**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**

**LUÍS GUSTAVO EGGER BARICHELLO**

# IDENTIFICAÇÃO DE OPORTUNIDADES DE REFATORAÇÃO E ANÁLISE DA QUALIDADE DE CÓDIGO POR MEIO DE MÉTRICAS DE CÓDIGO-FONTE

**DOIS VIZINHOS**

**2022**

**LUÍS GUSTAVO EGGER BARICHELLO**

# IDENTIFICAÇÃO DE OPORTUNIDADES DE REFATORAÇÃO E ANÁLISE DA QUALIDADE DE CÓDIGO POR MEIO DE MÉTRICAS DE CÓDIGO-FONTE

## Identification of Refactoring Opportunities and Code Quality Analysis through Source Code Metrics

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Gustavo Jansen de Souza Santos

**DOIS VIZINHOS**
**2022**

**LUÍS GUSTAVO EGGER BARICHELLO**

# IDENTIFICAÇÃO DE OPORTUNIDADES DE REFATORAÇÃO E ANÁLISE DA QUALIDADE DE CÓDIGO POR MEIO DE MÉTRICAS DE CÓDIGO-FONTE

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 23/junho/2022

---

Gustavo Jansen de Souza Santos
doutorado
Universidade Tecnológica Federal do Paraná

---

Francisco Carlos Monteiro Souza
doutorado
Universidade Tecnológica Federal do Paraná

---

Evandro Miguel Kuszera
doutorado
Universidade Tecnológica Federal do Paraná

**DOIS VIZINHOS**
**2022**

Dedico este trabalho aos meus pais que muito
me incentivaram ao longo desta caminhada.

## AGRADECIMENTOS

Agradeço primeiramente aos meus pais e a minha irmã, por tudo que já fizeram por mim. Pela luta, educação, comprometimento e tudo que me ensinaram ao longo desta caminhada para que eu estivesse em uma universidade federal hoje. Especialmente minha mãe, que sempre insistiu na minha formação e por ela estou aqui hoje.

Sou imensamente grato a todos que fizeram parte desta jornada, aos meus professores, família, amigos, colegas de trabalho e em especial ao meu orientador, professor Gustavo Jansen de Souza Santos por sempre me auxiliar nas decisões deste trabalho sem as quais eu não teria concluído este projeto.

A todos que fizeram parte da minha formação, o meu muito obrigado.

*Things don't have to change the world to be important.*
- Steve Jobs.

## RESUMO

Manter um software com uma estrutura interna complexa é uma tarefa difícil de se executar. Para isso, técnicas de refatoração são aplicadas com o objetivo de melhorar a estrutura interna, aumentar a qualidade e assim facilitar a manutenção do software. No entanto, os mecanismos para aplicar a refatoração de forma manual são complexos e exigem muito esforço técnico, e assim as atividades de refatoração acabam sendo deixadas de lado ou pouco executadas. Muitas IDEs utilizadas hoje facilitam a refatoração, mas a falta de confiança do desenvolvedor em utilizar as mesmas ainda é um obstáculo para se manter uma boa estrutura interna de um software. Este trabalho visa atender às necessidades citadas fornecendo uma ferramenta chamada *RefactorExtension* no formato de *plugin* para identificar a necessidade de refatorações e aplicá-las de forma automática em arquivos *JavaScript*. Para atingir esse objetivo, o *plugin* realiza a análise do código-fonte utilizando a técnica de conversão para AST, e para validar a efetividade da refatoração aplicada, são utilizadas métricas coletadas do código-fonte. Para avaliar a efetividade das refatorações aplicadas pelo *plugin*, foi realizado um estudo de caso aplicando a ferramenta em três diferentes projetos. A ferramenta apresentou índices de uma melhora efetiva nas métricas coletadas após a aplicação das refatorações nos projetos. Foi concluído com este trabalho que a aplicação de refatorações em um código-fonte pode resultar em grandes melhorias nas métricas de qualidade de um código-fonte, facilitando assim a sua manutenção e evolução natural.



**Palavras-chave:** refatoração; manutenção de software; métricas de código-fonte; javascript.

## ABSTRACT

Maintaining software with a complex internal structure is a difficult task to perform. For this, refactoring techniques are applied in order to improve code, increase quality and thus facilitate software maintenance. However, the mechanisms for manually refactoring are complex and require a great deal of technical effort; consequently, they are often left out or under-implemented by developers. Many IDEs used today make refactoring easier but do not apply them automatically, the developer still needs to identify refactoring opportunities and manually apply them. This study aims to provide a tool called *RefactorExtension* to use as a *plugin* for identifying refactoring opportunities and applying them automatically in JavaScript files. This *plugin* performs the analysis of the source code using the conversion to AST, and to validate the improvement of the applied refactoring, an analysis of the metrics of the source code is performed. To validate the *RefactorExtension plugin*, an experimental evaliation was conducted in three different projects. The tool showed signs of an effective improvement in the source code metrics after the application of refactorings. It was finished with this work that the application of refactorings directly affect the quality metrics of a source code, thus facilitating its maintenance and natural evolution.



**Keywords:** refactoring; software maintenance; source code metrics; javascript.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE QUADROS

## LISTAGEM DE CÓDIGOS FONTE

## LISTA DE ABREVIATURAS E SIGLAS

**Siglas**

| | |
|---|---|
| ACCM | Média da Complexidade Ciclomática por Método |
| AMLOC | Média da Complexidade Ciclomática por Método |
| AST | Árvore de Sintaxe Abstrata |
| CI/CD | *Continuous Integration / Continuous Delivery* |
| DCE | *Dead Code Elimination* |
| DCR | *Dead Code Removal* |
| DETEC | DETEC |
| ERR | Erro na refatoração |
| GFC | Gráfico de Fluxo de Controle |
| ICSME | *International Conference on Software Maintenance and Evolution* |
| IDE | Ambiente de Desenvolvimento Integrado |
| IMPAC | Impacto da refatoração |
| LLOC | Linhas de Código de Processamento Lógico |
| LOC | Linhas de Código |
| MS | Mapeamento Sistemático |
| NOM | Número de Métodos |
| PRE | Prediz a necessidade de refatorar determinado código-fonte |
| QP | Questão de Pesquisa |
| REFAC | Refatoração |
| RFC | Resposta de uma Classe |
| SANER | *Software Analysis, Evolution and Reengineering* |
| SCAM | *Source Code Analysis and Manipulation* |

| | |
|---|---|
| SUG | Sugestão de refatoração |

**SUMÁRIO**

# 1 INTRODUÇÃO

Durante o ciclo de vida de um software, melhorias e correções são necessárias para manter o software útil no contexto em que foi projetado. Para um software alcançar uma boa qualidade não basta este ser somente funcional, é necessário que o mesmo esteja bem estruturado, claro, coerente e de fácil manutenibilidade. Com o intuito de se atingir estes objetivos, pode-se utilizar de várias técnicas, como a refatoração (FOWLER, 2018). Segundo Fowler (2009) a refatoração se trata de uma mudança feita na estrutura interna do software, *i.e.*, no código fonte, fazendo com que a mesma se torne mais fácil de entender e mais barata de modificar, sem que altere seu comportamento observável.

Contudo, a etapa de manutenção do software é a etapa com maior custo financeiro e aplicação de esforço técnico. Erlikh (2000) identificou que do total dos investimentos realizados em um software nos Estados Unidos, 70% dos recursos eram destinados à manutenção, número que foi posteriormente ampliado para até 80% (INCOSE, 2015). A falta de boas práticas e a alta complexidade estrutural influenciam diretamente o aumento do indicador. Mesmo assim, atividades como refatoração são pouco executadas ou até mesmo esquecidas (FRANCO; HIRAMA; ROSSI, 2018).

Por se tratar de uma tarefa demorada e tediosa, que demanda de um grande esforço e com grande impacto no funcionamento atual do software, esta acaba sendo deixada de lado no dia-a-dia de um desenvolvedor (SANTOS, 2017). Como é um esforço que não resultará em uma nova funcionalidade ao software, mas sim que irá melhorar somente a estrutura interna, a refatoração é vista como um custo desnecessário e que não irá agregar ao software em um curto ou médio prazo.

À medida em que mudanças ocorrem no software, essas mudanças geralmente precisam ser feitas sob pressão de tempo e de orçamento. Ao aplicar mudanças sem considerar o impacto das mesmas sobre o software, o desenvolvedor compromete a qualidade estrutural do software. Mesmo que uma única alteração não tenha um impacto negativo, pequenas alterações acumuladas em futuras iterações do ciclo de vida do software podem causar problemas sérios a longo prazo (MENS; TOURWÉ, 2004).

Adicionar atividades de refatoração e análise de código-fonte ao processo de desenvolvimento é uma ação que garante uma boa qualidade do software, e assim evita o conjunto de decisões equivocadas e errôneas durante a implementação (SIMON; STEINBRUCKNER; LEWERENTZ, 2001; MAFFORT *et al.*, 2016). As leis de Lehman, também conhecidas como Leis da Evolução, afirmam algumas características importantes a se considerar durante o ciclo de desenvolvimento de um software (LEHMAN, 1996). Em especial, selecionamos três leis que interferem ativamente no ciclo de vida do software, e que impactam diretamente no planejamento, desenvolvimento e manutenção de um software:

- Lei da Mudança Contínua: Um software deve ser continuamente adaptado, senão torna-se aos poucos cada vez menos satisfatório.
- Lei da Complexidade: Se não forem tomadas medidas para reduzir ou manter a complexidade de um software, conforme ele é alterado sua complexidade irá aumentar progressivamente.
- Lei do Declínio da Qualidade: A qualidade do sistema decai ao longo do tempo, a menos que um esforço explícito seja realizado para que o sistema acomode melhor novas mudanças.

De modo geral, a evolução do software é naturalmente complexa (SANTOS, 2017). Desenvolvedores localizados em diferentes posições geográficas e a falta de comunicação só aumentam essa complexidade ao prestar manutenção (CHATZIGEORGIOU; MANAKOS, 2014). Por fim, o software se torna confuso, com rotinas emaranhadas e de complexo entendimento. Esses problemas durante a implementação precisam ser detectados e eliminados para que não se tornem um problema ou até mesmo uma barreira em uma evolução do software.

O presente projeto tem como objetivo suprir as necessidades citadas acima, fornecendo uma ferramenta focada em realizar refatorações em um código-fonte, com objetivo de realizar transformações de forma eficiente, tornando este código mais claro e compreensível. Para se realizar as transformações, serão utilizados conceitos de Engenharia de Software e métricas coletadas do código-fonte para classificar a qualidade de código, e a partir disso aplicar-se as transformações necessárias.

Trabalhos anteriores já identificaram que software que adota boas práticas de programação e refatoração de código possuem uma menor dívida técnica[1] e pouca demanda de esforço na manutenção corretiva (KALINOWSKI *et al.*, 2010). O uso destas boas práticas melhora o desempenho do software com respeito a custo, prazo, produtividade, qualidade, satisfação do cliente e retorno do investimento, assim consequentemente a empresa terá um processo mais enxuto de desenvolvimento.

Tradicionalmente para se atingir tal objetivo, alguns processos são estabelecidos no ciclo e desenvolvimento, como inspeção de um novo código implementado, ou até mesmo a utilização de ferramentas para análise de impacto de novas mudanças. Visando facilitar o processo referenciado acima, atualmente existem ferramentas que detectam oportunidades de refatoração e sugerem uma transformação em determinado ponto do código, porém poucas sugerem e realizam as transformações de forma automática.

Contudo, pesquisas identificaram que há uma certa desconfiança por parte dos desenvolvedores em utilizar essas ferramentas de transformação (NEGARA *et al.*, 2013). Os desenvolvedores preferem refatorar manualmente do que utilizar uma ferramenta automatizada, sendo que

[1] Metáfora utilizada para definir quando se compromete a qualidade do código-fonte para algum benefício imediato, contrai-se uma dívida onde o valor do empréstimo é o esforço economizado na execução da tarefa (ZAZWORKA *et al.*, 2013).

no final iria gerar o mesmo resultado com um menor esforço e aplicação de tempo (SANTOS, 2017).

O objetivo deste trabalho é apresentar uma ferramenta para detectar oportunidades de refatoração com base nas características de um código-fonte, assim sugerindo para o desenvolvedor tomar uma ação e aplicar um conjunto de refatorações. Tendo em vista que as ferramentas mais utilizadas nos dias de hoje para refatoração são mais focadas no design do código-fonte (TAVARES; FERREIRA; FIGUEREDO, 2018), este estudo propõe uma ferramenta focada na complexidade interna do código-fonte. A identificação das oportunidades de refatoração são detectadas com base nas características de um código-fonte, ou seja, métricas serão coletadas do mesmo para então apresentar a sugestão para o desenvolvedor de forma não invasiva, ficando a critério dele se aplica ou não a refatoração.

O projeto final será construído em três seções principais:

- Módulo para coleta de métricas de qualidade de código.
- Módulo para identificar oportunidades de refatoração.
- Módulo para aplicar refatoração de forma automática para a linguagem *JavaScript*, com enfoque nas refatorações de Complexidade Ciclomática, *Move Method*, *Inline Method* e Remoção de código-fonte morto.

Por fim, a ferramenta irá facilitar a tediosa e complexa atividade de refatorar um código-fonte, em que muitas vezes é executada de forma manual por desenvolvedores, transformando em uma atividade de execução automática, aplicando a refatoração no código-fonte de forma simples e eficiente.

Para se atingir o objetivo geral, os seguintes objetivos específicos foram definidos:

- Levantamento dos aspectos conceituais.
- Seleção das métricas de código-fonte.
- Seleção de técnicas para identificar oportunidades de refatoração.
- Levantamento de ferramentas de refatoração automática.
- Desenvolvimento do módulo para coleta de métricas.
- Desenvolvimento do módulo para identificar as oportunidades de refatoração.
- Desenvolvimento do módulo para aplicar refatoração automática.
- Avaliação experimental.

Atingindo tais objetivos específicos, surge uma ferramenta funcional e objetiva para auxiliar a manter um código-fonte de qualidade.

# 2 CONTEXTO

Um dos princípios básicos no desenvolvimento de software é a necessidade de evolução. A medida em que mudanças ou alterações são realizadas, a estrutura do software e seu código-fonte podem se tornar cada vez mais complexas, assim diminuindo a qualidade e aumentando os custos com a manutenção (MENS; TOURWÉ, 2004). Ao se implementar uma nova funcionalidade, desenvolvedores podem basicamente seguir duas opções: implementar rapidamente sem se preocupar com os impactos no legado do software, ou pode-se ajustar a estrutura do software para que a nova funcionalidade se acomode melhor na estrutura atual, e que seja fácil de compreender e manter futuramente. Ao seguir o primeiro caso, surgirá um débito técnico no projeto.

Uma alternativa para melhorar a qualidade interna de um software é a utilização de ferramentas e técnicas para preservar uma boa estrutura, como a refatoração. Utiliza-se refatoração com o objetivo de se alcançar benefícios como: reutilização de código-fonte, reduzir o acoplamento entre classes, adoção de boas práticas durante o desenvolvimento do software, e a facilitação do processo de manutenção e evolução do software. Por fim, a refatoração ajuda a obter um melhor entendimento do código para futuras manutenções.

Por outro lado, as refatorações devem ser aplicadas continuamente, e não ser aplicada poucas vezes durante o ciclo de vida de um software, práticas como essa, quando aplicadas, geram grandes impactos e assim são consideradas como ineficazes ou que o retorno não vale o esforço (MENG; KIM; MCKINLEY, 2013). Para que a aplicação de refatoração tenha melhores resultados, deve ser aplicada de forma contínua durante o ciclo de vida do software. Assim, os problemas de manutenibilidade são identificados e corrigidos de forma mais rápida. Embora importante, a refatoração não é um processo simples, uma vez que se deve garantir que o comportamento do software seja o mesmo no pré e pós-refatoração, para que a refatoração não gere um resultado inverso ao esperado.

No entanto, o processo de se aplicar refatoração de forma manual é complexo por ser uma atividade repetitiva, demorada e que se deve aplicar as transformações com atenção para evitar que cause novos problemas ao usuário. Desta forma, estabelecer um processo de refatoração durante o desenvolvimento do software, se torna inviável e é deixado de lado (MENG; KIM; MCKINLEY, 2013).

A principal motivação para a execução deste projeto parte do interesse em melhorar a qualidade de um software e facilitar a manutenção, assim diminuindo custos e aumentando a satisfação dos usuários. Para se atingir os objetivos descritos acima, as ferramentas de refatoração automática podem auxiliar os desenvolvedores durante a manutenção, trazendo economias e maior segurança durante o desenvolvimento.

A avaliação experimental deste estudo foi realizada aplicando a ferramenta desenvolvida em alguns projetos *open-source* selecionados, a fim de validar a efetividade das refatorações propostas. As refatorações focadas serão as que possam melhorar a qualidade interna de um

software, e alguns principios da metodologia de *Clean Code*, como por exemplo: *Dry (Don't Repeat Yourself)*, funções pequenas e objetivas, curtos trechos de comentários, entre outros.

# 3 ASPECTOS CONCEITUAIS

Este capítulo visa apresentar os conceitos fundamentais para o entendimento do trabalho proposto, além da sua contextualização com o estado da arte. O capítulo está organizado da seguinte forma. Na Seção 3.1 são apresentados os conceitos relacionados à refatoração de código-fonte. A Seção 3.2 apresenta o conceito de métricas de código-fonte e como pode-se extrair e utilizar as mesmas. Na Seção 3.3 serão apresentadas algumas ferramentas para aplicação automática de refatoração de código-fonte.

## 3.1 Refatoração

Segundo Fowler (2009), refatoração é uma mudança feita na estrutura interna do software, fazendo com que a mesma se torne mais fácil de entender e mais barata de modificar, sem que altere seu comportamento observável. Aplicamos refatoração em um código-fonte com o objetivo de manter um software bem estruturado com o passar do tempo e com as mudanças que o software virá a sofrer.

No estudo conduzido por Sae-Lim, Hayashi e Saeki (2019), os autores separaram a refatoração em duas abordagens diferentes, a refatoração reativa e a refatoração proativa.

- **Refatoração Reativa**: É definida como refatoração reativa aquela em que os desenvolvedores se concentram em trechos de código que já estão complexos e difíceis de se prestar manutenção, em vez de lidar com todas as partes do código-fonte. Isto é, os desenvolvedores não impedem a ocorrência de novos problemas no design do código-fonte. Uma vantagem desta abordagem é que os desenvolvedores terão seu foco aplicado somente nas partes mais problemáticas do código-fonte, em vez de lidar com o software como um todo. Porém, os desenvolvedores só irão refatorar quando o trecho do código-fonte já estiver complexo.

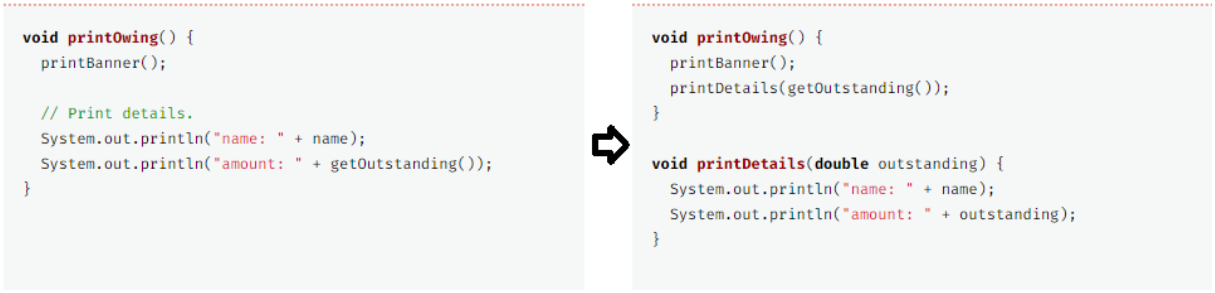
- **Refatoração Proativa**: É definida como refatoração proativa aquela em que o desenvolvedores refatoram o código-fonte antes que ele fique complexo ou com um design ruim, assim evitando dívidas técnicas que possam impedir o crescimento do software. Esta abordagem permite que se tenha uma visão geral da qualidade do código-fonte, e não somente dos módulos com problemas no design. Isto é, os desenvolvedores irão controlar para que nenhuma parte do código-fonte fique com um design ruim, em vez de esperar que fique complexo para que somente assim aplicar a refatoração.

Fowler (2009) formalizou os principais tipos de refatoração em um catálogo, e enfatiza que a refatoração não é uma tarefa especial a se executar em um plano de projeto, se realizada bem, é uma parte regular do processo de desenvolvimento. Abaixo serão listados alguns

tipos de refatoração existentes na maioria das Ambiente de Desenvolvimento Integrado (IDE) atualmente:

***Extract Method.*** A refatoração consiste em extrair um determinado trecho de código de um método, e criar um novo método na mesma classe. O objetivo da refatoração é a criação de métodos menores reduzindo o método original, a redução de duplicação em trechos repetidos numa mesma classe, além da criação de métodos mais objetivos e com nomes mais claros. A Figura 1 apresenta um código-fonte onde foi aplicado o tipo de refatoração *Extract Method*.

**Figura 1 – Exemplo de uma sugestão de refatoração utilizando Extract Method**




**Fonte: (REFACTORING.GURU, 2014).**

***Inline Method.*** Quando o corpo do método é mais obvio que o próprio método, implementamos em somente um método, substituímos a chamada do método antigo e o deletamos.

A Figura 2, apresenta um código-fonte onde foi aplicado o tipo de refatoração *Inline Method*.

**Figura 2 – Exemplo de uma sugestão de refatoração utilizando a técnica *Inline Method***


**Fonte: (REFACTORING.GURU, 2014).**

***Move Method.*** Extrair determinado método de uma classe para a classe onde é mais usado.

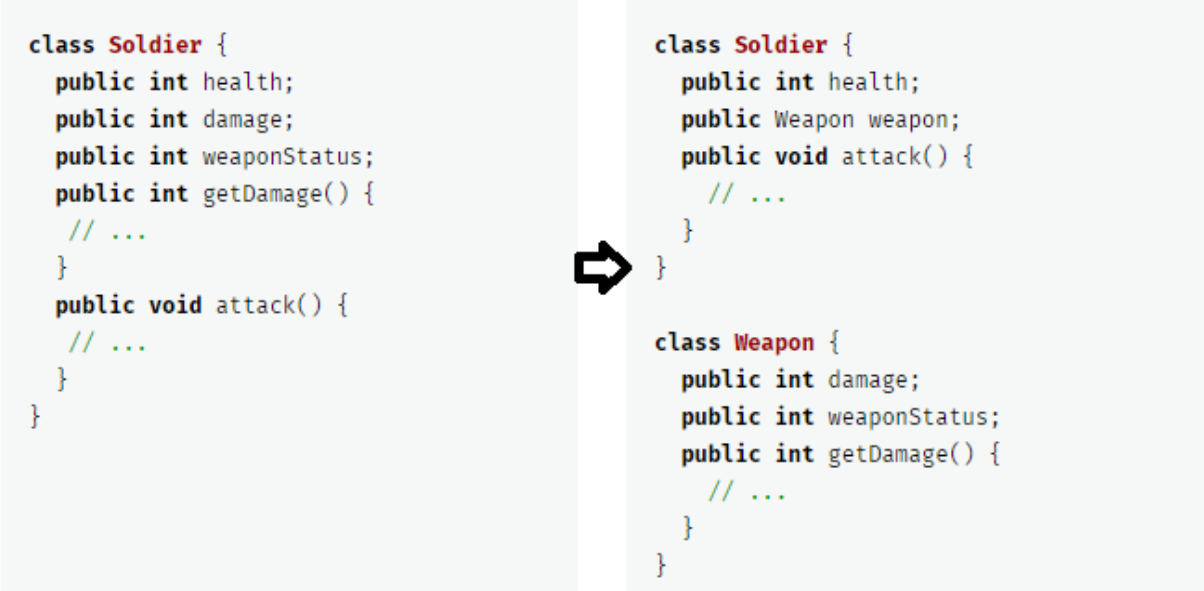
***Extract Class.*** Quando uma classe faz a atividade de duas, em vez disso deve ser criado uma nova classe e mover os atributos e métodos respectivos.

