UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PEDRO MAGNUS PEDROSO NOGUEIRA

ABORDAGEM PARA EXTENSÃO DA FERRAMENTA *REFACTORING AND MEASUREMENT TOOL* DE DETECÇÃO DE PONTOS DE INSERÇÃO E APLICAÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2019

PEDRO MAGNUS PEDROSO NOGUEIRA

ABORDAGEM PARA EXTENSÃO DA FERRAMENTA *REFACTORING AND MEASUREMENT TOOL* DE DETECÇÃO DE PONTOS DE INSERÇÃO E APLICAÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Drª. Simone Nasser Matos

PONTA GROSSA

2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

ABORDAGEM PARA EXTENSÃO DA FERRAMENTA *REFACTORING AND MEASUREMENT TOOL* DE DETECÇÃO DE PONTOS DE INSERÇÃO E APLICAÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE

Por

Pedro Magnus Pedroso Nogueira

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 18 de novembro de 2019 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Profa. Dra. Simone Nasser Matos
Orientador(a)

_____
Profa. Dra. Simone de Almeida
Membro titular

_____
Profa. Dra. Eliana Claudia Mayumi Ishikawa
Membro titular

_____
Prof. MSc Geraldo Ranthum
Prof. Responsável
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dra. Mauren Louise Sguario
Coordenadora do curso

AGRADECIMENTOS

Agradeço a Professora Dr.ª Simone Nasser Matos pela oportunidade, conhecimento compartilhado, comentários, críticas, sugestões e apoio.

Aos meus pais por sempre me apoiarem nos momentos difíceis.

Aos meus amigos por sempre estarem ao meu lado quando mais precisei.

RESUMO

NOGUEIRA, P. P. M. **Abordagem para Extensão da Ferramenta *Refactoring and Measurement Tool* de Detecção de Pontos de Inserção e Aplicação de Padrões de Projeto Em Código-Fonte.** 2019. 54 f. Trabalho de Conclusão de Curso Bacharelado em Ciência da Computação - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2019.

Refatoração de software é o processo de modificar um código-fonte sem alterar seu comportamento externo, visando otimizar a sua estrutura interna. A refatoração pode ser realizada por vários métodos, dentre eles os fundamentados em padrões de projeto que permitem analisar o código-fonte para detectar e inserir padrões que ajudam a melhorar a sua flexibilidade, manutenibilidade, entre outros requisitos de qualidade. O processo de aplicação de padrões de projeto pode ser realizado automaticamente por meio da ferramenta *Refactoring and Measurement Tool* que procura pontos de inserção e aplica os padrões de projeto de acordo com os métodos da literatura. A diferença deste processo é que contempla em um único ambiente métodos para inserção e detecção de padrões de projeto. A ferramenta possui dois métodos implementados, podendo ser estendida para contemplar outros de modo a torná-la mais completa. Este trabalho propõe a criação de uma abordagem de extensão para a *Refactoring and Measurement Tool* a fim de que novos métodos de detecção e inserção de padrões de projeto sejam incorporados a ela. A abordagem proposta foi utilizada para inserir o método de *minipatterns* e os testes usando esta nova inserção utilizou cinquenta projetos *open-souce*. A abordagem proposta contém um conjunto de etapas que permitem facilitar o processo de inserção de novos métodos à *Refactoring and Measurement Tool*.

Palavras-chave: Refatoração de Software. Ferramenta de Refatoração. Padrões de Projeto.

ABSTRACT

NOGUEIRA, P. P. M. **Approach for Extension of the Refactoring and Measurement Tool for Detection of Insertion Points and Application of Design Patterns in Source Code**. 2019. 54 p. Work of Conclusion Course Graduation in Computer Science - Federal Technology University - Paraná. Ponta Grossa, 2019.

Software refactoring is the process of modifying a source code without altering its external behavior, aiming to optimize its internal structure. Refactoring can be performed by several methods, including those based on design standards that allow the source code to be analyzed to detect and insert patterns that help improve its flexibility, maintainability, among other quality requirements. The process of applying design patterns can be performed automatically, by the Refactoring and Measurement Tool that looks for insertion points and applies the design patterns according to the methods in the literature. The difference of this process is that it contemplates a single environment method for insertion and detection of design patterns. The tool has two methods implemented and can be extended to include others to make it more complete. This work proposes the creation of an extension approach to the *Refactoring and Measurement Tool* so that new methods of detection and insertion of design patterns can be incorporated into it. The proposed approach was used to insert the mini patterns method and the tests using this new insertion used fifty open-source projects. The proposed approach contains a set of steps to facilitate the process of inserting new methods into the *Refactoring and Measurement Tool*.

Keywords: Software Refactoring. Refactoring Tool. Design Patterns.

LISTA DE ILUSTRAÇÕES

Figura 1 - Aplicação da minitransformation Abstraction 17
Figura 2 - Funcionamento do Método Cinnèide (2000) 18
Figura 3 - Importar projetos no Client App 21
Figura 4 - Seleção de candidatos a refatoração 21
Figura 5 - Aplicar a refatoração ao candidatos 21
Figura 6 - Visão geral da abordagem da ferramenta RMT 22
Figura 7 - Fluxo de Extensão da Ferramenta RMT 25
Figura 8 - Aba para configuração do JBoss 26
Figura 9 - Seleção de builds para o server 27
Figura 10 – Divisão de subprojetos 29
Figura 11 - Pacote do método de Cinnèide (2000) 29
Figura 12 - Aba de servidores 31
Figura 13 - Grafo de chamada de métodos 32
Figura 14 - Representação do pacote do autor do novo método 33
Figura 15 - Representação do pacote verifier 34
Figura 16 - Representação da classe executers 34
Figura 17 - Classe de testa CTest 37
Figura 18 - Implementação da minitransformation Abstract Access 38
Figura 19 - Implementação da minitransformation Encapsulate Construction 38
Figura 20 - Implementação da minitransformation Partial Abstraction 39
Figura 21 - Pacote definindo candidatos de Cinnèide (2000) 41
Figura 22- Pacote de verifiers Cinnèide (2000) 41
Figura 23 - Pacote de Executors Cinnèide (2000) 42
Figura 24 - Diagrama de classe da implementação do método de Cinnèide (2000) .42

LISTA DE TABELAS

Tabela 1 – Projetos de Cenários de Testes 43
Tabela 2 – Projetos que possuem candidatos a refatoração com o padrão Singleton 44
Tabela 3 – Projetos com candidatos a refatoração e avaliação de métricas.....47

SUMÁRIO

1 INTRODUÇÃO....9

1.1 Objetivos....11

1.2 Organização do Trabalho....11

2 REFATORAÇÃO DE SOFTWARE....12

2.1 Importância de refatoração....13

2.2 Formas de Aplicação da Refatoração de Software....13

2.3 Métodos de Refatoração Baseados em Padrões de Projetos....14

2.4 Ferramenta de Refatoração....18

2.5 Considerações do Capítulo....23

3 ABORDAGEM DE EXTENSÃO DA FERRAMENTA RMT....24

3.1 Visão Geral....24

3.2 Instalar a Ferramenta RMT....25

3.3 Testar a Ferramenta RMT....27

3.4 Analisar o Projeto da Ferramenta RMT....28

3.5 Analisar o fluxo da Ferramenta....30

3.6 CONSIDERAÇÕES DO CAPÍTULO....35

4 RESULTADOS....36

4.1 Implementação do Método de Cinnèide (2000)....36

4.2 Aplicação da Abordagem de Extensão Proposta....39

4.3 Testes da Ferramenta RMT com o Método de Cinnèide....42

4.4 Análise da Abordagem Proposta....48

4.5 CONSIDERAÇÕES DO CAPÍTULO....48

5 CONCLUSÃO....49

5.1 TRABALHOS FUTUROS....49

REFERÊNCIAS....50

1 INTRODUÇÃO

Refatoração é o processo de modificar um software de maneira que o comportamento externo do código não é alterado e ainda otimiza-se sua estrutura interna. A princípio um sistema deve ser projetado antes de ser programado, mas mesmo com um bom projeto ocasionalmente problemas ou até mesmo aplicações erradas podem aparecer. Nesses casos, a refatoração deve ser aplicada para corrigir o código e fazê-lo se adequar ao projeto inicial (FOWLER, 2018).

Um sistema normalmente é desenvolvido por uma equipe o que pode causar problemas na legibilidade do código. Nesse caso, a refatoração é importante para manter o código dentro do padrão estipulado (FOWLER, 2018; KERIEVSKY, 2004). Como definido por Fowler (2018, tradução própria, p.46) a refatoração é “uma mudança na estrutura interna do software para deixá-lo mais barato e de fácil entendimento sem mudar seu comportamento.”

A refatoração pode ser aplicada usando técnicas como descrita por Fowler (2018) ou por padrões de projetos como relatado por Kerievsky (2004). A refatoração por técnicas realiza diferentes mudanças ao código, como deixá-lo mais simples e melhorar sua legibilidade. Essas técnicas englobam conceitos como renomeação de campo, variável e função; encapsulamento; remoção de código não utilizado, entre outros (FOWLER, 2018).

A refatoração baseada em padrões de projetos aplica soluções e abstração de problemas comuns visando a sua reutilização, sempre levando em consideração as restrições impostas ao problema (GAMMA *et al*. 2009; KERIEVSKY 2004). Os padrões permitem a criação de um código que seja de fácil manutenção e reusável.

A refatoração de software com padrões de projeto pode ser feita usando métodos e para encontrar os pontos dentro do código-fonte que podem ser melhorados é necessário debugar o código. Para confirmar que nenhuma funcionalidade foi afetada, testes devem ser realizados e o tempo para sua realização varia de acordo com o tamanho do sistema e da quantidade de programadores envolvidos, sem contar que a refatoração manual é propensa a erros (MENS; TOURWE, 2004; MOOR IVAN, 1996). A refatoração deve ser aplicada por meio de software para a otimização de tempo e a minimização do erro humano.

Baseando-se na ideia de refatoração de software aplicada com padrões de projetos e procurando ferramentas automatizadas, Beluzzo (2018) realizou um

mapeamento sistemático para encontrar os trabalhos relacionados a refatoração de software baseada em padrões de projetos. O mapeamento inclui trabalhos de 1997 até 2017, e em seus resultados verificou que existem vários métodos de refatoração baseado em padrões de projeto, porém não existe um ambiente integrado que os reúna. Ou seja, se um desenvolvedor precisa refatorar seu código para inserir padrões de projeto necessita usar várias ferramentas para atingir seu objetivo.

Por isto, Beluzzo (2018) propôs uma ferramenta denominada de RMT (*Refactoring and Measurement Tool*) que procura encontrar pontos de inserção e aplicação de padrões de projetos que contempla métodos de refatoração para padrões de projeto, possui interação com o usuário e métricas para avaliar o código-fonte. A ferramenta foi implementada com os métodos de Liu *et al*. (2014) e Zafeiris *et al*. (2017). O método de Liu *et al*. (2014) contém refatorações para o padrão *Factory Method* e *Strategy*; enquanto o de Zafeiris *et al*. (2017) possui refatorações para o padrão *Template Method.*

A ferramenta RMT precisa ser estendida para incorporar mais métodos com o objetivo de deixá-la mais completa e disponibilizá-la para uso de profissionais de computação que desenvolvem aplicações. Desta forma, este trabalho propõe uma abordagem de extensão para a ferramenta RMT composta por várias etapas que podem ser executadas e permite que novos métodos de detecção e inserção de padrões de projeto possam ser adicionados a ela.

A abordagem proposta foi utilizada para incorporar a ferramenta RMT o método desenvolvido por Cinnèide (2000) que é fundamentado em *minipatterns* e *minitransformations* para detecção e inserção de padrões de projeto. Durante este processo foram criados novos pacotes e classes dentro do *detection-method,* que é um subprojeto responsável para a verificação e execução das refatorações dentro da ferramenta RMT. Foram realizados testes com a ferramenta RMT com projetos disponíveis no *GitHub* para testar se o novo método implementado conseguia detectar e inserir os padrões.

1.1 OBJETIVOS

Desenvolver uma abordagem de extensão a fim de permitir que novos métodos de detecção e inserção de padrões de projeto possam ser incorporados a ferramenta RMT criada por Beluzzo (2018).

Os objetivos específicos são:

- Codificar o método de Cinnèide (2000) de forma individual para o entendimento de seu funcionamento.
- Aplicar a abordagem proposta na inserção do método de Cinnèide (2000) dentro da ferramenta RMT.
- Avaliar as contribuições e fragilidades trazidas com a abordagem proposta.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado em cinco capítulos. O capítulo 2 apresenta conceitos sobre refatoração de software, sua importância e como aplicá-la. Relata também uma descrição do método de refatoração escolhido para ser incorporado à ferramenta RMT. Descreve também sobre as ferramentas de refatoração presentes na literatura dando ênfase à que foi desenvolvido por Beluzzo (2018), foco deste trabalho.

O capítulo 3 descreve a abordagem de extensão para que novos métodos de detecção e inserção de padrões de projeto possam ser incorporados a ferramenta RMT.

O capítulo 4 relata a aplicação da abordagem proposta para inserir o método de Cinnèide (2000) a ferramenta RMT, bem como apresenta os resultados dos testes e análise. Por fim, o último capítulo descreve as conclusões deste trabalho e as pesquisas que podem ser realizadas.

2 REFATORAÇÃO DE SOFTWARE

Como descrito por Fowler (2018, tradução própria, p.9) a refatoração é “o melhoramento do projeto após o código ser escrito”. Pontos onde a refatoração é necessária podem surgir ao longo do desenvolvimento do projeto independentemente de como projeto foi realizado, principalmente em sistemas desenvolvidos por equipes.

