INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
CAMPUS CAJAZEIRAS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANALISANDO REFATORAÇÕES EM QUEBRAS DE CONFINAMENTO NA LINGUAGEM JAVA

JOSÉ DAVID DE OLIVEIRA SOUSA

Cajazeiras
2022

JOSÉ DAVID DE OLIVEIRA SOUSA

ANALISANDO REFATORAÇÕES EM QUEBRAS DE CONFINAMENTO NA LINGUAGEM JAVA

Trabalho de Conclusão de Curso apresentado junto ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - Campus Cajazeiras, como requisito à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador

Prof. Me. Ricardo de Sousa Job.

Cajazeiras
2022

FICHA CATALOGRÁFICA

Dados Internacionais de Catalogação na Publicação (CIP)

S725a Sousa, José David de Oliveira

Analisando refatorações em quebras de confinamento na linguagem JAVA/José David de Oliveira Sousa. – Cajazeiras/PB: IFPB, 2022.

58f.: il.

Trabalho de Conclusão de Curso de Tecnologia em Análise e Desenvolvimento de Sistemas - Instituto Federal de Educação, Ciência e Tecnologia da Paraíba-IFPB, Campus Cajazeiras. Cajazeiras, 2022.

Orientador(a): Professor Me. Ricardo de Sousa Job.

1. Automação – Linguagens de programação 2. Automação – Sistema operacional 3. Automação – JAVA 4. Automação – Extract Method Refactoring I. Refatoração de sistemas II. Título

CDU: 004.45

Ficha catalográfica elaborada pelo Departamento de Bibliotecas DBIBLIO/IFPB

SERVIÇO PÚBLICO FEDERAL
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
COORDENAÇÃO DO CURSO SUPERIOR EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - CAMPUS CAJAZEIRAS

ATA 34/2022 - CADS/UNINFO/DDE/DG/CZ/REITORIA/IFPB

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO (TCC)
CURSO: ANÁLISE E DESENVOLVIMENTO DE SISTEMAS (ADS)

Às 15h30 do dia 04 do mês de MAIO do ano de 2022, o(a) aluno(a) **JOSÉ DAVID DE OLIVEIRA SOUSA**, matrícula **201912010007**, apresentou, como parte dos requisitos para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, seu trabalho de conclusão de curso, tendo como título **"APLICANDO REFATORAÇÃO EM QUEBRAS DE CONFINAMENTO NA LINGUAGEM JAVA"**. Constituíram a banca examinadora os professores **Ricardo de Sousa Job** (orientador), **Leandro Luttiane da Silva Linhares** (examinador) e **Diogo Dantas Moreira** (examinador).

Após a apresentação e as observações dos membros da Banca Examinadora, ficou definido que o trabalho foi considerado **APROVADO** com nota **90**, com a condição de que o (a) aluno (a) entregue, no prazo máximo de 30 dias, a versão final do trabalho com as correções sugeridas pelos membros da banca examinadora. Eu, FÁBIO ABRANTES DINIZ, Coordenador do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, lavrei a presente ata, que segue assinada digitalmente por mim e pelos membros da banca examinadora.

Cajazeiras, 12 de maio de 2022.

Documento assinado eletronicamente por:

- Ricardo de Sousa Job, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 12/05/2022 18:39:30.
- Leandro Luttiane da Silva Linhares, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 12/05/2022 19:39:41.
- Diogo Dantas Moreira, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 25/05/2022 20:48:33.
- José David de Oliveira Sousa, ALUNO (201912010007) DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - CAJAZEIRAS, em 20/06/2022 13:16:28.

Este documento foi emitido pelo SUAP em 10/05/2022. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/autenticar-documento/ e forneça os dados abaixo:

Código Verificador: 293024
Código de Autenticação: 73067f1bdb

JOSÉ DAVID DE OLIVEIRA SOUSA

ANALISANDO REFATORAÇÕES EM QUEBRAS DE CONFINAMENTO NA LINGUAGEM JAVA

Trabalho de Conclusão de Curso apresentado junto ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - Campus Cajazeiras, como requisito à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador

Prof. Me. Ricardo de Sousa Job.

Aprovada em: **data da apresentação.**

Prof. Me. Ricardo de Sousa Job - Orientador

Prof. Me.Nome do examinador 1 - Avaliador
Instituição do avaliador- Campus do avaliador

Prof. Me.Nome do examinador 2 - Avaliador
Instituição do avaliador- Campus do avaliador

Dedico este TCC à minha família: meu irmão Daniel, meu pai Agenor e minha mãe Luciene, por todo o amor e aprendizado em cada passo da caminhada.

AGRADECIMENTOS

A Deus, por me permitir experienciar tudo isso. À minha família, essencialmente a fonte de tudo o que sou hoje. E ao corpo docente do IFPB, *campus* Cajazeiras, mas especialmente ao orientador Ricardo Job, pela paciência e contribuição para que eu me formasse.

”If I have seen further it is by standing on the shoulders of Giants.”

Isaac Newton

RESUMO

Linguagens de programação orientadas a objetos oferecem uma ampla variedade de estruturas para se utilizar em um determinado projeto de software. Graças a isto é que é necessário realizar um decisão bem pensada na hora de escolher com que estrutura trabalhar no momento de implementar o projeto. Uma consequência são os possíveis problemas em que classes alteram o estado de atributos privados de outras classes. De modo particular, os projetos acadêmicos escritos em Java oferecem uma maior chance de apresentar problemas de quebras de confinamento. As quebras de confinamento produzem um custo adicional na hora de realizar a manutenção do software, pois mudanças futuras no código podem gerar mais facilmente erros, refletindo na flexibilidade do código. Ao final do estudo, foi constatado que uma ferramenta para reduzir esta inflexibilidade automaticamente é algo viável uma vez que reduz grandemente, em números, o problema nos projetos de código aberto estudados.

Palavras-chave: Quebra de confinamento. *Extract Method Refactoring*. Manutenção de software.

ABSTRACT

Object-oriented programming languages offer a wide variety of frameworks to use in a given software project. Thanks to this, it is necessary to make a well thought-out decision when choosing which structure to work with when implementing the project. A consequence that illustrates this idea well is the possible problems in which classes change the state of private attributes of other classes. In particular, academic projects written in java offer a greater chance of presenting confinement break problems. Confinement breaks incur an additional cost when it comes to software maintenance, as future code changes can more easily generate errors, reflecting the code's inflexibility. At the end of the study, it was found that a tool to automatically reduce this inflexibility is something viable since it greatly reduces, in numbers, the problem in the open source projects studied.

Keywords: Confinement break. Extract Method Refactoring. Software economy.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
ADS	Análise e Desenvolvimento de Sistemas
IFPB	Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
NBR	Norma Brasileira
TCC	Trabalho de Conclusão do Curso
PSI	Program Structure Interface
MCP	Method Call Parser
POO	Programação Orientada a Objetos
IDE	Integrated Development Environment
SDK	Software Development Kit

SUMÁRIO

1 INTRODUÇÃO

Em Ciência da Computação, os algoritmos são etapas escritas em código de computador que servem para atingir determinados objetivos dentro de um conjunto finito de dados.

Quando essas rotinas de código possuem trechos similares ao longo do programa, sugere-se um ato muito comum entre vários desenvolvedores de inúmeras linguagens de programação: o de construir funções. Os **métodos**, em programação orientada a objetos (*POO*), são um conceito similar às funções e são grupos de comandos que, dentre outras coisas, tem como um de seus cargos ditar o comportamento da classe associada.

No contexto de *POO*, existe um conjunto de regras chamado **Lei de *Demeter*** citado no artigo de Pessoa et al. (2019). Associada a Ian Holland, essa lei defende um conjunto de princípios que um programa orientado a objetos deve seguir para que se tenha, ao longo do processo de desenvolvimento, um código mais coeso e principalmente adaptável.

Quando a Lei de *Demeter* é violada, aumenta-se a complexidade do programa, logo tem-se um código difícil de gerenciar, violações começam a aparecer e acoplamentos entres classes e métodos se destacam. Estes sintomas estão fortemente presentes nos *Code Smells* que são trechos de código perigosos para a manutenibilidade do projeto. Esses trechos de código são nocivos em alta incidência pois quando eles aparecem **aumentam o custo** de mudanças futuras. Observar o algoritmo 5 da seção 3 Ferramenta Desenvolvida.

Quando a classe `C` altera o atributo privado `elements` da classe `A`, está se manifestando uma quebra de confinamento, de acordo com (PESSOA et al., 2019). O citado atributo é privado, logo, confinado. A classe `C` está o alterando, ou seja, quebrando seu confinamento, quando ele deveria ter visibilidade restrita em relação a qualquer classe que não seja a classe `A`.

Quando, no ato de desenvolvimento, o indivíduo não tem conhecimento dos citados princípios, é uma tendência natural que o código se torne **menos econômico**, isto é, de baixa compreensão e custoso de se trabalhar. É indispensável que os desenvolvedores façam uma análise bem criteriosa sobre como os componentes de seu programa estão relacionados e se estes apresentam a tão necessária baixa

dependência; pois, *contrario sensu*, uma pequena alteração pode representar um grande sacrifício.

Logo, como pode-se concluir a esta altura, mudar a forma como os componentes se relacionam e, ainda ter certeza de que essa alteração não subtraia do algoritmo o comportamento desejado é uma tarefa que requer substancial análise das formas como pode ser implementada.

Para reduzir os problemas de manutenibilidade causados pelo *Smell* da quebra de confinamento, é uma possível solução refatorar o código onde há ocorrências da mesma. A refatoração é o processo de realizar transformação, em outras palavras, **reescrita do código**, sem mudar o comportamento do mesmo. (FOWLER et al., 2002)

Se a expressão `getElements().add()` for extraída em um novo método e este for movido para a classe `A`, classe dona do atributo, se resolveria certamente o problema da fragilidade do código ocasionado pela localização da expressão em questão. Pois, de acordo com a Lei de Demeter, o método `m1` só pode alterar **diretamente** os atributos da classe que o contém.

Atualmente existem tecnologias que permitem uma automaticidade neste processo de mudar o código. Neste momento, pode-se citar o Kit de Desenvolvimento de Software (*SDK*) do *IntelliJ Idea*[1]. Atualmente na versão 2022.1.1, O *IntelliJ Idea* é um *IDE* robusto que oferece edição de código inteligente, incluindo código Java. O *SDK* do *IntelliJ* oferece suporte à modificação de arquivos abertos e, com isso, é possível refatorar as quebras de confinamento em Java.

Considerando o discutido até aqui, neste trabalho é implementado um algoritmo de **refatoração** de **quebras de confinamento** na linguagem **Java**. Isso é feito com o Kit de Desenvolvimento de Software do *IntelliJ Idea*, através do desenvolvimento do *plugin* nomeado ***Enclose*** para o citado *IDE*.

1.1 MOTIVAÇÃO

Vários desenvolvedores buscam produzir um código com qualidade, principalmente aqueles que conhecem os padrões de *design de software*, ou seja, boas práticas de organização do código que se repetem entre projetos. Um código com mais qualidade é aquele melhor **manutenível**, propriedade esta cujo custo corresponde a cerca de **67%** do custo total de um programa (MELLADO et al., 2015). Logo, pode-se inferir que um código de qualidade é aquele código com **pouco acoplamento** (dependências

[1] https://www.jetbrains.com/pt-br/idea/

entre componentes), com nenhuma ou **poucas quebras de confinamento** (conceito que será discutido mais adiante, mais especificamente na fundamentação, ao falarmos sobre *message chain*) e **balanceadas métricas** de linhas de código (*LOC*) e tamanho de classe.

A análise estática de código parece ser a abordagem mais notável neste caso. Ela funciona extraindo informação de um texto de determinado algoritmo em análise e demonstra necessidade de conhecimento de como é organizado um trecho de código que **viola A Lei de *Demeter***. Já existem ferramentas que trabalham com análise estática de código, a exemplo do ***MCP***[2], discutido na seção de fundamentação. Mas o que se sugere neste TCC é o desenvolvimento de uma ferramenta que proporcione ao desenvolvedor reorganizar o trecho de código onde é criado o alerta de violação.