A Figura 3, apresenta um código-fonte onde foi aplicado o tipo de refatoração *Extract Class*.

Pode-se consultar as refatorações no catálogo de refatorações do Refactoring Guru[1].

[1] https://refactoring.guru/refactoring

**Figura 3 – Exemplo de uma sugestão de refatoração utilizando a técnica Extract Class**




**Fonte: (REFACTORING.GURU, 2014).**

## 3.2 Métricas de código-fonte

Métricas de código-fonte são um conjunto de características que podem ser utilizadas na avaliação e melhoria da qualidade do projeto (SILVA; TSANTALIS; VALENTE, 2016). A utilização correta das métricas extraídas de um código-fonte, além de permitir uma análise da qualidade do mesmo, auxilia na identificação de oportunidades de refatoração e são geralmente aplicáveis ao paradigma orientado a objetos.

Abaixo serão listadas as principais métricas que podem ser extraídas utilizando ferramentas automatizadas para análise estática de código (MEIRELLES, 2013), o nome completo de cada uma das métricas foi especificado na lista de siglas.

- **Linhas de Código (LOC)**: Número de linhas executáveis do código-fonte, sem considerar espaços em branco ou comentários. Métrica de tamanho comumente utilizada.
- **Média da Complexidade Ciclomática por Método (AMLOC)**: Avalia a distribuição de linhas de código executável entre os métodos de uma classe. Indicador utilizado para identificar a coesão e a reutilização de código-fonte.
- **Número de Métodos (NOM)**: Número de métodos de uma classe, pode representar uma classe bem ou mal estruturada.
- **Média da Complexidade Ciclomática por Método (ACCM)**: Calcula a complexidade média dos métodos de uma classe, com base na complexidade dos fluxos implementados nos métodos.

- **Resposta de uma Classe (RFC)**: Calcula o número total de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto de uma classe.

Acima foram apresentadas métricas de código-fonte relacionadas ao tamanho, coesão, acoplamento e atributos estruturais, que dão uma visão geral sobre a qualidade do código. Como discutido em cada definição, as métricas têm uma grande relevância por ter associação direta com alguns problemas na qualidade de um código-fonte, como propensão a erros de implementação, baixo nível de compreensão ou estrutura complexa (MALERBA, 2010).

## 3.3 Aplicação de Refatoração de Forma Automática

Embora existam ferramentas para aplicar refatoração, cerca de 90% das refatorações são realizadas de forma manual (GE; DUBOSE; MURPHY-HILL, 2012). Transformações manuais são mais propensas a erros pelo fato de ser uma atividade complexa de se realizar. Desta forma, se realizadas sem cautela, a refatoração pode ter um efeito contrário do pretendido, gerando novos defeitos e podendo causar problemas ao usuário final.

Assim, a utilização de mecanismos para aplicar a refatoração de forma automatizada surgem como alternativa segura. Essas ferramentas auxiliam no dia-a-dia de um desenvolvedor, com o objetivo de facilitar esse processo de refatoração e garantir uma melhor qualidade do software. Muitas *IDE's* utilizadas hoje, como *Visual Studio Code (VsCode)*[2], *IntelliJ*[3] e *Eclipse*[4], se baseiam no catálogo de Fowler (2009), e disponibilizam alguns métodos simples de refatoração.

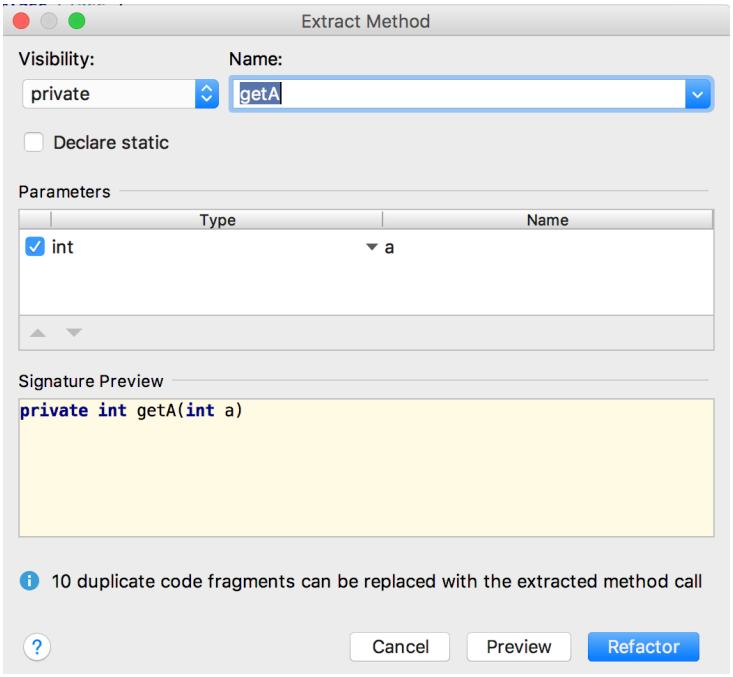
Um exemplo disso é a Figura 4, que mostra a *IDE IntelliJ*, em que esta sugere a utilização da técnica Extract Method de refatoração.

A refatoração apresentada na Figura 4 tem como finalidade criar um método para realizar o acesso a um atributo do método. Para aplicar a refatoração, o desenvolvedor deve selecionar o trecho de código-fonte que deseja extrair. Após aplicar a técnica de extração, a *IDE* apresenta uma pré-visualização da alteração para o desenvolvedor confirmar a alteração, como mostra a Figura 5.

Nesse exemplo, a *IDE* automatizou a técnica de refatoração Extract Method corretamente, porém o desenvolvedor precisou identificar o trecho de código-fonte desejava refatorar e qual o tipo de refatoração que desejava aplicar. Este trabalho propõe uma abordagem similar de refatoração, onde deve-se identificar oportunidades de refatoração (a exemplo da consequência do Extract Method em outros trechos de código-fonte) e aplicar automaticamente as sugestões. Porém, neste trabalho o foco é dado em refatorações com o objetivo de diminuir complexidade como *Extract Method* e *Move Attribute*, em que as refatorações sugeridas ao desenvolvedor se-

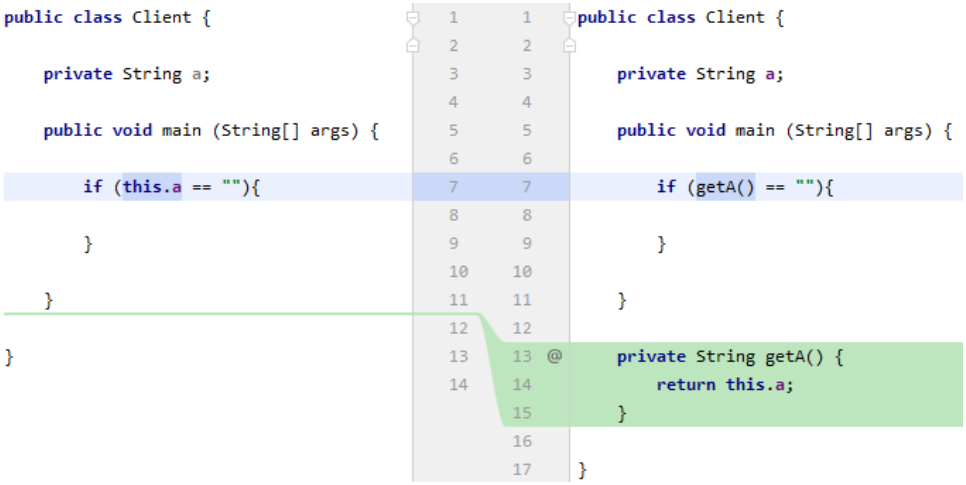
[2] https://code.visualstudio.com
[3] https://www.jetbrains.com/pt-br/idea
[4] https://www.eclipse.org

**Figura 4 – Exemplo de uma aplicação de refatoração utilizando a técnica Extract Method na *IDE IntelliJ***




**Fonte: Autoria própria.**

**Figura 5 – Exemplo de uma pré-visualização após aplicar a técnica de refatoração Extract Method na *IDE IntelliJ***




**Fonte: Autoria própria.**

rão com base nas características coletadas do código-fonte implementado. Consequentemente, pretende-se tornar o código-fonte mais limpo e de fácil entendimento.

# 4 MAPEAMENTO SISTEMÁTICO

A fim de produzir um conteúdo científico de qualidade, é necessário fazer um levantamento de evidências referente a área de pesquisa em que se pretende propor uma solução. Além de reunir, também é importante realizar uma avaliação dos dados coletados. Nesse contexto, existem diferentes técnicas da Engenharia de Software Baseada em Evidências (ESBE) que auxiliam neste processo (KITCHENHAM, 2004).

Dentre as técnicas destacamos o Mapeamento Sistemático (MS). O MS executado neste trabalho conduziu a busca por sólidas referências que evidenciam a utilização de ferramentas de refatoração de código-fonte utilizando métricas. Para realizar o MS foi utilizado o método proposto por (KITCHENHAM, 2004), que sugere três fases: Planejamento, Condução e Análise.

Este capítulo é organizado da seguinte maneira: na Seção 4.1 descreve-se o planejamento do MS, destacando a Questão de Pesquisa (QP); a estratégia e as fontes de busca utilizadas; bem como critérios de seleção dos estudos; a concepção das categorias dos estudos e a extração de dados. Na Seção 4.2 é apresentado o processo de condução do MS de acordo com a QP. A Seção 4.3 aborda a síntese e análise dos estudos primários. Na Seção 4.4 apresenta-se as ameaças à validade deste MS. E por fim, a Seção 4.5 dispõe as considerações finais deste Capítulo.

## 4.1 Planejamento do Mapeamento Sistemático

Nesta fase é de suma importância que seja estabelecido um protocolo que oriente a condução do MS. Esse artefato descreve os processos e métodos a serem empregados nessa atividade como: definição da QP, as bases de busca a serem consultadas, a *string* de busca, os critérios para inclusão e exclusão de estudos, além da forma que serão extraídos os dados.

A partir dos conceitos apresentados acima, criou-se as seguintes Questões de Pesquisa:

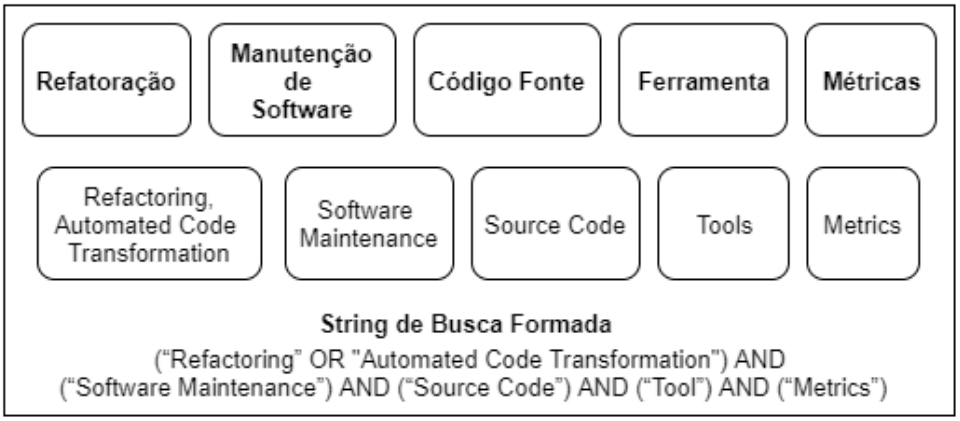
- **QP1: Quais são as evidências existentes de ferramentas para identificação de oportunidade de refatoração utilizando métricas de código-fonte?**
- **QP2: Quais são as evidências existentes de ferramentas para aplicação automática de refatoração em código-fonte?**

Com a QP1 espera-se identificar estudos propondo metodologias, ferramentas, notação textual e/ou abordagens para identificar oportunidades de refatoração em um código-fonte utilizando métricas coletadas a partir de um código-fonte. A QP2 já é atendida por um trabalho anterior (TAVARES; FERREIRA; FIGUEREDO, 2018), mas com o novo estudo feito com a QP2 espera-se identificar estudos propondo metodologias, ferramentas, notação textual e/ou abordagens para aplicação automática de refatoração em código-fonte que foram construídas recentemente.

### 4.1.1 Estratégia para Busca Automática

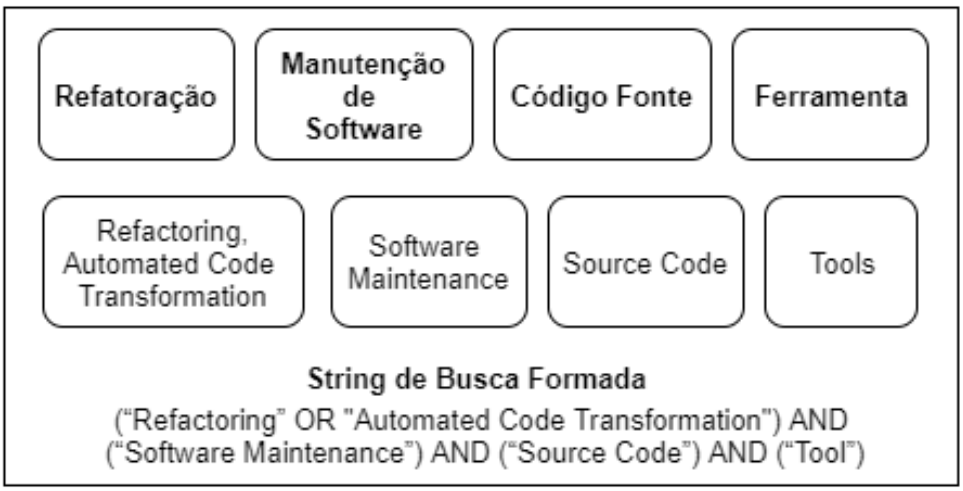
A estratégia desenvolvida constitui na escolha das fontes de busca automática e na construção das *strings* de buscas correspondentes a cada QP. Foram construídas duas QPs com o objetivo de separar as buscas para atender duas atividades planejadas para este estudo. Para construir a *string* de busca, foram seguidos três passos: (i) reconhecimento das palavras-chaves (*keywords*) relevantes à cada QP; (ii) utilização dos operadores "AND" e "OR" para concatenar os principais termos, conforme pode ser visto nas Figuras 6 e 7.

**Figura 6 – Definição da primeira string de busca**




**Fonte: Autoria própria.**

**Figura 7 – Definição da segunda string de busca**




**Fonte: Autoria própria.**

Após construção das strings de busca aplicamos a busca automática. Na Tabela 1 estão listadas as fontes de busca utilizadas. Executamos a busca automática somente nas fontes de busca listadas na Tabela 1, pois a área de transformação automática de código já é bem consolidada, e existem bastantes conferências para a área de Manutenção, a exemplo da *Soft-*

**Tabela 1 – Fontes de Busca Automática do Mapeamento Sistemático**

| Fonte/Base de Busca | Endereço Eletrônico |
|---|---|
| IEEE | www.ieeexplore.ieee.org |
| ACM | https://dl.acm.org |

**Fonte: Autoria própria.**

*ware Analysis, Evolution and Reengineering* (SANER), *International Conference on Software Maintenance and Evolution* (ICSME) e *Source Code Analysis and Manipulation* (SCAM).

### 4.1.2 Critérios para Seleção de Estudos Primários

Após a criação e execução das strings nas fontes de busca, o resultado passou por uma seleção dos estudos retornados. Este processo foi conduzido através de uma filtragem, que consiste na inclusão ou exclusão dos estudos que de fato possam contribuir para a elucidação das Questões de Pesquisa. Os critérios de inclusão e exclusão apresentados abaixo, foram utilizados para a filtragem de ambas QP. No contexto desse MS, o processo de filtragem seguiu os seguintes critérios:

- **Critérios de Inclusão (CI)**
  - **CI1:** estudos primários que evidenciam a utilização de ferramentas de refatoração automática utilizando métricas de código-fonte.
  - **CI2:** estudos primários que apresentam alguma metodologia, ferramenta, notação textual ou abordagem que viabilize o desenvolvimento de uma ferramenta para aplicar refatoração de forma automática.
- **Critérios de Exclusão (CE)**
  - **CE1:** estudos primários que não sejam trabalhos completos.
  - **CE2:** estudos primários incompletos.
  - **CE3:** estudos primários indisponíveis para download.

### 4.1.3 Extração de Dados

Para a conduzir a extração dos dados dos estudos, foram realizados os seguintes passos: leitura do título e *abstract*; leitura da introdução e conclusão; e leitura na íntegra.

Após a leitura integral de cada estudo foi criada uma planilha utilizando a ferramenta Planilhas do Google Drive, a fim de organizar a extração de dados como: título, ano da publicação, objetivo e autores.

## 4.2 Condução do Mapeamento Sistemático

A identificação e seleção de estudos relevantes à QP1 reuniu um total de nove estudos, sendo que em cada fonte de busca retornaram: sete estudos na IEEE e dois na ACM. A partir da execução da string nas fontes de busca, obtivemos um total de 125 estudos. Em seguida, quatro estudos foram excluídos por serem repetidos. Dos 121 estudos restantes foram lidos títulos e os resumos, e aplicados os critérios de inclusão e exclusão, assim foram incluídos 27 estudos. No passo 2, foram lidas a introdução e conclusão, e novamente aplicados os critérios de inclusão e exclusão aos 27 estudos, restando 16 estudos incluídos. No passo 3, dos 16 estudos lidos na íntegra, nove estudos foram incluídos. A Figura 8 ilustra a condução da pesquisa para a QP1.

A identificação e seleção de estudos relevantes à QP2 reuniu um total de 17 estudos, sendo que em cada fonte de busca retornaram: 10 estudos na IEEE e sete na ACM. A partir da execução da string nas fontes de busca, obtivemos um total de 810 estudos. Em seguida, 54 estudos foram excluídos por serem repetidos. Dos 756 estudos restantes foram lidos títulos e os resumos, e aplicados os critérios de inclusão e exclusão, assim foram incluídos 53 estudos. No passo 2 foram lidas a introdução e conclusão, e novamente aplicados os critérios de inclusão e exclusão aos 53 estudos, restando 26 estudos incluídos. No passo 3, dos 26 estudos lidos na íntegra, 17 estudos foram incluídos. A Figura 9 ilustra a condução da pesquisa para a QP2.

## 4.3 Análise e Síntese dos Estudos Primários

### 4.3.1 Questão de Pesquisa 1

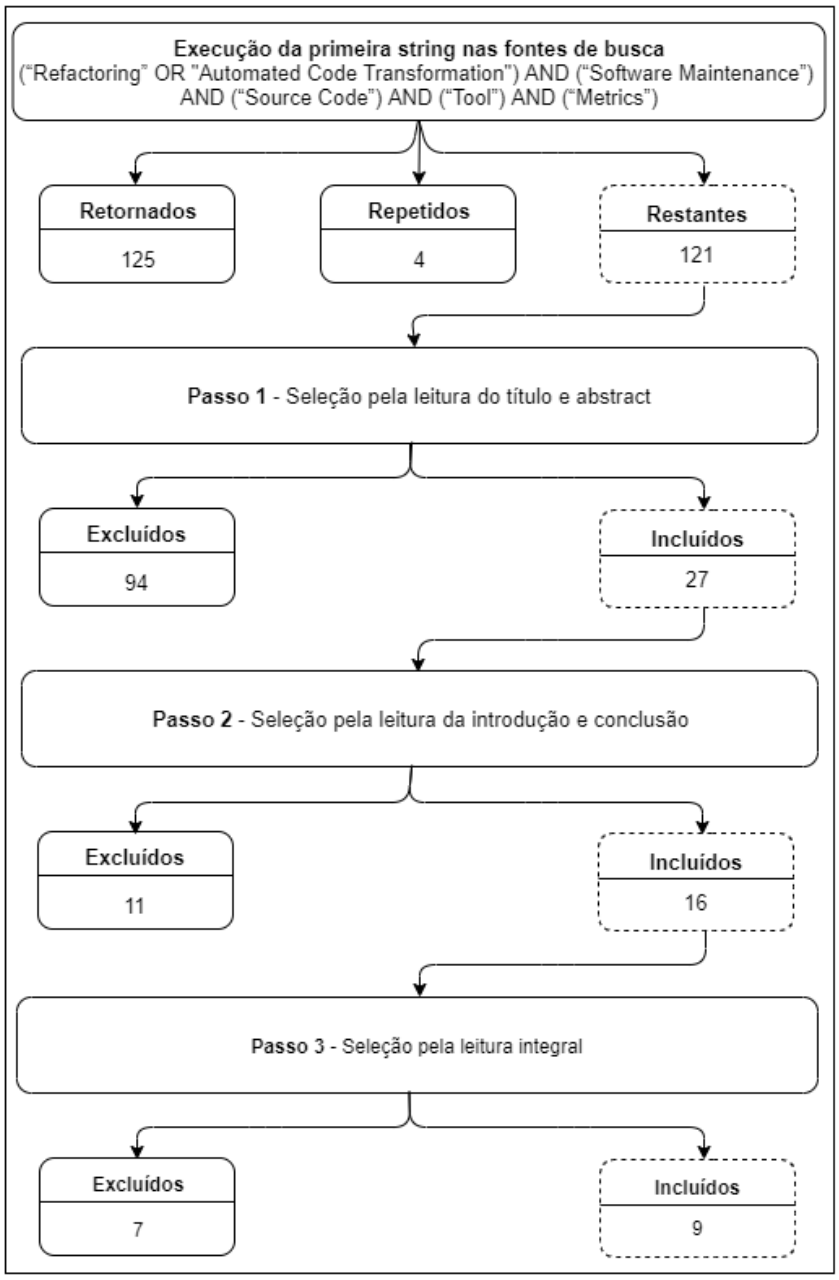
Com a busca automática realizada para a QP1, identificamos um total de nove estudos que propuseram metodologias, ferramentas, notação textual e/ou abordagens para identificar oportunidades de refatoração em um código-fonte utilizando métricas coletadas a partir de um código-fonte. A partir disso identificamos que o assunto começou a ser estudado em 2005 e é um tema que possui pesquisas recentes, como se pode observar na Figura 10.

Os estudos podem ser divulgados de diferentes formas, como: congressos, workshops, simpósios, periódicos, conferências, entre outros. Neste MS foram identificados dois tipos de publicação: conferência e periódico. Sendo a maior parte dos estudos resultantes foram publicadas em conferências, como pode ser observado na Figura 11.

A partir da busca conduzida, conclui-se que os estudos evidenciam a existência de ferramentas que identificam oportunidades de refatoração utilizando métricas de código-fonte. A grande maioria das ferramentas ou dependem de uma entrada das métricas coletadas utilizando outra ferramenta para análise de código-fonte.