A refatoração procura resolver problemas encontrados na produção de softwares, quando um software se torna maduro e precisa ser evoluído dois conflitos surgem: i) O software cumprir todos os requisitos, e ii) O software precisa ser reusado. É possível implementar novas funcionalidades ao software sem refatorá-lo, mas eventualmente ele se tornara inflexível (GAMMA *et al*., 2009).

A refatoração pode encontrar códigos duplicados e linhas de códigos parecidas, facilita a implementação de novos requisitos, modulariza classes sobrecarregadas e encontra métodos e classes desnecessárias.

A refatoração pode ser realizada por meio de técnicas ou métodos baseados em padrão de projetos. Em técnicas de refatoração o autor mais citado, com aproxidamente 8.354 citações, segundo a plataforma *Google Scholar* é Fowler (2018), que criou um catálogo separando as refatorações em 7 (sete) grupos.

Fowler possui um catálogo *online* que engloba as refatorações descritas em seus dois livros: *Refactoring: improving the design of existing code* (FOWLER, 2018) e *Refactoring: Ruby Edition: Ruby Edition* (FIELDS; HARVIE; FOWLER, 2009).

A refatoração por métodos baseados em padrões de projetos procuram no código-fonte pontos de inserção em que os padrões possam ser aplicados. Essa refatoração visa principalmente a manutenibilidade, legibilidade e reusabilidade. Vários autores propõem seu método de refatoração baseado em padrões de projeto, tais como Cinnèide (2000), Gamma *et al*. (2009) e Ouni *et al*. (2017).

Este capítulo apresenta informações sobre refatoração de software. A seção 2.1 discute a importância da refatoração. A seção 2.2 relata sobre a aplicação da refatoração. A seção 2.3 descreve os métodos de refatoração e seus funcionamentos explicados. Por fim, a seção 2.4 descreve as considerações finais do capítulo.

2.1 IMPORTÂNCIA DE REFATORAÇÃO

O primeiro passo da refatoração é a criação de testes, pois mesmo seguindo as estruturas da refatoração para evitar as oportunidades de criar *bugs*, humanos erram e causam graves problemas em produção. Na produção de sistemas a manutenibilidade é um grande fator que influência nos custos e uma das soluções propostas para reduzí-la é aumentar a qualidade de software.

A refatoração pode ser aplicada junto com o processo desenvolvimento e usada para aumentar a qualidade em que o programador é forçado a encontrar "*bad smells"* (Uma característica do código que indica um problema) (WILKING *et al*., 2017).

No estudo desenvolvido por Szőke *et al*. (2017), seis sistemas foram analisados para verificar a eficácia da refatoração de software visando manutenibilidade. Para obter os dados antes das refatorações, Szőke *et al*. (2017) analisou todos os sistemas utilizando um analisador de código chamado *SourcerMeter tool* baseada na ferramenta Columbus (FERENC *et al*., 2002). Essas refatorações foram feitas manualmente e os desenvolvedores foram informados sobre todos os dados incluindo a lista com os fragmentos de códigos problemáticos.

Nesse estudo os desenvolvedores analisaram um total de 2,5 milhões de linhas, realizando 732 revisões ao código em que 315 foram refatorações e realizaram 1.273 operações de refatorações. Após finalizar as análises Szőke *et al*. (2017), verificou que a refatoração de fato melhorou a manutenibilidade e a ferramenta foi capaz de encontrar melhora em 5 dos 6 sistemas testados. Como conclusão, o estudo demonstra que a refatoração realmente melhora a manutenibilidade o que ajuda no cumprimento de requisitos futuros dos softwares.

2.2 FORMAS DE APLICAÇÃO DA REFATORAÇÃO DE SOFTWARE

Como já mencionado a refatoração pode ser realizada por técnicas e métodos baseados em padrões de projetos. As refatorações por técnicas são comuns e podem ser encontradas nas principais IDEs, como o Eclipse, o qual disponibiliza refatorações automáticas baseadas no modelo criado por Fowler (2018). Mesmo possuindo ferramentas automáticas, as refatorações podem ser implementadas manualmente o

que resulta em uma demora na sua aplicação. A demora para a aplicação da refatoração manual é utilizada como argumentos contra a refatoração, esses podem ser encontrados (WILKING *et al.*, 2017).

Nas refatorações utilizando métodos baseados em padrões de projetos, existem vários autores que propõem métodos de refatoração de software aplicando padrões, cada um realizado de uma forma diferente. Alguns desses autores são: Liu *et al*. (2014), Zafeiris *et al*. (2017), Cinnèide (2000) e Ouni *et al*. (2017).

Esses métodos de refatoração procuram encontrar pontos de inserção de padrões em código-fonte, ou seja, busca códigos onde padrões de projetos possam ser implementados para melhorar a manutenibilidade, legibilidade e reusabilidade.

As refatorações baseadas em padrões de projetos diferentes das refatorações por técnicas procuram por partes dos códigos em que um padrão de projeto pode ser aplicado. A utilização do padrão de projeto é muito importante, pois a faz com que a evolução do software seja realiza de forma mais natural não criando problemas na implementação de novos requisitos. Existem vários padrões de projetos cada um com uma funcionalidade diferente, como: *factory method, proxy, observer, adpter, visitor, builder, adapter,* entre outros (GAMMA *et al*., 2009).

Como descrito no mapeamento realizado por Beluzzo, (2018), diversos autores propõem métodos de refatorações baseadas em padrões de projetos. Por exemplo, o método Moore e o por *Minipatterns and Mnitransformations* sendo que para cada método existe uma ferramenta que o implementa. Se o desenvolvedor desejar refatorar seu código-fonte terá que executar *n* ferramentas, onde *n* é a quantidade total de métodos.

O foco da refatoração por padrões de projetos é diferente da refatoração por técnicas, a refatoração por padrões procura encontrar uma forma de tornar um sistema fácil de ser “atualizado”. Um software com dificuldades para ser “atualizado” pode custar muito para a empresa, tanto em valor quanto em necessidades para novos cliente e competitividade com outras empresas (CINNÈIDE, 2000).

2.3 MÉTODOS DE REFATORAÇÃO BASEADOS EM PADRÕES DE PROJETOS

Conforme mencionado, Beluzzo (2018) identificou métodos para detecção e inserção de padrões de projeto. Dentre os métodos identificados por ele, este trabalho

elencou 2 (dois) para ser incorporados a ferramenta, a saber, Liu *et al*. (2014) e Zafeiris *et al*. (2017).

Dentre os métodos identificados no mapeamento sistemático de Beluzzo (2018), este trabalho incorporou a ferramenta RMT o de Cinnèide (2000), pois este apresenta trabalhos que mostram seu funcionamento e é um dos autores que possuem mais estudos recentes sobre refatoração fundamentada em padrões de projeto. Este método é detalhado na próxima seção.

2.3.1 Método de refatoração por *Minipatterns* e *Minitransformations*

Cinnèide (2000) apresenta um método de inserção e detecção de padrões de projeto que é dividido em duas partes: *minipatterns* e *minitransformations*.

Minipatterns é um padrão de projeto dividido em vários *motifs* que são características comuns entre padrões de projetos e a combinação diferenciada produz diferentes padrões de projetos. *Minipatterns* são *motifs,* pois um padrão de projeto é uma coleção de *motifs,* então pode se dizer que um padrão de projeto pode ser considerado uma coleção de *minipatterns*. A refatoração aplicada ao *minipattern* é chamada de *minitransformation*. Por exemplo, o *Factory Method* é composto pelos *minipatterns*:

1. *Abstraction:* a classe *Product* deve ter uma interface que reflita como a classe Criador utiliza as instâncias do Produto que ele cria.
2. *EncapsulatedConstruction:* na classe *Creator*, a construção de objetos do Produto deve ser encapsulada dentro de métodos dedicados e substituíveis, que são denominamos métodos de construção.
3. *AbstractAccess:* além de estar dentro dos métodos de construção descritos na classe *Creator*, não deve ter conhecimento da classe *Product*, exceto por meio da interface definida pelo “usuario”.
4. *PartialAbstraction:* a classe *Creator* deve herdar de uma classe abstrata em que os métodos de construção são declarados abstratos.

Minitransformation são as refatorações primitivas de um certo método e é aplicada utilizando uma corrente de *precondition* (pré-condição) e *postcondition* (pós-condição) as quais são definidas diferentemente dependendo dos padrões. A pré-condição é o que precisa ser verificado antes da refatoração ser aplicada, como por

exemplo, na transformação de uma classe em uma interface, o primeiro passo é verificar se a classe existe, e o segundo passo verifica se não existe uma interface com o nome da interface a ser implementada. A pós-condição verifica se a refatoração foi realizada, verificando por exemplo, se a interface existe e se ela está instanciada.

A junção do *minipatterns* com as *minitransformations* começa pela menor refatoração possível e evolui conforme mais iterações são realizadas, como no processo descrito no quadro 1.

Quadro 1 – Exemplo de minitransformação para o padrão *Factory Method*

Classe *abstraction*:

```
Abstraction (Class c,String newName){
        //instancia uma nova classe
        Interface inf = abstractClass(c, newName);
        //transforma a classe me um interface e adiciona o link a classe anterior a qual não era
        uma interface.
        addInterface(inf); addImplementsLink(c,inf);
}
```

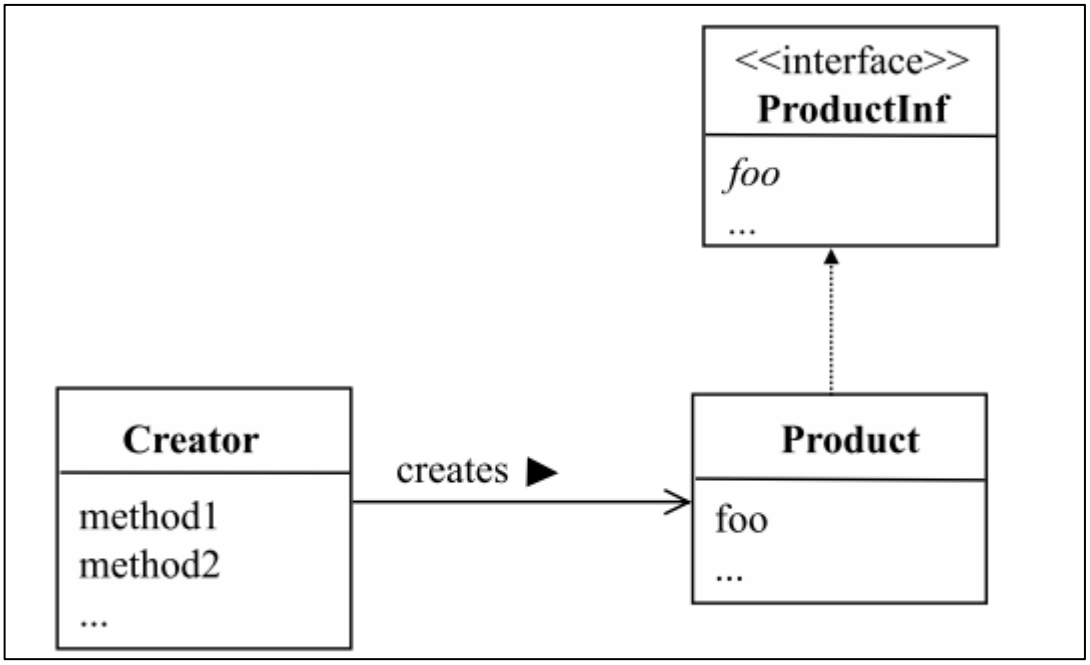
Fonte: Autória própria

Para começar a refatoração as precondições devem ser verdadeira. A primeira verificação é se não existe uma classe ou interface com o mesmo nome. Após a aplicação das precondições as pos-condições são aplicadas, sendo:

- Uma nova interface *inf* chamada “newName” existe.
- A classe *c* e a interface *inf* tem a mesma interface pública.
- Verifica se a ligação existe da classe *c* com a interface *inf*.

Logo após, uma nova interface é adicionada e permite uma visão abstrata da classe.

Figura 1 - Aplicação da *minitransformation Abstraction*

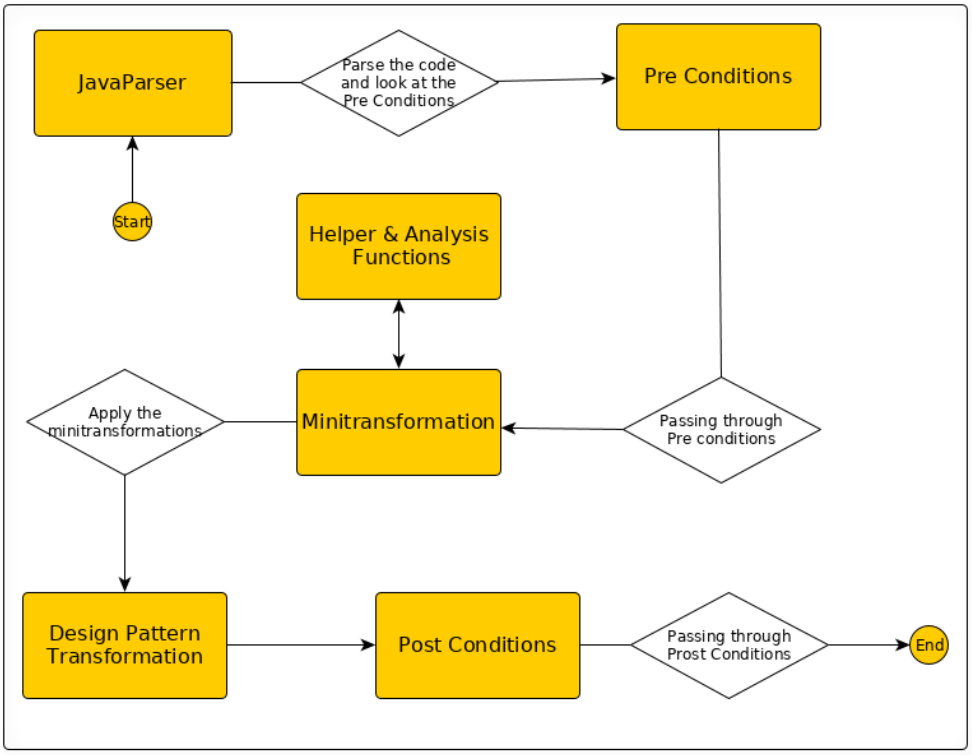


Fonte: Cinnèide (2000)

A figura 2 representa o fluxo seguido pelo método de Cinnéide (2000) para realizar uma refatoração. O fluxo inicia com a utilização do *JavaParser*, uma biblioteca de parser que permite analisar o código de um arquivo Java dentro do código Java.