1.2 PROBLEMA

Atuando na mão de programadores, existem instrumentos atuais capazes de disponibilizar dados comparativos acerca do ônus de determinadas estruturas de código. Por exemplo o *Metrics Reloaded*[3], que informa *LOC (Lines of Code)* cuja tradução é linhas de código, e número de classes. Porém, estas se limitam a apenas **apresentar**, sugerindo a construção de uma nova ferramenta que, na prática, **implemente** a insuflada alteração.

A necessidade de implementar estas escolhas de boas métricas pode ser evidenciada porque escolher uma boa forma de organização nos momentos iniciais do projeto de software é importante, já que isso reduz a probabilidade de surgir a necessidade de alterações futuras; isso pode trazer como vantagem também, no caso de surgirem alterações futuras, que essas alterações sejam muito mais econômicas e rápidas de se implementarem, no sentido de que um código **melhor planejado** acaba se tornando um código mais **flexível** e, logo, **econômico**.

Para considerar um projeto de software como econômico, podemos verificar se ele tem baixo acoplamento, isto é, um número baixo de dependências entre componentes; ou se ele não tem quebras de confinamento consideráveis. Para isso, existem ferramentas que nos mostram informações e métricas acerca de nosso código. Essas ferramentas utilizam a análise estática do código através da qual pode-se verificar inclusive se os princípios da lei de *Demeter* estão sendo seguidos. Entretanto, apesar de apresentar estas informações ao desenvolvedor, estes utilitários, como o *MCP* e

[2] https://github.com/gpes/mcp
[3] https://github.com/BasLeijdekkers/MetricsReloaded

o *MetricsReloaded* especialmente, não oferecem a funcionalidade de **reorganizar** a estrutura do código-fonte em um processo que melhore qualidade de software.

Para isto, nesse trabalho serão abordados alguns algoritmos *open-source* que manuseiam análise-estática como sua forma principal de processamento dos textos planos de software; além de serem estudados os trabalhos de notáveis autores da área de computação e de qualidade de software, a exemplo de (TERRA et al., 2017) e de (FOWLER et al., 2002). Ao final do trabalho, este conjunto de tarefas resultará em uma ferramenta confiável voltada para desenvolvedores que queriam melhorar a qualidade de seu software.

1.3 OBJETIVO

O objetivo principal deste trabalho é analisar os **índices de quebras de confinamento** antes e depois da aplicação de uma refatoração que extraia estas quebras. Isto é, considerando que existe uma ferramenta que **aponte** onde estes *Smells* estão localizados e que existem, também, outros utilitários que oferecem certos tipos de refatoração como *Move Method* e *Extract Method*, o que se busca neste trabalho é investigar uma forma viável de **realizar mudanças** que resolvam estas violações da Lei de *Demeter*. Somente desta forma será possível medir o grau de êxito destas refatorações.

Além disso, para medir a taxa de correção das quebras de confinamento, será necessário tomar nota de **quantas quebras de confinamento existem** antes de e após essas refatorações. De modo a satisfazer essa necessidade, serão utilizados o *Enclose* e o *MCP*, que podem fornecer essa informação nos projetos nos quais são utilizados. Também será necessário **selecionar estes projetos** que serão sujeitos ao teste de extração de suas quebras.

Como objetivo específico, verificou-se então a necessidade de **construção de uma ferramenta** que concretize estas refatorações já conhecidas por vários desenvolvedores, de modo a compartilhar este conhecimento consideravelmente benéfico com a diminuição do já discutido problema de grande acoplamento.

Outro objetivo específico básico é compreender o estado da arte das refatorações e de *Code Smells*, como a quebra de confinamento. A fim de que isso seja possível, **realizar-se-ão fichamentos** de artigos relevantes e junto disso, serão estudadas ferramentas que se voltam para a refatoração de códigos a exemplo de *JMove*[4] e

[4] https://github.com/aserg-ufmg/jmove

AntiCopyPaster[5].

1.4 SOLUÇÃO

Neste trabalho foram feitas análises sobre os **índices de quebra de confinamento** no código-fonte de 13 projetos de código aberto, antes de e após aplicar refatoração nesses códigos. Para isso, foi desenvolvido o utilitário conhecido como *Enclose*, ferramenta desenvolvida para extrair esses *Smells*.

A solução proposta equivale a duas *RQs (Research Questions)* para compreender as quebras de confinamento quando são expostas às transformações da ferramenta *Enclose*. Sendo uma delas **Qual a taxa de correção de quebras de confinamento em projetos acadêmicos?**. A segunda questão de pesquisa (*RQ*) é **Qual a taxa de corretude na refatoração das quebras de confinamento?**;

Porém, antes mesmo de implementar a refatoração das quebras, o trabalho tem início no **estudo de *Code Smells*** e dos inúmeros artigos que giram em torno do tema. Isto inclui aqueles trabalhos cuja metodologia envolve ferramenta que refatora código. Exemplos de ferramentas observadas incluem o ***MetricsReloaded***[6] e ***AntiCopyPaster***; que, respectivamente, mostra métricas do projeto e extrai código duplicado. Os *softwares* mencionados estão disponíveis no *GitHub*.

Para responder às *RQs*, o trabalho conta com a **construção de uma ferramenta** para o *IntelliJ IDEA*, o *Enclose*. O *Enclose* tem a finalidade de sugerir substituição automática de trechos de código onde ocorrem quebras de confinamento. As expectativas são de que cada vez mais desenvolvedores tenham acesso à ferramenta e de que seja observada uma utilização mais rigorosa de **boas práticas** de manutenibilidade de código, de modo que seja impulsionada a **qualidade** do projeto em desenvolvimento pelos mesmos.

1.5 RESULTADOS

Ao longo dos **fichamentos** de trabalhos relacionados, é possível concluir que códigos com métodos estrategicamente definidos certamente proporcionam projetos melhores de se manter. Se isso é verdade no cenário acadêmico, é ainda mais notável para possíveis empresas que lidam com grandes soluções.

[5] https://github.com/JetBrains-Research/anti-copy-paster
[6] https://github.com/BasLeijdekkers/MetricsReloaded

Ao se executar a avaliação da solução proposta, houve uma consideração sobre os tipos de resultados obtidos. A ferramenta *Enclose*, capaz de transformar o código com quebras de confinamento, foi executada em treze projetos *open-source* com o propósito de ter suas refatorações analisadas.

Os resultados mais importantes incluem mudanças com **cem porcento de corretude** em sete projetos acadêmicos, projetos dos quais houve quatro com chamadas acopladas a **expressões de referência**. Além disso, houve dois projetos com **mais refatorações do que quebras** entregues pelo *MCP* e foram criados quinze métodos com o **mesmo retorno**. Outro resultado importante foi o de não-refatoração por **aninhamento de classes**.

Estes foram os resultados observados nos projetos acadêmicos. Já no projeto *JMeter*[7] foram observadas **sessenta e duas vírgula cinco refatorações**, o que representa sete das dezesseis quebras detectadas pelo *Enclose* no projeto. Seis das outras nove quebras continham a **classe-alvo declarada no *JDK*** e houve três mudanças com conflito em **tipo de parâmetros**.

Até aqui foi apresentado breve exposição dos resultados, para maior detalhamento, visitar a seção Avaliação onde haverá com mais especifidade os resultados para cada projeto na amostra. Mas antes há na seção Fundamentação uma exploração dos conceitos chave para o trabalho.

[7] https://github.com/apache/jmeter

2 FUNDAMENTAÇÃO

Para esse estudo, alguns conceitos se destacaram como essenciais para a que o mesmo atingisse o seu propósito experimental. Uma destas ideias é a de **manutenibilidade de software**. A manutenibilidade é uma propriedade do produto que diz respeito a praticidade que ele oferece de evoluir e de se transformar, ou como também pode ser chamada, **flexibilidade**.

Quanto ao conceito de manutenibilidade, cerca de **67%** do custo total de um software corresponde aos custos dos esforços necessários para a realização de sua manutenção (MELLADO et al., 2015). Para reduzir este problema é que autores formularam ideias que ajudam a tornar o código mais coeso e com menos **acoplamento**, ou dependência entre suas partes, método através do qual é possível se ter um *software* com menos problemas de *design*.

Uma dessas heurísticas veio a ser conhecida como **Lei de *Demeter*** que propõe **reduzir o conhecimento** que um componente tem acerca de outro dentro do programa (MELLADO et al., 2015). Esta Lei oferece algumas sugestões para diminuir acoplamento entre as partes do código; dado um método `m` definido em `C`, em um cenário ideal `m` só pode acessar as **campos de C**, os **parâmetros de `m`**, ou ainda algum método de alguma **classe instanciada em `C`**, por exemplo. O algoritmo 1 é um exemplo onde essa Lei **não** é seguida, já que o método `m1` não apenas acessa o método `getElements()`, da outra classe com a qual se relaciona, além disto, ainda acessa o método `add()` em uma espécie de **encadeamento de chamadas**. Ainda vale ressaltar que esta Lei visa, junto com a diminuição de problemas relacionados à manutenção, proporcionar um aumento de qualidade de software, ou uma melhora no produto final que será o *software* criado pelo programador.

Algoritmo 1 – Classe C, cliente

```
1 public class C {
2     private A a;
3     public void m1(){
4         a.getElements().add(new A());
5     }
6 }
```

Fonte: Elaborado pelo autor

Para tentar reduzir os custos de manutenibilidade nos projetos de software, existem ferramentas que realizam **análise estática** do código-fonte. Estas são muito

úteis, uma vez que, dados os pontos do código que precisam de ajustes, ajudam os desenvolvedores a escrever o código com maior eficiência.

Uma dessas ferramentas é a *MetricsReloaded*, construída para **extrair métricas** em dado código fonte. Outra ferramenta que diz respeito é o ***MCP***, ferramenta resultado de uma pesquisa sobre quebras de confinamento em Java; é um notável exemplo de ferramenta que trabalha com a Lei de *Demeter*. O utilitário percorre, através de análise estática, um determinado código-fonte Java e em seguida extrai métricas, as quais incluem a quantidade de quebras de confinamento detectadas (PESSOA et al., 2019). Este termo, quebras de confinamento, será melhor desenvolvido em *Code Smells*, na fundamentação.

Outra ferramenta a se considerar é o ***AntiCopyPaster*** cuja funcionalidade principal é **verificar** onde um conjunto de linhas de código foi duplicado e aplica um conjunto de métricas nesta cópia; *a posteriori*, decide se a estratégia de refatoração de métodos chamada Extract Method se encaixa bem na situação que isso acontece. A refatoração *Extract Method* é uma forma de reescrever o código para evitar trechos repetidos. Isto também será esclarecido ao decorrer da fundamentação.

Estas ferramentas citadas oferecem funcionalidades que se unidas mostram uma capacidade de refatorar as quebras de confinamento. Mais a seguir pode-se conferir um aprofundamento no que seriam *Code Smells*, que ameaçam a manutenibilidade do software ao, por exemplo, aumentar acoplamento entre componentes como métodos.

2.1 *CODE SMELLS*

Ao escrever o código, pode ser notado haver algumas falhas na estruturação do mesmo, como partes escritas de uma forma difícil de ler. E é para isso que existe a chamada refatoração, que visa melhorar a qualidade do código (FOWLER et al., 2002). Uma forma básica de denominar refatoração é explicar que se trata de **reescrever o código** para melhorar sua leitura.

Estas partes mal escritas, também nomeadas como *Code Smells* já são conhecidas entre os estudiosos de engenharia de software e já são bem catalogadas neste meio. Existe por exemplo um estudo feito por Guggulothu e Moiz (2019) que se preocupa em **detectar** esses anti-padrões usando *Machine Learning* para tal.

Para nos familiarizarmos mais com estes conceitos, algumas classificações são descritas a seguir, a exemplo de código duplicado, descrito no algoritmo 2, e o

feature envy, algoritmo 3 (FOWLER et al., 2002).