Mohan, Greer e McMullan (2019) abordam a cobertura de uma atividade de refatoração. Os autores citam que o desenvolvedor não deve ficar preso transformando somente uma parte

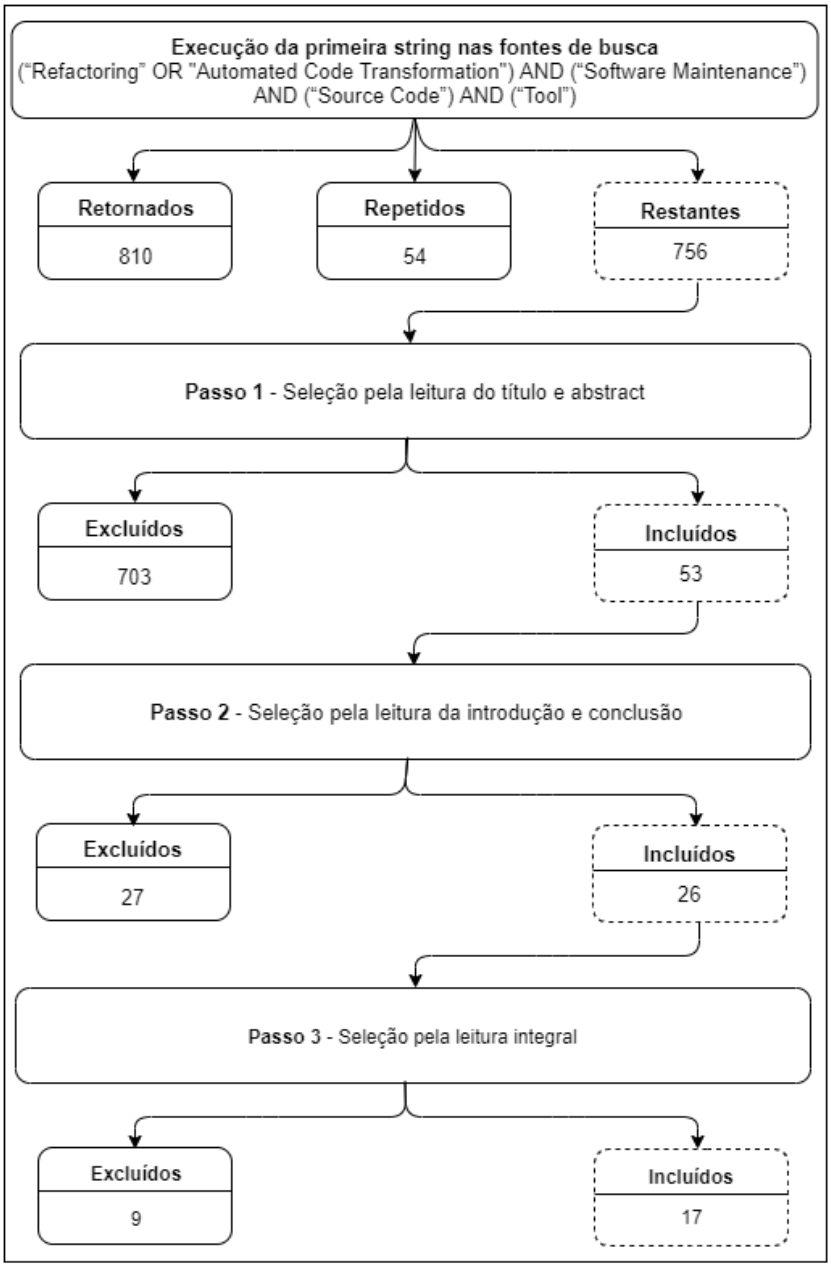
**Figura 8 – Processo de seleção dos estudos primários**




**Fonte: Autoria própria.**

do software, mas pelo contrário a refatoração deve atingir a maior parte do software possível. Para isso é proposta uma ferramenta chamada *MultiRefactor* tendo como objetivo aumentar a cobertura das refatorações e diminuir o número de refatorações redundantes. A entrada para a ferramenta funcionar é um código-fonte em Java, com todas as dependências suficientes para ele compilar, e a saída será um código-fonte refatorado, com base em métricas de qualidade. A ferramenta utiliza métricas como acoplamento entre as classes, coesão, abstração como funções para medir a qualidade do código. Após a identificação das refatorações a ferramenta utiliza as métricas para medir qual refatoração traria mais resultados com um menor impacto.

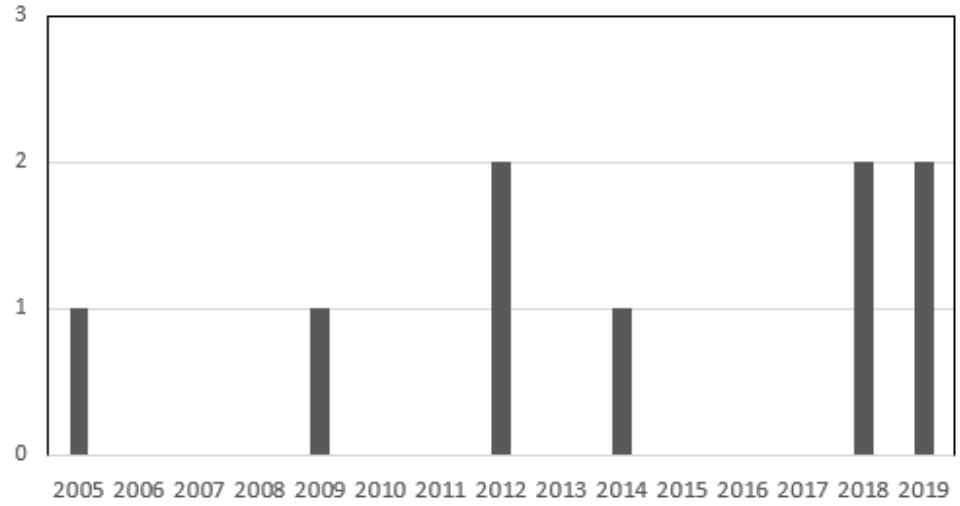
**Figura 9 – Processo de seleção dos estudos primários**




**Fonte: Autoria própria.**

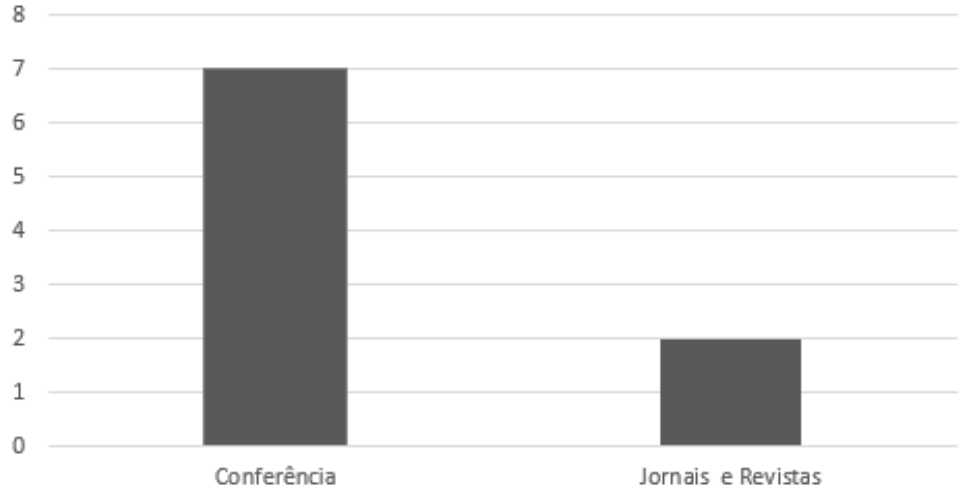
Junior, Filho e Araujo (2016) relatam a dificuldade em se prestar manutenção em um software complexo, e os transtornos que isso causa ao decorrer do projeto. O estudo apresenta a utilização da métrica de complexidade ciclomática e como sua aplicação em atividades de refatoração auxiliam em tornar um código-fonte mais fácil de se prestar manutenção. Para isso é apresentado no estudo uma abordagem para detectar a complexidade desnecessária em um código-fonte, a partir de um Gráfico de Fluxo de Controle (GFC), em que a entrada é um GFC da estrutura de um código-fonte, e a saída seria apresentar os pontos na qual se tem uma complexidade desnecessária.

**Figura 10 – Número de publicações por ano para QP1**




**Fonte: Autoria própria.**

**Figura 11 – Quantidade de estudos por tipo de publicação para a QP1**




**Fonte: Autoria própria.**

Herbold, Grabowski e Neukirchen (2009) apresentam a importância da atividade de inspeção do código-fonte baseado em métricas, para a atividade de verificação e validação. Com o objetivo de aumentar o alcance da análise de código-fonte, o estudo propõe uma ferramenta chamada AddFix utilizada em forma de plugin na IDE Eclipse, em que a proposta da ferramenta é sugerir correções rápidas no código-fonte para o desenvolvedor e sugestões para resolver erros de semântica no código-fonte. A ferramenta espera uma entrada de um arquivo XML contendo os marcadores gerados por uma ferramenta de análise estática de código-fonte, ou seja, uma ferramenta externa faz uma análise do código utilizando métricas como acoplamento e complexidade ciclomática e por fim gera um arquivo marcando a linha que deve ser refatorada.

Cinnéide *et al.* (2012), Nyamawe *et al.* (2018) e Sae-Lim, Hayashi e Saeki (2019) relatam que ao implementar novas funcionalidades em um software, as antigas são esquecidas, ou pouco mantidas, e relaciona isso com a decadência da estrutura interna do software. Os autores

argumentam que muitas ferramentas disponíveis hoje são efetivas na sugestão de refatoração, mas deixam a decisão final para o desenvolvedor, e que os mesmos muitas vezes escolhem a opção mais fácil de se implementar por questão de tempo. Os artigos propõem abordagens para recomendar soluções de refatoração, e validam a efetividade da aplicação da mesma utilizando métricas antes e depois de aplicar a refatoração. As métricas utilizadas para identificar oportunidades de refatoração são LOC, Encapsulamento e Complexidade Ciclomática.

Liu, Xu e Zou (2018) recomendam a refatoração para garantir uma boa qualidade do software e diminuir o desperdício em um projeto de software. Para isso, o estudo aplica uma ferramenta que utiliza conceitos de aprendizagem de máquina, para identificar estruturas ruins no código-fonte. A ferramenta utiliza métricas como coesão, estabilidade e acoplamento para classificar as principais refatorações a se aplicar.

Munro (2005) e Danphitsanuphan e Suwantada (2012) relatam a importância de se acompanhar um software no processo de manutenção, utilizando análise de código-fonte para evitar que a sua estrutura interna venha a decair. Os autores relatam a dificuldade em encontrar trechos do código com uma estrutura complexa. Para isso o estudo propõe uma ferramenta para detectar trechos do código que precisam ser refatorados. Para realizar a detecção foram utilizadas as seguintes métricas: número de métodos, linhas de código, complexidade ciclomática, coesão e profundidade da árvore de herança. Após a detecção, é apresentado ao desenvolvedor para que o mesmo aplique as transformações necessárias.

### 4.3.2 Questão de Pesquisa 2

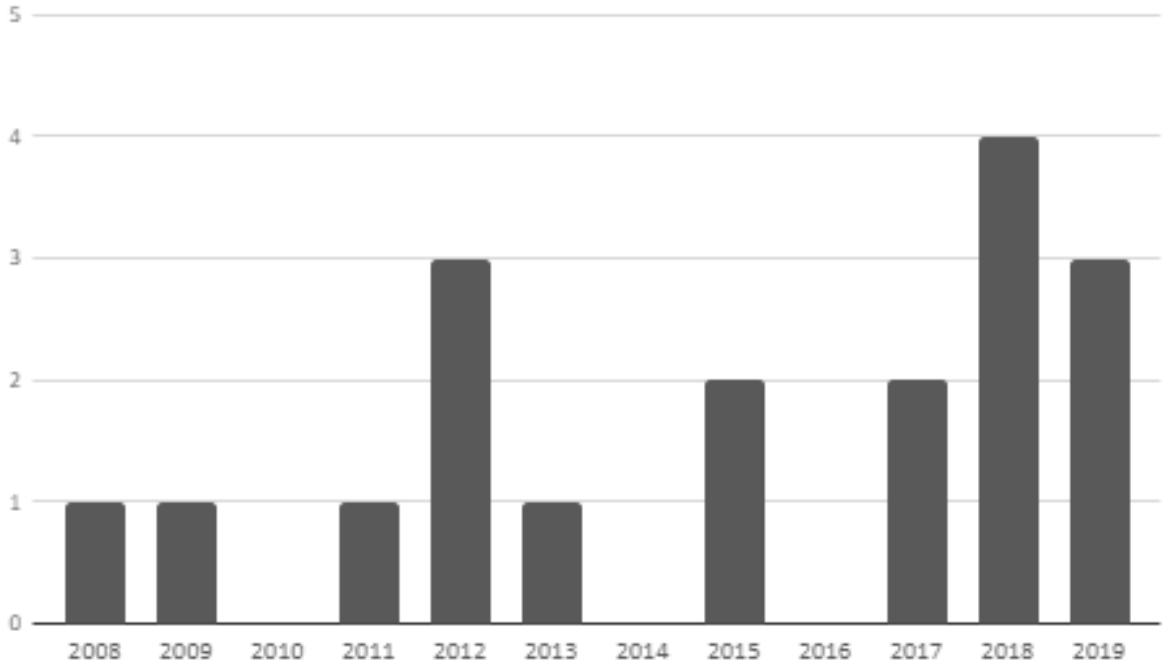
Com a busca automática realizada para a QP2, identificamos um total de 17 estudos que apresentam metodologias, ferramentas, notação textual e/ou abordagens para aplicação automática de refatoração em um código-fonte. A partir disso identificamos que o assunto começou a ser estudado em 2008 e é um tema que possui pesquisas recentes, como se pode observar na Figura 12.

Os estudos podem ser divulgados de diferentes formas, como: congressos, workshops, simpósios, periódicos, conferências, entre outros. Neste MS foram identificados os seguintes tipos de publicação: conferência, periódicos e workshops. Sendo a maior parte dos estudos resultantes foram publicadas em conferências, como pode ser observado na Figura 13.

Durante o estudo realizado na QP2 foi encontrado um MS realizado no ano de 2018, que tem como objetivo verificar quais as ferramentas para apoiar a refatoração de código-fonte disponíveis na literatura (TAVARES; FERREIRA; FIGUEREDO, 2018) que será evidenciado no decorrer desta seção.

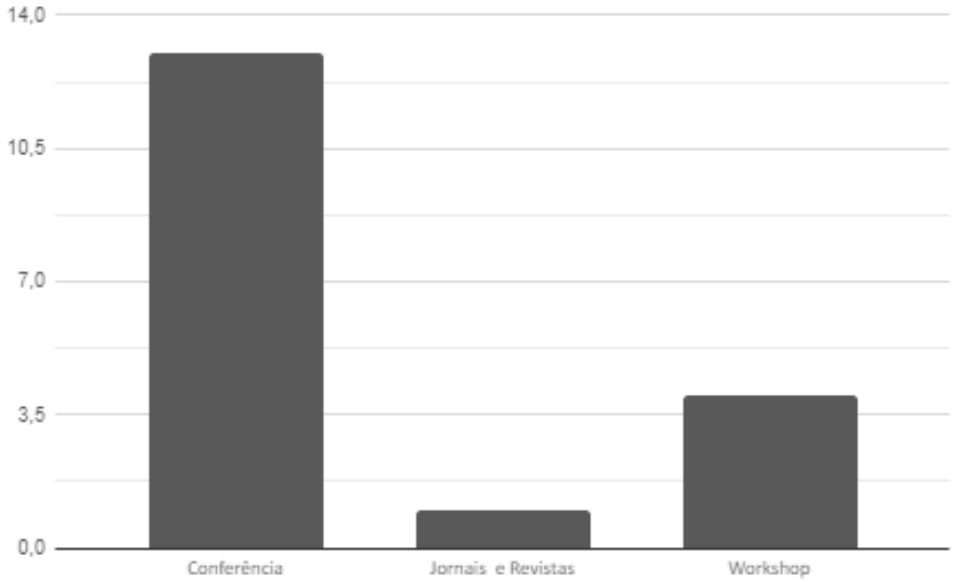
A partir da busca conduzida, conclui-se que os estudos evidenciam a existência de ferramentas para aplicação automática de refatoração em um código-fonte, com o objetivo de facilitar a transformação e diminuir o risco de erros.

**Figura 12 – Número de publicações por ano para a QP2**




**Fonte: Autoria própria.**

**Figura 13 – Quantidade de estudos por tipo de publicação para a QP2**




**Fonte: Autoria própria.**

Mohan, Greer e McMullan (2019) propuseram um estudo que também foi selecionado para a QP1 em que utiliza métricas para aplicar as refatorações. Com o objetivo de melhorar a qualidade da estrutura interna, a ferramenta utiliza técnicas de refatoração como Extract Class, Extract Interface, Extract Method e Move Attribute. Malathi e Sudhakar (2018) apresentam uma

abordagem semelhante de ferramenta para refatoração, mas utilizando a ferramenta Feature-Oriented Domain Analysis (FODA).

Kimura *et al.* (2012) apresentam a dificuldade em manter uma boa qualidade para a estrutura interna do software que acompanhe a evolução do mesmo. Para isso, os autores sugerem uma ferramenta que aplique refatorações utilizando algumas técnicas de refatorações, como Extract Method, Move Class, Move Atribute entre outras técnicas descritas por Fowler (2018), e com a validação da ferramenta os autores argumentaram que a mesma é capaz de aplicar refatorações em trechos candidatos de forma automática.

Herbold, Grabowski e Neukirchen (2009) apresentam a importância da atividade de inspeção do código-fonte baseado em métricas, para a atividade de verificação e validação. Com o objetivo de aumentar o alcance da análise de código-fonte, o estudo propõe uma ferramenta chamada AddFix utilizada em forma de plugin na *IDE* Eclipse, em que a proposta da ferramenta é sugerir correções rápidas no código-fonte para o desenvolvedor, sugestões para resolver erros de semântica no código-fonte. A ferramenta utiliza a técnica de refatoração Extract Method.

Wyrich e Bogner (2019), Ribeiro e Borba (2008) e Sharma (2012) relatam que a refatoração automática oferece muito potencial para aumentar a eficácia e eficiência da melhoria da qualidade do código. No estudo de Wyrich e Bogner (2019), foi apresentada a ferramenta Refactoring-Bot, que integra com os desenvolvedores por meio de solicitações *pull* no repositório do projeto para que posteriormente os desenvolvedores façam uma revisão da refatoração. A ferramenta aplica refatorações como: Move Method e Move Class, também é utilizada para remover comentários desnecessários e parâmetros/variáveis não utilizadas. Os autores descrevem que devido à complexa integração com ambientes existentes que os desenvolvedores trabalham, a ferramenta não teve uma boa aderência e não foi aceita pelos desenvolvedores pelo fato de propor uma mudança no processo atual.

Nyamawe *et al.* (2018) proporam um estudo que também foi selecionado na QP1. A ferramenta utiliza as técnicas de refatoração Extract Class e Move Method, e valida a refatoração aplicada utilizando as métricas já avaliadas anteriormente, ou seja, a ferramenta identifica as oportunidades de refatoração utilizando métricas do código-fonte e avalia se após a refatoração se tem um valor melhor para a métrica avaliada anteriormente.

Erdemir, Tekin e Buzluca (2011) relata a importância da utilização de métricas para medir a qualidade interna de um software, o estudo foi selecionado na QP1 por utilizar métricas no processo de apresentar a qualidade do código-fonte para o desenvolvedor. A ferramenta em forma de plugin tem seu foco em melhorar a qualidade interna do projeto e utiliza técnicas de refatoração como Extract Method, Extract Interface, Move Method e Move Attribute. Com o objetivo de aumentar o alcance da análise de código-fonte, o estudo propõe um plugin para a *IDE* Eclipse, em que a proposta da ferramenta é apresentar a qualidade do código-fonte para o desenvolvedor com base nas métricas coletadas.

Danphitsanuphan e Suwantada (2012), Orchard e Rice (2013), Szőke *et al.* (2015) e Ge *et al.* (2017) relatam que a refatoração tem que se tornar parte integrante do ciclo de vida do soft-

ware para garantir a qualidade contínua da estrutura interna. Os estudos apresentam a utilização de ferramentas para garantir a qualidade do código-fonte e evitar problemas de complexibilidade e de estrutura interna. Os autores descrevem também que no processo de desenvolvimento devemos adicionar a revisão de código-fonte para garantir a qualidade e além disso alinhar o conhecimento entre a equipe, evitando erros e implementações erradas. No estudo Orchard e Rice (2013) a ferramenta construída foca em dois tipos de refatoração: Move Attribute para evitar a utilização de variáveis globais e Extract Method evitando duplicação de código-fonte.

Pantiuchina (2019), Pantiuchina *et al.* (2018) relatam que a complexidade do software tende a aumentar com o tempo devido a manutenção contínua e a evolução. Para lidar com a complexidade, falhas no projeto e falta de padrões, o estudo propõe uma ferramenta chamada Code Smell Predicator, que é uma abordagem que apresenta problemas na estrutura interna do software antes que se torne um problema para a manutenção do software. A ferramenta tem como objetivo identificar estruturas que irão se tornar complexas, para sugerir ao desenvolvedor uma refatoração específica em determinado trecho, seja para reescrever ou aplicar a técnica Extract Method, assim evitando que o código complexo seja um problema para a evolução do software.

Liu, Xu e Zou (2018) apresentam a importância de aplicar as atividades de refatoração sem redundância. O estudo foi selecionado na QP1 por utilizar métricas no processo de identificar estruturas ruins no código-fonte. A ferramenta identifica a similaridade dos métodos de diferentes classes para sugerir que seja movido o método; ou seja, a ferramenta recomenda refatorações Move Method.

Magalhães *et al.* (2017) relatam a dificuldade em se prestar manutenção em um software complexo, e os transtornos que isso causa ao decorrer do projeto. É proposta uma ferramenta para detectar a complexidade desnecessária em um código-fonte. A ferramenta aborda a identificação da complexidade desnecessária, ao identificar a ferramenta gera dois Gráficos de Fluxo de Controle (GFC), um representando a estrutura original e outro otimizado sem complexidade desnecessária, a partir disso a ferramenta apresenta ao desenvolvedor para que o mesmo transforme aquele código o tornando mais claro.

Santos *et al.* (2015) relatam que durante o ciclo de vida de um software, a manutenção é uma tarefa que deve ser executada de forma contínua. Apesar de *IDE's* como *IntelliJ* e *Eclipse* oferecerem ferramentas de refatoração, há uma certa desconfiança dos desenvolvedores em utilizar as mesmas. Para isso o estudo apresenta uma ferramenta chamada MacroRecorder, que grava uma sequência de transformações e reproduz esses padrões de transformações em outros trechos de código. O gravador funciona como uma extensão do Pharo *IDE*, responsável por monitorar edições e alterações no código-fonte. A ferramenta utiliza técnicas de refatoração como Extract Method e Move Class.

Ainda na QP2, durante a pesquisa, foi identificado outro Mapeamento Sistemático com enfoque em ferramentas para Refatoração de Software. O estudo de (TAVARES; FERREIRA; FIGUEREDO, 2018) tem como objetivo identificar as ferramentas de refatoração disponíveis

na literatura, e também sintetizar com base nas características das mesmas. O estudo destaca também que a área de refatoração já é bem consolidada com grandes pesquisas, e que atualmente ainda é um assunto muito pesquisado.