O *JavaParser* foi utilizado para criar a *abstract sintax tree,* que permite analisar o código e com isso as pre-condições para um método são verificadas, as pre-condições são executas. Caso validadas a aplicação segue para o próximo passo. Após ser a analisado o código é encaminhado para as *minitransformations* as quais são aplicadas com base na refatoração a ser realizada. Essa aplicação acontece dentro da transformação que é representada por um conjunto de *minitransformations.* Após a transformação o *abstract sintax tree* é enviada pelas pós-condições para verificar se todas as mudanças que necessitavam serem feitas realmente aconteceram.

Figura 2 - Funcionamento do Método Cinnèide (2000)



Fonte: Autoria Própria

As refatorações propostas por Cinnèide (2000) são: *Factory Methods*, *Singleton*, *Abstract Factory*, *Builder*, *Prototype*, *Bridge* e *Strategy*.

2.4 FERRAMENTA DE REFATORAÇÃO

A refatoração manual é um processo utilizado por desenvolvedores, mas elas podem introduzir erros ao código e até mesmo mudar a funcionalidade do sistema. Esse problema acontece porque os desenvolvedores não estão preparados para realizar a refatoração (GE; DUBOSE; MURPHY-HILL, 2014).

A refatoração manual pode demorar muito tempo para ser realizada, pois o código inteiro deve ser revisado. Por esse motivo, ferramentas são criadas para facilitar a aplicação das refatorações, podendo atuar na identificação *bad smells* ou identificar e refatorar o código-fonte como criado por Beluzzo (2018).

Um mapeamento para encontrar técnicas de refatorações foi realizado, e em seus resultados várias ferramentas foram encontradas (BELUZZO, 2018). Constatou-

se que ferramentas já foram desenvolvidas para realizar a refatoração, como a Elbereth desenvolvida por Korman e Griswold (1998) que automatiza algumas refatorações a extração de código, englobando a extração de método (*Extract Method*); extração de superclasses abstratas, substituição de uma classe existente, e adição de uma nova subclasse.

Visando a interação com o usuário para a aplicação da refatoração Murphy-Hill e Black (2008) criaram uma ferramenta para ser rápida, resistente a erros e que seja prazerosa de utilizar. Para tornar a ferramenta prazerosa foram implementadas marcas no código feitas por retângulos, flechas e colorindo partes do código. Essas 3 marcas são utilizadas em conjunto para demonstrar ao usuário de uma forma limpa onde e qual é o problema para que as refatorações sejam aplicadas.

Existem também ferramentas contendo interface gráfica e iteração com o usuário, como Rani; Kaur e Prof (2014), em que o intuito é detectar *bad smells.* Esta ferramenta foi desenvolvida em C# e encontra *bad smells* como *long method, large class, lazy class* e *comment lines*, esse *smells* podem ser encontrados em códigos escritos em JAVA e *.net*.

Muitas ferramentas de refatoração de código são baseadas no catálogo de refatorações criado por Fowler (2019). Mas, existem outros autores que propõem novas refatorações. Entretanto, a maioria dos autores concorda que a refatoração realizada manualmente é muito demorada e difícil de ser realizada sem criar novos *smells* ao código, por esse motivo as ferramentas são criadas.

Ge e Murphy-Hill (2014) em conjunto com oito desenvolvedores de software com diferenças experiências de trabalhos utilizam a ferramenta de *GostFactor* para auxiliar no processo de refatoração e foi observado que esse processo diminuiu em 23,3% os erros na aplicação da refatoração.

A ferramenta RMT criada por Beluzzo (2018) permite integrar métodos de inserção e detecção de padrões de projeto em um único ambiente. A ferramenta tem uma iteração com usuário para que ele saiba o que irá acontecer no código após a refatoração e ao mesmo tempo saber se a refatoração irá realmente melhorar o código ou não. Esta ferramenta é o foco desta pesquisa porque constitui um ambiente integrado de métodos de detecção e inserção de padrões de projeto e é detalhada na próxima seção.

O trabalho de Sangeetha e Chandrasekar (2019) não descrevem o funcionamento da ferramenta implementada, mas a utiliza para realizar o teste e a validação das técnicas utilizadas.

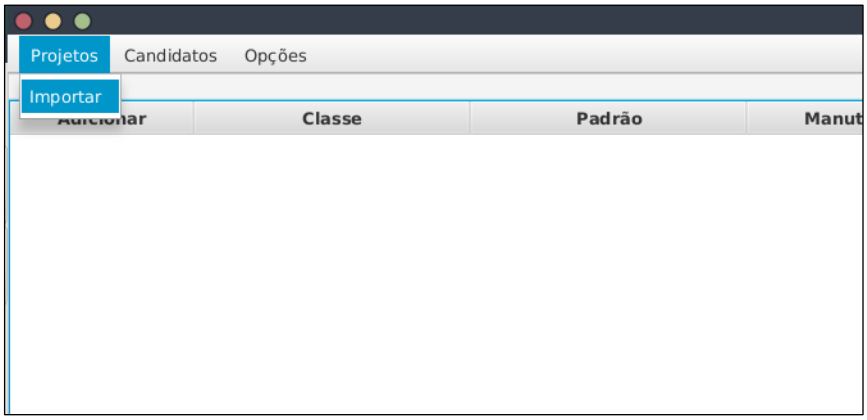
2.4.1 RMT: Uma Breve Descrição

A ferramenta RMT (*Refactoring and Measurement Tool*) foi criada para a detecção de pontos de inserção de padrões de projetos utilizando os métodos selecionados pelo autor (BELUZZO, 2018). Os métodos de refatorações que já foram implementados são os desenvolvido por Zafeiris *et al*. (2017) e Liu *et al*. (2014). O método de Zafeiris *et al*. (2017) contempla a refatoração para o padrão *Template Method* e o método de Liu *et al*. (2014) abrange dois padrões sendo eles: *Strategy* e *Factory Method*.

Além da aplicação das refatorações, a ferramenta utiliza o extrator de métricas CK (CK GITHUB, 2019) que engloba as métricas de profundidade da árvore de herança (PAH), complexidade ciclomática (CC) e tamanho do programa em linhas de código (TPLC). Após o usuário selecionar o projeto a ser refatorado as métricas são aplicadas para demonstrar ao usuário o efeito das mudanças a serem aplicadas. Os atributos de qualidade externa utilizados foram manutenibilidade, confiabilidade e reusabilidade.

A iteração com usuário que acontece dentro do *Client App* pode ser executada de forma independente, pois se comunica com o *Intermediary Service* que oferece o serviço de comunicação da camada do usuário (*Client*) com a camada de processamento (*Services*). Para realizar a operação o *Intermediary Service* utiliza uma fila para cada *service*, sendo uma fila para o *Metrics Service* e uma para o *Detection Methods Service*. O funcionamento do *Client App* é dividido em 3 etapas. A primeira etapa é a importação do projeto, apresentada na Figura 3.

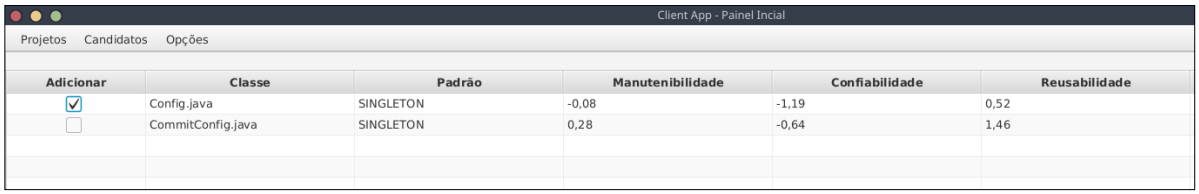
Figura 3 - Importar projetos no *Client App*



Fonte: Beluzzo (2018)

Na segunda etapa o projeto é avaliado e o retorno é mostrado para o usuário, que pode encontrar informações adicionais das refatorações candidatas como nome da classe, pacote a ser refatorado, entre outras, conforme exibo na Figura 4.

Figura 4 - Seleção de candidatos a refatoração



Fonte: Beluzzo (2018)

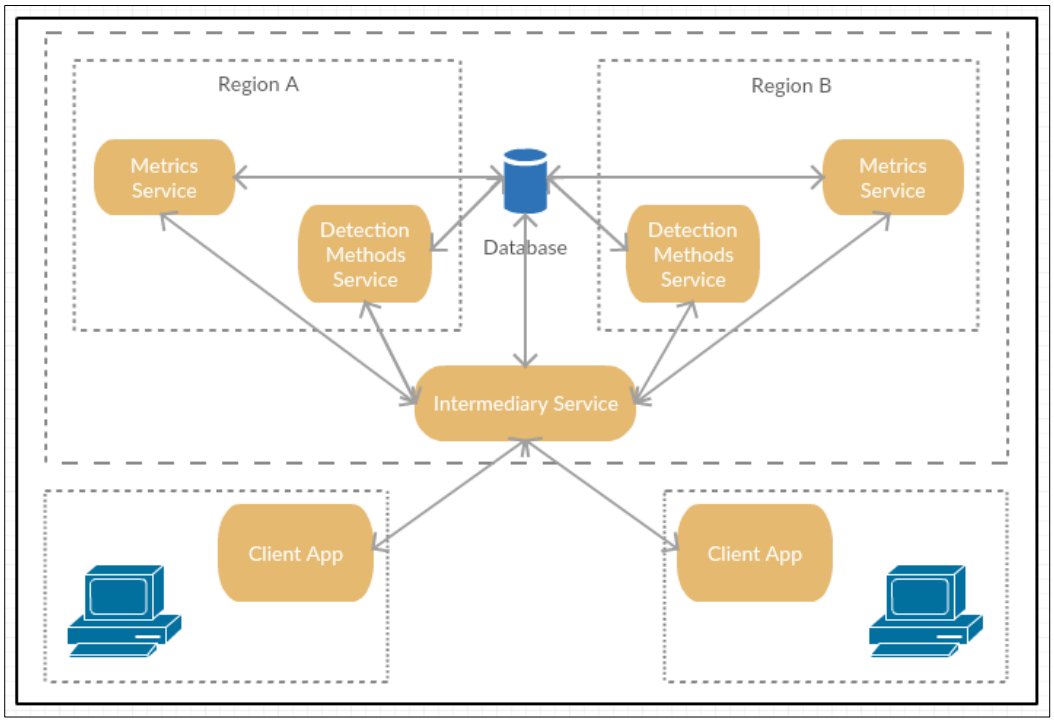
Na terceira e última etapa o usuário pode escolher os candidatos para aplicar a refatoração. Após as refatorações terem sido aplicadas, um novo projeto é salvo em um diretório escolhido pelo usuário, conforme apresenta a Figura 5.

Figura 5 - Aplicar a refatoração ao candidatos

Fonte: Beluzzo (2018)

O modelo criado por Beluzzo (2018) também propõem um sistema de *Regions* para que se possa proporcionar uma estrutura com maior tolerância a falhas, pois quanto mais *Regions* existirem, mais membros para processamentos existirão. Outro fator é a alta disponibilidade da ferramenta, em que as *Regions* deveriam ser colocadas em servidores diferentes, prevenindo sobrecarregamento do servidor. A figura 6 representa a divisões de *regions* e também dos serviços da aplicação.

Figura 6 - Visão geral da abordagem da ferramenta RMT



Fonte: Beluzzo (2018)

O fluxo da arquitetura da Figura 6 se inicia no *Client App* e seguem para o *Intermediary Service* o qual irá escolher o serviço de processamento a ser utilizado dependendo da ação do usuário, podendo escolher “Solicitar a avaliação do código-fonte” e “Solicitar aplicação de padrões de projeto”.

O processo *Intermediary Service* quando recebe um candidato a refatoração pelo *Client App* envia a solicitação para o *Detection Methods Service* o qual busca no banco de dados o código-fonte e obtém os candidatos a refatoração. Os candidatos são retornados ao *Intermediary Service* que os envia para o *Metrics Service* para a avaliação das métricas.

Após o cálculo das métricas, as mesmas são enviadas para o *Intermediary Service* o qual reenvia para *Client App* disponibilizar as métricas e candidatos a

refatoração. Caso o usuário selecione algum candidato a refatoração, o *Client App* envia as solicitações para o *Intermediary Service* com os candidatos a serem refatorados no *Detection Methods Service* e cria um projeto com as refatorações realizadas.

Beluzzo (2018) se preocupou em abstrair a ferramenta para que se possa realizar uma expansão sem maiores problemas, mas nenhum processo de extensão foi definido por ele. Por isto, a criação deste processo de extensão é foco de pesquisa deste trabalho.