Algoritmo 2 – Exemplo do *Smell* de código duplicado

```
1 public class A {
2     private A a;
3     public void method() {
4         System.out.println("Ola, mundo!");
5         ...
6         System.out.println("Ola, mundo!");
7     }
8 }
```

Fonte: Elaborado pelo autor

O código duplicado é uma forma de *Code Smell* que pode ter sido detectado no artigo de Golubev e Bryksin (2021). Este estudo foi implementado para **detectar** clones em diferentes projetos, tanto a nível de arquivo quanto a nível de método. Esse *Smell* envolve trechos de **códigos iguais** ao longo do projeto. Se isso acontece, é quase certo que o código precisa ser mudado. Uma possível solução é aplicar a refatoração *Extract Method*, que será explicada mais adiante, na seção refatoração.

Já no tocante ao outro *Smell*, que também pode ser resolvido ao se aplicar refatoração, o *Feature Envy*, pode-se afirmar se trata de um método que se interessa mais por **outra classe, que não a que o contém** (algoritmo 3). Para esse possível problema, uma solução candidata envolve *Extract Method*, por exemplo, juntamente a *Move Method*.

Outros exemplos de *Bad Smells* que cobrem fortes **dependências** entre componentes do *software* podem fazer menção às ideias presentes no tema das *Message Chains*, tema da próxima seção, e que precisa de uma atenção maior já que têm a estrutura de uma **possível quebra de confinamento**.

2.1.1 *Message Chain*

Code Smells são divulgados na literatura como formas de se tornar sistemas orientados a objetos difíceis de se manter (ABBES et al., 2011), semelhantemente ao que ocorre com as quebras de confinamento.

Quebra de confinamento, termo cunhado por Pessoa et al. (2019), em seu trabalho, se refere a um encademanto de chamadas *(Message Chain)* na qual se observa estarem envolvidos um **método acessor** `getElements()` no algoritmo 1, que nos devolve um atributo privado de determinada classe, e em seguida, após ter

Algoritmo 3 – Exemplo do *Smell* de *feature envy*

```
1  public class A {
2      private C c;
3      public void myMethod() {
4          System.out.println("Ola, mundo!");
5      }
6      public void method() {
7          c.showMessage();
8      }
9      public void show() {
10         c.sayHello();
11     }
12     public void doSomething() {
13         c.queryData();
14     }
15     public void showHelloMessage() {
16         this.metodo();
17     }
18 }
```

Fonte: Elaborado pelo autor

acesso a esse campo, a chamada a um outro método, desta vez `add()`, **modifica o valor** deste atributo, alterando o estado da instância.

Existem diversos tipos de *Code Smells* e um exemplo a se destacar são as ***Message Chains***, elaborado no algoritmo 4. Essas por sua vez podem ser definidas como extensos encadeamentos de chamadas a métodos que aumentam acoplamento, ou como também pode ser chamado, **dependência**, entre componentes do código. Com isso, trazem consigo a questão de uma **manutenibilidade mais dificultosa** no projeto de *software*.

Algoritmo 4 – Código Exemplo com *Message Chain*

```
1  public class A {
2      private A a;
3      public void method() {
4          a.getB().getC();
5      }
6  }
```

Fonte: Elaborado pelo autor

De fato, foi descoberto no artigo de Palomba et al. (2018), (estudo realizado visando coletar dados acerca de quais *Smells* aconteciam mais em conjunto,) que o *Message Chain* era o *Smell* que mais aparecia em conjunto com outros *Smells*. Nos

resultados listou-se que aparecia **32%** das vezes com *Complex Class*, outros **25%** das vezes com *Refused Bequest*, **17%** com *Spaghetti Code* e **13%** com *Blob*, concluindo assim a **ameaça** desse *Smell*.

Finalmente, na seção das conclusões do artigo sobre co-ocorrências de *Smells*, foi referenciado que a remoção das co-ocorrências pode estar relacionada a alguma atividade de **refatoração** que remove os componentes (classes, métodos) que as contém. A refatoração é um processo essencial para a nossa proposta e será detalhada no tópico imediatamente a seguir.

2.2 REFATORAÇÃO

Um conceito intimamente relacionado à facilidade de manutenção que um código oferece é a refatoração. A refatoração é o processo de realizar alterações no código do programa sem alterar o comportamento do mesmo (FOWLER et al., 2002). Programadores de vários níveis técnicos utilizam este conhecimento para continuamente subirem de nível o seu projeto em termos de qualidade.

Um dos motivos pelos quais existem ferramentas de refatoração é o de que fazer mudanças no código custa muito, mesmo com ferramentas de testes em mãos (FOWLER et al., 2002). Se a refatoração for simplificada a ponto de se tornar um ato tão simples quanto o de mudar alguns aspectos da estrutura do código, ajustes podem ser feitos de forma similar a mudanças pequenas, porém, os efeitos serão **positivamente muito grandes**.

É ainda mais notável a importância desse conceito quando são publicados artigos como o de Silva et al. (2021). A razão de existir desse artigo é de que uma ferramenta que aponte, entre duas versões de código-fonte existentes, quais as refatorações presentes é muito útil para ensinar a programadores mais inexperientes novas técnicas de refatoração de código. Assim como o artigo de Tsantalis et al. (2022), que busca para o mesmo fim, fazer **análises** no código-fonte.

Tomando como base a refatoração ***extract method***, também pode-se relacionar o artigo de Henrique et al. (2021), estudo realizado para coletar dados que indiquem as motivações por trás da aplicação desta refatoração pelos desenvolvedores em mensagens de *commit*. Esse tipo de refatoração se refere àquela em que é criado um novo método para **extrair** trechos do códigos que se repetem, por exemplo.

Já com relação a quando é indicado **aplicar** este tipo de reescrita do código, pode-se afirmar como resposta os seguintes requisitos: O programa tem **difícil leitura**

na sua forma atual; ou a sua estrutura compromete a facilidade com que **eventuais alterações** sejam feitas. Ainda é interessante afirmar, quanto a este conceito, que o mesmo pode ser classificado em **manual** quando o programador não lança-mão de nenhum outro software para este fim; ou ainda, **semi-automático ou automático**, quando um programa serve de apoio para que que a refatoração seja feita.

Ainda considerando-se *extract method*, tem-se um belo exemplo de como a automatização de uma refatoração pode ser vantajosa. Quando a *extract method* é realizada, o programador precisa verificar se ela pode ser aplicada ao seu caso em específico; depois é necessário tomar nota de todos os **tipos dos parâmetros** que vão estar no método criado; e além destes tipos, também é necessário conhecer o **tipo que o novo método tem que retornar**; sem mencionar que o desenvolvedor terá, nesse caso, que escrever desnecessariamente a **estrutura padrão** dos métodos de determinada linguagem. Diferentemente de uma refatoração automática com uma ferramenta, cenário em que a verificação é feita pela própria tecnologia, só restando ao usuário ter que definir **os nomes** dos novos componentes do método criado.

Outro tipo de refatoração existente a se considerar é a refatoção mover método (do inglês ***Move Method Refactoring***), que como o próprio nome diz, se preocupa em mover métodos de um lugar do código para outro. Quanto a *Move Method Refactoring*, artigos já foram escritos no que tocam à possibilidade de ser propício sugerir a mudança.

Um exemplo destes foi o artigo escrito por Terra et al. (2017), no qual se propõe comparar similaridade entre conjunto de dependências importadas entre métodos de cada classe. Vários algoritmos de similaridade foram utilizados no artigo, que teve como resultado um recall bastante grande (79%) quando não era necessário detectar a classe-alvo mas só o método. Também pode-se concluir que o JMove, ferramenta resultado do artigo e que é capaz de **mover os métodos** entre classes, era melhor recomendada para detecção de métodos maiores, conclusão essa retirada de uma comparação entre a ferramenta e mais duas outras: JDeodorant e Methodbook.

Estes dois tipos de refatoração: *Extract Method* e *Move Method* serão considerados para o desenvolvimento de nosso experimento, uma vez que o nosso objetivo principal é propor a **criação de um novo método** em uma certa classe em Java para realizar a alteração da estrutura de dados definida neste mesmo local do código.

Para medir a utilidade de uma ferramenta que aplique estes tipos de refatoração, ainda é importante saber que existem os critérios prático e também o técnico (FOWLER et al., 2002). O Critério **técnico** diz respeito à capacidade de a ferramenta alterar o

código preservando seu comportamento original, para isso, um banco de dados ou repositório apoiado por *parse trees* se faz necessário.

Uma *parse tree* é uma árvore criada a partir do processamento de um texto plano e com a finalidade de coletar os elementos do código que serão importantes para a refatoração. Com essa abordagem é possível acessar locais possivelmente comprometidos no código-fonte, no momento em que a refatoração for executada.

Além disso, também pode-se considerar a precisão do utilitário. Afirma-se que não é possível uma mudança no código não deformar nem que seja um pouco o comportamento anterior à transformação, já que, embora o programa possa até passar nos testes, até o tempo de resposta do programa pode diferenciar-se alguns milissegundos em comparação com o seu formato anterior à alteração.

Outro notável critério é o **prático**. Este confirma a velocidade e a acessibilidade de uma ferramenta que proporcione refatorar o código, afirmando que esta tem que oferecer velocidade ao usuário. Além disso, o usuário também necessita ter em mãos a capacidade de se desfazer da mudança causada pelo uso da ferramenta.

Relatados os aspectos inerentes à refatoração, a seguir está descrita uma tecnologia usada para realizar alteração nas quebras de confinamento para melhorar a qualidade do código buscando não mudar o seu comportamento, que seria o *SDK* do *IntelliJ*.

2.3 *INTELLIJ PLATFORM SDK*

O ambiente em que comumente vê-se código ser desenvolvido tem por nome *IDE (Integrated Development Enviroment)*. Existem vários *IDE*s, mas é interessante ressaltar o *JetBrains IntelliJ Idea*. Um *SDK*, sendo um conglomerado de ferramentas, é um conceito importante neste documento. O **Kit de Desenvolvimento de Software do *IDE IntelliJ***, escrito na linguagem Java, permite a expansão de funcionalidades através de *plugin*s.

Para proporcionar a realização de uma refatoração onde ocorrem as quebras de confinamento, sugeriu-se o desenvolvimento de um *plugin* para o produto da *Jet Brains*. Para isso, necessitou-se primo compreender o funcionamento da estrutura dos programas abertos.

A ***PSI (Program Structure Interface)***, ou interface da estrutura do programa, é o nome dado a um pacote de Classes e métodos para tratamento e substituição de

elementos, como *tokens*, na árvore de expressões do arquivo aberto. Alguns destas classes incluem:

- ***PsiElement***: Fornece um conjunto de métodos como `getText()`, que retorna uma *String* com o nome do elemento na árvore, ou a exemplo de `getFirstChild()`, cujo retorno representa, caso exista, o primeiro filho do elemento. Assim, pode ser visto com uma generalização de vários tipos de elementos *PSI*, descritos a seguir;
- ***PsiExpression*** e suas heranças: representam uma expressão na árvore *PSI*, que pode ser um aninhamento de chamadas de métodos ao estilo `a.call1().call2()`, assim como também pode vir na forma de uma expressão de referência, que não é uma chamada de método, como a expressão `a` do encadeamento;
- ***PsiStatement* e suas especificações**: geralmente um *return statement* com um *value*, ou até mesmo um *for statement* ou um *if statement*. Alguns *statements* possuem corpo, como é o caso de um *if* ou *for*, porém outros não como assim visto em um *return*.

Antes mesmo de manipular a *PSI*, uma possível maneira de criar um *plugin* para o *IntelliJ*, é por meio do *Gradle Intellij Plugin*[1]. É uma opção útil uma vez que oferece operações de rodar a *IDE* com o *plugin* em desenvolvimento ou publicar o *plugin*.

Depois da criação do projeto do *plugin*, a **extensão das funcionalidades** se dá registrando novos pontos de extensão como, por exemplo, uma nova ação de intenção, no caso do *Enclose*. A ação de intenção aparece no menu de contexto ao clicar com o botão direito do mouse no código.

No *IntelliJ*, uma ação de intenção é representada pela interface `IntentionAction`, a qual contém para serem implementados os métodos `invoke` e `isAvailable`. O primeiro faz tudo quando a ação é executada, já o segundo é capaz de verificar se a ação deve estar disponível no contexto. É através desses métodos que se tem acesso ao elemento *Psi* sob o cursor no momento que o usuário clica com o botão direito no código.