A *string* de busca construída pelo autor do estudo foi: **Qual é o perfil das ferramentas de refatoração propostas recentemente?**, a busca realizada tinha como enfoque principal as características das ferramentas, em que se obteve um total de 2800 artigos, após isso foram aplicados 5 critérios de inclusão e exclusão, chegando a um total de 51 estudos que foram abordados no MS do autor.

Os resultados obtidos no MS de (TAVARES; FERREIRA; FIGUEREDO, 2018), vêm de encontro com alguns resultados obtidos neste MS, em que é destacado que a maioria das ferramentas disponíveis na literatura são focadas para a linguagem Java, mas existem também para outras linguagens como JavaScript, PHP e C. Outro ponto que os MS obtiveram resultados parecidos foi nos tipos de refatorações suportados pelas ferramentas disponíveis, que são: Move Method, Pull Up Method e Extract Class.

Da parte das ferramentas identificadas, os estudos se completam, na qual algumas sumarizadas aqui não foram localizadas lá e vice-versa. Como pode-se observar na Tabela 2 as ferramentas foram classificadas em quatro categorias: **Programa**: Nome da ferramenta de refatoração. **Linguagem**: Linguagem de programação que a ferramenta abrange nas refatorações. **Plugin**: se a ferramenta é apresentada como um *plugin* de alguma *IDE* conhecida. **Característica**: quais as funcionalidades das ferramentas, tais como sugestão de refatoração, realiza refatoração, detecta refatoração, detecta erro na refatoração, estima impacto da refatoração e prediz a necessidade de realizar refatoração. **Operação**: como a ferramenta realiza a operação: de forma automatizada, semi-automatizada, manual e/ou passo a passo. **Identificado na QP2**: se a ferramenta levantada no estudo foi identificada na QP2.

Como pode ser visto na Tabela 2, 37 ferramentas de 42 apoiam a linguagem Java com ferramentas para realizar refatorações. E em sua grande maioria não é utilizada em forma de plugins com sugestões de refatoração de forma automática.

## 4.4 Ameaças à Validade

- **Validade de Conclusão**: é relacionada ao tratamento dos resultados obtidos. Como a análise dos estudos foram feitas somente por um pesquisador, a interpretação do mesmo pode ser falha, e acabar realizando o processo de filtragem de forma imprecisa.

- **Validade Externa**: é relacionada com as chances da generalização dos estudos primários obtidos. Como foram consultadas apenas duas bases de dados com ambas as strings de busca, existe a possibilidade de existirem estudos pertinentes às QPs que não foram incluídos no MS, pois estão em outras bases de dados. A questão de pes-

**Tabela 2 – Ferramentas identificadas no estudo de Tavares, Ferreira e Figueredo (2018)**

| Perfil das ferramentas analisadas | | | | | |
|---|---|---|---|---|---|
| Programa | Ling. | Plugin | Característica | Operação | Ident. QP2 |
| LambdaFicator | Java | Sim | Sugestão de refatoração (SUG)/Refatoração (REFAC) | Manual | Sim |
| Cider | Java | Não | DETEC (DETEC) | Manual | Não |
| HAN | Java | Não | DETEC | Manual | Não |
| PILGRIM | Java | Não | REFAC | Manual | Não |
| Obey | Java | Sim | REFAC | Manual | Não |
| WebDyn | PHP | Sim | REFAC | Automática | Sim |
| KRISHNAN | Java | Sim | REFAC | Manual | Sim |
| JSRefactor | JS | Sim | Erro na refatoração (ERR) | Semi-Automática | Não |
| GLIGORIC | Java | Sim | ERR | Manual | Não |
| Xll | Java | Não | REFAC | Automática | Não |
| MONDAL | Java | Não | DETEC | Automática | Não |
| BAVOTA | Java | Não | REFAC | Automática | Não |
| Morex | Java | Não | SUG | Manual | Não |
| Ripe | Java | Não | Impacto da refatoração (IMPAC) | Manual | Não |
| JExtract | Java | Não | SUG | Manual | Sim |
| WANG | Java | Não | SUG | Automática | Não |
| TAO | Java | Não | REFAC | Automática | Não |
| Morpheus | C | Não | REFAC | Manual | Não |
| Reflective Refactoring | Java | Sim | REFAC | Semi-Automática | Sim |
| YANG | Java | Não | REFAC | Manual | Não |
| Ad-Room | Ecore | Sim | DETEC | Automática | Sim |
| B-Refactoring | Java | Não | REFAC | Automática | Sim |
| MORALES | Java | Não | REFAC | Manual | Sim |
| MAZINANIAN | Java | Sim | DETEC | Manual | Sim |
| OUNI | Java | Sim | SUG | Automática | Não |
| KEBIR | Java | Não | REFAC | Automática | Sim |
| MKAOUER | Java | Sim | REFAC | Automática | Sim |
| KHATCHA DOURIAN | Java | Sim | REFAC | Automática | Não |
| ZAFEIRIS | Java | Sim | REFAC | Automática | Não |
| C-JrefRec | Java | Não | SUG | Manual | Não |
| More | Java | Não | SUG | Automática | Não |
| MANSOOR | Java | Não | REFAC | Manual | Sim |
| DALLAL | Java | Não | Prediz a necessidade de refatorar determinado código-fonte (PRE) | Manual | Não |
| RefDiff | Java | Não | PRE | Manual | Não |
| ZAFEIRIS | Java | Não | DETEC | Manual | Não |
| FENSKE | Java | Não | REFAC | Semi-Automática | Sim |

**Fonte: Autoria própria.**

quisa e os critérios de inclusão e exclusão, sendo previamente definidos, evita que haja parcialidade em relação aos artigos encontrados.

## 4.5 Considerações Finais

Nesse MS buscamos encontrar os estudos que têm como objetivo auxiliar o dia-a-dia de um desenvolvedor propondo ferramentas para aplicar refatorações em um código-fonte de forma automática. Para isso, separamos em duas QPs, na qual uma busca os estudos que apresentam ferramentas para identificar oportunidades de refatoração com base em métricas coletadas de um código-fonte, e outra QP que propõe identificar ferramentas para aplicar refatorações de forma automática em um código-fonte.

De modo geral, pelos estudos selecionados pode-se concluir que é uma área com pouca variedade de ferramentas, em que muitas aboradam da mesma forma para as mesmas linguagens (Java ou C), e outras que não acompanharam muito a evolução e são defasadas para se utilizar no dia-a-dia de um desenvolvedor nos dias de hoje, visto que existe uma desconfiança de desenvolvedores em relação a ferramentas que modificam o código-fonte automaticamente (NEGARA *et al.*, 2013).

A partir das contribuições identificadas, pode-se aplicar no desenvolvimento da ferramenta algumas técnicas já identificadas pelos autores selecionados no MS, técnicas como: principais métricas a se utilizar na identificação de refatoração, técnicas de refatoração e a utilização de testes de unidade para validar se a refatoração não gerou impactos indesejados ao software.

Desta forma, a utilização das métricas de código-fonte durante a refatoração de um código-fonte, pode surgir como uma mitigação da desconfiança dos desenvolvedores, aumentando assim a confiabilidade dos mesmos e surgindo como um diferencial em relação as ferramentas disponíveis na literatura atualmente.

# 5 O *PLUGIN REFACTOREXTENSION*

Conforme apresentado no Capítulo 4, são escassas as ferramentas que automatizam o processo de refatoração de um código-fonte, e as ferramentas que têm esse objetivo acabam com uma baixa aderência por parte dos desenvolvedores devido à baixa confiabilidade dos mesmos em relação às ferramentas.

Performance, baixa complexidade, qualidade do código-fonte são fatores de suma importância para a produção de um software. Os *code smells* podem prejudicar fortemente em tais aspectos, principalmente na qualidade e na complexidade interna do código-fonte. A fim de identificar e remover os *code smells* de um código-fonte, foi criada uma ferramenta capaz de identificar e sugerir técnicas de refatoração de forma automática em um código-fonte.

Para o desenvolvimento da solução para este trabalho, optou-se pelo desenvolvimento de uma ferramenta no formato de *plugin* para utilizar no ambiente de desenvolvimento (ou *Integrated Development Environment, IDE*) do desenvolvedor. Com o *plugin*, é possível aplicar algumas técnicas de refatoração e identificação de *code smells* para auxiliar no dia-a-dia do desenvolvedor, tornando o código implementado mais claro, objetivo e menos complexo.

No presente capítulo será descrito o processo de desenvolvimento do *plugin* chamado *RefactorExtension*. Na Seção 5.1 são descritas as técnicas utilizadas para análise do código-fonte, na Seção 5.2 é descrito o processo de identificação dos *code smells*, na Seção 5.3 é descrito o processo de aplicação da refatoração pelo *plugin*, e por fim, na Seção 5.4 é apresentado o *plugin* e seu funcionamento.

## 5.1 Análise do código fonte e seleção das métricas

Nesta seção será descrito o processo de seleção das técnicas para análise do código-fonte, em que na Subseção 5.1.1 é apresentado o processo de conversão do código-fonte para uma AST e na Subseção 5.1.2 é descrito o processo de análise da qualidade do código-fonte por meio de métricas.

### 5.1.1 Árvore de Sintaxe Abstrata

Uma AST é uma representação do código-fonte construída em forma de nós, em que cada nó representa uma parte da estrutura do código-fonte. Este processo é basicamente a conversão de uma informação bruta (o código-fonte) para uma estrutura tipada, facilitando análises e manipulações do código-fonte.

A Figura 14 apresenta uma representação em AST de uma função simples, apresentada na parte superior da Figura. Já na parte inferior, é possível verificar a estrutura do programa tipada no padrão da AST, a qual apresenta os identificadores de cada nó do programa, identifi-

cadores como tipo da estrutura, onde inicia e termina o nó e o trecho de código implementado em tal nó.

**Figura 14 – Representação de nós AST no formato JSON**

```
function test() {
  return "this is a test";
}

{
  "type": "Program",
  "start": 0,
  "end": 46,
  "body": [
    {
      "type": "FunctionDeclaration",
      "start": 0,
      "end": 46,
      "id": {
        "type": "Identifier",
        "start": 9,
        "end": 13,
        "name": "test"
      },
      "expression": false,
      "generator": false,
      "async": false,
      "params": [],
      "body": {
        "type": "BlockStatement",
        "start": 16,
        "end": 46,
        "body": [
          {
            "type": "ReturnStatement",
            "start": 20,
            "end": 44,
            "argument": {
              "type": "Literal",
              "start": 27,
              "end": 43,
              "value": "this is a test",
              "raw": "\"this is a test\""
            }
          }
        ]
      }
    }
  ],
  "sourceType": "module"
}
```

**Fonte: Autoria própria.**

Para identificar os possíveis *code smells* no código-fonte como, por exemplo, estruturas condicionais *if-else*, o *plugin* obtém e utiliza a AST a partir do código-fonte do sistema sob análise. Ferramentas similares realizam análise estática por meio de conversão do código para AST, a exemplo das ferramentas *JSRefactor* e *JExtract* identificadas no Mapeamento Sistemático. A identificação dos *code smells* no *RefactorExtension* é abordada na Seção 5.2.

### 5.1.2 Extração de métricas do código-fonte

Para avaliar a eficiência das refatorações sugeridas pela ferramenta, foram elencadas algumas métricas que foram coletadas antes e depois de aplicar a refatoração. Assim, é possível se ter um panorama da melhoria real na qualidade do código-fonte proposta pela ferramenta. Para isso, foram definidas algumas boas métricas para a análise do resultado da refatoração, o que facilitará e servirá de insumo para um acompanhamento mais preciso da qualidade do código-fonte sob análise.

São consideradas boas métricas aquelas que facilitam a medição dos parâmetros de qualidade definidos para um determinado software (MUNRO, 2005). Para facilitar a análise, foi limitada a quantidade de métricas, até porque o tratamento de um grande volume de dados pode ser humanamente impossível (MEIRELLES, 2013). Abaixo são listadas as métricas utilizadas e a forma que são calculadas.

#### 5.1.2.1 Complexidade Ciclomática

A complexidade ciclomática é uma métrica de código-fonte, desenvolvida por *Thomas J. McCabe* em 1976, que utiliza a contagem da quantidade de caminhos independentes que o mesmo pode executar até o seu fim. Um caminho independente é aquele que contém pelo menos uma nova possibilidade de desvio de execução, ou um novo fluxo para execução (MCCABE, 1976). É possível verificar no Código 1 e na Figura 19 o mapeamento do grafo de fluxo de controle do mesmo.

**Listagem 1 – Exemplo de código-fonte para gerar o Grafo de Fluxo de Controle da Figura 19**

```
1  function main() {
2      let number = 1;
3      while(number < 10) {
4        number++;
5      }
6      if (number === 10) {
7        number = 0;
8      }
9      return number;
10 }
```

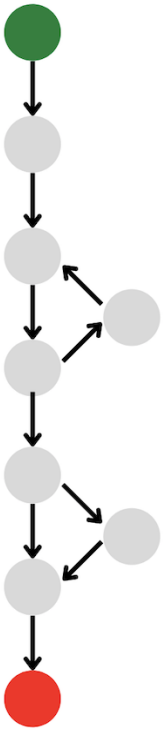
**Fonte: Autoria própria.**

Matematicamente pode-se exemplificar este cálculo da seguinte forma:

$$C = S - N + 2 \times P \tag{1}$$

| |
|---|
| $S$**:** Quantidade de arestas ou caminhos para execução; |
| $N$**:** Quantidade de nós do código-fonte; |
| $P$**:** Quantidade de componentes conectados; |
| $C$**:** Complexidade Ciclomática de um código-fonte; |

**Figura 15 – Grafo de Fluxo de Controle do Código-Fonte 1, no qual os desvios no fluxo ocorrem devido às estruturas de controle e condicional implementadas**



**Fonte: Autoria própria.**

Quanto mais estruturas de decisão o código-fonte possui, maior a complexidade de um software, o que implicará nos custos de manutenção e cobertura de testes do mesmo. Portanto, uma redução na métrica de complexidade ciclomática pode indicar uma melhoria no código-fonte.

#### 5.1.2.2 Dificuldade Halstead

As métricas de complexidade Halstead foram introduzidas por *Maurice Howard Halstead* em 1977, e têm como objetivo identificar propriedades mensuráveis de um código-fonte e a relação entre elas. Assim, suas métricas não estão relacionadas somente à complexidade do código-fonte, mas sim de informações gerais do mesmo. Logo, esta métrica de dificuldade está relacionada à dificuldade de escrita e compreensão de um código-fonte; pode-se relacionar esta métrica com a atividade de revisão de um código-fonte.

Matematicamente pode-se exemplificar este cálculo da seguinte forma:

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \tag{2}$$

$\eta_1$**:** Quantidade de operadores distintos;

$\eta_2$**:** Quantidade de operandos distintos;

$N_2$**:** Quantidade de total de operandos;

$D$**:** Dificuldade Halstead de um código-fonte;

No Código 2 é possível verificar um exemplo do que seriam os operandos e operadores em um código-fonte.

**Listagem 2 – Exemplo de código para exemplificar a métrica de Dificuldade Halstead**

```
1 function main() {
2     int a, b, c, avg;
3
4     scanf("%d %d %d", &a, &b, &c);
5     avg = (a+b+c)/3;
6
7     printf("avg = %d", avg);
8 }
```

**Fonte: Autoria própria.**

$\eta_1$**:** main, (), {}, int, scanf, =, +, &, /, printf, “ , ”, “ ; ” ;

$\eta_2$**:** a, b, c, 3, avg, “%d %d %d” , “avg = %d”;

$N_2$**:** a, b, c, avg, “%d %d %d”, &a, &b, &c, avg, 3, a, b, c, “avg = %d”, avg;

**Totalizadores Código 2:**

$\eta_1$**:** 12;

$\eta_2$**:** 7;

$N_2$**:** 15;

$D$**:** 12.85;

Dessa forma, quanto mais operandos e operadores o código-fonte possuir, a exemplo de variáveis, parâmetros e operações lógicas, maior a dificuldade para compreender o que o código é responsável por fazer. Consequentemente, uma redução no valor da métrica de dificuldade pode indicar uma melhoria no código-fonte.

#### 5.1.2.3 Tempo de Halstead

Assim como a métrica de dificuldade, a métrica de tempo também foi proposta por *Maurice Howard Halstead* em 1977. Esta métrica também está diretamente relacionada à dificuldade de leitura e interpretação do código-fonte, se assemelhando também a uma atividade de revisão de código.

Matematicamente pode-se exemplificar este cálculo da seguinte forma:

$$
\begin{aligned}
N &= N_1 + N_2 \\
\eta &= \eta_1 + \eta_2 \\
V &= N \times \log_2 \eta \\
E &= D \times V \\
T &= \tfrac{E}{18} segundos
\end{aligned}
\tag{3}
$$

| |
|---|
| $N_1$**:** Quantidade de total de operadores; |
| $N_2$**:** Quantidade de total de operandos; |
| $N$**:** Tamanho do código-fonte; |
| $\eta_1$**:** Quantidade de operadores distintos; |
| $\eta_2$**:** Quantidade de operandos distintos; |
| $\eta$**:** Total do vocabulário do programa; |
| $V$**:** Volume do código-fonte; |
| $D$**:** Dificuldade Halstead de um código-fonte; |
| $E$**:** Esforço para compreensão de um código-fonte; |
| $T$**:** Tempo Halstead em segundos; |

Abaixo, é possível verificar o resultado da métrica de Tempo Halstead, aplicando-a no Código 2. É estimado que um desenvolvedor leve aproximadamente dois minutos para compreender o código-fonte.

$N_1$: 27;

$N_2$: 15;

$N$: 42;

$\eta_1$: 12;

$\eta_2$: 7;

$\eta$: 19;

$V$: 178.4;

$D$: 12.85;

$E$: 2292.44;

$T$: 127.357.

Dessa forma, quanto mais operandos e operadores o código-fonte possuir, a exemplo de variáveis, declarações, atribuições e operadores matemáticos, maior o tempo para compreender o que o código é responsável por fazer. Consequentemente, uma redução no valor da métrica de tempo pode indicar uma melhoria no código-fonte.

#### 5.1.2.4 Linhas de Código-Fonte para Processamento Lógico

O número de linhas de código para processamento lógico é uma métrica de tamanho, assim como a métrica de número de LOC. A métrica LOC realiza uma contagem de todas as linhas de código-fonte excluindo os comentários, enquanto o número de LLOC realiza a contagem de todas as instruções que são executáveis no código-fonte.

Para exemplificar essas diferentes métricas, analisando o Código 3, temos como resultado: LOC é igual a 2 que é o resultado da soma da linha número 2 com a linha número 4. Enquanto LLOC é igual a 4 que é o resultado da soma da linha número 2 com as 3 execuções na linha número 4, ou seja, o compilador interpreta como uma linha para a validação, outra linha para a execução do *If* e outra para a execução do *Else*.

**Listagem 3 – Exemplo de código para exemplificar a métrica de LLOC**

```
1 function main() {
2     let number = 1;
3
4     return number === 1 ? "Valido" : "Invalido";
5 }
```

**Fonte: Autoria própria.**

Dessa forma, quanto maior o número de LOC de uma função ou método, mais complexo e mais difícil de manter ele será. Consequentemente, uma redução no tamanho das funções pode indicar uma melhoria no código-fonte.

## 5.2 Identificação de *code smells*

Como resultado do Mapeamento Sistemático, descrito no Capítulo 4, foram identificadas algumas técnicas de refatoração e exemplos de *code smells* que tem pouca abrangência nas ferramentas disponíveis na literatura, então foi identificada essa carência e abordada na ferramenta desenvolvida.

Nesta seção, serão elencadas as técnicas de identificação de *code smells* que foram implementadas na ferramenta para melhorar a qualidade do código-fonte.

### 5.2.1 Uso desnecessário de estruturas *If-else*

Durante a codificação, é comum que um desenvolvedor faça verificações no código-fonte utilizando a estrutura de controle *if-else*. Porém, quando essas estruturas são utilizadas repetidamente com expressões curtas e objetivas, o código-fonte acaba ficando mais extenso e menos objetivo, aumentando assim a complexidade do mesmo.

Abaixo é possível verificar no Código 4, um exemplo de tal *code smell*[1], no qual existe uma simples validação utilizando uma estrutura de controle que acaba deixando o código-fonte mais extenso, dificultando a compreensão do mesmo.

**Listagem 4 – Exemplo de Operador Condicional em JavaScript**

```
1 function validate(n) {
2     if (n === 1) {
3         return "Valido";
4     } else {
5         return "Invalido";
6     }
7 }
```

**Fonte: Autoria própria.**

Uma forma de resolver esse problema seria a utilização de operadores ternários (FOWLER, 2018). Essa solução será apresentada na Seção 5.3.1.

[1] Switch Statements - https://refactoring.guru/smells/switch-statements

### 5.2.2 Código Morto

Durante o ciclo de vida de um software, é comum que alguns trechos de código-fonte acabem ficando obsoletos e sua execução não gera impactos nos resultados de determinada função ou classe. Abaixo é possível verificar no Código 5, um exemplo de tal *code smell*[2], no qual existe um operador condicional em que sua expressão sempre vai ser falsa; ou seja, um código-fonte que nunca será executado.

**Listagem 5 – Exemplo de código-fonte com um trecho de código morto**

```
1  function validate() {
2      console.log("Eu sou um codigo vivo");
3
4      if (false) { //Inicio codigo morto
5          let number = 1;
6
7          if (number === 1) {
8              number = 2;
9          }
10
11         console.log("Eu sou um codigo morto");
12     }
13 }
```

**Fonte: Autoria própria.**

Uma forma de resolver esse problema seria a aplicação da técnica de remoção de código-fonte morto (FOWLER, 2018). Essa solução será apresentada na Seção 5.3.2.

### 5.2.3 Classes Alternativas com Diferentes Interfaces

Tal *code smell*[3] ocorre quando diferentes classes usam o mesmo subconjunto da interface de uma classe ou duas classes têm parte de suas interfaces em comum (FOWLER, 2018), isso acontece quando o desenvolvedor cria uma classe com um objetivo, sem saber da existência da outra, ficando assim duas classes com o mesmo objetivo e não padronizadas.

Abaixo é possível verificar no Código 6, um exemplo de tal *code smell*, em que são apresentadas duas diferentes classes, com o mesmo objetivo mas implementadas de forma diferente, que acaba aumentando assim a complexidade interna do projeto.

Uma forma de resolver esse problema seria a aplicação da técnica de refatoração *Extract Interface* (FOWLER, 2018). Essa solução será apresentada na Seção 5.3.3.

[2] Dead Code - https://refactoring.guru/smells/dead-code

[3] Alternative Classes with Different Interfaces - https://refactoring.guru/smells/alternative-classes-with- different-interfaces

**Listagem 6 – Exemplo de código-fonte sem uma interface implementada**

```
1  public class validate {
2      Integer number;
3
4      public Boolean isEquals(Integer value) {
5          return number == value;
6      }
7
8      public Boolean call() {
9          return true;
10     }
11 }
12
13 public class verificate() {
14     Integer value;
15
16     public Boolean isEquals(Integer value) {
17         if (value > 0) {
18             return 0;
19         }
20
21         return this.value == value;
22     }
23
24     public Boolean test() {
25         return true;
26     }
27 }
```

**Fonte: Autoria própria.**

### 5.2.4 Middle Man (Intermediário)

Tal tipo de *code smell*[4] ocorre quando um método delega a execução de um simples trecho de código-fonte para outro método, ficando somente como um intermediário na execução.