2.5 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo relatou a importância da refatoração e também descreveu os métodos de refatoração baseados em padrões de projetos, assim como a descrição do funcionamento de um método de refatoração que foi implementado na ferramenta RMT.

A refatoração é importante para manter o código-fonte sem *smells* (“mal-cheiros”), mantendo-o em um padrão de qualidade. O principal foco da refatoração é a manutenção e reusabilidade do código, pois um código com poucos *smells* pode ser mudado mais facilmente.

Foi abordada a importância do uso de uma ferramenta de refatoração de software para que se possa obter o máximo da refatoração sem prejudicar o trabalho já realizado, dando ênfase a ferramenta RMT que foi foco desta pesquisa.

A ferramenta RMT integra vários métodos de detecção e inserção de padrões de projeto em um único ambiente, assim o desenvolvedor de aplicações pode aplicar em seu código-fonte padrões de projeto sem que precise utilizar várias ferramentas de refatoração que possuem esta finalidade.

3 ABORDAGEM DE EXTENSÃO DA FERRAMENTA RMT

Para poder realizar a extensão de uma ferramenta deve-se primeiro criar uma estratégia de abordagem para ser usada no processo de extensão e ser seguida durante o desenvolvimento da ferramenta. A abordagem deve conter as informações da instalação até a implementação, explicando os passos para que possam ser reutilizados por outros autores sem conhecimento prévio da ferramenta.

Este capítulo apresenta a abordagem proposta para realizar a extensão da ferramenta RMT no que tange a inclusão de novos métodos de inserção e detecção de padrões de projeto. A seção 3.1 descreve a visão geral da abordagem de extensão proposta neste trabalho. As seções 3.2, 3.3, 3.4 e 3.5 relatam detalhamente cada etapa da abordagem proposta. Por fim, a última seção apresenta as considerações do capítulo.

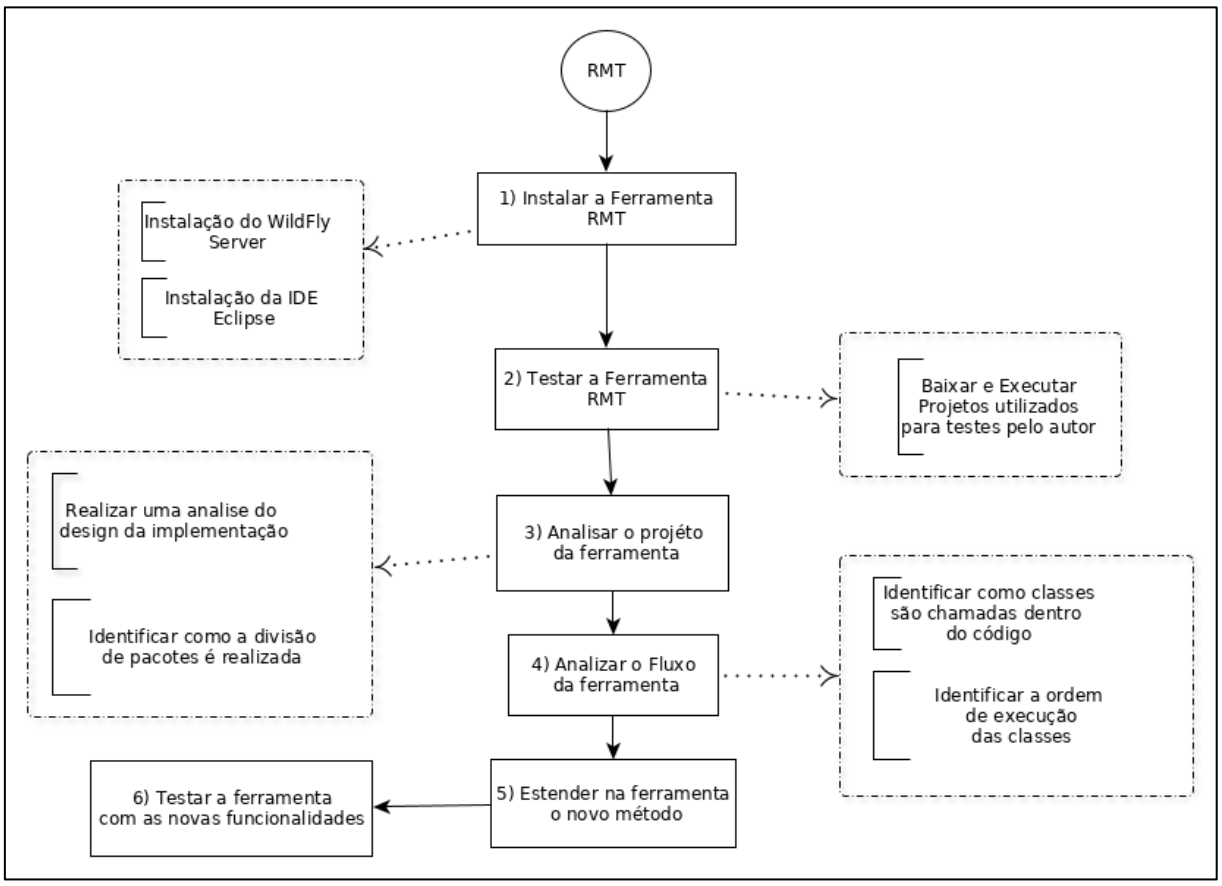
3.1 VISÃO GERAL

A abordagem de extensão proposta neste trabalho foi dividida em 5 etapas as quais descrevem o processo utilizado para a incorporação de um novo método de detecção e inserção de padrões de projeto na ferramenta de refatoração. As etapas são executadas sequencialmente representadas por: 1) Instalar a ferramenta RMT; 2) Testar a ferramenta RMT; 3) Analisar o projeto da ferramenta; 4) Analisar o fluxo da ferramenta; 5) Estender na ferramenta o novo método e 6) Testar a ferramenta com as novas funcionalidades. Essas etapas estão apresentadas na Figura 7. Em cada etapa encontram-se caixas pontilhadas com uma breve descrição do que se deve ser realizado na etapa.

Na primeira etapa é realizada a instalação da ferramenta e seus respectivos requisitos. Na segunda etapa os testes são realizados utilizando projetos utilizados como teste por Beluzzo (2018) descritos na seção 3.3. Na terceira etapa uma análise de projeto da ferramenta é feita para se entender como o funcionamento em relação as divisões de pacotes e classes. Na quarta etapa é realizada um analise do fluxo onde se identifica a ordem pela qual as classes são executadas e os modos de

chamada de classe, como uma instanciação da classe ou uma reflexão. Por fim, a ultima etapa é implementada a qual é descrita no Capítulo 4.

Figura 7 - Fluxo de Extensão da Ferramenta RMT



Fonte: Autoria Própria

3.2 INSTALAR A FERRAMENTA RMT

O primeiro passo para incorporar um novo método de detecção e inserção de padrões de projeto a ferramenta RMT proposta por Beluzzo (2018) é necessário realizar o processo de instalação. Para a instalação é necessário instalar a IDE eclipse 4.13.1 para ler o projeto da ferramenta RMT e executá-la. Como a ferramenta foi desenvolvida utilizando o *Project Manager* Maven também é necessário instalá-lo para poder construir as aplicações de serviço. Como os serviços da RMT são executados pelo *WildFly (*WILDFLY, 2019*)* a sua instalação é necessária, na versão *WildFly 17.0.0.Final*.

Tendo estas aplicações instaladas, o próximo passo é importar o projeto da ferramenta RMT para o Eclipse. Após importar o projeto, deve-se executar a opção é a *build*. Beluzzo (2018) disponibilizou um *bash script* (*script* utilizado para executar comandos *Linux* sequencialmente) para construir todos os serviços da aplicação, este *script* está disponível na pasta raiz do projeto da ferramenta.

Para executar o projeto dentro da IDE eclipse é preciso instalar a extensão *JBoss* a qual executa o *WildFly*, possibilitando a utilização de ferramentas que facilitam o desenvolvimento como *hot reaload*. Para a instalação do *plugin* é necessário seguir os seguintes passos: acessar o menu *Help* > Eclipse *MarketPlace*; dentro da opção do *marketplace* pesquisar por *JBoss* e instalar a extensão *JBoss Tools 4.12.0.Final*.

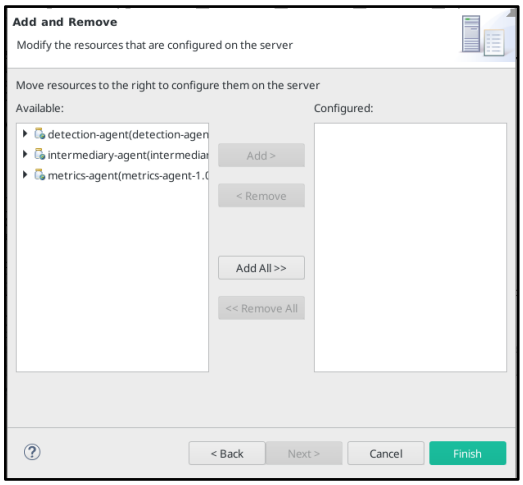
Para configurar o *JBoss* é preciso acessar *Window* > *Show view* > *Other* e selecionar *server.* Uma opção chamada *servers* aparece como representado na Figura 8.

Figura 8 - Aba para configuração do JBoss

Fonte: Autoria Própria

Em configuração deve selecionar a versão instalada do *WildFly* e incluir as *builds* realizadas anteriormente pelo *bash script* ao *server*. As *builds* devem aparecer automaticamente caso o projeto já tenha sido importado para o Eclipse, como exibe na Figura 9 todas a *builds* devem ser incluídas.

Figura 9 - Seleção de *builds* para o *server*



Fonte: Autoria Própria

Após realizar as etapas de instalação e iniciar o servidor a aplicação está funcionando. Ressalta-se que o servidor executa somente os serviços, o *Client App* deve ser executado manualmente pela classe *MainApp.java*.

3.3 TESTAR A FERRAMENTA RMT

O objetivo da realização dos testes é entender como a aplicação funciona sem que nenhuma modificação seja realizada. A ferramenta nesta etapa não pode ser modificada.

Para verificar o comportamento da aplicação foram escolhidos de forma aleatória projetos que já foram executados e testados por Beluzzo (2018), pois o mesmo testou 50 projetos em sua ferramenta e os separou em uma tabela contendo projetos com candidatos e outra sem candidatos. Sugere-se que sejam escolhidos de forma aleatória o mínimo de 3 (três) projetos que não possuem candidatos as refatorações e 3 (três) projetos que possuem candidatos a refatoração, esses projetos são Docker-java, J2V8, Jfairy.

Para iniciar o teste é necessário executar o servidor *WildFly*. Ressalta-se que o servidor demora em média 30 segundos para iniciar. Para que seja realizada a importação de um projeto a ferramenta RMT o *Client App* precisa ser executado. Não existe uma ordem correta para iniciar a ferramenta, podendo ligar o *Client App* antes

de se iniciar o servidor. Mas, para que se possa importar um projeto é necessário esperar a inicialização do servidor, caso o contrário o servidor retorna erros.

Os comportamentos que devem ser testados são:

- Importar o projeto: foi verificado que após a importação do projeto que o mesmo passa por um processo de análise e retorna candidatos a refatoração. Caso existam candidatos, pode-se selecioná-los. Os candidatos são exibidos em uma tabela *scrollable* e são visualizados utilizando o nome da classe candidata.
- Aplicar a refatoração: após a seleção dos candidatos a refatoração é possível aplicar as refatorações. Caso selecionada, uma interface gráfica é disponibilizada e solicita ao usuário o nome do arquivo .ZIP em que as classes refatoradas serão gravadas.

Vale considerar que os testes foram realizados para poder seguir o fluxo do aplicativo detalhado na seção 2.4.

3.4 ANALISAR O PROJETO DA FERRAMENTA RMT

Para que se possa iniciar a implementação de um novo método de detecção e inserção de padrões de projeto é necessário entender como a divisão de pacotes e classes é realizada na ferramenta RMT, para que as mudanças sejam realizadas de forma que não interfira nos métodos que já estão em funcionamento.

O projeto da ferramenta RMT é dividido em quatro subprojetos (Figura 10): *client-module, metrics-agent, intermediary-agent* e *detection-agent*. Dentre os subprojetos o *detection-agent* é o que deve ser modificado caso a abordagem de extensão deseje somente incluir novos métodos de refatoração, pois nele são encontrados pacotes para cada uma das abordagens de refatorações implementadas por Beluzzo (2018).

Os demais projetos funcionam de forma independente e podem ser alterados para adicionar novas funcionalidades, com por exemplo, caso uma nova métrica necessita ser adicionada ela deve ser adicionada ao projeto *metrics-agent* e no *client-module*, para ser exibido. O cálculo das métricas é realizado independente da refatoração realizada, ele é verificado a partir das mudanças realizadas no código e

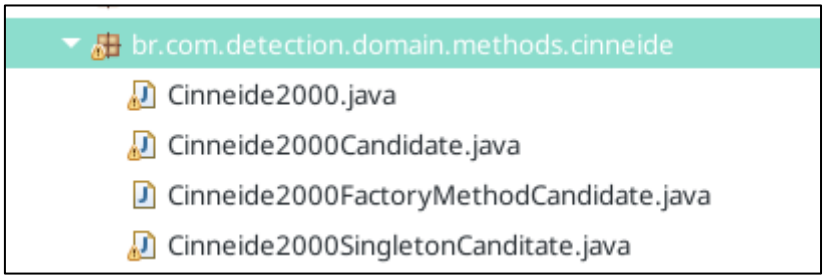
salvas no banco de dados, ou seja, nenhuma mudança precisa ser realizada. O mesmo se aplica para o *intermediary-agent*, o seu trabalho é enviar requisições para os demais serviços, por esse motivo nenhuma mudança precisa ser feita.