Utilizando a manutenção da árvore estrutural de arquivos, será possível realizar a refatoração ***extract method*** para **quebras de confinamento** nos mesmos, aplicando assim, a Lei de *Demeter* nos casos necessários.

[1] https://github.com/JetBrains/gradle-intellij-plugin

3 FERRAMENTA DESENVOLVIDA

A ferramenta resultante deste estudo - a chamada *Enclose* - verifica-se como um *plugin* para a *IDE IntelliJ* que serve, como principal funcionalidade, para realizar a refatoração *extract method* onde são encontradas quebras de confinamento.

Existe um conceito constantemente mencionado a partir deste ponto e este se trata de ***Call***. Uma *Call* é um modelo de chamada que representa uma quebra de confinamento. Ela aparece na forma <classe-alvo, método-alvo, tipo de retorno do método-alvo, classe-cliente, método-cliente, tipo de retorno do método-cliente, método da coleção> e foi inicialmente introduzido no trabalho de (PESSOA et al., 2019). Tomando como base o algoritmo 1 da seção Fundamentação, alguns dos elementos de uma Call são esclarecidos a seguir:

- **método da coleção** - é como é chamado o método que realiza a alteração do atributo exposto pelo método acessor `getElements()`, neste caso, o método `add()`;
- **método-cliente** - forma pela qual é chamado o método que contém o método da coleção, ou seja, `m1`;
- **classe-cliente** - é a classe em que o método-cliente está localizado - `C`;
- **método-alvo** - este nada mais é do que o método que nos expõe o atributo privado da classe `A`, o método `getElements()`;
- **classe-alvo** - é a classe com o atributo privado, neste caso é a classe `A`.

Inicialmente, é gerada uma lista de *Calls*, pelo *Enclose*; em seguida, o *plugin* é acionado quando o cursor se encontra sobre a chamada que causa a quebra. Nesse momento, é oferecida a sugestão "*Refactor Law of Demeter violation*". A partir daí, a ferramenta realizará o *extract method refactoring* (refatoração *extract method*), caso o usuário aceite.

3.1 VERSÃO ESTÁTICA

Ao partir-se para a implementação do utilitário em questão, houve a necessidade de primeiro compreender a dinâmica da interação da *PSI Tree* com um pré

determinado arquivo Java. O resultado disto foi a própria ferramenta em sua versão estática.

Considerando as duas classes presentes no algoritmo 5:

Algoritmo 5 – Classes A, alvo e C, cliente

```
1
2  import java.util.List;
3  public class A {
4      private List<A> elements;
5      public List<A> getElements() {
6          return this.elements;
7      }
8  }
9
10 public class C {
11     private A a;
12     public void m1(){
13         a.getElements().add(new A());
14     }
15 }
```

Fonte: Elaborado pelo autor

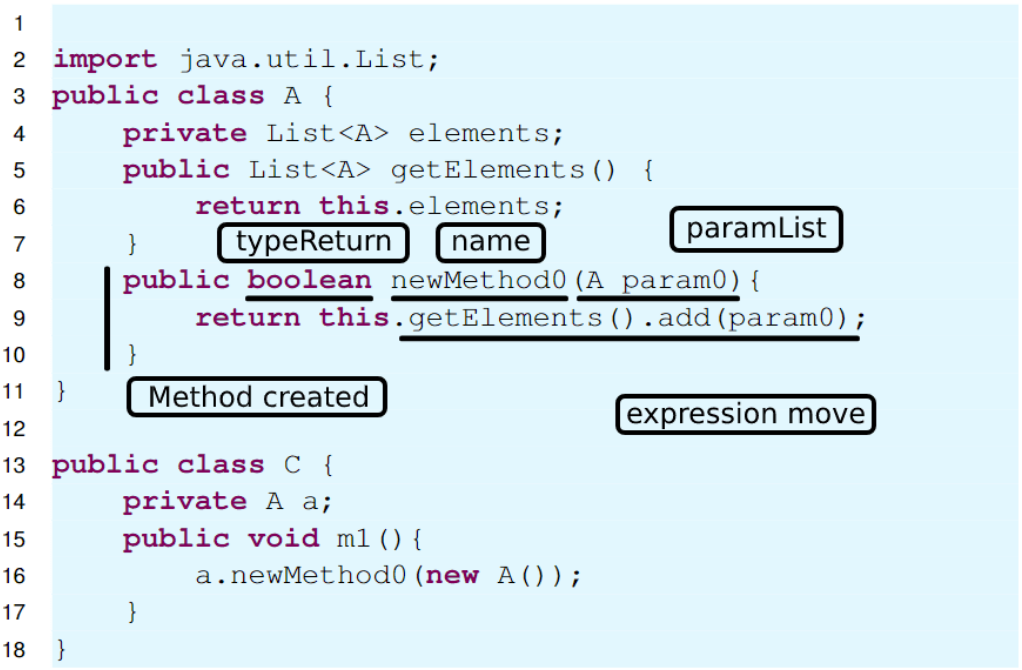
A primeira coisa a se perceber é a quebra de confinamento ocorrente em `m1`. O que acontece é que o os elementos de `A` são um atributo privado da mesma classe, logo, não deveria ser alterado em um lugar de outra classe, como acontece no cenário apresentado em `C`.

Antes de partir para o passo-a-passo, ressalta-se que o objetivo aqui é **extrair em um novo método em `A`**, a expressão que quebra o confinamento, e esse novo método será responsável estritamente por modificar o atributo em questão da mesma classe.

Vamos nomear os elementos presentes no passo a passo da figura 1 da seguinte forma:

Method created vai ser o nome dado a `public typeReturn name (paramList)`, que é o novo método em `A`, e ***expression move*** é como se chama a expressão `getElements().add()`, presente no código acima;

No *method created*, alguns elementos a se nomear são o tipo de retorno, que se chamará ***typeReturn***; **name**, que significa o nome dado ao novo método; e finalmente, **paramList**, que é o conjunto de parâmetros que o método vai receber como




```
1
2  import java.util.List;
3  public class A {
4      private List<A> elements;
5      public List<A> getElements() {
6          return this.elements;
7      }
8      public boolean newMethod0(A param0){
9          return this.getElements().add(param0);
10     }
11 }
12
13 public class C {
14     private A a;
15     public void m1(){
16         a.newMethod0(new A());
17     }
18 }
```

Figura 1 – Os elementos envolvidos na extração de uma quebra de confinamento

entrada. O passo a passo do algoritmo se reduz às etapas caracterizadas abaixo:

1. Vão ser armazenados os argumentos passados para *expression move* em *paramList* e *typeReturn* vai receber o tipo de retorno da expressão *expression move*;

2. Caso o *typeReturn* não corresponda a *void*, é criado um *return statement* e para esse se passa *expression move* como *return value*;

    a) Logo após, se anexa a *method created* o *return statement*. Nota-se que caso o *typeReturn* assuma o valor de *void*, o procedimento não entra nesta etapa. Ao invés disso, se anexa diretamente *expression move* como um *statement* no *method created*;

3. Em um último movimento, o algoritmo define em `A` o novo método *method created*. Dessa forma, só resta ao algoritmo criar uma nova chamada de método em `C` que substitua a *expression move*, e que, desta forma, aplique definitivamente a refatoração nas duas classes conectadas através da chamada.

Explicado o passo a passo para esse único formato de entrada, o das classes `A` e `C`, agora partiremos para as mecânicas que envolvem assumir com maior **variedade** os códigos que contêm quebras de confinamento.

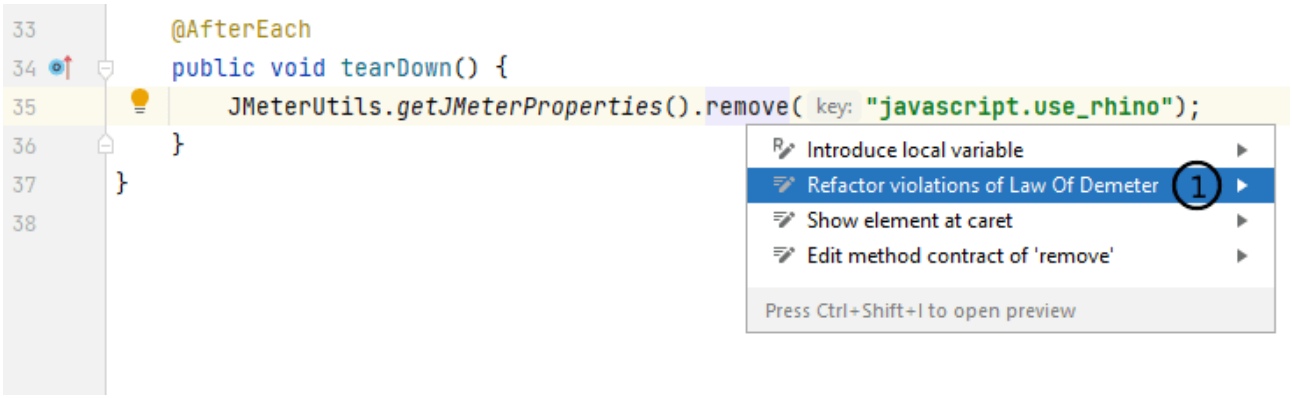


Figura 2 – A Ação de Intenção do Enclose

3.2 VERSÃO DINÂMICA

Em um conjunto variado de códigos-fontes Java, não se pode definir por antecedência o tipo de retorno de *expression move*, e nem que os argumentos da mesma serão sempre `new A()`, por exemplo.

Neste ponto pode-se afirmar que este é o verdadeiro diferencial da versão dinâmica para a versão estática. Assumindo uma variedade maior de entradas foi possível tornar a ferramenta mais funcional.

A **Interface de Usuário** é uma questão discutida a partir daqui. A ilustração da figura 2 segue como introdução ao tema.

Atentanto-se à figura 2 é possível visualizar o único item numerado, o qual se trata da intenção em si a qual pode ser localizada no menu de contexto ao clicar com o botão direito do mouse com o cursor acima da chamada selecionada, e em seguida selecionar **visualizar ações de contexto**. A mesma depois de ativa por clique, vai estar realizando a mudança ou extração da chamada selecionada.

A teoria construída até aqui trata da capacidade da ferramenta Enclose de mudar o código; no tópico em seguida o ponto principal a ser discutido é a **sumarização de chamadas** do projeto.

3.3 LISTAGEM DE *CALLS*

Para complementar o funcionamento da ferramenta *Enclose*, o objetivo de implementar uma listagem de *Calls* foi estabelecido. Disto resultou uma nova **janela de ferramentas**, ilustrada na figura 3. Uma janela de ferramentas no *IntelliJ* nada mais é do que um painel que apresenta informações acerca do projeto aberto na *IDE*.

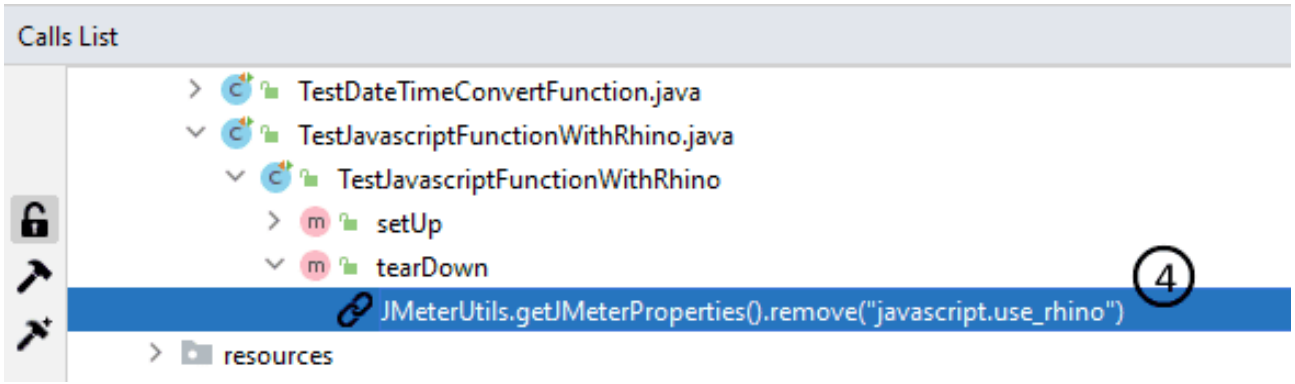


Figura 3 – A Janela de Ferramentas gerada pelo Enclose

A figura 3 exibe o que seria a janela de ferramentas do *Enclose*. Ela serve para localizar as chamadas de métodos que quebram ou não confinamento (a funcionalidade de filtragem de quebras será discutida mais adiante) com um formato de árvore cuja raiz é a pasta do projeto aberto na *IDE*. Também oferece algumas funcionalidades através dos botões à sua esquerda também discutidos mais na frente.