Abaixo é possível verificar no Código 7, um exemplo de tal *code smell*, em que um método com uma simples execução, delega sua execução para outro método, que acaba aumentando assim a complexidade interna do projeto e a quantidade de métodos desnecessários no projeto.

Uma forma de resolver esse problema seria a aplicação da técnica de refatoração *Inline Method* (FOWLER, 2018). Essa solução será apresentada na Seção 5.3.4.

[4] Middle Man - https://refactoring.guru/smells/middle-man

**Listagem 7 – Exemplo de código-fonte com uma função intermediária**

```
1 function main() {
2     let number = 1;
3
4     return validate(number);
5 }
6
7 function validate(number) {
8     return number > 0;
9 }
```

**Fonte: Autoria própria.**

## 5.3 Aplicação das técnicas de refatoração

Nesta seção, será explicado a forma que a ferramenta desenvolvida identifica as oportunidades de refatoração elencadas na seção 5.2, a partir de um código-fonte utilizando as características do mesmo.

### 5.3.1 Converter expressão para operador condicional ternário

Para remover o *code smell* 5.2.1 do código-fonte, a ferramenta realiza a conversão do mesmo para uma AST. Após realizar a conversão, são identificados os nós do tipo *IfStatement*, que possuem instruções objetivas, essas instruções objetivas são identificadas pela métrica de LOC, em que se existirem até 4 linhas de código-fonte é realizada a conversão dessa estrutura de validação ao operador ternário. No Código 4 é possível analisar uma estrutura de validação do tipo *if-else*, e no Código 8 o mesmo código-fonte, mas já convertido ao operador condicional ternário pelo *plugin*.

**Listagem 8 – Exemplo de Operador Condicional Ternário em JavaScript**

```
1 function validate(n) {
2     return n === 1 ? "Valido" : "Invalido";
3 }
```

**Fonte: Autoria própria.**

Para tal técnica de refatoração, a ferramenta implementada aborda somente as estruturas do tipo *if-else*. Outras variações dessa estrutura como *if-elseif-else* ou *if-elseif-elseif-else*, geram uma maior complexidade para o design do código-fonte, dificultando assim o seu entendimento.

Após a ferramenta rodar a refatoração, é exibido no método refatorado as métricas do código-fonte refatorado. Para facilitar a leitura das métricas essas informações foram simplificadas na Tabela 3, onde são exibidas as métricas do Código 4 e do Código 8.

Analisando as métricas da Tabela 3, pode-se verificar a diminuição no nível de dificuldade ao interpretar o método, a diminuição de linhas no processamento lógico e uma diminuição

**Tabela 3 – Métricas extraídas do código-fonte antes e depois de refatorar utilizando a técnica de refatoração para converter ao operador ternário**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 4 | Código 8 |
| Complexidade Ciclomática | 2 | 2 |
| Dificuldade Halstead | 4.125 | 3.438 |
| LLOC | 6 | 3 |
| Halstead Time | 18.323 | 13.427 |

**Fonte: Autoria própria.**

do tempo para o entendimento da instrução, ou seja, o método ficou mais claro e objetivo e com uma diferença considerável no tempo de entendimento.

### 5.3.2 Remover código-fonte morto

O processo de remoção de código-fonte morto (*Dead Code Removal* (DCR), *Dead Code Elimination* (DCE)) (BECK; FOWLER; BECK, 1999), tem como objetivo remover todo código que não altera os resultados do código-fonte implementado. A aplicação dessa técnica de refatoração traz diversos benefícios a aplicação que está sendo desenvolvida, benefícios como: Redução do tamanho da aplicação desenvolvida e evitar a execução/compilação de uma operação desnecessária. Consequentemente ocasiona na melhora do tempo de execução do programa e na otimização do código-fonte implementado, o tornando mais simples e objetivo (FOWLER, 2018).

Para remover o *code smell* 5.2.2 do código-fonte, a ferramenta realiza a conversão do mesmo para uma AST. Exemplificando com o Código 5, pode-se verificar na Figura 16 a AST do mesmo, onde o existe uma expressão do tipo *IfStatement* com um teste literal em que o mesmo é sempre falso, ou seja, algo que nunca será executado. A partir disso o *plugin* identifica o final da instrução com base na métrica de LOC e remove o bloco de código-fonte morto.

Após a ferramenta realizar a remoção do *code smell* do código-fonte 5, pode-se verificar como ficou o mesmo código-fonte no Código 9 com o *code smell* removido.

**Listagem 9 – Exemplo de código-fonte com o código morto removido**

```
1 function validate() {
2     console.log("Eu sou um codigo vivo");
3 }
```

**Fonte: Autoria própria.**

É possível verificar as métricas do código-fonte antes e depois da refatoração na Tabela 4, onde são exibidas as métricas do Código 5 e do Código 9.

Analisando as métricas da Tabela 4, é possível verificar a diminuição no nível de complexidade ciclomática, diminuição no grau de dificuldade do método, a diminuição de LLOC

**Figura 16 – Representação de nós AST do Código 5 no formato JSON.**


**Fonte: Autoria própria.**

**Tabela 4 – Métricas extraídas do código-fonte antes e depois de refatorar utilizando a técnica de remoção de código morto**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 5 | Código 9 |
| Complexidade Ciclomática | 3 | 1 |
| Dificuldade Halstead | 4.875 | 1 |
| LLOC | 6 | 1 |
| Halstead Time | 23.717 | 0.645 |

**Fonte: Autoria própria.**

e uma diminuição do tempo para interpretação da instrução, ou seja, o método ficou objetivo e menos complexo, facilitando assim o seu entendimento.

### 5.3.3 Extract Interface

Para remover o *code smell* 5.2.3 do código-fonte, a ferramenta identifica a quantidade de classes existentes no código-fonte por meio das métricas coletadas do mesmo, e então cria uma interface para cada classe adicionando a assinatura dos métodos existentes na mesma, e então anota a classe com a implementação da interface.

É possível verificar no Código 6 uma classe com a implementação de dois métodos, após o *plugin* rodar a refatoração, pode-se verificar no Código 10 como ficou a implementação do mesmo código-fonte após a implementação da interface.

Após aplicar essa refatoração no Código 6, pode-se observar na Tabela 5 que não é identificada uma melhora significativa nas métricas do código-fonte. Um provável motivo seria porque este tipo de refatoração busca trazer uma padronização da atribuição da função executada pelas classes implementadas no projeto, facilitando assim o entendimento e a interpretação do código-fonte.

**Listagem 10 – Exemplo de código-fonte com uma interface implementada**

```
1  public interface Extracted {
2      public Boolean isEquals(position);
3      public Boolean call();
4  }
5
6  public class validate implements Extracted {
7      Integer number;
8
9      @Override
10     public Boolean isEquals(Integer value) {
11         return number == value;
12     }
13
14     @Override
15     public Boolean call() {
16         return true;
17     }
18 }
19
20 public class verificate implements Extracted {
21     Integer value;
22
23     public Boolean isEquals(Integer value) {
24         if (value > 0) {
25             return 0;
26         }
27
28         return this.value == value;
29     }
30
31     public Boolean call() {
32         return true;
33     }
34 }
```

**Fonte: Autoria própria.**

### 5.3.4 Inline Method

Esse tipo de refatoração traz alguns benefícios para o código-fonte como a redução da quantidade de métodos implementados, consequentemente diminuindo o número de linhas de código-fonte e o tempo de entendimento do mesmo, tornando assim o código-fonte mais limpo e objetivo (FOWLER, 2018).

Para aplicar a refatoração no código-fonte e identificar as funções que podem ter sua execução simplificada, a ferramenta captura as métricas do código-fonte, e a partir disso, busca identificar os métodos simples que delegam sua execução para outro método utilizando a métrica de LOC em conjunto com a AST. Para definir um método simples, por exemplo, pode-se

**Tabela 5 – Métricas extraídas do código-fonte antes e depois de refatorar utilizando a técnica de refatoração Extract Interface**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 6 | Código 10 |
| Complexidade Ciclomática | 0.667 | 0.667 |
| Dificuldade Halstead | 0.889 | 0.857 |
| LLOC | 13 | 13 |
| Halstead Time | 25.717 | 25.717 |

**Fonte: Autoria própria.**

levar em consideração a quantidade de *return* que o método pode ter, ou seja, quanto mais possibilidades de retorno, mais regras foram aplicadas em uma determinada função.

É possível verificar no Código 7 um código-fonte antes de aplicar a técnica de refatoração *Inline Function*, após o *plugin* rodar a refatoração, pode-se verificar no Código 11 como ficou a implementação do mesmo código-fonte após a remoção do *code smell*.

**Listagem 11 – Exemplo de código-fonte depois de aplicar a técnica de refatoração *Inline Method***

```
1 function main() {
2     let number = 1;
3
4     return number > 0;
5 }
```

**Fonte: Autoria própria.**

Após aplicar essa refatoração no Código 7, pode-se observar na Tabela 6 que não é identificada uma melhora significativa nas métricas do código-fonte. Um provável motivo seria porque este tipo de refatoração busca trazer uma padronização e organização no código-fonte, sem alterar a lógica de execução, apenas reorganizando a disponibilidade e execução dos métodos no código-fonte, facilitando assim o entendimento e a interpretação do código-fonte.

**Tabela 6 – Métricas extraídas do código-fonte antes e depois de refatorar utilizando a técnica de refatoração Inline Function**

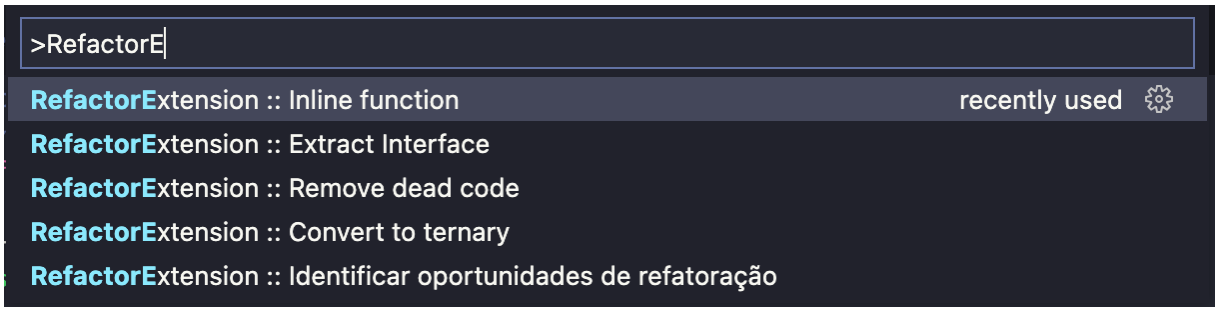
| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 6 | Código 10 |
| Complexidade Ciclomática | 1 | 1 |
| Dificuldade Halstead | 1 | 1 |
| LLOC | 3 | 2 |
| Halstead Time | 0.645 | 0.645 |

**Fonte: Autoria própria.**

## 5.4 *RefactorExtension*

Após realizar o levantamento dos tipos de refatoração e *code smells* que seriam abordados na solução deste estudo, foi implementado o *plugin* para a *IDE Visual Studio Code*[5] ou como é popularmente conhecida por *VSCode*. A utilização da ferramenta, uma vez instalada, ocorre por meio dos atalhos de comandos úteis do *VSCode*, nos quais os comandos do *plugin* podem ser facilmente identificados pelo prefixo *RefactorExtension*, como pode ser visto na Figura 17.

**Figura 17 – Lista de comandos úteis do *plugin RefactorExtension***




**Fonte: Autoria própria.**

Ao acionar o comando de identificar as oportunidades de refatoração no código-fonte, o *plugin* vai analisar o código, e apresentar ao desenvolvedor as refatorações que podem ser aplicadas no código-fonte, ficando a critério do mesmo se deseja ou não aplicá-las. É importante destacar que para o funcionamento correto do *plugin*, não pode haver erros de sintaxe no código-fonte.

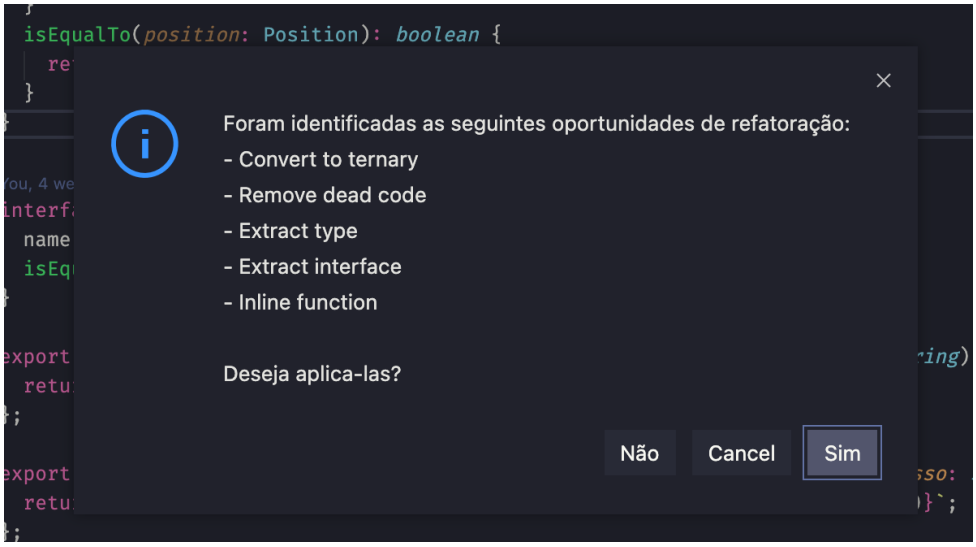
A funcionalidade de validar o arquivo inteiro de código-fonte, aplica as três etapas do estudo em uma única operação, na qual são elencadas as métricas do código-fonte, identificadas as oportunidades de refatoração e sugeridas as refatorações necessárias ao desenvolvedor por meio de um alerta na *IDE*. É possível verificar este funcionamento na Figura 18.

Para exemplificar o funcionamento do *plugin RefactorExtension*, o processo está representado na Figura 19. Os retângulos cinzas representam o processamento por parte da ferramenta, enquanto os azuis representam a ação do usuário. Após o desenvolvedor produzir o código-fonte, a ferramenta coleta as métricas do mesmo, e após isso é gerada a árvore AST do código-fonte e identificadas as oportunidades de refatoração. Após isso, é realizada uma nova coleta das métricas do código-fonte que tem o objetivo de coletar evidências de que a refatoração melhorou a qualidade do código-fonte, e então são apresentadas as sugestões de refatoração ao desenvolvedor, ficando ao critério deste se aceita as recomendações ou não.

Todos os trechos de código-fonte utilizados para exemplificar as refatorações neste presente capítulo, foram refatorados utilizando a ferramenta *RefactorExtension*, as métricas de re-

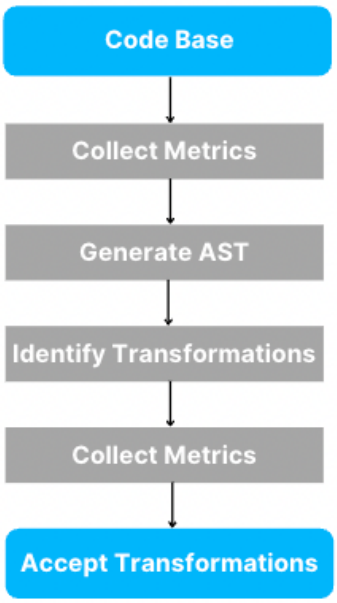
[5] https://code.visualstudio.com/

**Figura 18 – Lista de refatorações e code smells identificadas para refatoração pelo *plugin***




**Fonte: Autoria própria.**

**Figura 19 – Fluxo de execução do *plugin***




**Fonte: Autoria própria.**

sultados também foram coletadas pela mesma; assim, pode-se ter uma ideia do funcionamento da mesma.

O *plugin RefactorExtension* foi disponibilizado em código aberto no GitHub[6] e pode ser acessado a partir do link https://github.com/LuisGustavo802/refactoringextension.

[6] O GitHub é uma plataforma para hospedagem de códigos.https://github.com/

## 5.5 Ferramentas Utilizadas

Para a realização deste projeto foram utilizadas algumas bibliotecas *JavaScript* para auxiliar no processo de desenvolvimento. Descritas no Quadro 1, essas bibliotecas foram fortemente utilizadas para as etapas de identificação de oportunidades de refatoração e *code smells* e na análise e coleta das métricas do código-fonte (Seção 5.3).

**Quadro 1 – Bibliotecas *JavaScript* utilizadas**

| **Bibliotecas** | **Versão** | **Descrição** |
|---|---|---|
| **typhonjs-escomplex** [7] | 0.1.0 | Utilizada para realizar a coleta das métricas do código-fonte. |
| **abstract-syntax-tree** [8] | 2.19.1 | Utilizada para realizar a conversão do código-fonte para AST. |

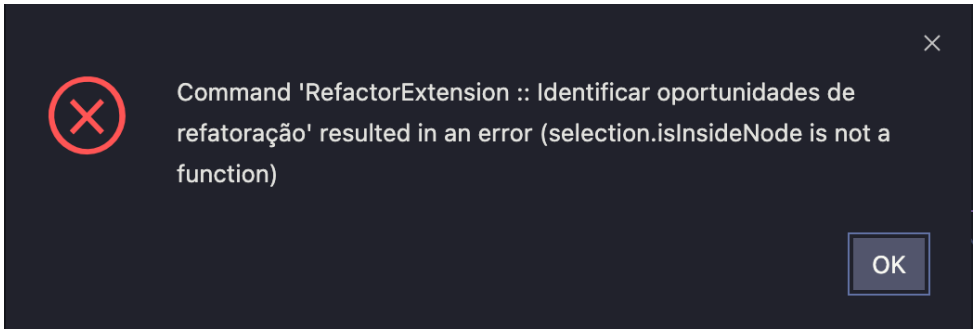
**Fonte: Autoria própria.**

## 5.6 Limitações da ferramenta

Durante a construção da ferramenta, foram observados alguns pontos dos quais poderiam se tornar um grande desafio para a eficácia e o funcionamento correto da ferramenta. Sendo o primeiro deles a limitação para a linguagem *JavaScript*, em que nem mesmo o seu superconjunto sintático, *TypeScript*, pôde ser considerado por haver incompatibilidade na conversão do código fonte para AST.

Após considerar para análise, exclusivamente, código escrito em *JavaScript*, uma nova limitação foi detectada: o uso de *frameworks* externos podem apresentar uma sintaxe não compatível para a conversão do código-fonte para AST. A Figura 20 exibe a forma que a ferramenta exibe o erro encontrado ao converter o Código 12 para AST. Isso ocorre pois o código-fonte está implementado utilizando o *framework React*.

**Figura 20 – Exemplo de erro para o erro de sintaxe encontrado**




**Fonte: Autoria própria.**

Como última limitação encontrada, alguns arquivos de código-fonte de testes unitários também apresentam erros ao realizar a conversão para AST. Desta forma, esses arquivos de

**Listagem 12 – Exemplo de código-fonte que ocorre erro ao gerar a AST**

```
export const FormControlFeedback = ({
  children,
  className,
  testId
}: FormControlVariationProps) => {
  const defaultClasses =
    'absolute top-0 right-0 z-2 block w-8 h-8 leading-8 ' +
    'text-center pointer-events-none text-green-700';

  const classes = [defaultClasses, className].join(' ');
  return (
    <span className={classes} data-testid={testId}>
      {children}
    </span>
  );
};
```

**Fonte: Autoria própria.**

código-fonte que se tornaram um problema para o funcionamento correto da ferramenta foram ignorados durante o processamento da mesma. Entende-se que estes problemas devem ser solucionados no futuro, para assim garantir uma maior abrangência da ferramenta.

# 6 ESTUDO DE CASO

O objetivo deste trabalho é auxiliar o desenvolvedor no seu dia-a-dia e melhorar a qualidade interna do software, por meio de um *plugin* que sugere a aplicação de técnicas de refatoração em um código-fonte. Para avaliar a solução proposta, foi realizado um estudo de caso (WOHLIN *et al.*, 2012) por seu caráter analítico; ou seja, a partir do código-fonte já produzido pelo desenvolvedor são identificadas as oportunidades de refatoração, e realizado um comparativo das métricas de qualidade do código-fonte antes e depois da refatoração. Neste estudo foram analisados os resultados do comparativo das métricas após aplicar a refatoração, a fim de validar a efetividade das refatorações propostas.

## 6.1 Configuração para experimento

Para execução da ferramenta desenvolvida, a configuração para a realização dos estudos é simples. A ferramenta foi versionada no Github[1] e para executar basta utilizar o compilador do próprio Visual Studio Code[2], e utilizar os comandos de atalhos identificados pelo prefixo *RefactorExtension*.

Para o presente trabalho, é possível utilizar a ferramenta somente possuindo o código-fonte e executando o mesmo localmente. Mas para trabalhos futuros, é possível publicar o *plugin* na loja disponível dentro da *IDE* ficando assim disponível para todos os desenvolvedores que utilizam a mesma usufruírem da ferramenta.

## 6.2 Projetos Javascript Selecionados

Para avaliar a efetividade das refatorações propostas em projetos reais, foram selecionados três projetos *open-source* na plataforma *GitHub* escritos na linguagem *JavaScript*. A seleção foi realizada com base na relevância e popularidade, a partir do número de *stars* na plataforma *GitHub*. Os projetos selecionados para esse estudo de caso são apresentados no Quadro 2.

**Quadro 2 – Projetos *JavaScript* selecionados para o estudo de caso**

| **Nome do Projeto** | **Autor** | **Stars** | **NAP JS**[3] | **NAA JS**[4] |
|---|---|---|---|---|
| **FreeCodeCamp** [5] | freeCodeCamp.org | 334.000 | 450 | 88 |
| **javascript-algorithms** [6] | Oleksii Trekhleb | 126.000 | 340 | 340 |
| **JavaScript** [7] | The Algorithms | 15.700 | 509 | 509 |

**Fonte: Autoria própria.**

[1] https://github.com/LuisGustavo802/refactoringextension
[2] https://code.visualstudio.com/

O projeto *open-source freeCodeCamp* foi criado pela comunidade freeCodeCamp.org, com o objetivo de ensinar desenvolvimento *Web* para qualquer pessoa. O projeto relata que já ajudou mais de 10.000 pessoas a seguir a carreira de desenvolvedor, e atualmente possui cerca de 4.335 contribuidores e mantedores do projeto. Com enfoque educacional, os projetos *open-source JavaScript-Algorithms* e *The Algorithms/Javascript* possuem um grande acervo de algoritmos e estrutura de dados disponíveis em seu repositório, projetos ainda mantidos atualmente, e possuem 143 e 194 respectivamente.

Por limitações encontradas durante a execução da ferramenta proposta, ao exemplo citado na Seção 5.6, foi necessário realizar uma pré-validação nos arquivos que seriam processados pela ferramenta. Caso a análise sintática do arquivo falhar, ele é ignorado e o processo de análise segue.

Ao final desta etapa, foram implementadas algumas restrições para o estudo de caso: *i)* O projeto *freeCodeCamp* possuía alguns arquivos cuja sintaxe era incompatível com o esperado, e assim foram desconsiderados; para a realização do estudo de caso para esse projeto foi considerado somente o módulo chamado *api-server* presente na raiz do projeto; *ii)* Os demais projetos (*JavaScript-Algorithms* e *The Algorithms/Javascript*) não apresentaram nenhuma incompatibilidade de sintaxe, portanto, todos os arquivos *JavaScript* foram considerados para a realização do estudo.