Figura 10 – Divisão de subprojetos

Fonte: Beluzzo(2018)

O pacote do *detection-agent* é dividido por funções e contém o nome dos autores, das classes dos candidatos a refatoração e suas implementações para cada padrão de projeto. A Figura 11 ilustra um exemplo deste pacote para o método Cinnèide (2000) que foi incorporado a RMT.

Figura 11 - Pacote do método de Cinnèide (2000)



Fonte: Autoria Própria

O pacote *verifiers* armazena as classes utilizadas para verificar se um projeto possui candidatos a refatoração, possuindo uma classe de verificação abstrata a qual é estendida para cada nova refatoração adicionada. O pacote *executers* contém a lógica da aplicação dos métodos da literatura para refatoração, essas separadas por padrões de projetos.

Além dos pacotes utilizados para a realização das refatorações, o projeto *detection-methods* possui pacotes de configurações e I/O como o pacote de *datastore* que se comunica com a *database* para alocar os projetos candidatos a refatoração. O pacote *boundaries* cuida dos *endpoints* da API. Esses pacotes são importantes para o funcionamento da aplicação, mas não precisam ser modificados para realizar a extensão da ferramenta, então não é necessário detalhá-los.

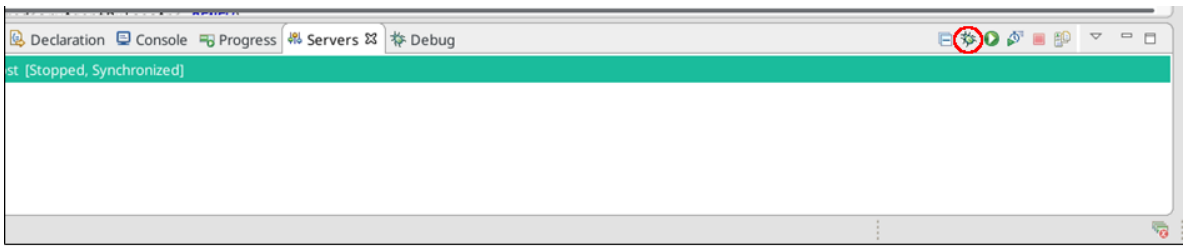
Os resultados obtidos na terceira etapa foram importantes para a realização da etapa de análise do fluxo da ferramenta, pois seus resultados tornam mais fácil a escolha do projeto.

3.5 ANALISAR O FLUXO DA FERRAMENTA

A análise de fluxo da ferramenta é a parte mais difícil dentre as etapas para o processo de extensão, pois é necessário percorrer por todas as chamadas da aplicação e é necessário entender como a comunicação de um objeto acontece com o outro.

Para poder identificar o fluxo é necessário realizar a análise de projeto da ferramenta que foi realizada por meio do *debug*. O processo de *debug* é feito após realizar a configuração do *WildFly* para suportar a depuração. O servidor pode ser iniciado em *debug mode* clicando no opção “Debug”, próximo ao botão de iniciar o servidor.

Figura 12 - Aba de servidores



Fonte: Autoria Própria

Com o *debug mode* ligado é possível se definir *breakpoints* no código, que são pontos atribuídos a linhas de códigos, a qual o desenvolvedor quer que a aplicação pare de executar automaticamente e sua execução comece a ser controlada pelo desenvolvedor. Nesse modo, é possível navegar na aplicação linha a linha observando o fluxo do programa, seguindo-o para dentro de instâncias e métodos executados em outras classes, mas sempre obedecendo a ordem de execução normal do programa. Para que se possa entender como o fluxo de dados está sendo realizado o *debug mode* disponibiliza um campo com as variáveis alocadas na memória e seus respectivos valores, os quais podem ser atualizados em cada execução de linha.

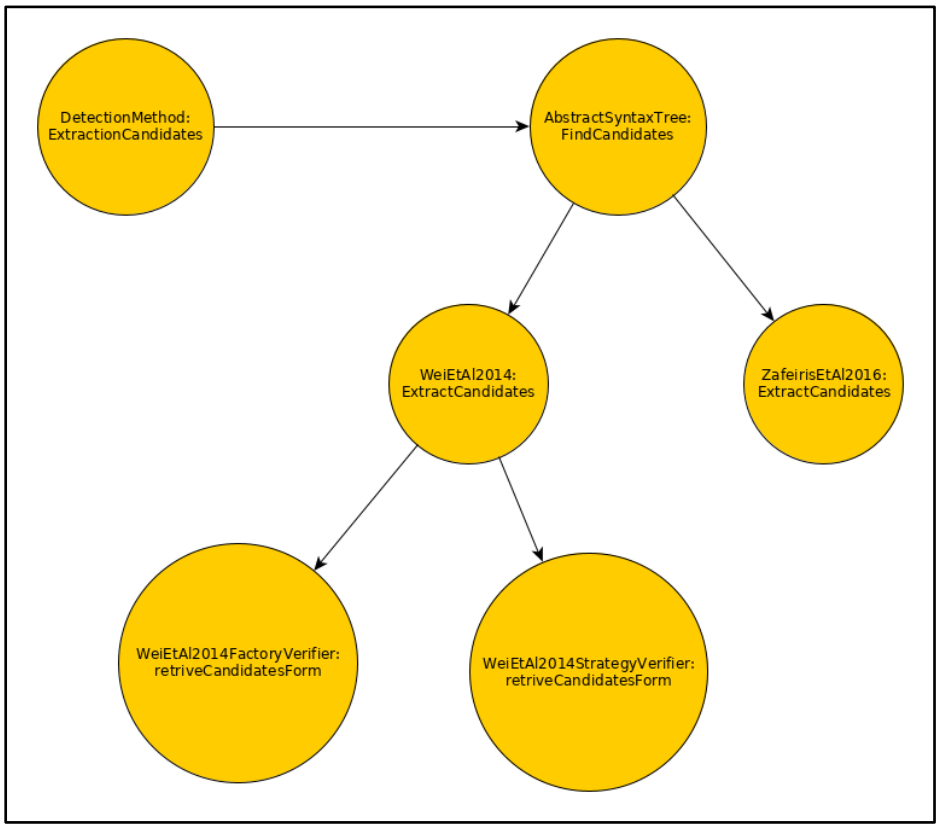
Para iniciar o *debug* é necessário utilizar um *breakpoint* definido na Classe *DetectorBoundary.java* dentro do método *requestEvaluation*. Esse método recebe uma requisição de iniciação da procura por candidatos a refatoração do *Client App*.

O *debug* foi útil para encontrar o início do fluxo até chegar a classe *DetectionMethod.java*, responsável por começar a chamada para análise dos projetos e por retornar o projeto candidato. Após entrar nessa classe o *debug* começou a ter um comportamento inesperado, parando de exibir o código da aplicação e exibindo uma tela branca, a qual não desapareceu mesmo continuando a execução do projeto até o seu término. Tendo esse problema a análise manual teve de ser utilizada, em que se leu o código da classe *DetectionMethod.java*. Com a leitura foi necessário verificar os métodos chamados pelas classes de forma manual para poder rastrear as trocas de mensagens entre os objetos.

Utilizando a análise manual de todos os métodos chamados dentro do código tiveram suas devidas implementações encontradas. Um grande problema encontrado foi quando haviam múltiplas chamadas de métodos seguidos, isso facilitava a perda nas chamadas, dificultando saber qual classe chamou qual método. Por esse motivo,

foi necessário criar grafos conforme exibe a Figura 13 que demonstra o fluxo de chamadas da aplicação.

Figura 13 - Grafo de chamada de métodos



Fonte: Autoria Própria

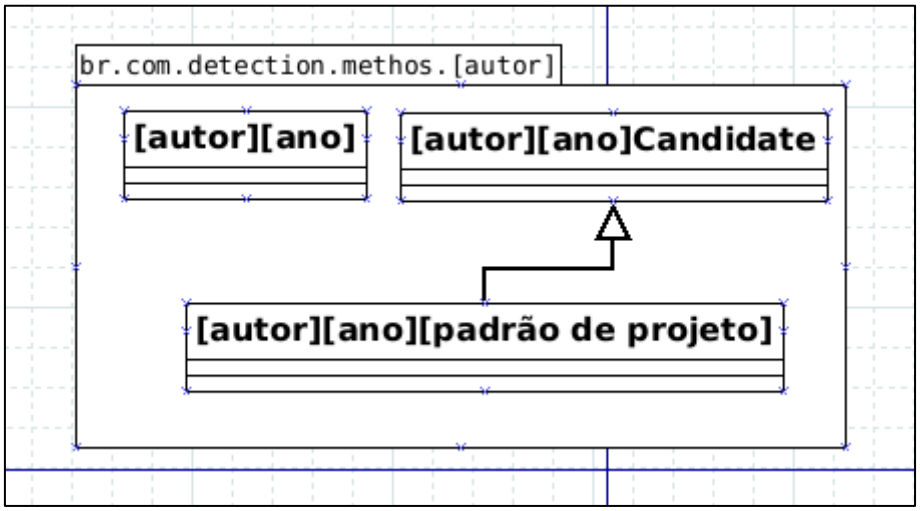
A criação do grafo apresentado na Figura 13 foi elaborado para identificar as chamadas e suas respectivas classes, para compreender a comunicação entre os objetos e também comentar métodos que precisam ser utilizados na implementação do novo método na ferramenta. Cada vértice do gráfico representa uma classe e o método o qual foi chamado pela classe anterior a ele. Apesar de ser um grafo direcionado, se a folha do grafo possuir uma função de retorno o grafo pode demonstrar esse fluxo adicionando arestas com o fluxo contrário para se ligar com o nível de cima.

Um problema encontrado na análise manual foi a falta de documentação do código, como por exemplo, a utilização de *Javadocs* para descrever a função principal do método. Para alguns métodos foram adicionados *Javadocs* para que se pudesse identificar suas funções.

Foram criados diagramas representando os quatro pacotes necessários para estender os novos métodos na ferramenta RMT. Toda abordagem necessita desses pacotes representados nas Figuras 14, 15, 16, mas ao mesmo tempo não significa que serão os únicos criados.

A figura 14 representa o pacote do autor, o qual contém a classe com nome e ano do trabalho realizado pelo autor. Nessa classe é armazenado o nome do artigo, em conjunto com o ano e os autores do artigo, também inclui as refatorações propostas pelo autor. As classes que identificam os candidatos a refatoração também são criadas nesse pacote, a classe pai possui *fields* para a identificação do candidato, como *CompilationUnit, path, package,* entre outros. Na classe filha é implementado métodos necessários e ou novos *fields*, dependendo da necessidade do novo método que será inserido.

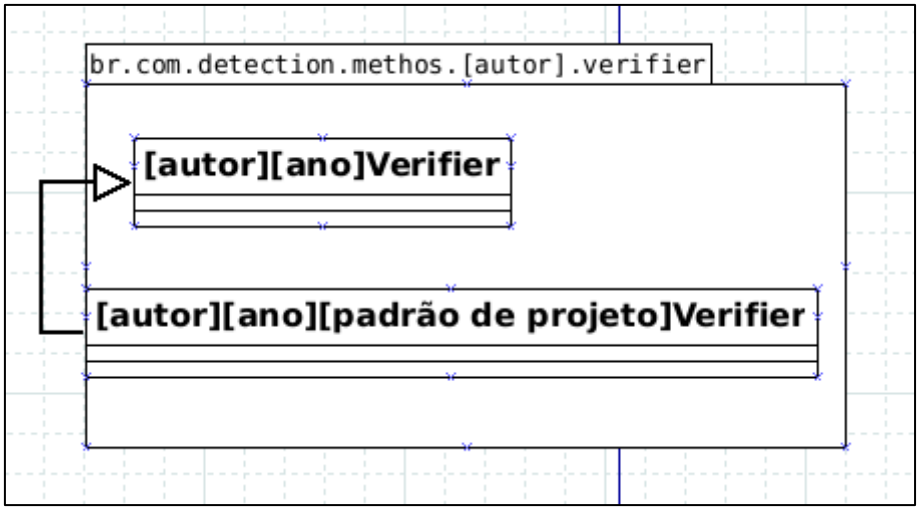
Figura 14 - Representação do pacote do autor do novo método



Fonte: Autoria Própria

A Figura 15 exibe o pacote de *verifiers*, o qual é composto pelo pai (classe abstrata), seus filhos e suas as implementações das verificações. Dentro da classe filha deve conter a implementação das funções que verificam o código-fonte em busca de refatorações. Essas funções devem retornar instâncias de candidatos para a classe pai a qual envia os candidatos para execução.