Para contextualizar melhor, o número 4 faz referência ao item na árvore que corresponde a uma chamada de método, entre várias outras chamadas listadas na janela de ferramentas constante na figura. Inclusive, ao clicar na chamada é possível ser levado diretamente ao lugar no código que a chamada na árvore está apontando.

Além da árvore mencionada, o grupo de ações presentes na janela são discutidos a partir daqui. Para isto, primeiro deve-se atentar à figura 4.

A numeração nela presente pode ser explanada de maneira que: o item 1 fique sendo um botão com a principal funcionalidade de ativar ou desabilitar a **filtragem das chamadas** capturadas neste modelo de árvore, de tal forma que o mesmo apenas mostre chamadas que quebrem confinamento, por isso o símbolo de cadeado; o item 2 recebe a possibilidade de **realizar a mudança** correspondente ao item selecionado na árvore; e o item 3, com a tarefa de aplicar esta **mudança para cada item**, ou chamada, elencado na estrutura.

De forma semelhante ao que a janela de ferramentas faz, foi implementado na Ferramenta, uma ação de listar *Calls*, a qual abre um **diálogo** no *IntelliJ Idea* com as chamadas encontradas no projeto selecionado e informando quais destas quebram confinamento. Conforme ilustrado na figura 5.

Para utilizar esta funcionalidade, o usuário precisa estar clicando em **Tools > Listar Chamadas** no menu superior da *IDE*. Após isto, é possível ter acesso a uma janela de diálogo informando uma listagem de *Calls* que remete ao formato da

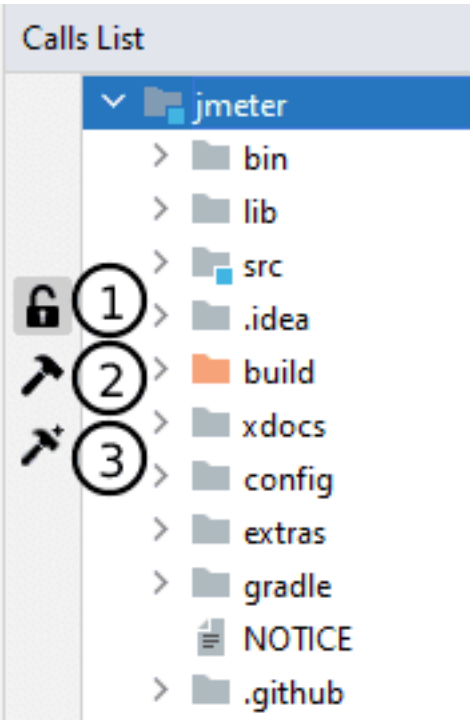


Figura 4 – Opções da Janela de Ferramentas gerada pelo Enclose

ferramenta de (PESSOA et al., 2019).

Novamente, isto foi feito para o usuário poder estar vendo e acessando de forma prática suas chamadas do projeto. Além disso, voltando à janela de ferramentas, também é possível mostrar apenas aquelas que quebram confinamento. Isto só foi possível graças ao implementado algoritmo de filtragem de *Calls*, o qual se baseia no algoritmo implementado em (PESSOA et al., 2019) e que se baseia em uma classificação de quais métodos alteram ou não o estado de uma Lista, por exemplo.

No tópico a seguir será apresentado mais sobre a avaliação da presente proposta do trabalho, a qual teve como auxílio esse suporte que a ferramenta oferece ao localizar as mudanças possíveis e ao aplicar estas transformações em cada projeto participante do experimento que será analisado na citada seção.

3.4 FUNCIONALIDADE DA LISTAGEM DE CALLS

Em caracterização, as chamadas são bem diversas, podendo partir das mais simples, como chamadas a métodos declarados na própria classe em que são invocados (caso este em que as chamadas podem não necessariamente possuir qualificador); como até mesmo um encadeamento bem longo, ou *message chain*, com a presença, neste caso, necessária de expressão qualificadora. Com qualificador ou expressão qualificadora, o trabalho faz menção a toda uma expressão que segue a chamada do

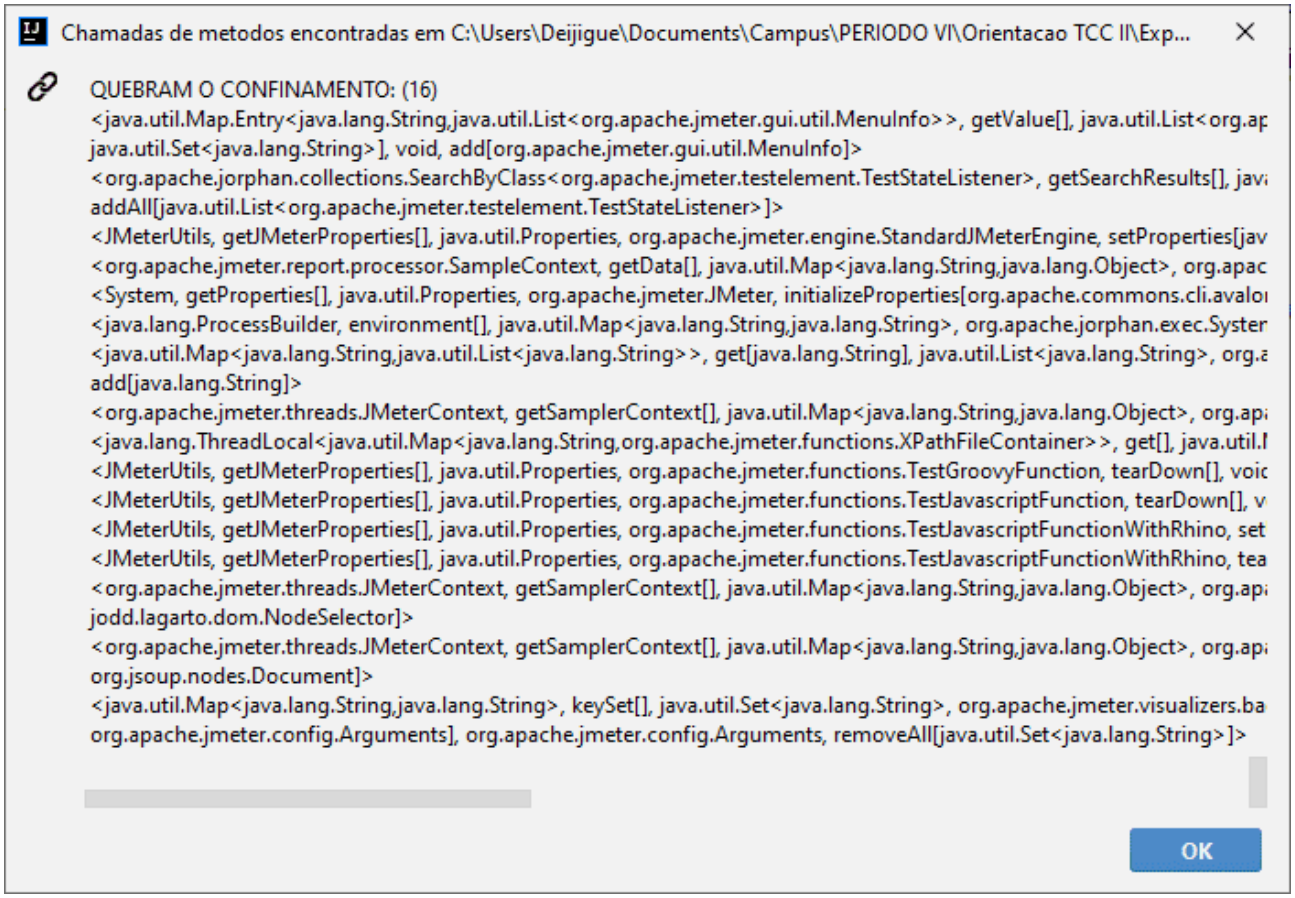


Figura 5 – Figura apresentando o diálogo com *Calls*

método até o `.` que precede o nome do método invocado. Estes exemplos estão desenhados no algoritmo 6, nas linhas 4 e 10, respectivamente. Deve-se atentar ao fato de o método `exibir()` da linha 4 no citado algoritmo não possuir o `.`, diferentemente da chamada `a.getElements().forEach(System.out::println)`, na qual a chamada ao método `forEach()` é qualificada pela expressão `a.getElements()`.

Para um maior detalhamento destas *Calls*, conferir o **apêndice A - *Calls* extraídas por *MCP* e *Enclose***.

Chamadas de métodos encadeadas: um exemplo de chamadas de métodos encadeadas está no projeto zephyr, retratada no algoritmo 7 nas linhas 9 e 13 por exemplo, e tanto o *MCP* (PESSOA et al., 2019), quanto o *Enclose* identificam normalmente acoplamentos entre chamadas de métodos.

Chamadas não qualificadas por outras chamadas: Este tipo de chamada de métodos compreende aquelas qualificadas por campos da classe em que estão declaradas ou variáveis do escopo, por exemplo.

O *MCP* (PESSOA et al., 2019) registra estas chamadas com os valores espe-

Algoritmo 6 – Código Exemplificando chamadas

```
1  class C {
2      private A a;
3      public metodo() {
4          exibir();
5      }
6      public void exibir() {
7          System.out.println( Hello , World! );
8      }
9      public void imprimirA() {
10         a.getElements().forEach(System.out::println);
11     }
12 }
```

Fonte: Elaborado pelo autor

Algoritmo 7 – Código Exemplo com acoplamentos entre chamadas

```
1
2      public List<Emprestimo> procurarUsuario(String palavra) throws
           IOException, ClassNotFoundException {
3          List<Emprestimo> respos = new ArrayList<>();
4          List<Aluno> alunos = bibliotecaDAO.getAnulos();
5          List<Funcionario> funcionarios = bibliotecaDAO.
               getFuncionarios();
6          List<Emprestimo> emprestimos = new ArrayList<>();
7
8          for (Aluno i : alunos) {
9              emprestimos.addAll(i.getEmprestimo());
10         }
11
12         for (Funcionario i : funcionarios) {
13             emprestimos.addAll(i.getEmprestimo());
14         }
15
16         for (Emprestimo i : emprestimos) {
17             if (i.getUsuario().getCPF().equals(palavra) || i.
                   getUsuario().getMatricula().equals(palavra)) {
18                 respos.add(i);
19             }
20         }
21
22         return respos;
23     }
```

Fonte: retirado de projeto zephyr

rados. Já o *Enclose* coleta o qualificador como nulo. O que pode ser justificado pela máxima de que para quebras de confinamento ocorrerem, o encadeamento precisa partir de uma chamada de método acessor, pois os campos modificados precisam ter visibilidade restrita em relação a quem realiza o acesso a eles. Ver o algoritmo 8 (linha 18).

Algoritmo 8 – Código Exemplo com qualificador sendo expressões de referência

```
1
2     public List<Emprestimo> procurarUsuario(String palavra) throws
          IOException, ClassNotFoundException {
3         List<Emprestimo> respos = new ArrayList<>();
4         List<Aluno> alunos = bibliotecaDAO.getAnulos();
5         List<Funcionario> funcionarios = bibliotecaDAO.
              getFuncionarios();
6         List<Emprestimo> emprestimos = new ArrayList<>();
7
8         for (Aluno i : alunos) {
9             emprestimos.addAll(i.getEmprestimo());
10        }
11
12        for (Funcionario i : funcionarios) {
13            emprestimos.addAll(i.getEmprestimo());
14        }
15
16        for (Emprestimo i : emprestimos) {
17            if (i.getUsuario().getCPF().equals(palavra) || i.
                  getUsuario().getMatricula().equals(palavra)) {
18                respos.add(i);
19            }
20        }
21
22        return respos;
23    }
```

Fonte: retirado do projeto poo-biblioteca

Mais chamadas suspeitas: Com relação ao número total de quebras detectadas, há alguns fatores observados que explicam a razão pela qual o número obtido pelo *Enclose* é maior do que o que foi feito pelo *MCP* (PESSOA et al., 2019). A versão do *MCP* (PESSOA et al., 2019) utilizada, não faz registro de chamadas de métodos da interface *Map*, coisa observada como contrária ao *Enclose*. Exemplo na linha 13:

Múltiplas chamadas em um mesmo corpo de método: Outro número identificado como diferente entre as duas abordagens aqui analisadas foi o de quebras de confinamento existentes em um mesmo método. Neste aspecto, a versão do *MCP* utilizada só salva um registro dessas ocorrências (possivelmente apenas a primeira).