## 6.3 Planejamento

As variáveis elencadas para este experimento estão dispostas a seguir:

**C:** Complexidade ciclomática média do projeto;

**C(Novo):** Complexidade ciclomática do código-fonte após a refatoração;

**C(Antes):** Complexidade ciclomática do código-fonte antes da refatoração;

**D:** Dificuldade média para interpretação do código-fonte;

**D(Novo):** Dificuldade para interpretação do código-fonte após a refatoração;

**D(Antes):** Dificuldade para interpretação do código-fonte antes da refatoração;

**T:** Tempo médio despendido para a compreensão do código-fonte;

**T(Novo):** Tempo para a compreensão do código-fonte após a refatoração;

**T(Antes):** Tempo para a compreensão do código-fonte antes da refatoração;

**L:** Quantidade média de linhas de código-fonte para processamento lógico;

**L(Novo):** Quantidade de linhas de código-fonte para processamento lógico após a refatoração;

**L(Antes):** Quantidade de linhas de código-fonte para processamento lógico antes da refatoração;

As questões a serem respondidas são identificadas como $R_1$, $R_2$, $R_3$ e $R_4$, bem como suas respectivas métricas e hipóteses:

- $R_1$) As refatorações aplicadas reduziram a complexidade interna do código-fonte produzido?

  **Métrica**: Diminuição da métrica de Complexidade Ciclomática, representa a diminuição na complexidade interna do código-fonte.

  - **Hipótese nula** ($H_0$): A aplicação da refatoração do código-fonte não diminuiu a complexidade ciclomática interna.

$$C(Novo) \geq C(Antes) \tag{4}$$

  - **Hipótese alternativa** ($H_1$): A aplicação da refatoração do código-fonte diminuiu a complexidade ciclomática interna.

$$C(Novo) < C(Antes) \tag{5}$$

- $R_2$) As refatorações aplicadas reduziram a dificuldade para interpretação do código-fonte produzido?

  **Métrica**: Diminuição da métrica de Dificuldade Halstead, representa a diminuição da dificuldade para interpretação do código-fonte.

  - **Hipótese nula ($H_0$)**: A aplicação da refatoração do código-fonte não diminuiu a dificuldade para interpretação do código-fonte.

$$D(Novo) \geq D(Antes) \tag{6}$$

  - **Hipótese alternativa ($H_1$)**: A aplicação da refatoração do código-fonte diminuiu a dificuldade para interpretação do código-fonte.

$$D(Novo) < D(Antes) \tag{7}$$

- $R_3$) As refatorações aplicadas reduziram o tempo médio para compreensão do código-fonte?

  **Métrica**: Diminuição da métrica de Halstead Time, representa a melhora no tempo de compreensão do desenvolvedor.

  - **Hipótese nula ($H_0$)**: A aplicação da refatoração do código-fonte não melhorou o tempo de compreensão do código-fonte.

$$T(Novo) \geq T(Antes) \tag{8}$$

  - **Hipótese alternativa ($H_1$)**: A aplicação da refatoração do código-fonte diminuiu o tempo de compreensão do código-fonte.

$$T(Novo) < T(Antes) \tag{9}$$

- $R_4$) As refatorações aplicadas reduziram a quantidade média de linhas de código-fonte para processamento lógico?

  **Métrica**: Diminuição da métrica *LLOC*, representa a redução na quantidade de linhas de código-fonte para processamento lógico.

  - **Hipótese nula ($H_0$)**: A aplicação da refatoração do código-fonte diminuiu a quantidade de linhas de código-fonte para processamento lógico.

$$L(Novo) \geq L(Antes) \tag{10}$$

  - **Hipótese alternativa ($H_1$)**: A aplicação da refatoração do código-fonte diminuiu a quantidade de linhas de código-fonte para processamento lógico.

$$L(Novo) < L(Antes) \tag{11}$$

Uma vez que os repositórios dos projetos utilizados na validação dos resultados foram clonados, eles não voltaram a ser atualizados, para assim evitar a discrepância nos resultados, tendo em vista que o código-fonte dos projetos selecionados sofrem alterações constantemente.

Para melhor conclusão dos resultados, foi aplicado o teste estatístico **Teste T** para amostras pareadas ou **Teste T** paramétrico. Tal teste consiste em realizar mais de uma medida em uma mesma unidade amostral a fim de verificar se houve diferença entre essas medidas. Especificamente no caso das métricas, o teste ajuda a identificar se a amostra de métricas antes das refatorações é diferente da amostra depois das refatorações.

No contexto deste estudo, são comparadas as amostras referentes às variáveis elencadas acima, utilizando o comparativo de antes e depois de aplicar a refatoração. É sabido que métricas de código-fonte seguem uma distribuição exponencial (FERNANDES, 2018). Além disso, as duas amostras são dependentes, uma vez que as funções analisadas são as mesmas antes e depois da refatoração. Portanto, uma função com complexidade alta poderá ainda ter complexidade alta após a refatoração. Para variáveis que não seguem distribuição normal e com amostras dependentes, recomenda-se a aplicação do teste de **Wilcoxon**.

Se as amostras possuem distribuições, médias e variâncias diferentes, é possível concluir que as duas amostras são estatisticamente diferentes, e consequentemente pode-se comparar as variáveis elencadas para avaliar se houve uma melhora devido às refatorações aplicadas.

## 6.4 Validação

Os projetos *JavaScript* selecionados foram analisados automaticamente pela ferramenta com o objetivo de coletar as métricas antes e depois de refatorar o código-fonte. Além da visível melhoria no código-fonte, após executar a refatoração, são elencados os arquivos de código-fonte, e analisadas as métricas do antes e depois de aplicar a refatoração.

A partir deste comparativo das métricas, é possível chegar a um veredito se as refatorações aplicadas naquele arquivo de código-fonte foram ou não efetivas e trouxeram um resultado positivo para a qualidade interna do software.

Como mencionado anteriormente, a validação será feita aplicando o teste de **Wilcoxon** nas amostras coletadas. O primeiro passo é dado pela análise das amostras, a fim de validar se as mesmas são diferentes. Caso o teste retorne um *p-value* menor que 5% a hipótese nula ($H_0$) é descartada, caso for maior, isso significa que não há indícios para descartar a mesma. O segundo passo consiste em comparar as amostras por meio de suas médias, validando assim se houve ou não uma diminuição no valor da métrica após a refatoração.

# 7 RESULTADOS

O Capítulo apresenta os resultados para cada projeto selecionado, informando o número de refatorações, e o resultado da melhoria com base nas métricas de código-fonte obtidas após aplicar a refatoração. Também é apresentado o resultado total das métricas do software como um todo, a fim de validar a efetividade da utilização da ferramenta no contexto geral da aplicação.

Na Seção 7.1 são apresentadas as oportunidades de refatoração e *code smells* encontrados nos projetos, na Seção 7.2 são descritas as refatorações aplicadas nos projetos, na Seção 7.3 é descrita a análise da comparação das métricas antes e depois de identificar e aplicar as técnicas de refatoração nos projetos, na Seção 7.4 é discutido sobre os resultados obtidos na Seção anterior, e por fim, na Seção 7.5 são apresentadas as ameaças à validade do projeto identificadas durante a execução da ferramenta.

## 7.1 Identificação de *Code smells*

Para a realização do experimento, foi utilizado o projeto *freeCodeCamp* considerando apenas um subdomínio de sua extensão. Ao validar o subdomínio, foram encontrados 88 arquivos de código-fonte, somando um total de 759 funções codificadas em *Javascript*. Desses 88 arquivos, a ferramenta identificou *code smells* em 22 arquivos diferentes. Ou seja, aproximadamente 25% dos arquivos de código-fonte do projeto sofreram alguma refatoração. Das 759 funções existentes no projeto, a ferramenta identificou *code smells* em 56 funções diferentes. Ou seja, aproximadamente 8% das funções existentes sofreram alguma alteração.

Já para o projeto *Javascript-Algorithms*, foram considerados todos os arquivos de código-fonte de seus subdomínios. Ao realizar a validação, foram encontrados 340 arquivos de código-fonte, somando um total de 1451 funções codificadas em *JavaScript*. Desses 340 arquivos, a ferramenta identificou *code smells* em 72 arquivos diferentes. Ou seja, aproximadamente 22% dos arquivos de código-fonte do projeto sofreram alguma refatoração. Das 1451 funções existentes no projeto, a ferramenta identificou *code smells* em 74 funções diferentes. Ou seja, aproximadamente 5% das funções existentes sofreram alguma alteração.

Por fim, para o projeto *TheAlgorithms/Javascript*, também foram considerados todos os arquivos de código-fonte de seus subdomínios. Ao realizar a validação, foram encontrados 509 arquivos de código-fonte, somando um total de 2260 funções codificadas em *JavaScript*. Desses 509 arquivos, a ferramenta identificou *code smells* em 94 arquivos diferentes. Ou seja, aproximadamente 20% dos arquivos de código-fonte do projeto sofreram alguma refatoração. Das 2260 funções existentes no projeto, a ferramenta identificou *code smells* em 115 funções diferentes. Ou seja, aproximadamente 5% das funções existentes sofreram alguma alteração.

Todos os projetos apresentaram algum *code smell* no código-fonte, dentre os quatro selecionados. Portanto, isso reforça a importância e frequência que esses fenômenos tendem

a ocorrer mesmo em projetos populares e com manutenção frequente. Pode-se observar no Quadro 3 os *code smells* identificados por projeto.

**Quadro 3 – *Code smells* identificados por projeto**

| Nome do Projeto | UD If-Else [1] | CM [2] | CADI [3] | MM [4] |
|---|---|---|---|---|
| **freeCodeCamp** | 24 | 0 | 1 | 9 |
| **JavaScript-Algorithms** | 39 | 1 | 48 | 12 |
| **TheAlgorithms/Javascript** | 54 | 0 | 34 | 24 |

**Fonte: Autoria própria.**

## 7.2 Aplicação das técnicas de refatoração

Para o projeto *freeCodeCamp*, conforme apresentado no Quadro 4, foram aplicadas 35 refatorações nos arquivos de código-fonte. Das 35 refatorações, 24 foram conversão ao fator ternário, nenhum código-fonte morto foi identificado, para a única classe existente no projeto foi criada uma interface e 9 funções foram removidas.

Já para o projeto *Javascript-Algorithms*, foram aplicadas 100 refatorações nos arquivos de código-fonte. Das 100 refatorações, 39 foram conversão ao fator ternário, 1 código-fonte morto foi identificado e removido, para 48 classes foram criadas interfaces e 12 funções foram removidas.

Por fim, para o projeto *TheAlgorithms/Javascript*, foram aplicadas 112 refatorações nos arquivos de código-fonte. Das 112 refatorações, 54 foram conversão ao fator ternário, nenhum código-fonte morto foi identificado, para 34 classes foram criadas interfaces e 24 funções foram removidas.

**Quadro 4 – Refatorações aplicadas por projeto e por tipo de refatoração**

| Nome do Projeto | CFT [5] | RCM [6] | IEX [7] | IMT [8] |
|---|---|---|---|---|
| **freeCodeCamp** | 24 | 0 | 1 | 9 |
| **JavaScript-Algorithms** | 39 | 1 | 48 | 12 |
| **TheAlgorithms/Javascript** | 54 | 0 | 34 | 24 |

**Fonte: Autoria própria.**

Na próxima seção é realizada uma análise das métricas do código-fonte antes e depois de aplicar as refatorações mencionadas acima, a fim de validar se a aplicação das refatorações resultaram em uma diminuição no valor das mesmas.

## 7.3 Validação de Hipótese

Para realizar a mensuração da experimentação proposta neste estudo, foram calculadas as médias das variáveis elencadas antes e depois de se aplicar a refatoração, para assim possuir um panorama da efetividade da ferramenta.

Na Seção 7.3.1 são apresentados os resultados obtidos para a métrica de Complexidade Ciclomática, na Seção 7.3.2 são apresentados os resultados obtidos para a métrica de Dificuldade Halstead, na Seção 7.3.3 são apresentados os resultados obtidos para a métrica de Tempo Halstead, e por fim, na Seção 7.3.4 são apresentados os resultados obtidos para a métrica de *LLOC*.

### 7.3.1 Complexidade Ciclomática ($R_1$)

A fim de validar a variância entre as amostras da métrica de Complexidade Ciclomática média dos projetos, foi aplicado o teste estatístico nas amostras coletadas.

Para o projeto *freeCodeCamp*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 13. O teste resultou em um *p-value* menor que 5%; portanto há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias diferentes.

**Listagem 13 – Resultado teste estatístico da métrica de Complexidade Ciclomática para o projeto *freeCodeCamp***

```
1    Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 309932, p-value = 0.00133
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 21, a distribuição da Complexidade Ciclomática das 35 funções que sofreram alteração por meio de refatorações. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, a Complexidade Ciclomática média era de 1,61 e após a refatoração 1,51.

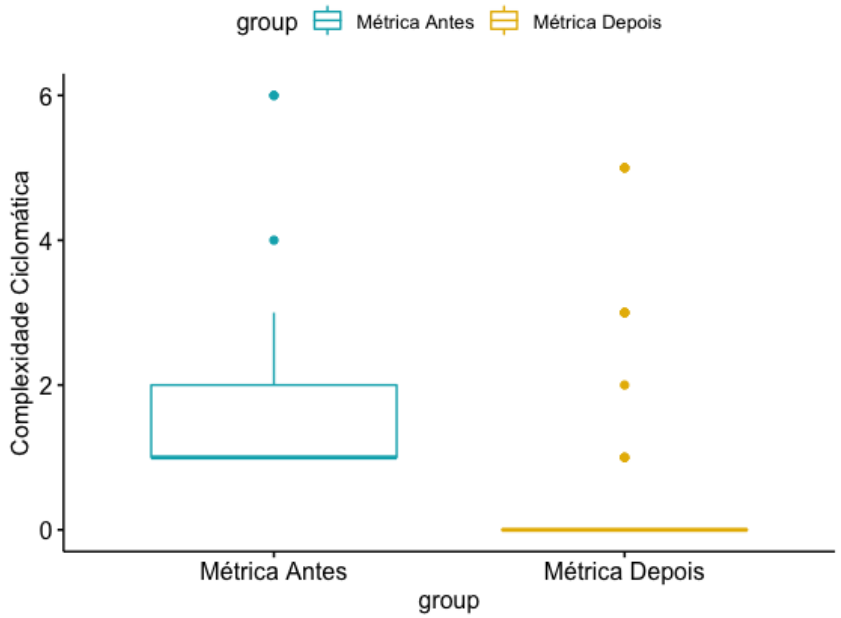
> **Sumário $R_1$ (*freeCodeCamp*):** A Complexidade Ciclomática média do projeto diminuiu de 1,61 para 1,51; representando uma diminuição de 6% na Complexidade Ciclomática interna do projeto. Portanto, foi rejeitada a hipótese nula ($H_0$).

Já para o projeto *JavaScript-Algorithms*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 14. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

**Listing 7.1 – Resultado teste estatístico da métrica de Complexidade Ciclomática para o projeto *JavaScript-Algorithms***

```
1    Wilcoxon rank sum test with continuity correction
```

**Figura 21 – Comparativo da métrica de Complexidade Ciclomática para o projeto *freeCodeCamp***




**Fonte: Autoria própria.**

```
2
3 data:  metricaAntes and metricaDepois
4 W = 1056434, p-value = 0.8202
5 alternative hypothesis: true location shift is not equal to 0
```

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *JavaScript-Algorithms*. Pode-se observar no Código 28 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de Complexidade Ciclomática. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma diferença significativa na Complexidade Ciclomática das mesmas.

**Listagem 14 – Resultado teste estatístico da métrica de Complexidade Ciclomática para o projeto *JavaScript-Algorithms***

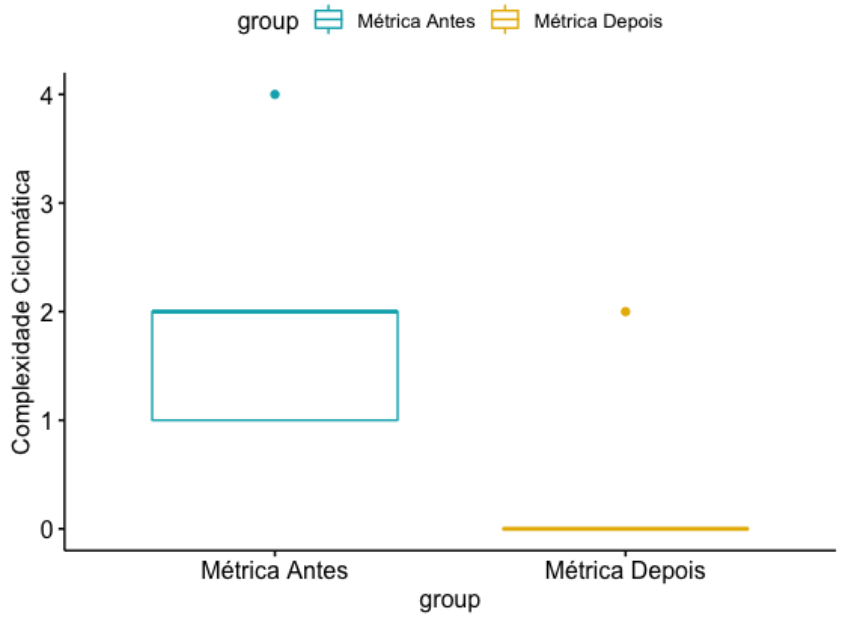
```
1    Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 1056434, p-value = 0.8202
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 22, a distribuição da Complexidade Ciclomática das 6 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram diferentes; especificamente, antes da refatoração, a Complexidade Ciclomática média era de 2 e após a refatoração 0,4.

**Sumário** $R_1$ **(*JavaScript-Algorithms*):** A Complexidade Ciclomática média do projeto diminuiu de 1,58 para 1,57; representando uma diminuição menor que 1% na Complexidade Ciclomática interna do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

**Figura 22 – Comparativo da métrica de Complexidade Ciclomática para o projeto *JavaScript-Algorithms***




**Fonte: Autoria própria.**

Por fim, para o projeto *TheAlgorithms/Javascript*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 15. O teste resultou em um *p-value* menor que 5%; portanto há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias diferentes.

**Listagem 15 – Resultado teste estatístico da métrica de Complexidade Ciclomática para o projeto *TheAlgorithms/Javascript***

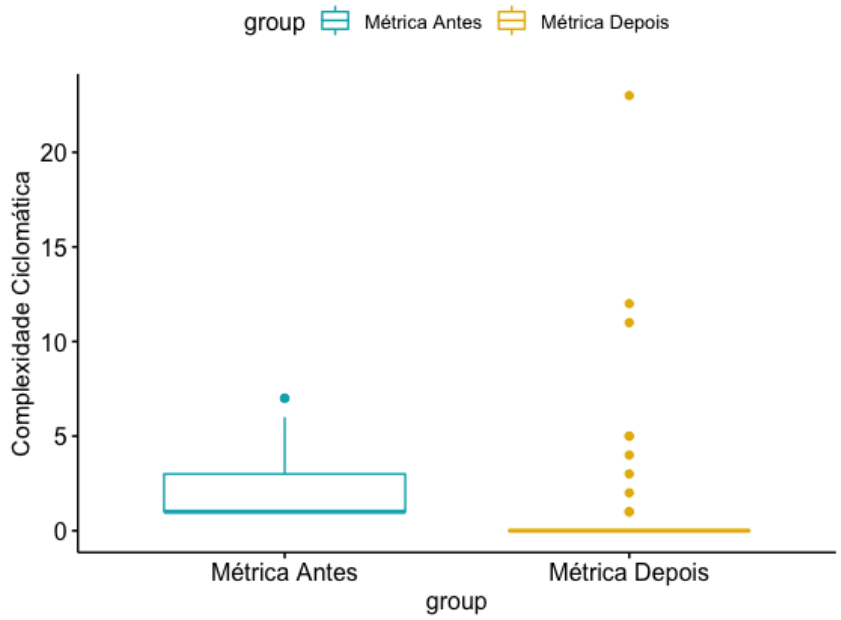
```
1     Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 2620962, p-value = 0.04305
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 23, a distribuição da Complexidade Ciclomática das 62 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, a Complexidade Ciclomática média era de 1,73 e após a refatoração 1,70.

| **Sumário** $R_1$ (***TheAlgorithms/Javascript***)**:** A Complexidade Ciclomática média do projeto diminuiu de 1,73 para 1,70; representando uma diminuição de 2% na Complexidade Ciclomática interna do projeto. Portanto, foi rejeitada a hipótese nula ($H_0$). |
|---|

**Figura 23 – Comparativo da métrica de Complexidade Ciclomática para o projeto *TheAlgorithms/-Javascript***




**Fonte: Autoria própria.**

### 7.3.2 Dificuldade Halstead ($R_2$)

A fim de validar a variância entre entre as amostras da métrica de Dificuldade Halstead média dos projetos, foi aplicado o teste estatístico nas amostras coletadas.

Para o projeto *freeCodeCamp*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 16. O teste resultou em um *p-value* menor que 5%; portanto há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias diferentes.

Pode-se observar na Figura 24, a distribuição da Dificuldade Halstead das 35 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, a Dificuldade Halstead média era de 3,82 e após a refatoração 3,59.

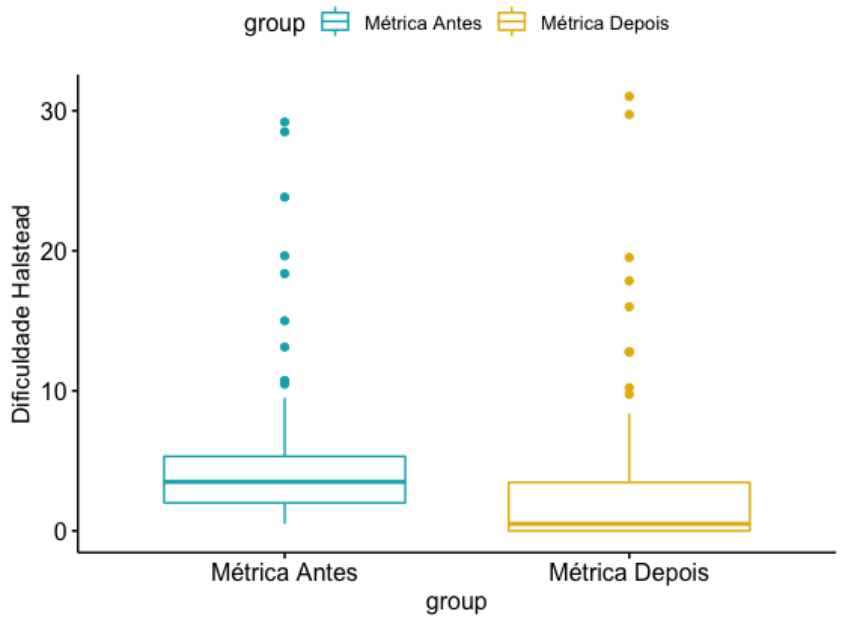
**Listagem 16 – Resultado teste estatístico da métrica de Dificuldade Halstead para o projeto *freeCodeCamp***

```
1         Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 306364, p-value = 0.03167
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

**Sumário $R_2$ (*freeCodeCamp*):** A Dificuldade Halstead média do projeto diminuiu de 3,82 para 3,59; representando uma diminuição de 6% na Dificuldade Halstead do projeto. Portanto, foi rejeitada a hipótese nula ($H_0$).