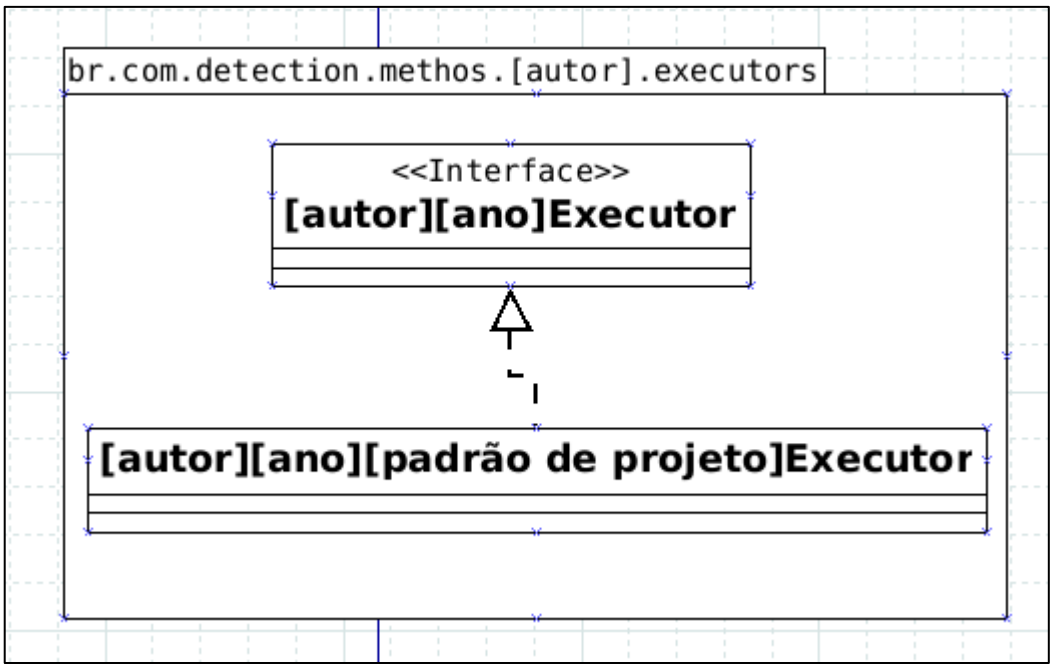
Figura 15 - Representação do pacote *verifier*



Fonte: Autoria Própria

A Figura 15 exibe o pacote de *executers* na qual as classes responsáveis por aplicar a refatoração devem ser armazenadas. O pacote é composto pela interface pai, e as que a implementam que são compostas com os métodos de execução da refatoração.

Figura 16 - Representação da classe *executers*



Fonte: Autoria Própria

3.6 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo relatou as etapas que devem ser seguidas para realizar a inserção de novos métodos na ferramenta RMT. As etapas propostas pela abordagem foram a instalação da ferramenta descrevendo os passos necessários para rodá-la; realização de testes iniciais para verificar se os resultados estão corretos; compreender a divisão de pacotes e de classes; e finalmente analisar como o fluxo da ferramenta se comporta para poder encontrar os pontos em que a implementação deve ser realizada.

As etapas representadas nas seções 3.4 e 3.5 são as principais para realizar a inserção de novos métodos de detecção e inserção de padrões de projeto da literatura na ferramenta RMT.

4 RESULTADOS

Seguindo as etapas da abordagem descrita no capítulo 3, este capítulo relata a inserção do método de refatoração baseado em padrões de projetos proposto por Cinnèide (2000) dentro da ferramenta RMT, bem como apresenta a execução de testes com o novo método inserido.

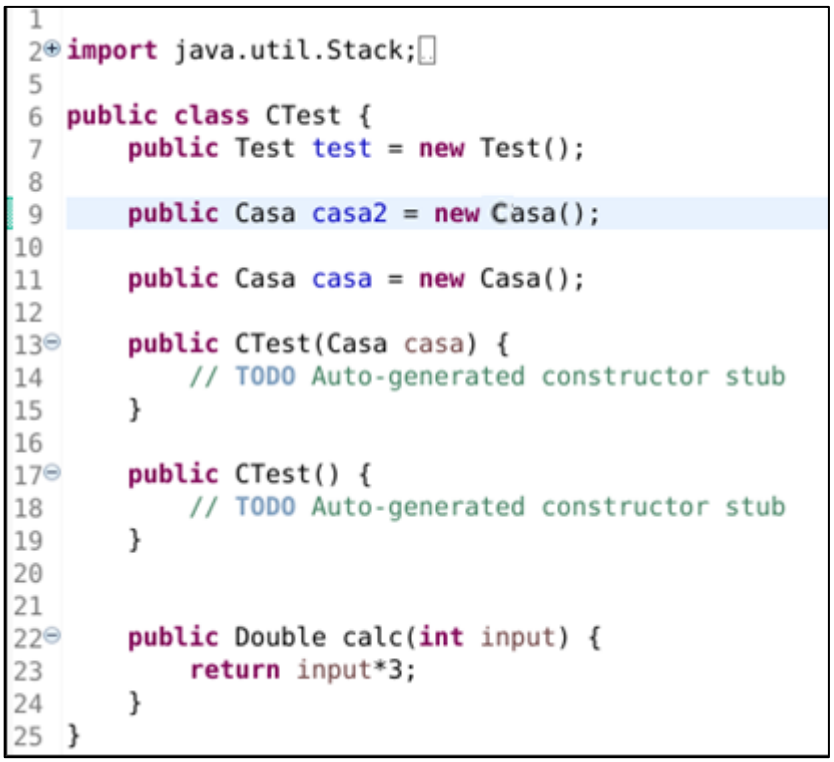
A seção 4.1 apresenta a implementação do método de Cinnèide (2000) sem incorporá-lo a ferramenta RMT. A seção 4.2 descreve como a abordagem de extensão foi aplicada na inserção do método de Cinnèide (2000). A seção 4.3 apresenta os resultados e testes da implementação do método de Cinnèide (2000) na ferramenta RMT. A seção 4.4 realiza uma análise da abordagem proposta. Por fim, a seção 4.5 relata as conclusões finais deste capítulo.

4.1 IMPLEMENTAÇÃO DO MÉTODO DE CINNÈIDE (2000)

Com o propósito de conhecer o método de Cinnèide (2000) foi necessário a leitura de seu trabalho e a implementação foi realizada em um projeto contendo três classes: uma para as implementações e duas classes de teste.

A primeira classe de teste chamada *Test* é uma classe vazia, somente com um construtor, essa classe é utilizada para testar o método *singleton* e *factory method.* Na segunda classe teste *CTest* é para verificar se ela é uma candidata a refatoração aos dois padrões citados.

Figura 17 - Classe de testa CTest



```
1
2  import java.util.Stack;
5
6  public class CTest {
7      public Test test = new Test();
8
9      public Casa casa2 = new Casa();
10
11     public Casa casa = new Casa();
12
13     public CTest(Casa casa) {
14         // TODO Auto-generated constructor stub
15     }
16
17     public CTest() {
18         // TODO Auto-generated constructor stub
19     }
20
21
22     public Double calc(int input) {
23         return input*3;
24     }
25 }
```

Fonte: Autoria Própria

A segunda classe *CTest* representada na figura 17, pois possui instância de duas classes, dois construtores e um método. A classe foi estruturada dessa maneira para poder testar as refatorações propostas (CINNÈIDE, 2000).

A classe *CTest* possui duas instâncias da classe *Casa* para poder testar o método utilizado para verificar candidatos ao padrão *singleton*, se deve ter uma instância de uma classe para ser eleito, a segunda instancia tem o objetivo de verificar se o contador está funcionando evitando a aplicação do método a classe Casa. A classe apresenta dois construtores, pois quando a refatoração para o padrão *Singleton é executada* é necessário verificar se a classe conte somente um construtor, caso contrário a refatoração é negada. O parâmetro foi adicionado a classe para verificar se estavam sendo passados para o método *getInstance()*. Da mesma forma o método na classe é utilizado para criar uma método abstrato.

As *minitransformations* foram implementadas no mesmo arquivo que constrói a árvore de sintaxe abstrata em forma de métodos. A Figura 18 ilustra a implementação da *minitransformations Abstract Access* mudando o tipo de *concrete* para *inf*.

Figura 18 - Implementação da *minitransformation Abstract Access*

```
package br.com.detection.domain.methods.cinneide.minitransformations;

import java.nio.file.Path;

public class AbstractAccess extends MinitransformationUitls {

    public void makeAbstractAccess(String clazz, ObjectCreationExpr concrete, String inf, DataHandler dataHandler,
            Collection<String> skipMethods) {
        final CompilationUnit cu = new CompilationUnit();
        final CompilationUnit baseCu = (CompilationUnit) dataHandler.getParsedFileByName(clazz);
        final Path file = dataHandler.getFile(baseCu);
        final ClassOrInterfaceDeclaration newInterface = cu.addClass(clazz);

        baseCu.findAll(FieldDeclaration.class).forEach(o -> {
            for (String skipMethod : skipMethods) {
                if (o.getElementType().toString().equals(concrete.getTypeAsString()) && !o.getElementType().toString().e
                    o.getVariables().forEach(v -> v.setType(inf));
                }
            }
        });

        this.writeChanges(baseCu, file);
    }
}
```

Fonte: Autoria Própria

A Figura 19 representa a implementação do padrão *Encapsulate Construction* o qual cria um método com os mesmos parâmetros e o mesmo corpo do construtor da classe *product* e substitui as instâncias da classe pelo método criado.

Figura 19 - Implementação da *minitransformation Encapsulate Construction*

```
1  package br.com.detection.domain.methods.cinneide.minitransformations;
2
3  import java.nio.file.Path;
15
16 public class EncapsulateConstruction extends MinitransformationUitls {
17
18     public void makeEncapsulateConstruction(String creator, String product, String createP, DataHandler dataHandler) {
19         final CompilationUnit creatorCu = (CompilationUnit) dataHandler.getParsedFileByName(creator);
20         final CompilationUnit productCu = (CompilationUnit) dataHandler.getParsedFileByName(product);
21         final Path file = dataHandler.getFile(creatorCu);
22
23         final ClassOrInterfaceDeclaration creatorClass = creatorCu.findFirst(ClassOrInterfaceDeclaration.class).get();
24
25         productCu.findAll(ConstructorDeclaration.class).forEach(c -> {
26             MethodDeclaration m = makeAbstract(c, createP);
27             creatorClass.addMember(m);
28         });
29
30         creatorCu.findAll(FieldDeclaration.class).forEach(e -> {
31             if (e.getElementType().toString().equals(product)
32                     && !e.getVariable(0).getInitializer().get().getChildNodes().get(0).toString().equals(createP)) {
33                 e.getVariable(0).setInitializer(createP);
34             }
35         });
36         this.writeChanges(creatorCu, file);
37     }
38 }
39
```

Fonte: Autoria Própria

A Figura 20 representa a implementação da *minitransformation partial abstraction* a qual constrói uma classe abstrata da classe *clazz*, criando todos os métodos da classe como abstratos e estendendo a nova classe na *clazz*.

Figura 20 - Implementação da minitransformation *Partial Abstraction*

```java
package br.com.detection.domain.methods.cinneide.minitransformations;

import java.nio.file.Path;

public class PartialAbstraction extends MinitransformationUitls {

    public void makePartialAbstraction(String clazz, DataHandler dataHandler) {
        final CompilationUnit cu = new CompilationUnit();
        final CompilationUnit baseCu = (CompilationUnit) dataHandler.getParsedFileByName(clazz);
        final Path file = dataHandler.getFile(baseCu);
        String newClassName = clazz;
        final String abstractClassName = String.format("Abstract%s", newClassName);
        ClassOrInterfaceDeclaration newClass = cu.addClass(abstractClassName);
        newClass.setAbstract(true);

        baseCu.findAll(MethodDeclaration.class).forEach(m -> {
            if (m.isAbstract()) {
                newClass.addMethod(m.getNameAsString()).setAbstract(true);
            }
        });

        baseCu.findAll(MethodDeclaration.class).forEach(m -> {
            if (!m.isAbstract()) {
                newClass.addMethod(m.getNameAsString()).setBody(m.getBody().get());
                m.remove();
            }
        });

        baseCu.findAll(ClassOrInterfaceDeclaration.class).forEach(c -> c.addExtendedType(newClass.getNameAsString()));

        this.writeChanges(cu, file.getParent().resolve(String.format("%s.java", abstractClassName)));
        this.writeChanges(baseCu, file);
    }

}
```

Fonte: Autoria Própria

Após a realização dos testes, a implementação foi inserida na ferramenta RMT, em suas respectivas classes.

4.2 APLICAÇÃO DA ABORDAGEM DE EXTENSÃO PROPOSTA

Para aplicar a abordagem de extensão proposta neste trabalho foram seguidas as etapas detalhadas no capítulo 3. A primeira etapa foi a instalação da ferramenta. Os programas requisitados foram baixados e instalados no sistema operacional *Manjaro Linux*. A única dificuldade encontrada foi na importação das *builds* para o servidor *WildFly* onde é necessário importar o projeto antes de adicionar as *builds* ao *server*. Após a solução desse problema, a ferramenta funcionou sem erros.

A segunda etapa de validação da ferramenta foi realizada por meio de testes com os mesmos 50 projetos utilizados por Beluzzo (2018). Os resultados foram similares, acredita-se que não foram idênticos porque os projetos utilizados estão disponíveis na plataforma *GitHub* e podem ter sido modificados para reparar problemas ou adicionar requisitos.

A terceira etapa foi analisar o projeto da ferramenta RMT, o que demandou maior tempo para compreensão por se tratar de uma arquitetura dividida em 3 projetos os quais se comunicam por meio de uma API.

Todos os pacotes foram analisados, mas o pacote “br.com.detection.domian.methods.[nome do autor]” chamaram atenção por possuirem nome dos autores citados na implementação no artigo (BELUZZO, 2018). A classe com a *keyword candidate* representam instâncias dos candidatos de refatoração, *keyword verification* representam instâncias para a verificação de possíveis candidatos, *keyworkd executors* representam as classes que executam a refatoração aos candidatos.

A etapa de validação da ferramenta é abstrata pois ela é utilizada para compreender como o desenvolvedor dividiu seu código, tentando encontrar padrões. Não foi especificado uma forma para rastrear o projeto, pois qualquer informação obtida se tornou importante na realização da próxima etapa.

A análise iniciou com a execução do *WildFly* em modo de *debbug* para poder executar o código linha por linha e um *breakpoint* foi adicionado ao arquivo *DetectorBoundary* responsável por receber as requisições do *Intermediary-Service*. A partir desse ponto as instruções foram seguidas linha por linha identificando as chamadas e as escrevendo no grafo de fluxo.