Algoritmo 9 – Código Exemplo com chamadas de *Map*

```
1  @Override
2      public void addEvent(String eventName, String artistName, String
          description, LocalDate date, int avaiableTickets, int price)
          throws InvalidPrivilegeException, EventAlreadyExistsException,
           ArtistNotFoundException {
3          if (!(currentUser instanceof Admin))
4              throw new InvalidPrivilegeException();
5          if(!artists.containsKey(artistName))
6              throw new ArtistNotFoundException(artistName);
7          if(events.containsKey(date) && events.get(date).containsKey(
              eventName))
8              throw new EventAlreadyExistsException();
9          Artist artist = artists.get(artistName);
10         Event e = new ConcertClass(eventName, artist, date,
              description, avaiableTickets, price);
11         if(!events.containsKey(date))
12             events.put(date, new HashMap<String, Event>());
13         events.get(date).put(eventName, e);
14         String[] artistNameArray = {artistName};
15         addEventToCollections(e, artistNameArray, EVENT_TYPE.CONCERT)
              ;
16     }
```

Fonte: retirado de POOFindConcertTicket

Em contrapartida, a metodologia exposta neste trabalho apresentou uma extração de múltiplas chamadas em um mesmo método.

Este exemplo faz relação ao algoritmo seguinte. Nas linhas 4 e 5 há chamadas ao mesmo método *put* de *Map*. É notável a variedade de chamadas de métodos abordadas pela ferramenta e isso contribui, no final, para uma detecção mais ampla de quebras de confinamento. Concluindo, o processo de verificação de quebra é o mesmo presente no trabalho *MCP* (PESSOA et al., 2019), fato que explica a considerável porcentagem de extrações equivalentes entre as duas ferramentas.

Algoritmo 10 – Código Exemplo com chamadas iguais em um mesmo corpo de método

```
1 private void addArtistToCollections(String name, Artist artist){
2         artists.put(name, artist);
3         artistEvents.put(name, new HashMap<EVENT_TYPE, OrderList<
             Event>>());
4         artistEvents.get(name).put(EVENT_TYPE.CONCERT, new
             OrderListClass<Event>(new DateComparator(),true));
5         artistEvents.get(name).put(EVENT_TYPE.FESTIVAL, new
             OrderListClass<Event>(new DateComparator(),true));
6     }
```

Fonte: retirado de POOFindConcertTicket

4 AVALIAÇÃO

Nesta seção o assunto abordado é a avaliação do trabalho, cuja concretização veio tendo por base a questão que será apresentada no tópico seguinte. Além desta, posteriormente serão apresentados, respectivamente, os **participantes do experimento**, a **instrumentação**, o **ferramental de suporte** e os **resultados junto à análise**.

4.1 QUESTÕES DE PESQUISA

A definição do escopo do experimento seguiu a abordagem **GQM** (*Goals, Questions and Metrics*) como definido por (BASILI et al., 1994). O objetivo desse experimento está descrito da seguinte forma:

> ***Analisar*** *as mudanças no código fonte com quebras de confinamento*
> ***Com a intenção*** *de avaliar a corretude*
> ***Com respeito*** *à qualidade do código*
> ***Do ponto de vista*** *do desenvolvedor*
> ***No contexto*** *de projetos acadêmicos que utilizam estruturas de dados definidas no JCF e possuem quebras de confinamento.*

Definido o objetivo da pesquisa, foi realizado o rastreamento entre ele e os dados que foram coletados por meio de exatamente duas perguntas, detalhadas a seguir.

Q1: Qual a taxa de correção de quebras de confinamento em projetos acadêmicos?

Buscando responder à primeira pergunta, foi extraída por meio da ferramenta *MCP* a métrica **M1: Quantidade de ocorrências de quebras de confinamento**, considerando uma classe que acessa diretamente métodos que modificam o estado de uma estrutura de dados pertencente a outra classe, acesso o qual se qualifica como quebra de confinamento. Já com relação à segunda pergunta, a mesma métrica foi extraída com o próprio *Enclose*, quando foi implementada a funcionalidade de listagem de *Calls*, conceito já explanado anteriormente como uma abstração das chamadas dos projetos.

Q2: Qual a taxa de corretude na refatoração das quebras de confinamento?

A partir deste recurso, pode-se nomear o novo conceito **taxa de correção**. A taxa de correção pode ser descrita como a proporção existente entre o total de mudanças realizadas nos objetos, com êxito, e o número total de mudanças realizadas em cada um dos mesmos. De modo semelhante, a **taxa de corretude** presente na questão *Q2* significa, dado um conjunto de refatorações realizadas nestas quebras, a contagem das que ocorreram com êxito em relação ao total de transformações.

Caracterizados os dados atinentes à realização do experimento, segue-se agora para a seleção dos participantes e em seguida à instrumentação do mesmo.

4.2 SELEÇÃO DOS PARTICIPANTES

A seguir estão relacionados os participantes eleitos para responder as preestabelecidas perguntas Q1 e Q2. Referindo-se aos doze participantes da Tabela 1, eles foram escolhidos porque são interessantes especificamente para este tipo de pesquisa, uma vez que foram declaradas quebras de confinamento nos mesmos pelo projeto *MCP*.

Tabela 1 – Projetos *open-source* acadêmicos selecionados de seus respectivos repositórios no Github

projetos	classes	LOC
Donnervoegel/java	69	10729
DaniloRaniery/Projeto	14	809
wensttay/poo-lanchonete	13	1696
LukasGrudtner/Yu-Gi-Oh	82	4813
gustavomeloGH/Projeto-POO-Livre-Leitura-Biblioteca	48	5592
namtan8888/Project	48	5759
wensttay/poo-biblioteca	29	7162
LiamDotPro/Access-Control-System	55	8529
prateek-khatri/OOADProject	20	2301
rstancioiu/Dev-OO	47	4747
alibaba/fastjson	2534	202021
json-iterator/java	214	21700

Para a segunda questão de pesquisa, há presente o projeto JMeter, participante sob experimentação para responder à segunda questão Q2. Diferentemente dos projetos acadêmicos o JMeter possui 1701 classes e 421467 linhas de código, neste projeto as quebras de confinamento foram detectadas pelo *Enclose*, graças à sua funcionalidade de listar chamadas e de filtragem das *Calls* do projeto.

Após detectadas estas ocorrências, a refatoração foi aplicada com o auxílio do Enclose em cada uma destas quebras, tanto nos projetos acadêmicos, quanto no projeto JMeter, e os resultados estão reunidos na seção de mesmo nome.

De acordo com o site oficial, o *JMeter* é uma aplicação feita puramente em Java altamente focada em testes de performance. Ela foi feita para avaliar o desempenho de Aplicações *Web*, porém acabou evoluindo e adaptou outras funcionalidades de testes.

4.3 INSTRUMENTAÇÃO

A realização dos experimentos ambientou-se no sistema operacional **windows 10.0.19042 Compilação 19042**, com o ***JDK* versão 11 *java version* 11.0.12 (*JRE* correspondente)**, nível de linguagem do projeto definido como **padrão do *SDK*** e tendo como *IDE* o ***IntelliJ Idea Community* 2020.1.3**. Ainda é necessário notar a necessidade de se ter um *JDK* configurado e próprio para cada um dos projetos sob teste.

4.4 FERRAMENTAL DE SUPORTE

Tendo em vista as necessidades apresentadas pela realização deste trabalho, desabrochou-se em forma de *plugin* o utilitário *Enclose* capaz de realizar as mudanças nos códigos fonte, tendo como finalidade, estas, a concretização de uma espécie de *extract method refactoring*, através da qual visou-se consertar as quebras anunciadas pelo projeto *open-source MCP*.

A fim de prover uma maior confiabilidade à metodologia utilizada, foi elaborada uma estratégia de avaliação que se embasa em um trabalho anterior. Neste contexto, foi utilizado o conjunto de resultados obtidos do *MCP* (PESSOA et al., 2019), o qual oferece uma listagem com chamadas de métodos, retirada de 32 projetos *open-source*.

4.5 RESULTADOS E ANÁLISE

Q1: Qual a taxa de correção de quebras de confinamento em projetos acadêmicos?

A resposta à esta questão revelou algumas resultados cujas peculiaridades serviram-lhes de categorização.

Foram analisados doze projetos, com um total de 3797 classes e encontraram-

se os seguintes resultados: o primeiro destaca a taxa de correção de todas as quebras de confinamento transformadas; por sua vez, o segundo resultado apresenta situações em que não houve mudanças do código; o terceiro resultado mostra mais mudanças que quebras detectadas pelo *MCP*; o quarto, métodos com mesmo *return* e, por último, o quinto, no qual pode-se conferir que há um aninhamento de classes. Todos estes tipos de observações serão analisados imediatamente a seguir.

O primeiro resultado diz respeito aos casos de mudanças que ocorreram conforme o esperado. Essas mudanças contêm um método de coleção e um método acessor ligados por uma *message chain*. Esta estrutura está representada no algoritmo 1.

As mudanças que não apresentavam nenhum erro somadas às mudanças que não aconteceram são, inicialmente, a maioria, representando 11 do total de 12 projetos que serviram para análise. Destes, 4 exibiram-se como não aptos a mudança, ou seja, a chamada do método da coleção partia diretamente de uma expressão de referência. Para mais detalhes, constatar a tabela 2, a qual elenca estes 11 projetos citados.

Tabela 2 – As mudanças e suas taxas de corretude

projects	mudanças	sucessos
Donnervoegel/java	0	0
DaniloRaniery/Projeto	2	2
LukasGrudtner/Yu-Gi-Oh	0	0
gustavomeloGH/Projeto-POO-Livre-Leitura-Biblioteca	8	8
namtan8888/Project	0	0
wensttay/poo-biblioteca	2	2
LiamDotPro/Access-Control-System	2	2
prateek-khatri/OOADProject	1	1
rstancioiu/Dev-OO	6	6
alibaba/fastjson	10	10
json-iterator/java	0	0

A ferramenta *Enclose* em sua versão atual não é disponível para estes casos de quebras cujas chamadas de método partem de uma expressão de referência. Este caso em que o *collection method* é chamada a partir de expressão de referência está especificado a seguir.

Houve quatro casos em que a ferramenta não refatorou e esta categoria de resultado está explicada a seguir e é exibido na tabela 3. Quando não há uma *message chain* com o método da coleção, isto é, o método da coleção está ligado a uma expressão de referência, o *Enclose* não faz mudanças. Expressão de referência pode ser uma variável ou campo de classe.

Tabela 3 – Projetos aos quais as mudanças mostraram-se indisponíveis

projects	quebras conf.	mudanças
Donnervoegel/java	4	0
LukasGrudtner/Yu-Gi-Oh	1	0
namtan8888/Project	2	0
json-iterator/java	1	0

Com relação aos casos de não mudanças, cem por cento destes se referem aos casos em que as chamadas partem de expressões de referência. O trecho de código do algoritmo 11 ilustra bem o citado cenário.

Algoritmo 11 – Código Exemplo com expressões de referência

```
1  public class C {
2    public void m1() {
3      elements.add(new Class());
4    }
5    public void m2(List<Class> elements) {
6      list.add(new Class());
7    }
8    public void m3() {
9      List<Class> elements;
10     ...
11     list.add(new Class());
12   }
13 }
```

Fonte: Elaborado pelo autor

Analisando-se o código, nota-se o supracitado caso em que o método da coleção, neste caso o `add()`, tem sua chamada qualificada pelo que o *PSI* categoriza como expressão de referência. Em todos os três métodos do código exemplo, a chamada tem como qualificador este tipo de expressão e nestes três métodos, em sua versão atual, a ferramenta não pode ser observada funcionando.