**Figura 24 – Comparativo da métrica de Dificuldade Halstead para o projeto *freeCodeCamp***




**Fonte: Autoria própria.**

Já para o projeto *JavaScript-Algorithms*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 17. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *JavaScript-Algorithms*. Pode-se observar no Código 18 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de Dificuldade Halstead. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma melhora efetiva na Dificuldade Halstead das mesmas.

**Listagem 17 – Resultado teste estatístico da métrica de Dificuldade Halstead para o projeto *JavaScript-Algorithms***

```
1          Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 1055188, p-value = 0.9121
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

**Listagem 18 – Resultado teste estatístico da métrica de Dificuldade Halstead para o projeto *JavaScript-Algorithms***

```
1          Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 600.5, p-value = 0.4759
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 25, a distribuição da Dificuldade Halstead das 34 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram aproximadas; especificamente, antes da refatoração, a Dificuldade Halstead média era de 6,72 e após a refatoração 6,71.

**Sumário $R_2$ (*JavaScript-Algorithms*):** A Dificuldade Halstead média do projeto diminuiu de 6,72 para 6,71; representando uma diminuição menor que 1% na Dificuldade Halstead do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

Por fim, para o projeto *TheAlgorithms/Javascript*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 19. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

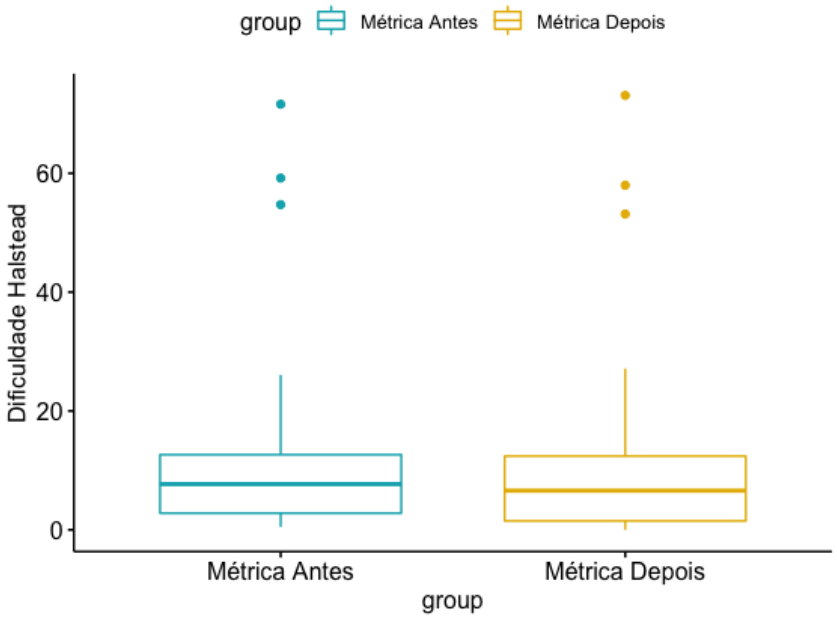
**Listagem 19 – Resultado teste estatístico da métrica de Dificuldade Halstead para o projeto *TheAlgorithms/Javascript***

```
1   Wilcoxon rank sum test with continuity correction
2
3 W = 2494793, p-value = 0.9745
4 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *TheAlgorithms/Javascript*. Pode-se observar no Código 20 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de Di-

**Figura 25 – Comparativo da métrica de Dificuldade Halstead para o projeto *JavaScript-Algorithms***




**Fonte: Autoria própria.**

ficuldade Halstead. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma melhora efetiva na Dificuldade Halstead das mesmas.

**Listagem 20 – Resultado teste estatístico da métrica de Dificuldade Halstead para o projeto *The-Algorithms/Javascript***

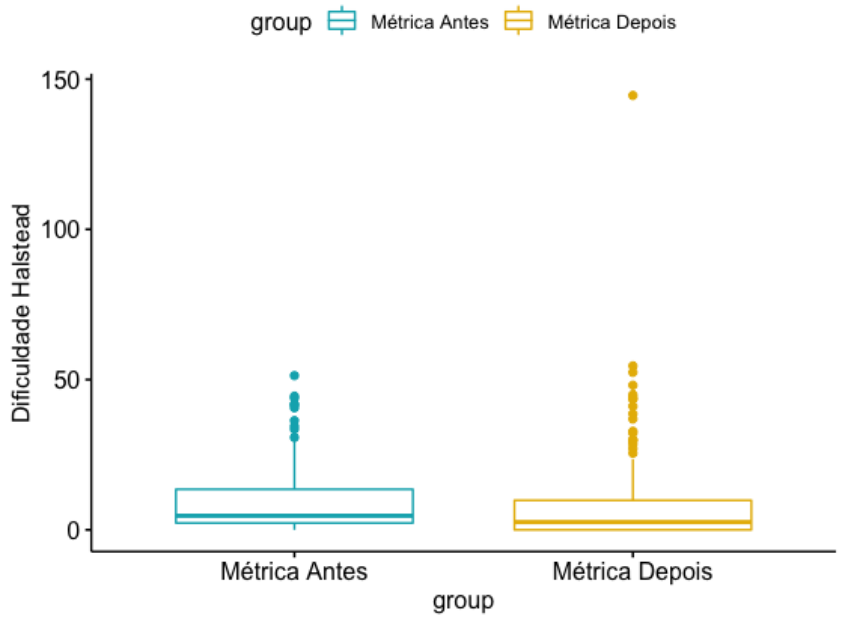
```
1          Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 12422, p-value = 0.0001022
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 26, a distribuição da Dificuldade Halstead das 141 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram diferentes; especificamente, antes da refatoração, a Dificuldade Halstead média era de 6,07 e após a refatoração 6,18. Na Figura 26 é possível observar que uma amostra elevou o valor da média, isso é dado devido alguns falsos positivos onde foi aplicado a técnica de refatoração *Inline Method* trazendo toda a implementação de outros métodos para um só.

**Sumário $R_2$ (*TheAlgorithms/Javascript*):** A Dificuldade Halstead média do projeto aumentou de 6,07 para 6,18; representando um aumento de 2% na Dificuldade Halstead do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

**Figura 26 – Comparativo da métrica de Dificuldade Halstead para o projeto *TheAlgorithms/Javascript***




**Fonte: Autoria própria.**

### 7.3.3 Tempo Halstead ($R_3$)

A fim de validar a variância entre entre as amostras da métrica de Tempo Halstead médio dos projetos, foi aplicado o teste estatístico nas amostras coletadas.

Para o projeto *freeCodeCamp*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 21. O teste resultou em um *p-value* menor que 5%; portanto há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias diferentes.

**Listagem 21 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *freeCodeCamp***

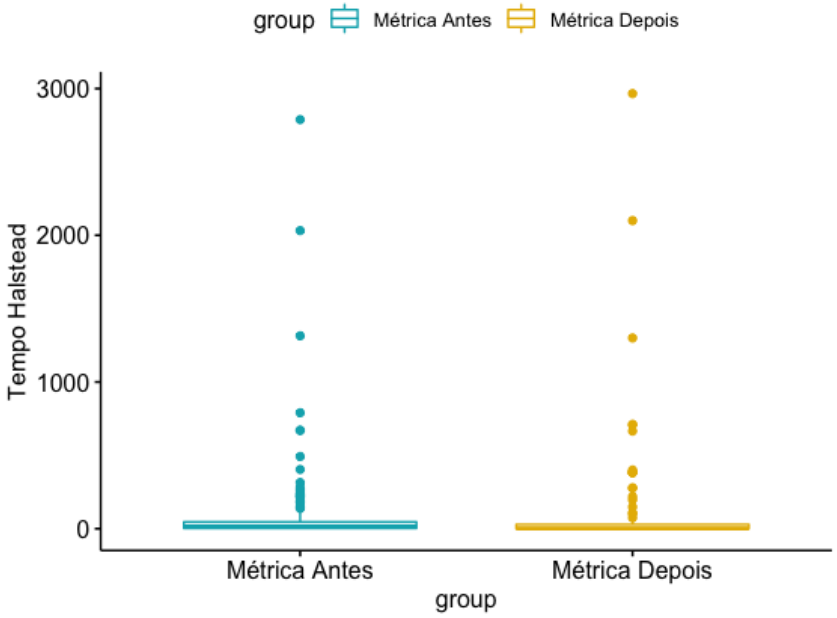
```
1         Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 306712, p-value = 0.02874
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 27, a distribuição do Tempo Halstead das 35 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, o Tempo Halstead médio era de 66,2 e após a refatoração 64,6.

| **Sumário** $R_3$ (***freeCodeCamp***)**:** O Tempo Halstead médio do projeto diminuiu de 66,2 para 64,6; representando uma diminuição de 3% no Tempo Halstead médio do projeto. Portanto, foi rejeitada a hipótese nula ($H_0$). |
|---|

**Figura 27 – Comparativo da métrica de Tempo Halstead para o projeto *freeCodeCamp***




**Fonte: Autoria própria.**

Já para o projeto *JavaScript-Algorithms*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 22. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

**Listagem 22 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *JavaScript-Algorithms***

```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 1055696, p-value = 0.8944
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *JavaScript-Algorithms*. Pode-se observar no Código 23 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de Tempo Halstead. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma diferença significativa na Tempo Halstead das mesmas.

Pode-se observar na Figura 28, a distribuição do Tempo Halstead das 37 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram

**Listagem 23 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *JavaScript-Algorithms***

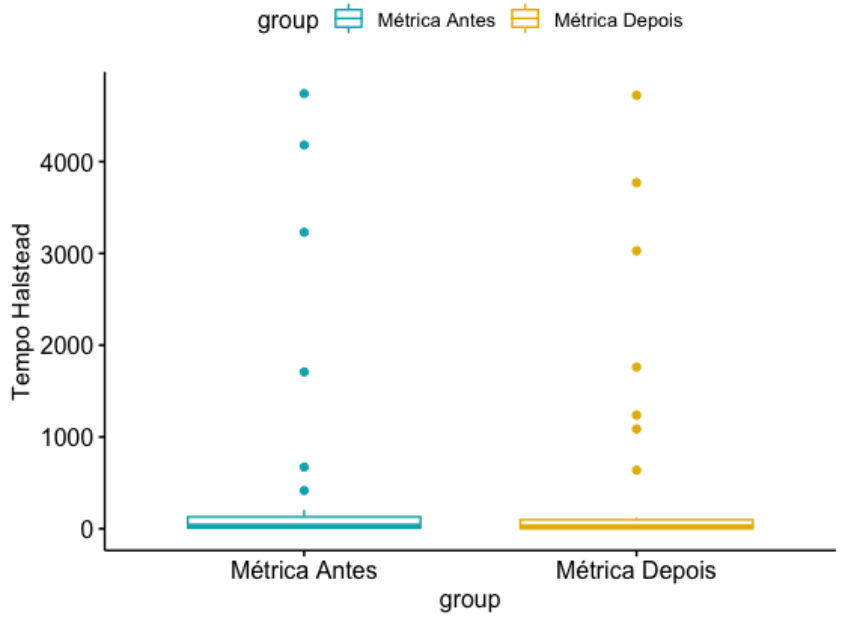
```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 732.5, p-value = 0.3436
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

próximas; especificamente, antes da refatoração, o Tempo Halstead médio era de 208 e após a refatoração 209.

> **Sumário $R_3$ (*JavaScript-Algorithms*):** O Tempo Halstead médio do projeto aumentou de 208 para 209; representando um aumento menor que 1% no Tempo Halstead médio do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

**Figura 28 – Comparativo da métrica de Tempo Halstead para o projeto *JavaScript-Algorithms***




**Fonte: Autoria própria.**

Por fim, para o projeto *TheAlgorithms/Javascript*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 24. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *TheAlgorithms/Javascript*. Pode-se observar no Código 25 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de

**Listagem 24 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *TheAlgorithms/Javascript***

```
1         Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 2494806, p-value = 0.9748
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Tempo Halstead. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma melhora efetiva na Tempo Halstead das mesmas.

**Listagem 25 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *TheAlgorithms/Javascript***

```
1         Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 15132, p-value = 0.0001916
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 29, a distribuição do Tempo Halstead das 157 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram diferentes; especificamente, antes da refatoração, o Tempo Halstead médio era de 112 e após a refatoração 122. Na Figura 29 é possível observar que uma amostra elevou o valor da média, isso é dado devido alguns falsos positivos onde foi aplicado a técnica de refatoração *Inline Method* trazendo toda a implementação de outros métodos para um só.

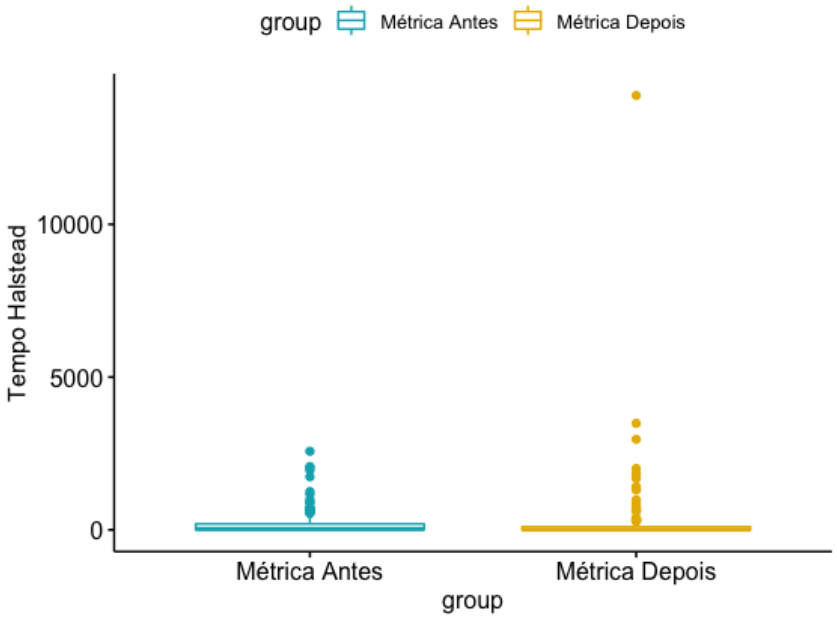
**Sumário $R_3$ (*TheAlgorithms/Javascript*):** O Tempo Halstead médio do projeto aumentou de 112 para 122; representando uma aumento de 9% no Tempo Halstead médio do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

### 7.3.4 Linhas de Código-Fonte para Processamento Lógico ($R_4$)

A fim de validar a variância entre entre as amostras da métrica de *LLOC* médio dos projetos, foi aplicado o teste estatístico nas amostras coletadas.

Para o projeto *freeCodeCamp*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 26. O teste resultou em um *p-value* menor que 5%; portanto há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias diferentes.

**Figura 29 – Comparativo da métrica de Tempo Halstead para o projeto *TheAlgorithms/Javascript***




**Fonte: Autoria própria.**

**Listagem 26 – Resultado teste estatístico da métrica de *LLOC* para o projeto *freeCodeCamp***

```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 307840, p-value = 0.0199
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

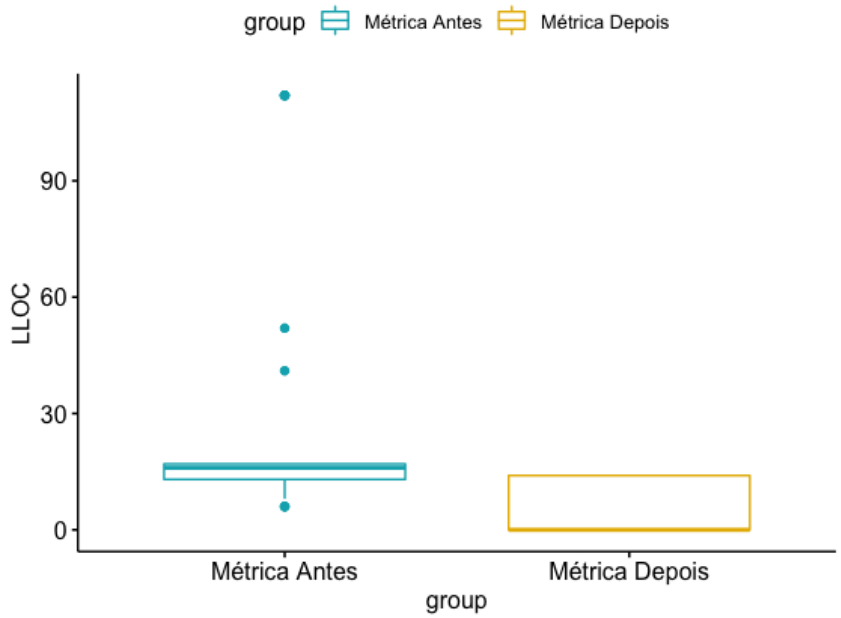
Pode-se observar na Figura 30, a distribuição da métrica de *LLOC* das 35 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, o *LLOC* médio era de 35,1 e após a refatoração 32,5.

> **Sumário $R_4$ (*freeCodeCamp*):** O *LLOC* médio do projeto diminuiu de 35,1 para 32,5; representando uma diminuição de 8% no *LLOC* médio do projeto. Portanto, foi rejeitada a hipótese nula ($H_0$).

Já para o projeto *JavaScript-Algorithms*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 27. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *JavaScript-Algorithms*. Pode-se observar no Código 28 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de

**Figura 30 – Comparativo da métrica de *LLOC* para o projeto *freeCodeCamp***




**Fonte: Autoria própria.**

**Listagem 27 – Resultado teste estatístico da métrica de *LLOC* para o projeto *JavaScript-Algorithms***

```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 1055394, p-value = 0.9042
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

*LLOC*. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma melhora efetiva na *LLOC* das mesmas.

**Listagem 28 – Resultado teste estatístico da métrica de *LLOC* para o projeto *JavaScript-Algorithms***

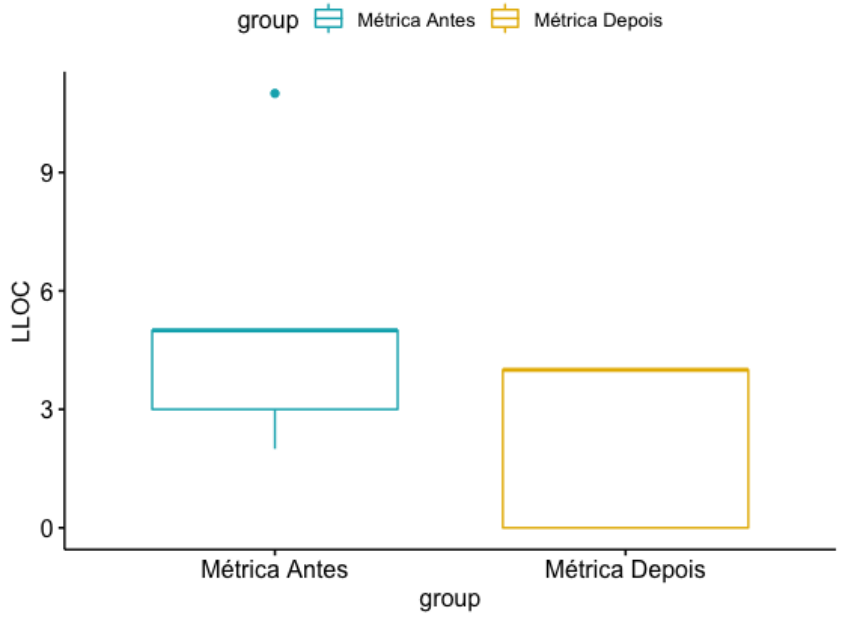
```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 76, p-value = 0.04774
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 31, a distribuição da métrica de *LLOC* das 11 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram próximas; especificamente, antes da refatoração, o Tempo Halstead médio era de 6,18 e após a refatoração 6,17.

**Sumário $R_4$ (*JavaScript-Algorithms*):** O *LLOC* médio do projeto diminuiu de 6,18 para 6,17; representando uma diminuição menor que 1% no *LLOC* médio do projeto. Portanto, não há evidências para rejeitar a hipótese nula ($H_0$).

**Figura 31 – Comparativo da métrica de *LLOC* para o projeto *JavaScript-Algorithms***




**Fonte: Autoria própria.**

Por fim, para o projeto *TheAlgorithms/Javascript*, após aplicar o Teste de **Wilcoxon** nas amostras coletadas, foi obtido o resultado apresentado abaixo no Código 29. O teste resultou em um *p-value* maior que 5%; portanto não há evidências para descartar a $H_0$, ou seja, as duas distribuições têm variâncias e médias iguais ou aproximadas.