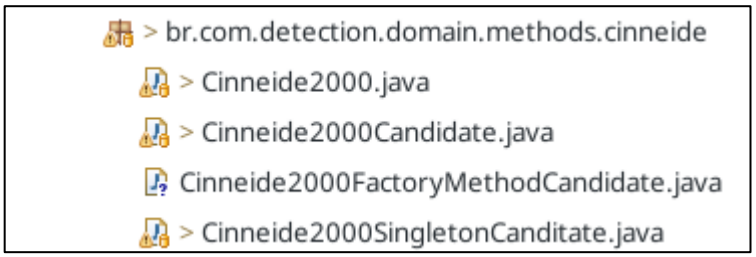
A princípio o processo de *debugging* funcionou até o momento que passou a não mostrar mais o código que estava sendo analisado. Outros *breakpoints* foram colocados em vários arquivos para tentar solucionar o problema, mas não se obteve êxito. Várias dificuldades foram encontradas principalmente quando os métodos foram chamados por meio de reflexão computacional a qual demorou para ser encontrada. Por isto, passou-se a utilizar a forma manual. A reflexão computacional é a capacidade de um programa se observar e modificar em tempo de execução.

O projeto da ferramenta é contém 188 classes, o *detection-service* possui 50 classes. O problema começa quando é necessário entender o funcionamento dos métodos das classes, os quais são de melhor compreensão se comentários no estilo *Javadocs* ou somente comentários na linha da classe fosse encontrado. Por isto, recomenda-se colocar *JavaDocs* ao projeto para que se possa incentivar as pessoas utilizarem a ferramenta.

A Figura 21 representa as classes do pacote com o nome do autor, nesse caso Cinnèide. A classe *Cinneide2000* contém as informações sobre o autor e o artigo,

como nome do autor, data da publicação, padrões de projetos, título do artigo. A classe *candidate* é a classe que guarda as informações necessárias para criar um candidato a refatoração. Nas classes *Factory Method* e *Singleton* são adicionadas a parte específica de cada candidato, como novos parâmetros para o construtor ou implementação de novos métodos.

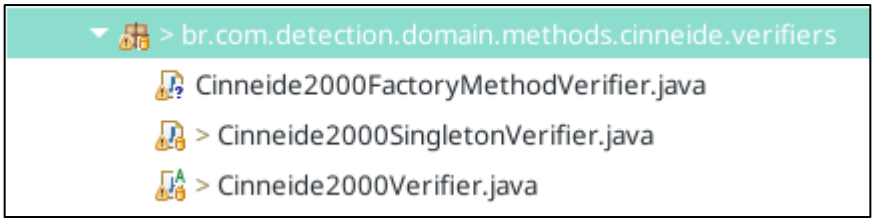
Figura 21 - Pacote definindo candidatos de Cinnèide (2000)



Fonte: Autoria Própria

A Figura 22 representa as classes do pacote *verifiers* do autor, nesse caso *Cinneide.verifiers*. A classe *Cinneide2000Verifier* é a classe abstrata que implementa as funções para localizar os candidatos, já as classes filhas implementam a regra de para decidir se a mesma a ser analisada é um candidato ou não a refatoração.

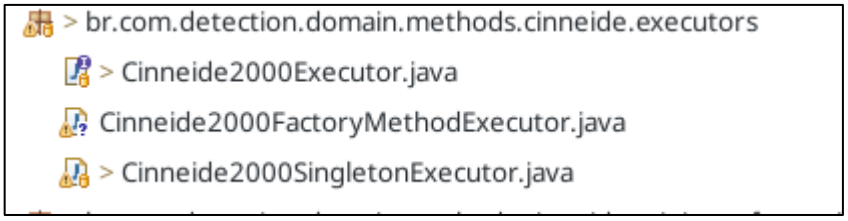
Figura 22- Pacote de *verifiers* Cinnèide (2000)



Fonte: Autoria Própria

A Figura 23 representa as classes do pacote *executors* do autor, nesse caso *Cinneide.executors* na qual uma interface define as classes a serem implementadas nos filhos que implementam a regra para realizar a refatoração dos candidatos.

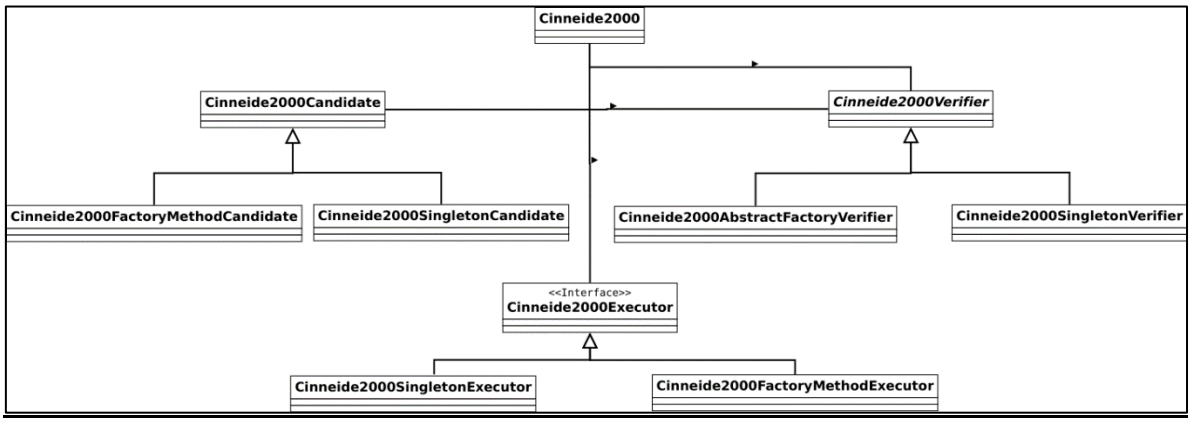
Figura 23 - Pacote de *Executors* Cinnèide (2000)



Fonte: Autoria Própria

A Figura 24 exibe o diagrama de classes citadas nas descrições dos pacotes das Figuras 21 até a 23. Essas classes foram utilizadas para realizar extensão da ferramenta RMT adicionando o método de Cinnèide (2000).

Figura 24 - Diagrama de classe da implementação do método de Cinnèide (2000)



Fonte: Autoria Própria

4.3 TESTES DA FERRAMENTA RMT COM O MÉTODO DE CINNÈIDE

Para a realização de testes foram utilizados projetos *open source*. Como não existem exemplos para serem testados na tese de Cinnèide (2000), não há como se obter confirmação de que a proposta abordada nestes trabalhos tem o mesmo comportamento que a desenvolvida pelo autor.

Na avaliação da nova versão da RMT foram utilizados trabalhos retirados da plataforma *GitHub* (GITHUB, 2019) que permite a criação de repositórios públicos e privados para a gestão de projetos. Grandes empresas como *Microsoft*, *Google* e *Docker* o utilizam, disponibilizando aplicações *open souce* para a comunidade. Os

trabalhos foram selecionados por Beluzzo (2018) aleatoriamente, somente filtrando a pesquisa pela linguagem de programação Java e foram estes utilizados para testes.

Todos os projetos foram submetidos a avaliação da nova versão da ferramenta. Na Tabela 1 são listados os projetos utilizados para testes. Os mesmos estão divididos em 3 colunas, a primeira possui sequência numérica e referenciados com a letra “P” de projeto, a segunda é composta pelo *link* para os repositórios dos projetos, e a terceira a quantidade de classes contidas no projeto.

Tabela 1 – Projetos de Cenários de Testes

Cód	Repositório	Nº de Classes
P1	https://github.com/happyfish100/fastdfs-client-java.git	34
P2	https://github.com/kungfoo/geohash-java.git	24
P3	https://github.com/soundcloud/java-api-wrapper.git	38
P4	https://github.com/Froussios/Intro-To-RxJava.git	93
P5	https://github.com/eclipsesource/J2V8.git	99
P6	https://github.com/lzyzsd/JsBridge.git	9
P7	https://github.com/udacity/Just-Java.git	1
P8	https://github.com/shyiko/mysql-binlog-connector-java.git	97
P9	https://github.com/opentracing/opentracing-java.git	74
P10	https://github.com/kpelykh/docker-java.git	31
P11	https://github.com/firebase/firebase-admin-java.git	358
P12	https://github.com/codeborne/selenide.git	407
P13	https://github.com/sendgrid/sendgrid-java.git	113
P14	https://github.com/dawei101/shadowsocks-android-java.git	40
P15	https://github.com/swagger-api/swagger-parser.git	152
P 16	https://github.com/twilio/twilio-java.git	1499
P 17	https://github.com/pedrovgs/Algorithms.git	166
P18	https://github.com/isee15/captcha-ocr.git	21
P19	https://github.com/cucumber/cucumber-java-skeleton.git	3
P20	https://github.com/xpinjection/java8-misuses.git	52
P21	https://github.com/MagnusS/Java-BloomFilter.git	3
P22	https://github.com/dmpe/JavaFX.git	32
P23	https://github.com/DomHeal/JavaFX-Chat.git	32
P24	https://github.com/scream3r/java-simple-serial-connector.git	7
P25	https://github.com/jcip/jcip.github.com.git	144
P26	https://github.com/etcd-io/jetcd.git	141
P27	https://github.com/jsonld-java/jsonld-java.git	40
P28	https://github.com/bwaldvogel/liblinear-java.git	31

Tabela 1- Projetos de Cenários de Testes

(Continua)

Cód	Repositório	Nº de Classes
P29	https://github.com/jasonross/Nuwa.git	8
P30	https://github.com/taskadapter/redmine-java-api.git	138
P31	https://github.com/wg/scrypt.git	19
P32	https://github.com/oxo42/stateless4j.git	45
P33	https://github.com/soundcloud/java-api-wrapper.git	38
P34	https://github.com/evant/gradle-retrolambda.git	23
P35	https://github.com/Progether/JAdventure.git	61
P37	https://github.com/google/caliper.git	291
P48	https://github.com/joscha/play-authenticate.git	369
P49	https://github.com/caprica/vlcj.git	469
P50	https://github.com/careermonk/DataStructureAndAlgorithmsMadeEasyInJava.git	160

Fonte: Beluzzo (2018)

Dentre os projetos da Tabela 1 que contém candidatos a refatoração para o padrão *Singleton* foram transferidas para a Tabela 2. Isto foi necessário para testar o novo método (CINNÈIDE, 2000) inserido na ferramenta detectada o padrão *Singleton*. Na primeira coluna são informados os códigos referentes aos projetos, na segunda coluna está disponível o *link* dos repositórios de cada projeto, a terceira coluna contém a quantidade de classes candidatas a serem refatoradas.

Tabela 2 – Projetos que possuem candidatos a refatoração com o padrão *Singleton*

Cód	Repositório	Candidatos
P1	https://github.com/happyfish100/fastdfs-client-java.git	4
P2	https://github.com/kungfoo/geohash-java.git	3
P8	https://github.com/shyiko/mysql-binlog-connector-java.git	21
P10	https://github.com/kpelykh/docker-java.git	2
P15	https://github.com/swagger-api/swagger-parser.git	11
P16	https://github.com/twilio/twilio-java.git	15
P17	https://github.com/pedrovgs/Algorithms.git	41
P19	https://github.com/cucumber/cucumber-java-skeleton.git	1
P20	https://github.com/xpinjection/java8-misuses.git	4
P22	https://github.com/dmpe/JavaFX.git	3
P23	https://github.com/DomHeal/JavaFX-Chat.git	3
P25	https://github.com/jcip/jcip.github.com.git	7
P26	https://github.com/etcd-io/jetcd.git	15
P27	https://github.com/jsonld-java/jsonld-java.git	9
P31	https://github.com/wg/scrypt.git	3

Tabela 2 – Projetos que possuem candidatos a refatoração com o padrão *Singleton*

(Continua)

Cód	Repositório	Candidatos
P32	https://github.com/oxo42/stateless4j.git	1
P33	https://github.com/soundcloud/java-api-wrapper.git	8
P34	https://github.com/evant/gradle-retrolambda.git	3
P38	https://github.com/Devskiller/jfairy.git	24
P39	https://github.com/jpush/jpush-api-java-client.git	28
P42	https://github.com/raml-org/raml-java-parser	78
P44	https://github.com/maxmind/geoip-api-java.git	2
P45	https://github.com/kwhat/jnativehook.git	4
P46	https://github.com/cardillo/joinery.git	3
P48	https://github.com/joscha/play-authenticate.git	16
P49	https://github.com/caprica/vlcj.git	44
P50	https://github.com/careermonk/DataStructureAndAlgorithmsMadeEasyInJava.git	13

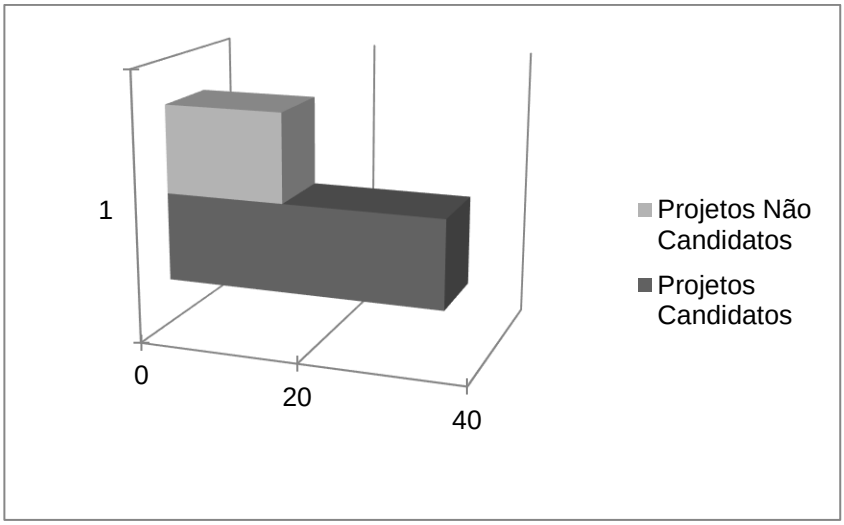
Fonte: Autoria Própria

Os resultados apresentados são referentes ao padrão *Singleton*, apesar dele não ter sido o único a ser implementado. A implementação de um segundo padrão foi realizado, o *factory method*, mas esse não apresenta resultado por causa de sua geração de candidatos ser genérica, onde qualquer classe que possui uma instancia de outra classe é um candidato a refatoração. Isso fez com que a ferramenta RMT parasse de executar e sugere-se que sejam realizadas pesquisas futuras sobre como distribuir o processamento de novos métodos inseridos a RMT de forma que a execução ocorra em um menor tempo.