Ao ser executado o experimento, foi notado haver em dois projetos uma quantidade maior de refatoração do que as quebras detectadas pelo *MCP*. Estes estão na tabela 4.

Tabela 4 – Quantidade de quebras e mudanças correspondentes aos projetos

projects	quebras conf.	mudanças
gustavomeloGH/Projeto-POO-Livre-Leitura-Biblioteca	3	8
rstancioiu/Dev-OO	3	6

No projeto denominado Projeto-POO-Livre-Leitura-Biblioteca, tinha-se três quebras acusadas pelo *MCP*. Porém, no momento de se realizar as mudanças, constatou-

se um montante de 8 linhas de código nas quais aplicou-se o *Enclose*.

Da afirmação anterior extraiu-se a conclusão de que o *MCP* só captura quebras de confinamento por origens diferentes. O que quer dizer que para cada conjunto método-cliente, classe-cliente e método da coleção, tem-se uma quebra coletada pelo *MCP*.

Algoritmo 12 – Código apresentando dois métodos de coleção de mesmo nome dentro de um único método *Java*

```
1 public void doSomething(int parameter1) {
2     // corpo do metodo omitido
3     a.getElements().add(new A());
4     a.getElements().add(new A());
5     // corpo do metodo omitido
6 }
```

Fonte: Elaborado pelo autor

O algoritmo 12 exibe dois métodos de coleção de mesmo nome dentro de um único método *Java*. O *MCP* não detectaria estas duas ocorrências, mas o *Enclose* refatoraria normalmente.

Este fato gerou a hipótese de que podería-se observar um número considerável de mudanças das quais se originariam *returns* iguais entre as mesmas, isto é, haveriam criações de métodos com mesma expressão de retorno. Este tipo de resultado tem uma relação intrínseca com o próximo item desta lista de resultados.

No momento do experimento, foram refatoradas quebras de confinamento que resultaram em métodos iguais, se forem descartadas diferenças no nome dos mesmos.

A tabela de número 5 representa um reflexo da ausência da funcionalidade "verificar se o *statement* extraído, ou chamada extraída, já existe"a qual poderá ser pensada em momentos futuros em próximas *releases*.

Tabela 5 – mudanças com êxito e quantidade de repetição de returns

projetos	# sem erro	# rep.	# máx. ret. iguais
DaniloRaniery/Projeto	2	1	2
wensttay/poo-lanchonete	5	3	2
gustavomeloGH/Projeto-POO-Livre-Leitura-Biblioteca	8	3	3
rstancioiu/Dev-OO	6	1	6

A coluna rep. representa o número das diferentes repetições de *return* observadas em um mesmo projeto depois de aplicar as mudanças do Enclose. Levando isto

em conta, a partir do trecho de código do algoritmo 13 pode ser extraída informação importante detalhada a seguir.

Algoritmo 13 – Código Exemplo de *returns* repetidos

```
1 public class A {
2     private A a;
3     public void newMethod1(int parameter1, A parameter2) {
4         return a.getElements().set(parameter1, parameter2);
5     }
6     public void newMethod2(int parameter1, A parameter2) {
7         return a.getElements().set(parameter1, parameter2);
8     }
9 }
```

Fonte: Elaborado pelo autor

Algo que se mostrou explícito nas linhas 4 e 7 do citado trecho de código é o fato de que a ferramenta, por não verificar se o *statement* extraído para o novo método é utilizado em outros trechos do código, acabe criando métodos novos a partir de um único *statement* que se repete.

Uma última categoria de resultado é descrita a seguir. Esta se refere a casos em que a classe-alvo não pode ser encontrada pela ferramenta por uma razão bastante específica.

Quando o desenvolvedor está criando uma nova classe, ele pode optar por declarar-la dentro de uma outra classe, a este conceito dá-se o nome de aninhamento de classes.

Durante a realização do experimento, houve apenas um projeto, (8.3% dos objetos de análise), o alibaba/fastjson, no qual duas de suas doze mudanças possíveis (16.6% destas) não puderam ser realizadas por conta do conceito de aninhamento de classes em Java. O que ocorreu é que a classe-alvo, por ser aninhada dentro de outra classe, não pode ser localizada como deveria, resultando em um erro na ferramenta. Este registro também deverá ser levado em consideração em uma futura versão da ferramenta.

Atentando-se ao algoritmo 14, é possível visualizar o que pode ser chamado de classes aninhadas. A classe `A` está localizada dentro do escopo da classe `X`, por esse motivo a ferramenta não refatorou no projeto.

Algoritmo 14 – Código Exemplo de classes aninhadas

```
1  public class X {
2    public class A {
3        // bloco de classe omitido
4    }
5  }
```

Fonte: Elaborado pelo autor

> ***Q1:*** *Foram realizadas mudanças na totalidade de corretude em sete projetos analisados. Dos projetos analisados, em quatro projetos tivemos chamadas por referências. Dentre os doze, em dois projetos tivemos mais de três mudanças além das quebras. Nas refatorações, foram criados quinze métodos com corpos iguais. Dos doze projetos, um projeto mostrou não-mudança por aninhamento de classes*

Q2: Qual a taxa de corretude na refatoração das quebras de confinamento?

Para esta questão de pesquisa foram analisadas 1701 classes do projeto JMeter. O projeto contém 421467 linhas de código e foi construído para avaliar o desempenho de Aplicações *Web*, porém acabou evoluindo e adaptou outras funcionalidades de testes.

Como principais resultados podemos destacar mudanças exitosas, o que foi sete das dezesseis quebras de confinamento detectadas, mudanças com problemas de tipos na criação do método na classe-alvo e a ausência de seis mudanças por classe-alvo ser do *JDK*.

Para avaliar a proposta do *Enclose* de realizar transformações em trechos do código que comprometem a manutenibilidade do mesmo, concretizou-se um teste na citada plataforma. Para a realização do mesmo, de modo inicial foram rastreadas as quebras de confinamento do mesmo com o auxílio da ferramenta apresentada neste trabalho, o *Enclose*. A seguir se apresentam as classes-cliente que contêm todas as 16 quebras detectadas no projeto.

- org.apache.jmeter.gui.util.MenuFactory
- org.apache.jmeter.engine.StandardJMeterEngine
- org.apache.jmeter.engine.StandardJMeterEngine

- org.apache.jmeter.report.processor.AbstractSampleConsumer
- org.apache.jmeter.JMeter
- org.apache.jorphan.exec.SystemCommand
- org.apache.jmeter.protocol.http.control.CacheManager
- org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl
- org.apache.jmeter.functions.XPathWrapper
- org.apache.jmeter.functions.TestGroovyFunction
- org.apache.jmeter.functions.TestJavascriptFunction
- org.apache.jmeter.functions.TestJavascriptFunctionWithRhino
- org.apache.jmeter.functions.TestJavascriptFunctionWithRhino
- org.apache.jmeter.extractor.JoddExtractor
- org.apache.jmeter.extractor.JSoupExtractor
- org.apache.jmeter.visualizers.backend.BackendListenerGui

Para melhor entendimento da natureza destas refatorações, neste teste, foram observados três tipos de resultados principais, descritos logo a seguir.

Primeiramente, será explicada a mudança com problema no tipo de parâmetro do método criado. Quando o tipo é totalmente qualificado, ou seja, há uma especificação de qual biblioteca contém a classe que representa aquele tipo.

Essa espécie de resultado foi aquela observada quando tentou-se reestruturar códigos e o resultado teve um tipo totalmente qualificado, assim exemplificado no algoritmo 15. Dos 16 resultados, esses representaram 3. As classes-cliente estão na lista a seguir.

- org.apache.jmeter.engine.StandardJMeterEngine
- org.apache.jmeter.extractor.JoddExtractor
- org.apache.jmeter.extractor.JSoupExtractor

Algoritmo 15 – Código Exemplo de tipos totalmente qualificados

```
1     public boolean newMethod1(List<org.apache.jmeter.testelement.
          TestStateListener> param0) {
2         return this.getSearchResults.addAll(param0);
3     }
```

Fonte: Elaborado pelo autor

Como pode ser visto no algoritmo 15, o `param0` possui um tipo que contém a localização completa na sua referência `org.apache.jmeter.testelement.TestStateListe`. Quando este método foi criado, houve um problema de compilação.

A partir daqui será discorrido sobre as quebras de confinamento que contém classe-alvo declarada no próprio *Java*, logo, não houve mudanças neste caso.

Este tipo de mudança não aconteceu, porque a classe-alvo detectada se localiza nas bibliotecas da própria linguagem *Java* como no algoritmo 16. As quebras estão localizadas nas classes elencadas na lista a seguir.

- org.apache.jmeter.gui.util.MenuFactory
- org.apache.jmeter.JMeter
- org.apache.jorphan.exec.SystemCommand
- org.apache.jmeter.protocol.http.control.CacheManager
- org.apache.jmeter.functions.XPathWrapper
- org.apache.jmeter.visualizers.backend.BackendListenerGui

Algoritmo 16 – Código Exemplo no qual classe-alvo é do próprio Java

```
1 public void doSomething() {
2     // corpo do metodo omitido
3     for (Map.Entry<String, List<A>> entry : a.entrySet() ) {
4         entry.getValue().add(new A());
5     }
6     // corpo do metodo omitido
7 }
```

Fonte: Elaborado pelo autor

Na linha 4 do algoritmo 16, pode-se constatar a utilização do método `add` de `List` e que a classe-alvo é do tipo `Map.Entry`, porém esta classe está declarada no *Java*, logo, não pode ser modificada.

A próxima categoria de resultados diz respeito às mudanças padrão realizadas no projeto *JMeter*.

A mudança padrão tem as seguintes características: a quebra é detectada em uma *message chain* e a classe-alvo não é declarada dentro do *JDK*. A categoria mudanças padrão trata das mudanças realizadas conforme o esperado. As classes nas quais as chamadas foram detectadas estão elencadas na seguinte lista. Esta mudança tem a estrutura de uma **quebra de confinamento básica** com método-alvo e classe-alvo que podem ser transformados conforme apresentado ainda no início do trabalho, mais precisamente no algoritmo 5, onde podem ser visualizadas tanto a classe que realiza a modificação do atributo privado (classe-cliente), como também a classe que devolve seu atributo (classe-alvo), com seu método acessor (método-alvo).

- org.apache.jmeter.engine.StandardJMeterEngine
- org.apache.jmeter.report.processor.AbstractSampleConsumer
- org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl
- org.apache.jmeter.functions.TestGroovyFunction
- org.apache.jmeter.functions.TestJavascriptFunction
- org.apache.jmeter.functions.TestJavascriptFunctionWithRhino
- org.apache.jmeter.functions.TestJavascriptFunctionWithRhino

Algoritmo 17 – Código Exemplo de uma mudança padrão no *JMeter*

```
1 public void setProperties(Properties p) {
2     // corpo do metodo omitido
3     JMeterUtils.getJMeterProperties().putAll(p);
4 }
```

Fonte: Elaborado pelo autor

No algoritmo 17, a linha 3 apresenta um exemplo de chamada que sofreu mudança padrão, ou seja, foi extraída e movida a expressão de quebra de confinamento e foi criado um método novo na classe-alvo.

Q2: *Das dezesseis mudanças possíveis, sete foram realizadas conforme o esperado. Em 3 destas dezesseis transformações possíveis tivemos conflito com tipo de parâmetro. Foram obtidas seis mudanças possíveis, porém que não ocorreram por classes-alvo declaradas no JDK*

5 CONSIDERAÇÕES FINAIS

Nesta seção pode-se dizer que a ferramenta em si, por mostrar uma boa taxa de corretude na amostra de projetos de código aberto estudadas, pode ser utilizada para corrigir as quebras de confinamento nas quais existe um método acessor para o atributo da coleção, na classe alvo. É perceptível também que a ferramenta só tem a se extender, e com isso, aumentar a taxa de corretude em versões futuras.

As principais contribuições do trabalho são o ***Enclose***, *plugin* para o *IntelliJ Idea*; reunião das ideias que englobam **refatoração de quebras de confinamento** na Fundamentação do trabalho e os **resultados** do experimento em treze projetos de código aberto.