**Listagem 29 – Resultado teste estatístico da métrica de *LLOC* para o projeto *TheAlgorithms/Javascript***

```
1        Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 2590964, p-value = 0.3956
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Vale considerar que o Teste de **Wilcoxon** apresentado acima, foi executado com base em todas as funções existentes no projeto *TheAlgorithms/Javascript*. Pode-se observar no Código 30 o Teste de **Wilcoxon** executado apenas nas funções que sofreram alteração na métrica de *LLOC*. Portanto, quando consideradas somente as funções que sofreram alteração na métrica, houve uma melhora efetiva na Tempo Halstead das mesmas.

**Listagem 30 – Resultado teste estatístico da métrica de Tempo Halstead para o projeto *TheAlgorithms/Javascript***

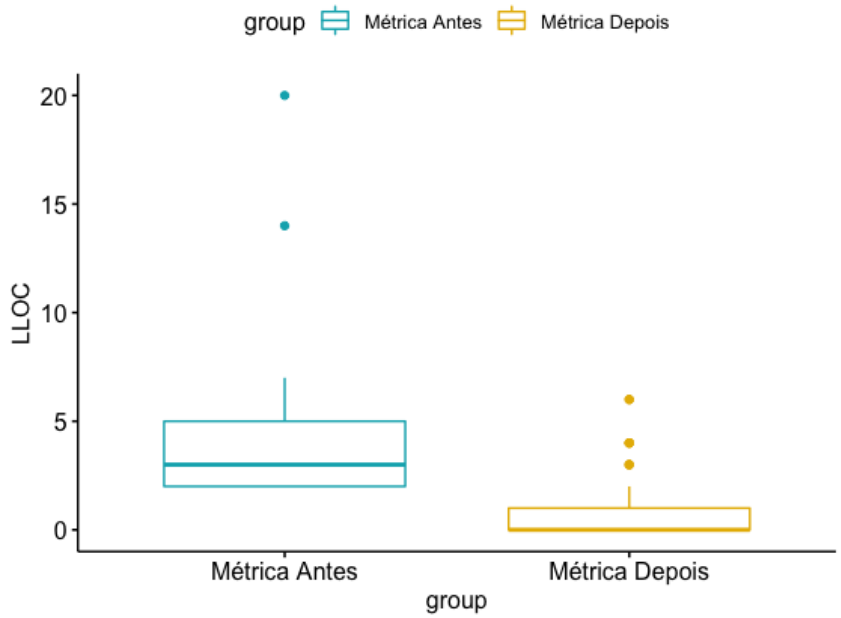
```
1         Wilcoxon rank sum test with continuity correction
2
3 data:  metricaAntes and metricaDepois
4 W = 3795, p-value = 5.022e-16
5 alternative hypothesis: true location shift is not equal to 0
```

**Fonte: Autoria própria.**

Pode-se observar na Figura 32, a distribuição da métrica de *LLOC* das 66 funções que sofreram alteração no valor da métrica. Nota-se que as médias dessas distribuições ficaram diferentes; especificamente, antes da refatoração, o *LLOC* médio era de 4 e após a refatoração 0.78.

**Sumário $R_4$ (*TheAlgorithms/Javascript*):** O *LLOC* médio do projeto diminuiu de 8,64 para 8,55; representando uma redução de 2% no *LLOC* médio do projeto. Portanto há evidências para rejeitar a hipótese nula ($H_0$).

**Figura 32 – Comparativo da métrica de *LLOC* para o projeto *TheAlgorithms/Javascript***




**Fonte: Autoria própria.**

## 7.4 Discussões

Garantir a qualidade interna de um de software é um grande e conhecido desafio (SCHWARZ; STEFFENS; LICHTER, 2018). A tarefa de identificação de *code smells* em um

código-fonte pode ser algo complexo, uma vez que a sua aplicabilidade é subjetiva e os resultados podem sofrer influência de fatores externos.

Saber o número real de *code smells* presentes em um código-fonte é fundamental para a aplicação das técnicas de refatoração (KHOMH *et al.*, 2009). Para isso, um importante passo na identificação dos mesmos seria o tratamento de falsos positivos ou falsos negativos. Porém para esse trabalho, todos os *code smells* identificados, foram considerados como verdadeiros positivos.

Para o projeto *freeCodeCamp*, quando realizado o processo de identificação dos *code smells*, pode não ter havido uma grande incidência desses falsos positivos, por isso os resultados obtidos foram expressivos. Também foi observada uma melhoria da qualidade interna do projeto, evidenciado pelas métricas escolhidas neste trabalho. Pode-se observar tal melhoria no Quadro 5.

Já para os projetos *JavaScript-Algorithms* e *TheAlgorithms/Javascript*, a incidência de falsos positivos parecem ter sido muito maiores, assim a aplicação das técnicas de refatoração nos *code smells* identificados, não trouxeram uma melhora significativa nas métricas coletadas do código-fonte. Analisando o total de funções refatoradas, apenas 5% das funções existentes nos projetos foram refatoradas; dessa forma, uma melhoria analisando todas as funções pode não ser percebida. Porém, outro importante ponto a destacar, é que a aplicação dessas técnicas de refatoração melhoram a qualidade do código-fonte, reduzindo assim a incidência de *code smells*, tal melhoria pode ser identificada por outras métricas, além das elencadas para esse projeto.

Ainda para os projetos *JavaScript-Algorithms* e *TheAlgorithms/Javascript*, como apresentado no Quadro 4, pode-se observar que a maior parte das técnicas de refatoração aplicadas foram *Extract Interface* e *Inline Method*. Como apresentado na Seção 5.3, tais técnicas de refatoração não apresentam uma expressividade nos resultados obtidos das métricas do código-fonte após a aplicação das técnicas de refatoração. Isso é dado porque, como apresentado anteriormente, essas técnicas de refatoração buscam trazer uma padronização e organização no código-fonte, sem alterar a lógica de execução, apenas reorganizando a disponibilidade e execução dos métodos no código-fonte, facilitando assim sua compreensão. Pode-se observar o resultado da média das métricas antes e depois de aplicar as refatorações no Quadro 6 e 7 respectivamente.

**Quadro 5 – Comparativo da média das métricas para o projeto *freeCodeCamp***

| **Métrica** | **Média Antiga** | **Nova Média** | ***p-value*** |
|---|---|---|---|
| **Complexidade Ciclomática** | 1,61 | 1,51 | **0,00133** |
| **Dificuldade Halstead** | 3,82 | 3,59 | **0,03167** |
| **Tempo Halstead** | 66,2 | 64,6 | **0,02874** |
| ***LLOC*** [9] | 35,1 | 32,5 | **0,0199** |

**Fonte: Autoria própria.**

**Quadro 6 – Comparativo da média das métricas para o projeto *JavaScript-Algorithms***

| Métrica | Média Antiga | Nova Média | *p-value* |
|---|---|---|---|
| **Complexidade Ciclomática** | 1,58 | 1,57 | 0,8202 |
| **Dificuldade Halstead** | 6,72 | 6,71 | 0,9121 |
| **Tempo Halstead** | 208 | 209 | 0,8944 |
| ***LLOC*** [10] | 6,18 | 6,17 | 0,9042 |

**Fonte: Autoria própria.**

**Quadro 7 – Comparativo da média das métricas para o projeto *TheAlgorithms/Javascript***

| Métrica | Média Antiga | Nova Média | *p-value* |
|---|---|---|---|
| **Complexidade Ciclomática** | 1,73 | 1,70 | **0,04305** |
| **Dificuldade Halstead** | 6,07 | 6,18 | 0,9745 |
| **Tempo Halstead** | 112 | 122 | 0,9748 |
| ***LLOC*** [11] | 8,64 | 8,55 | 0,3956 |

**Fonte: Autoria própria.**

No Código 31, é possível observar um trecho de código-fonte do projeto *freeCodeCamp*, onde existe um *code smell* do tipo *Uso desnecessário de estruturas If-else*. É possível observar no Código 32 o código-fonte refatorado após o *plugin* identificar o *code smell* e refatorar o mesmo utilizando a técnica de refatoração *Converter Expressão para Operador Condicional Ternário*. Na Tabela 7, é possível observar as métricas antes e depois de aplicar a refatoração e constatar a melhoria no resultado das métricas de *LLOC* e Tempo Halstead.

**Listagem 31 – Trecho de código-fonte do projeto *freeCodeCamp* antes de aplicar a refatoração**

```
1 export function getEncodedEmail(email) {
2   if (!email) {
3     return null;
4   }
5   return Buffer.from(email).toString('base64');
6 }
```

**Fonte: Autoria própria.**

**Listagem 32 – Trecho de código-fonte do projeto *freeCodeCamp* após aplicar a refatoração**

```
1 export function getEncodedEmail(email) {
2   return !email ? null : Buffer.from(email).toString('base64');
3 }
```

**Fonte: Autoria própria.**

No Código 33, é possível observar um trecho de código-fonte do projeto *JavaScript-Algorithms*, onde existe um *code smell* do tipo *Classes Alternativas com Diferentes Interfaces*. É possível observar no Código 34 o código-fonte refatorado após o *plugin* identificar o *code smell* e refatorar o mesmo utilizando a técnica de refatoração *Extract Interface*. Na Tabela 8, é possível observar as métricas antes e depois de aplicar a refatoração e constatar que não

**Tabela 7 – Métricas extraídas do código-fonte do projeto *freeCodeCamp* antes e depois de refatorar**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 31 | Código 32 |
| Complexidade Ciclomática | 2 | 2 |
| Dificuldade Halstead | 2.917 | 2.917 |
| LLOC | 5 | 3 |
| Halstead Time | 8.408 | 7.848 |

**Fonte: Autoria própria.**

houve alteração no valor das métricas. O corpo do código-fonte foi omitido pois a refatoração foi aplicada somente na assinatura da função, não havendo a necessidade de exibir o corpo do mesmo.

**Listagem 33 – Trecho de código-fonte do projeto *JavaScript-Algorithms* antes de aplicar a refatoração**

```
1  export default class Sort {
2    sort() {
3      return;
4    }
5  }
6  export default class BubbleSort {
7    sort() {
8      return;
9    }
10 }
```

**Fonte: Autoria própria.**

**Listagem 34 – Trecho de código-fonte do projeto *JavaScript-Algorithms* após aplicar a refatoração**

```
1  export default class Sort implements Extracted {
2    sort() {
3      return;
4    }
5  }
6  export default class BubbleSort implements Extracted {
7    sort() {
8      return;
9    }
10 }
11 interface Extracted {
12   sort();
13 }
```

**Fonte: Autoria própria.**

No Código 35, é possível observar um trecho de código-fonte do projeto *TheAlgorithms/Javascript*, onde existe um *code smell* do tipo *Middle Man*. É possível observar no Código 36

**Tabela 8 – Métricas extraídas do código-fonte do projeto *JavaScript-Algorithms* antes e depois de refatorar**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 33 | Código 34 |
| Complexidade Ciclomática | 3 | 3 |
| Dificuldade Halstead | 1.333 | 1.333 |
| LLOC | 6 | 6 |
| Halstead Time | 1.376 | 1.376 |

**Fonte: Autoria própria..**

o código-fonte refatorado após o *plugin* identificar o *code smell* e refatorar o mesmo utilizando a técnica de refatoração *Inline Function*. Na Tabela 9, é possível observar as métricas antes e depois de aplicar a refatoração e constatar a melhoria no resultado do valor de todas as métricas.

**Listagem 35 – Trecho de código-fonte do projeto *TheAlgorithms/Javascript* antes de aplicar a refatoração**

```
1  function isLetter (str) {
2    return str.length === 1 && str.match(/[a-zA-Z]/i)
3  }
4
5  function encrypt (message, key) {
6    for (let i = 0, j = 0; i < message.length; i++) {
7      const c = message.charAt(i)
8      if (isLetter(c)) {
9        return true;
10     } else {
11       return false;
12     }
13   }
14 }
```

**Fonte: Autoria própria.**

**Tabela 9 – Métricas extraídas do código-fonte do projeto *TheAlgorithms/Javascript* antes e depois de refatorar**

| Métricas extraídas do código-fonte | | |
|---|---|---|
| Métrica | Código 35 | Código 36 |
| Complexidade Ciclomática | 5 | 4 |
| Dificuldade Halstead | 11.8 | 10.833 |
| LLOC | 10 | 8 |
| Halstead Time | 65.259 | 63.112 |

**Fonte: Autoria própria..**

**Listagem 36 – Trecho de código-fonte do projeto *TheAlgorithms/Javascript* após aplicar a refatoração**

```
function encrypt (message, key) {
  for (let i = 0, j = 0; i < message.length; i++) {
    const c = message.charAt(i)
    if (c.length === 1 && c.match(/[a-zA-Z]/i)) {
      return true;
    } else {
      return false;
    }
  }
}
```

**Fonte: Autoria própria.**

## 7.5 Ameaças à Validade

A ameaça à validade de construção é referente a ligação entre a teoria e o que é observado durante a construção do estudo (WOHLIN *et al.*, 2012). Como principal ameaça a validade de construção do presente estudo, pode-se citar o fato da ferramenta implementada não abranger outros tipos de *code smells* não obtendo assim resultados expressivos em projetos reais.

A ameaça à validade interna está relacionada a fatores que impactam nos resultados e que não podem ser controlados (WOHLIN *et al.*, 2012). A principal ameaça é a ocorrência de falsos positivos ou falsos negativos na identificação dos *code smells*, uma vez que a ocorrência excessiva deles pode comprometer os dados e distorcer as análises. No presente trabalho, todos os *code smells* que a ferramenta identifica são considerados como verdadeiros positivos. Em um contexto real, geralmente é preciso uma análise de um especialista no código-fonte para validar se a identificação do *code smell* representa de fato um problema. No entanto, de certa forma, a remoção dos *code smells* é um indício de que realmente havia sintomas de mau design nos projetos avaliados, pois foi possível identificar melhorias nas métricas de código fonte em alguns dos projetos analisados.

A ameaça à validade externa está relacionada à generalização dos resultados para outros cenários (WOHLIN *et al.*, 2012). O estudo de caso foi realizado em alguns projetos específicos, por meio de uma avaliação na qual, quando aplicada em projetos com estruturas diferentes pode-se obter diferentes resultados e funcionamento do *plugin*.

# 8 CONSIDERAÇÕES FINAIS

Muitos trabalhos já foram realizados como forma de encontrar oportunidades de refatoração em um código-fonte e muitas ferramentas surgiram a partir desta referência. Porém, em sua grande maioria, poucos desses levam em considerações as métricas de qualidade de um software antes, durante ou depois de aplicar a refatoração, não tendo assim um panorama da real efetividade da refatoração aplicada naquele código-fonte, buscando apenas uma melhoria no design dele.

Neste trabalho foi desenvolvida uma ferramenta utilizando a abordagem de AST em conjunto com algumas métricas de qualidade de software de *Thomas J. McCabe* e *Maurice Howard Halstead* a fim de validar a qualidade da refatoração aplicada no código-fonte. Dessa forma, foi possível aplicar refatorações no código-fonte e avaliar a sua efetividade a nível de melhoria fornecida na qualidade interna do software validado, assim auxiliando no processo de mitigação da desconfiança por parte dos desenvolvedores na utilização de ferramentas que aplicam refatorações de forma automática.

Espera-se que este trabalho possa contribuir para o desenvolvimento de novos estudos a fim de trazer qualidade ao código-fonte produzido no dia-a-dia de um desenvolvedor, melhorando assim a qualidade interna dos softwares produzidos e diminuindo assim a dívida técnica[1], facilitando sua manutenção e evolução natural.

## 8.1 Trabalhos Futuros

Por meio da ferramenta desenvolvida, foram observados alguns pontos de melhorias técnicas que podem contribuir para aumentar a abrangência e efetividade da ferramenta ao identificar os *code smells*. Além disso, a abordagem da ferramenta e os padrões de projeto aplicados, podem favorecer a efetividade da ferramenta em aplicar as refatorações no código-fonte.

Utilizando outras abordagens de utilização como por exemplo aplicação da ferramenta em uma *pipeline* no processo de *Continuous Integration / Continuous Delivery* (CI/CD) dos projetos que utilizam a ferramenta trariam uma maior abrangência e padronização da utilização da ferramenta por todos os desenvolvedores envolvidos no projeto.

Entende-se também que existe uma grande limitação nas linguagens e *frameworks* que a ferramenta tem seu pleno funcionamento. É necessário buscar possibilidades de soluções para aumentar a abrangência da ferramenta para que possa ser utilizada por um universo maior de projetos. Como uma alternativa, existe um grande campo a ser explorado para se melhorar a qualidade de um código-fonte, utilizando abordagens baseadas em aprendizado de máquina.

[1] Metáfora utilizada para definir quando se compromete a qualidade do código-fonte para algum benefício imediato, contrai-se uma dívida onde o valor do empréstimo é o esforço economizado na execução da tarefa (ZAZWORKA *et al.*, 2013).

# REFERÊNCIAS


BECK, K.; FOWLER, M.; BECK, G. Bad smells in code. **Refactoring: Improving the design of existing code**, v. 1, n. 1999, p. 75–88, 1999.

CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of code smells in object-oriented systems. **Innovations in Systems and Software Engineering**, Springer, v. 10, n. 1, p. 3–18, 2014.

CINNÉIDE, M. Ó. *et al.* Experimental assessment of software metrics using automated refactoring. *In*: ACM. **ACM-IEEE international symposium on Empirical software engineering and measurement**. [*S.l.*], 2012. p. 49–58.

DANPHITSANUPHAN, P.; SUWANTADA, T. Code smell detecting tool and code smell-structure bug relationship. *In*: IEEE. **2012 Spring Congress on Engineering and Technology**. [*S.l.*], 2012. p. 1–5.

ERDEMIR, U.; TEKIN, U.; BUZLUCA, F. E-quality: A graph based object oriented software quality visualization tool. *In*: IEEE. **6th International Workshop on Visualizing Software for Understanding and Analysis**. [*S.l.*], 2011. p. 1–8.

ERLIKH, L. Leveraging legacy system dollars for e-business. **IT professional**, IEEE, v. 2, n. 3, p. 17–23, 2000.

FERNANDES, V. H. M. Um estudo empírico sobre métricas de código fonte do android api framework. 2018.

FOWLER, M. **Refatoração: Aperfeiçoamento e Projeto**. [*S.l.*]: Bookman Editora, 2009.

FOWLER, M. **Refactoring: improving the design of existing code**. [*S.l.*]: Addison-Wesley Professional, 2018.

FRANCO, E. F.; HIRAMA, K.; ROSSI, R. Uma análise qualitativa-sistemática da relação entre o acúmulo de dívida técnica e a satisfação de usuários ao longo da operação de pacotes de software empresarial. **Revista GEPROS**, v. 13, n. 4, p. 263, 2018.

GE, X.; DUBOSE, Q. L.; MURPHY-HILL, E. Reconciling manual and automatic refactoring. *In*: IEEE PRESS. **34th International Conference on Software Engineering**. [*S.l.*], 2012. p. 211–221.

GE, X. *et al.* Refactoring-aware code review. *In*: IEEE. **2017 IEEE Symposium on Visual Languages and Human-Centric Computing**. [*S.l.*], 2017. p. 71–79.

HERBOLD, S.; GRABOWSKI, J.; NEUKIRCHEN, H. Automated refactoring suggestions using the results of code analysis tools. *In*: IEEE. **1st International Conference on Advances in System Testing and Validation Lifecycle**. [*S.l.*], 2009. p. 104–109.

INCOSE, D. D. W. Systems engineering handbook: A guide for system life cycle processes and activities. **San Diego, US-CA: International Council on Systems Engineering**, John Wiley & Sons, Inc., 2015.

JUNIOR, H. d. S. C.; FILHO, L. R. V. M.; ARAUJO, M. A. P. An approach for detecting unnecessary cyclomatic complexity on source code. **IEEE Latin America Transactions**, IEEE, v. 14, n. 8, p. 3777–3783, 2016.

KALINOWSKI, M. *et al.* Mps. br: promovendo a adoção de boas práticas de engenharia de software pela indústria brasileira. *In*: SN. **XIII Congreso Iberoamericano en"Software Engineering"(CIBSE), Cuenca, Equador**. [*S.l.*], 2010.

KHOMH, F. *et al.* A bayesian approach for the detection of code and design smells. *In*: IEEE. **2009 Ninth International Conference on Quality Software**. [*S.l.*], 2009. p. 305–314.

KIMURA, S. *et al.* Move code refactoring with dynamic analysis. *In*: IEEE. **28th International Conference on Software Maintenance**. [*S.l.*], 2012. p. 575–578.

KITCHENHAM, B. Procedures for performing systematic reviews. **Keele, UK, Keele University**, v. 33, n. 2004, p. 1–26, 2004.

LEHMAN, M. M. Laws of software evolution revisited. *In*: SPRINGER. **European Workshop on Software Process Technology**. [*S.l.*], 1996. p. 108–124.

LIU, H.; XU, Z.; ZOU, Y. Deep learning based feature envy detection. *In*: ACM. **33rd ACM/IEEE International Conference on Automated Software Engineering**. [*S.l.*], 2018. p. 385–396.

MAFFORT, C. *et al.* Mining architectural violations from version history. **Empirical Software Engineering**, Springer, v. 21, n. 3, p. 854–895, 2016.

MAGALHÃES, N. M. *et al.* An automated refactoring approach to remove unnecessary complexity in source code. *In*: ACM. **2nd Brazilian Symposium on Systematic and Automated Software Testing**. [*S.l.*], 2017. p. 3.

MALATHI, S.; SUDHAKAR, P. Implementation of software refactoring using foda tool. *In*: IEEE. **3rd International Conference on Communication and Electronics Systems (ICCES)**. [*S.l.*], 2018. p. 839–842.

MALERBA, C. Vulnerabilidades e exploits: técnicas, detecção e prevenção. 2010.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MEIRELLES, P. R. M. **Monitoramento de métricas de código-fonte em projetos de software livre**. 2013. Tese (Doutorado) — Universidade de São Paulo, 2013.

MENG, N.; KIM, M.; MCKINLEY, K. S. Lase: Locating and applying systematic edits by learning from examples. *In*: **2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 502–511. ISBN 978-1-4673-3076-3. Disponível em: http://dl.acm.org/citation.cfm?id=2486788.2486855.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Transactions on software engineering**, IEEE, v. 30, n. 2, p. 126–139, 2004.

MOHAN, M.; GREER, D.; MCMULLAN, P. Maximizing refactoring coverage in an automated maintenance approach using multi-objective optimization. *In*: IEEE PRESS. **3rd International Workshop on Refactoring**. [*S.l.*], 2019. p. 31–38.

MUNRO, M. J. Product metrics for automatic identification of"bad smell"design problems in java source-code. *In*: IEEE. **11th International Software Metrics Symposium (METRICS'05)**. [*S.l.*], 2005. p. 15–15.

NEGARA, S. *et al.* A comparative study of manual and automated refactorings. *In*: **27th European Conference on Object-Oriented Programming**. Berlin, Heidelberg:

Springer-Verlag, 2013. (ECOOP'13), p. 552–576. ISBN 978-3-642-39037-1. Disponível em: http://dx.doi.org/10.1007/978-3-642-39038-8_23.

NYAMAWE, A. S. *et al.* Recommending refactoring solutions based on traceability and code metrics. **IEEE Access**, IEEE, v. 6, p. 49460–49475, 2018.

ORCHARD, D.; RICE, A. Upgrading fortran source code using automatic refactoring. *In*: ACM. **2013 ACM workshop on Workshop on refactoring tools**. [*S.l.*], 2013. p. 29–32.

PANTIUCHINA, J. Towards just-in-time rational refactoring. *In*: IEEE PRESS. **41st International Conference on Software Engineering: Companion Proceedings**. [*S.l.*], 2019. p. 180–181.

PANTIUCHINA, J. *et al.* Towards just-in-time refactoring recommenders. *In*: ACM. **26th Conference on Program Comprehension**. [*S.l.*], 2018. p. 312–315.

REFACTORING.GURU. **Refactoring Guru**. 2014. Disponível em: https://refactoring.guru. Acesso em: 15 de maio de 2022.

RIBEIRO, M.; BORBA, P. Recommending refactorings when restructuring variabilities in software product lines. *In*: ACM. **2nd Workshop on Refactoring Tools**. [*S.l.*], 2008. p. 8.

SAE-LIM, N.; HAYASHI, S.; SAEKI, M. Toward proactive refactoring: an exploratory study on decaying modules. *In*: IEEE PRESS. **3rd International Workshop on Refactoring**. [*S.l.*], 2019. p. 39–46.

SANTOS, G. *et al.* Recording and replaying system specific, source code transformations. *In*: IEEE. **15th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [*S.l.*], 2015. p. 221–230.

SANTOS, G. J. de S. **Assessing and improving code transformations to support software evolution**. 2017. Tese (Doutorado), 2017.

SCHWARZ, J.; STEFFENS, A.; LICHTER, H. Code smells in infrastructure as code. *In*: IEEE. **2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)**. [*S.l.*], 2018. p. 220–228.

SHARMA, T. Identifying extract-method refactoring candidates automatically. *In*: ACM. **5th Workshop on Refactoring Tools**. [*S.l.*], 2012. p. 50–53.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. *In*: ACM. **24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [*S.l.*], 2016. p. 858–870.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics based refactoring. *In*: IEEE. **5th European Conference on Software Maintenance and Reengineering**. [*S.l.*], 2001. p. 30–38.

SZŐKE, G. *et al.* Do automatic refactorings improve maintainability? an industrial case study. *In*: IEEE. **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S.l.*], 2015. p. 429–438.

TAVARES, C. S.; FERREIRA, F.; FIGUEREDO, E. Um mapeamento sistemático da literatura sobre ferramentas de refatoração de software. *In*: SBC. **Anais do XIV Simpósio Brasileiro de Sistemas de Informação**. [*S.l.*], 2018. p. 88–81.

WOHLIN, C. *et al.* **Experimentation in software engineering**. [*S.l.*]: Springer Science & Business Media, 2012.

WYRICH, M.; BOGNER, J. Towards an autonomous bot for automatic source code refactoring. *In*: IEEE PRESS. **1st International Workshop on Bots in Software Engineering**. [*S.l.*], 2019. p. 24–28.

ZAZWORKA, N. *et al.* A case study on effectively identifying technical debt. *In*: ACM. **17th International Conference on Evaluation and Assessment in Software Engineering**. [*S.l.*], 2013. p. 42–47.