Um dado constatado é que a RMT refatora todos os candidatos para poder exibir as métricas para o usuário. Quando as refatorações estavam sendo realizados os oito gigas de memória RAM do computador que se estava utilizando para os testes não aguentava e o mesmo parava de funcionar.

O Gráfico 1 descreve a quantidade de projetos que tiveram candidatos a refatoração, sendo um total 31 projetos com candidatos a refatoração para o padrão *Singleton* dos 50 projetos testados.

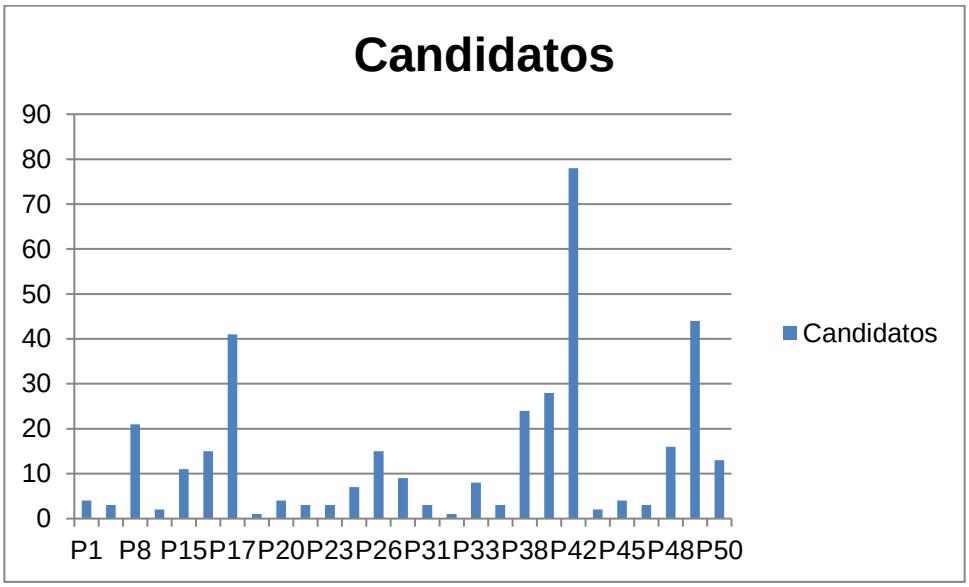
Gráfico 1 – Projetos Com ou Sem Candidatos a Refatoração



Fonte: Autoria Própria

O Gráfico 2 mostra a quantidade de candidatos a refatoração por projeto, sendo 28 candidatos e 22 não candidatos. Todos os candidatos listados são candidatos a refatoração para o padrão *Singleton*. As classes exibidas na aplicação são referentes as classes as quais está sendo instanciada e se tornará *Singleton,* caso o usuário deseje.

Gráfico 2 - Quantidade candidatos projeto



Fonte: Autoria Própria

Além de demonstrar quais classes podem ser refatoradas, a implementação calcula em porcentagem as mudanças feitas no código. As métricas calculadas para o projeto P1 estão exibidas na tabela 3.

Tabela 3 – Projetos com candidatos a refatoração e avaliação de métricas

Cód	Classe	Manutenibilidade	Confiabilidade	Reusabilidade
P1	StorageClient1	-2,83	-5,4	-0,54
P1	DownloadFileWritter	0,39	-0,58	1
P1	StoragteClient	-8,43	-13,8	-3,86
P1	TrackerGroup	0,12	-0,99	0,8
P1	GeoHashCircleQuery	0,12	-1,9	1,46
P1	BoundingBoxGeoHashterator	0,32	-1,16	1,62
P1	BoundingBoxSamper	0,34	-1,58	1,5

Fonte: Autoria Própria

Para se considerar a qualidade da refatoração a ser realizada é necessário verificar os valores das métricas, as quais são representadas em porcentagem. Quando a porcentagem é positiva indica que a refatoração trará benefícios ao código, e quando é negativa indica que se que a modificação causara uma piora as métricas e assim ao código.

Na análise das avalições foi constatado que a maioria das refatorações impacta de forma negativa na confiabilidade do código. A análise geral foi realizada em um conjunto de 10 projetos selecionados considerando a sua magnitude, pois foi percebido que quanto maior o projeto mais candidatos existiam, pois o método de seleção utilizado por Cinnèide (2000) é muito genérico.

Os projetos analisados foram P2 ,P6 ,P10 ,P19 ,P20 ,P22 ,P25 , P24 ,P29, P32 ,P39 e P44. Com a análise desses dados foi possível perceber que 58% das métricas de manutenibilidade são positivas demostrando que a aplicação do padrão de projeto será vantajosa em mais da metade dos casos.

Já a confiabilidade possui 1% das métricas positivas demonstrando a perda de confiabilidade no software. Essa taxa é preocupante, pois em praticamente todas refatorações realizadas a confiabilidade irá diminuir em média 7,5%. Por outro lado, a reusabilidade apresenta um número positivo expressivo, em que 98% das

refatorações causaram uma mudança positiva na métrica apresentando uma média de 1,5% de melhora no código.

4.4 ANÁLISE DA ABORDAGEM PROPOSTA

A abordagem permite que novos alunos ou desenvolvedores com interesse em implementar novos métodos na ferramenta RMT possam realizar a inclusão por meio de etapas que identificam onde as modificações devem ser aplicadas, deixando o foco na compreensão e implementação do novo método à ferramenta.

Um problema que pode acontecer com a instalação da ferramenta é que a mesma depende do sistema operacional. Isto pode ocasionar problemas na execução do software mesmo utilizando as mesmas versões dos programas necessárias para a RMT rodar.

Um ponto a ser abordado é o final da aplicação da terceira etapa (Analisar o projeto da ferramenta), pois não existe definição de ponto de parada, o que pode tornar esta etapa longa, mas ao mesmo tempo permite ao desenvolvedor escolher a sua necessidade.

4.5 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou a aplicação da abordagem de extensão utilizada na inserção do método de Cinnèide (2000) na ferramenta RMT.

A extensão conta com a implementação do padrão *Singleton* para a coleção de refatorações da ferramenta. A nova refatoração foi testada com a mesma base de projetos utilizada por (BELUZZO 2018). Os testes foram realizados em 50 projetos *open source*.

Os testes incluíram encontrar candidatos a refatoração e também refatorá-los. Essa análise foi feita manualmente selecionando projeto por projeto e guardando as informações de quantos candidatos a refatoração foram encontrados por projeto. Para todos os projetos o fluxo foi o mesmo, importar o projeto verificar se foram obtidos candidatos a refatoração. Caso sim, a mesma foi aplicada e um novo projeto refatorado é gerado.

5 CONCLUSÃO

Este trabalho propôs uma abordagem para extensão da ferramenta RMT que pode ser usada por alunos ou desenvolvedores a fim de facilitar a incorporação de novos métodos de detecção e inserção de padrões de projeto a ela tornando uma ferramenta mais completa.

A ferramenta RMT foi analisada procurando pontos de extensão. A análise foi dividida em 4 passos sendo eles: Instalar a ferramenta, Testar, Analisar o projeto da ferramenta e Analisar o fluxo da ferramenta.

Seguindo as etapas da abordagem proposta foi possível realizar a implementação do método de Cinnèide (2000) à RMT. O novo método foi testado utilizando os mesmos projetos de Beluzzo (2018).

Dentre as contribuições apresentadas nesta pesquisa, têm-se a criação de uma abordagem de extensão para uma ferramenta de refatoração de software e a incorporação do método de *minitransformations* criado por Cinnèide (2000).

Uma das dificuldades encontrada foi testar a implementação da refatoração, pois o artigo de Cinnèide (2000) não possui exemplos de testes para serem aplicados na ferramenta e assim permitir a comparação de resultados. Outra dificuldade foi entender o método de detecção de padrões de projetos criado pelo autor o qual o descreve de forma breve.

5.1 TRABALHOS FUTUROS

Algumas propostas para continuação do trabalho são:

- Implementar todas as refatorações de padrões de projetos propostos por Cinnèide (2000).
- Testar a abordagem na incorporação de novos métodos a ferramenta RMT identificando possíveis falhas ou benefícios.
- Distribuir o processamento de novos métodos inseridos à RMT melhorando seu desempenho.

REFERÊNCIAS

BELUZZO, L. B**. Abordagem para avaliar e detectar pontos de inserção e aplicação de padrões de projeto em código-fonte.** 2018. 101f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

CINNÈIDE, M. Ó. **Automated application of design patterns:** a refactoring approach. 2001. 242 f. Thesis (Doutorado) — Programa de Pós-Graduação University of Dublin, Trinity College Dublin, 2001.

CINNÈIDE, M. Ó. Automated refactoring to introduce design patterns. In: ACM. **Proceedings of the 22nd international conference on Software engineering**. 2000. p. 722–724.

CK GITHUB. Disponível em: <https://github.com/mauricioaniche/ck/>. Acesso em: 20 jun. 2019.

DEB, K. *et al*. A fast and elitist multiobjective genetic algorithm: NSGA-II. **IEEE Transactions on Evolutionary Computation**, v. 6, n. 2, p. 182–197, abr. 2002.

FERENC, R. *et al*. Columbus - reverse engineering tool and schema for C++. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Montreal. **Proceedings**… Montreal(Canadá) IEEE Comput. Soc, 2002.

FIELDS, J.; HARVIE, S.; FOWLER, M. **Refactoring : Ruby edition**. Addison-Wesley Professional: Boston, 2009.

FOWLER, M. **Refactoring:** Improving the Design of Existing Code. Addison-Wesley Professional: Boston, 2018.

GAMMA, E. *et al*. **Design Patterns Elements of Reusable Object-Oriented Software**. Addison Wesley: Boston, 2009.

GE, X.; DUBOSE, Q. L.; MURPHY-HILL, E. Reconciling manual and automatic refactoring. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), Zurich. **Proceedings**... Zurich(Switzerland) IEEE, 2012, p. 211-221.

GE, X.; MURPHY-HILL, E. Manual refactoring changes with automated refactoring validation. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York. **Proceedings**… New York (USA): ACM Press, 2014, p. 1095-1105.

GITHUB. Disponível em: <https://github.com/>. Acesso em: 20 set. 2019.

JAVA. Disponível em: <https://java.com/>. Acesso em: 10 out. 2019.

JAVAPARSER. Disponível em: <http://javaparser.org/>. Acesso em: 10 out. 2019.

KERIEVSKY, J. **Refactoring To Patterns. First.** Addison Wesley: Boston, 2004.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. 2007.

KORMAN, W.; GRISWOLD, W. **Elbereth: Tool support for refactoring Java programs**.1998. 48 f. Thesis(Mestrado) - Programa de pos graduação University of California, 1998.

LIU, W. *et al*. Automated pattern-directed refactoring for complex conditional statements. **J. Cent. South Univ**, v. 21, 2014.

MARTIN FOWLER. **Catalog of Refactorings**. Disponível em: <https://refactoring.com/catalog/>. Acesso em: 7 jun. 2019.

MENS, T.; TOURWE, T. A survey of software refactoring. **IEEE Transactions on Software Engineering**, v. 30, n. 2, p. 126–139, fev. 2004.

MOOR IVAN. Automatic inheritance hierarchy restructuring and method refactoring. In: OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES & APPLICATIONS,San Jose. **Proceedings**… San Jose(USA): ACM SIGPLAN Notices, 1996, p. 235–250.

MURPHY-HILL, E.; BLACK, A. P. Breaking the barriers to successful refactoring. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York. **Proceedings**... New York(USA): ACM Press, 2008, p. 421-430.

OUNI, A. *et al*. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. **Journal of Software: Evolution and Process**, v. 29, n. 5, p. e1843, maio 2017.

RANI, A.; KAUR, H.; PROF, A. Detection of Bad Smells in Source Code According To Their Object Oriented Metrics**. International Journal for Technological Research in Engineering**. Disponível em: <www.ijtre.com>. Acesso em: 12 jul. 2019.

Sangeetha, M.; Chandrasekar, C. An empirical investigation into code smells rectifications through ADA_BOOSTER. **Ain Shams Engineering Journal**, 2019.

SZŐKE, G. *et al*. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. **Journal of Systems and Software**, v. 129, p. 107–126, 1 jul. 2017

WILDFLY. Disponível em: < https://wildfly.org/>. Acesso em: 8 nov. 2019.

WILKING, D. *et al*. An Empirical Evaluation of Refactoring. **e-Informatica Software Engineering Journal**, v. 1, n. 1, p. 27–42, 2017.

ZAFEIRIS, V. E. *et al*. Automated refactoring of super-class method invocations to the Template Method design pattern. **Information and Software Technology**, v. 82, p. 19–35, 1 fev. 2017.