A ferramenta cumpriu com o nosso objetivo principal: o de verificar e analisar a taxa de corretude de mudanças em quebras de confinamento. Além disto, por realizar as mudanças além de detectar as quebras, a ferramenta pode ser expandida para detectar quebras em outras *IDE*s, por exemplo. Adiciona-se também o aprendizado proporcionado ao realizar este TCC, o que inclui o cumprimento de prazos, e o funcionamento da linguagem Java.

Como contribuição pode-se destacar o experimento em doze projetos acadêmicos, buscando responder à questão Q1. Neste experimento foram obtidos cinco tipos de resultados. O primeiro diz respeito a observação de uma proporção de um para um entre mudanças realizadas e quebras de confinamento apontadas pelo *MCP* em sete projetos acadêmicos. O segundo tipo de resultado é aquele em que a chamada era composta por uma expressão de referência, neste caso a mudança não aconteceu em quatro projetos. Também houve o resultado, apresentado em dois projetos, em se observaram mais mudanças realizadas do que as mudanças possíveis do *MCP*, por conta de o *MCP* só registrar uma quebra de confinamento por conjunto método-cliente e método da coleção. Outra observação foi a presença de métodos que continham o mesmo *return*, não sendo estes iguais apenas pela mínima diferença no nome do método. Este tipo de resultado foi conferido em quatro projetos acadêmicos com um valor máximo de seis *returns* iguais entre si. Por último, na questão de pesquisa Q1, houve o resultado de aninhamento de classes, isto é, um projeto não teve mudanças por conta de a classe-alvo estar declarada no escopo de outra classe.

Outra contribuição foi a validação da ferramenta através da execução da mesma no projeto *open-source JMeter*. Buscando verificar qual a taxa de corretude das mudan-

ças no projeto (Q2), obtiveram-se três tipos de resultados. São estes sete mudanças esperadas, três com conflitos na tipagem do parâmetro do método criado e seis quebras não refatoradas por conter classe-alvo declarada no próprio *JDK*.

Como forma de responder às questões de pesquisa Q1 e Q2, uma contribuição importante é a ferramenta *Enclose*. O *Enclose* é capaz de fazer uma listagem com as chamadas de métodos presentes em um projeto do *IntelliJ Idea*. Estas chamadas, por sua vez, ficam presentes em uma árvore na janela de ferramentas do *Enclose*. Cada chamada da árvore pode ser refatorada individualmente ou, se o usuário preferir, pode-se refatorar todas as chamadas de uma só vez. Ainda há a funcionalidade de filtragem de chamadas de métodos a qual permite que o usuário deixe visíveis na árvore apenas aquelas que quebram confinamento. Isto o *Enclose* identifica através do sistema de classificação de *Calls* divulgado por Pessoa et al. (2019). Para utilizar o algoritmo de filtragem de *Calls* do *MCP*, a ferramenta *Enclose* também conta com a conversão de chamadas de métodos em *Calls* na estrutura <classe-alvo, método-alvo, tipo de retorno do método-alvo, classe-cliente, método-cliente, tipo de retorno do método-cliente, método da coleção>, explicada na seção da ferramenta desenvolvida.

Para trabalhos futuros, sugere-se expandir a abordagem para outras linguagens orientadas a objetos a exemplo de *C#*, pois o conceito da Lei de Demeter é independente da linguagem e vale desde que exista uma orientação a objetos. Além disto, o algoritmo tem a possibilidade de ser implementado em outras tecnologias, a exemplo de um *plugin* para a *IDE Eclipse*. Para mencionar tudo, o experimento também pode ser expandido e abraçar outros contextos de projetos, como mais projetos de outras empresas, que não a *Apache*, que é a atual responsável por manter o JMeter.

REFERÊNCIAS

ABBES, M.; KHOMH, F.; GUéHéNEUC, Y.-G.; ANTONIOL, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: **2011 15th European Conference on Software Maintenance and Reengineering**. [S.l.: s.n.], 2011. p. 181–190.

BASILI, V. R.; CALDIERA, G.; ROMBACH, D. The goal question metric approach. **Encyclopedia of Software Engineering**, v. 1, 01 1994.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, d. Refactoring: Improving the design of existing code. 2002.

GOLUBEV, Y.; BRYKSIN, T. On the nature of code cloning in open-source java projects. In: **2021 IEEE 15th International Workshop on Software Clones (IWSC)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 22–28. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/IWSC53727.2021.00010>.

GUGGULOTHU, T.; MOIZ, S. A. Detection of shotgun surgery and message chain code smells using machine learning techniques. **International Journal of Rough Sets and Data Analysis**, v. 6, p. 34–50, 04 2019.

HENRIQUE, J.; DóSEA, M.; SANT'ANNA, C. Minerando motivações para aplicação de extract method: Um estudo preliminar. In: **Anais do IX Workshop de Visualização, Evolução e Manutenção de Software**. Porto Alegre, RS, Brasil: SBC, 2021. p. 21–25. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/vem/article/view/17212>.

MELLADO, R. P.; MOREIRA, G. de S. P.; CUNHA, A. M. da; DIAS, L. A. V. A software framework for identifying the law of demeter violations. **2015 12th International Conference on Information Technology - New Generations**, Jun. 2015.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. A large-scale empirical study on the lifecycle of code smell co-occurrences. **Information and Software Technology**, v. 99, 02 2018.

PESSOA, J. P.; S., R. Job de; MOREIRA, D. D. Mcp - uma ferramenta para deteccao de quebra de confinamento no uso das estruturas de dados. **Anais**, p. 1276–1282, Nov. 2019.

SILVA, D.; SILVA, J. P. da; SANTOS, G.; TERRA, R.; VALENTE, M. T. Refdiff 2.0: A multi-language refactoring detection tool. **IEEE Transactions on Software Engineering**, v. 47, n. 12, p. 2786–2802, 2021.

TERRA, R.; VALENTE, M.; MIRANDA, S.; SALES, V. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. **Journal of Systems and Software**, v. 138, 12 2017.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. **IEEE Transactions on Software Engineering**, v. 48, n. 3, p. 930–950, 2022.

APÊNDICE A – *CALLS* EXTRAÍDAS POR *MCP* E *ENCLOSE*

Neste apêndice está registrada uma comparação para cada tipo de *Call* registrado para os projetos acadêmicos, retiradas com o auxílio da ferramenta *MCP* (PESSOA et al., 2019) e com o *Enclose*, ferramenta deste trabalho, respectivamente.

A seguir estão elencadas na seguinte ordem: ***calls* padrão; *calls* com chamadas não qualificadas por outras chamadas; *calls* com método da coleção sendo da interface *Map*; *calls* com mesmo método da coleção em um mesmo método cliente.**

- ***Calls* padrão:**

Saída do *MCP*:

<java.util.List, remove[int], AccessSystem.People.BaseInformation.Person, SetupForms.MainMenu, deleteUserBtnActionPerformed[java.awt.event.ActionEvent], void, null>

<java.util.List, add[AccessSystem.Architecture.Campus], boolean, SetupForms.UniversitySetup, campusAddBtnActionPerformed[java.awt.event.ActionEvent], void, null>

Saída do *Enclose*:

QUEBRAM O CONFINAMENTO: (2)

<AccessSystem.PersonRegister, getRegister[], java.util.List<AccessSystem.People.BaseInformation.Person>, SetupForms.MainMenu, deleteUserBtnActionPerformed[java.awt.event.ActionEvent], void, remove[int]>

<AccessSystem.Architecture.University, getCampusList[], java.util.List<AccessSystem.Architecture.Campus>, SetupForms.UniversitySetup, campusAddBtnActionPerformed[java.awt.event.ActionEvent], void, add[AccessSystem.Architecture.Campus]>

- ***calls* com chamadas não qualificadas por outras chamadas:**

Saída do *MCP*:

Quebram o confinamento

<java.util.List, add[java.lang.String], boolean, br.edu.ifpb.ads.poo.biblioteca.Controllers.AdicionarExemplarControl, separar_Altores[java.lang.String], java.util.List<java.lang.String>, null>

<java.util.List, add[br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo], boolean, br.edu.ifpb.ads.poo.biblioteca.Controllers.FinalizarEmprestimoControl, procurarGeral[java.lang.String], java.util.List<br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo>, null>

<java.util.List, add[br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo], boolean, br.edu.ifpb.ads.poo.biblioteca.Controllers.FinalizarEmprestimoControl, procurarUsuario[java.lang.String], java.util.List<br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo>, null>

<java.util.List, add[br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo], boolean, br.edu.ifpb.ads.poo.biblioteca.Interface.UsuarioLoginFrame, enviarjButtonActionPerformed[java.awt.event.ActionEvent], void, null>

Saída do *Enclose*:

QUEBRAM O CONFINAMENTO: (1)

<br.edu.ifpb.ads.poo.biblioteca.Entidades.Usuario, getEmprestimo[], java.util.List<br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo>, br.edu.ifpb.ads.poo.biblioteca.Interface.UsuarioLoginFrame, enviarjButtonActionPerformed[java.awt.event.ActionEvent], void, add[br.edu.ifpb.ads.poo.biblioteca.Entidades.Emprestimo]>

- ***calls* com método da coleção sendo da interface *Map*:**

Saída do *MCP*:

Quebram o confinamento

0

Saída do *Enclose*:

QUEBRAM O CONFINAMENTO: (1)

<null, getScoresMap[], java.util.HashMap<participatingbody.Hostel,java.lang.Long>, main.Main, initializeScoreBoard[], void, put[participatingbody.Hostel, long]>

- ***calls* com mesmo método da coleção em um mesmo método cliente:**

Saída do *MCP*:

Quebram o confinamento

<java.util.ArrayList, add[java.lang.String], boolean, br.com.novaroma.projeto.negocio.ControladorLivro, emprestar[java.lang.String, java.lang.String, java.lang.String], void, null>

<java.util.ArrayList, add[java.lang.String], boolean, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, null>

<java.util.ArrayList, remove[int], java.lang.String, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, null>

3

Saída do *Enclose*:

QUEBRAM O CONFINAMENTO: (8)

<br.com.novaroma.projeto.entidades.Emprestimo, getNomeFuncionario[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, emprestar[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

<br.com.novaroma.projeto.entidades.Emprestimo, getLivros[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, emprestar[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

<br.com.novaroma.projeto.entidades.Emprestimo, getDataEmprestimos[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, emprestar[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

<br.com.novaroma.projeto.entidades.Emprestimo, getLivros[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, remove[int]>

<br.com.novaroma.projeto.entidades.Emprestimo, getDataEmprestimos[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, remove[int]>

<br.com.novaroma.projeto.entidades.Devolucao, getNomeFuncionario[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

<br.com.novaroma.projeto.entidades.Devolucao, getLivros[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

<br.com.novaroma.projeto.entidades.Devolucao, getDataEmprestimos[], java.util.ArrayList<java.lang.String>, br.com.novaroma.projeto.negocio.ControladorLivro, devolver[java.lang.String, java.lang.String, java.lang.String], void, add[java.lang.String]>

Os elementos que armazenam a classe com o atributo privado e a classe na qual acontece a alteração deste mesmo campo, até aqui conhecidos como *calls*, por informações na presente seção reunidas podem ser melhor analisadas e avaliadas.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Cajazeiras
Rua José Antônio da Silva, 300, Jardim Oásis, CEP 58.900-000, Cajazeiras (PB)
CNPJ: 10.783.898/0005-07 - Telefone: (83) 3532-4100

Documento Digitalizado Restrito

Trabalho de Conclusão de Curso Final com Ata

Assunto:	Trabalho de Conclusão de Curso Final com Ata
Assinado por:	Oliveira David
Tipo do Documento:	Anexo
Situação:	Finalizado
Nível de Acesso:	Restrito
Hipótese Legal:	Direito Autoral (Art. 24, III, da Lei no 9.610/1998)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- José David de Oliveira Sousa, ALUNO (201912010007) DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - CAJAZEIRAS, em 28/06/2022 12:06:20.

Este documento foi armazenado no SUAP em 28/06/2022. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 557927
Código de Autenticação: 103beb2a7c