Ministério da Educação
Universidade Tecnológica Federal do Paraná
Pró-Reitoria de Pesquisa e Pós-Graduação

RELATÓRIO FINAL DE ATIVIDADES DE INICIAÇÃO CIENTÍFICA

<Proposta de Melhorias para a Ferramenta RMT>

vinculado ao projeto

<ARCHMAPE: Criação de uma arquitetura integrada de refatoração de código-fonte baseada em padrões de projeto e metapadrões>

<Carlos Eduardo Rodrigues Cardoso>
< Bolsista Fundação Araucária>
<Ciência da Computação>
Data de ingresso no programa: 10/2021
Prof(ª). Dr(ª). <Simone Nasser Matos>

Área do Conhecimento: Ciência Exata e da Terra – Sistemas de Computação

CAMPUS PONTA GROSSA, <2021>

CARLOS EDUARDO RODRIGUES CARDOSO
SIMONE NASSER MATOS

PROPOSTA DE MELHORIAS A FERRAMENTA RMT

Relatório de Pesquisa do Programa de Iniciação Científica da Universidade Tecnológica Federal do Paraná.

PONTA GROSSA, 2021.

SUMÁRIO

INTRODUÇÃO

A refatoração de um código-fonte está relacionada a ação de reorganizar estruturalmente trechos de código de um sistema, sem modificar seu funcionamento externo. Segundo Fowler et al. [1] é importante refatorar para que as atividades como compreender o código ou inserir uma nova funcionalidade sejam executadas com baixo teor de dificuldade.

O processo de refatorar o código pode ser realizado manualmente por um programador ou de maneira automatizada por meio de programas que detectam pontos chaves dentro do código-fonte que necessitam de alteração e posteriormente efetua a mudança necessária. Para que o processo seja executado, a refatoração pode seguir o princípio de técnicas ou baseada em padrões de projetos. Os padrões possuem categorias como: criacionais, estruturais e comportamentais e são descritos por Gamma et al. [2]. A utilização de padrões de projeto define uma linguagem comum de interação, possibilitando uma comunicação mais eficiente entre os membros da equipe durante o desenvolvimento do projeto [2].

Um software que é programado sem a preocupação de como o código-fonte está organizado torna complexa a sua manutenibilidade e isso acarreta na diminuição do nível de qualidade atribuído ao projeto.

O intuito da refatoração é tornar o código mais flexível eliminando fatos como duplicação de linha de código, redundância de informação, métodos e classes mal estruturadas. Alguns atributos são utilizados para a avaliação da qualidade da refatoração tais como robustez, reutilização, extensibilidade e desempenho como características para a análise [3].

A partir do conhecimento de qual padrão é aplicado no projeto, informações como propósito do padrão, contexto do problema encarado, ideia de solução, forma de implementação e impacto de utilização ficam subentendidas pois cada padrão possui um cenário específico de aplicação. Para aplicar um padrão ao sistema é necessário compreender bem o contexto do domínio do seu problema pois utilizar o padrão de projeto onde não tem necessidade irá aumentar de maneira desnecessária a complexidade do sistema sem trazer benefício algum [4].

No livro “Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos”, Gamma et al [2] descrevem 23 padrões de projeto que são soluções gerais para problemas recorrentes utilizando orientação a objeto. Com o passar do tempo outros padrões foram descobertos e descritos na literatura por diversos autores.

Autores como Fowler et al. [1] e Gaitani et al. [6] desenvolveram métodos de inserção de objeto nulo em um determinado sistema, aplicando o conceito de refatoração de código fonte. O *NULL Object* tem a finalidade de encapsular a ausência de um objeto, utilizando objeto alternativo que possui um comportamento padrão (*default*) ou apenas o comportamento de “não fazer nada” [7]. Para detectar a ausência de objeto é utilizada condicionais de verificação nulas.

O padrão de projeto *NULL Object* possibilita a substituição de verificações de nulos por invocação de método polimórficos que são vinculados a um objeto real ou a um objeto nulo [6]. Esses métodos polimórficos possuem comportamentos que resolvem o problema de ausência de objeto sem a necessidade de efetuar verificações de nulos melhorando o desempenho do programa e evita a duplicação de código.

A classe do objeto nulo herda as funções do objeto real, assim, garante que todos os métodos da mesma sejam implementados. Esses métodos podem ter “corpo vazio” onde nenhuma linha é codificada ou retornar o valor padrão estipulado [7] e é comumente aplicado junto a outros padrões de projeto.

Na literatura existem vários métodos baseados em padrões de projeto capazes de realizar a refatoração do código-fonte. Cada método foi implementado de forma individual por seus idealizadores. Beluzzo [5] definiu um ambiente chamado de *Refactoring and Measurement Tool* (RMT) que reúne mais de um método da literatura capaz de refatorar um código-fonte escrito em linguagem Java, isto ajuda o desenvolvedor pois pode refatorar o seu código usando um único ambiente.

Na RMT, ao inserir um código-fonte como entrada, o software avalia e retorna para o usuário quais classes devem ser refatoradas, quais métodos são recomendados e qual o impacto de cada opção de refatoração na classe. Permite também visualizar de forma percentual o resultado positivo ou negativo da refatoração no código. O impacto é calculado aplicando a combinação entre métricas baseadas em atributos de qualidade.

Com o objetivo de propor melhorias para a ferramenta de refatoração RMT, é apresentado neste relatório um estudo sobre refatoração de software baseado em padrões de projeto e da ferramenta RMT a fim de avaliar quais características podem ser melhoradas visando elevar os atributos de qualidade da ferramenta e sua disponibilização gratuita.

REFERENCIAL TEÓRICO

O referencial teórico que norteou o desenvolvimento desta pesquisa está dividido nos seguintes conteúdos: refatoração de código-fonte, padrões de projeto e refatoração baseada em padrões de projeto e a ferramenta RMT os quais são detalhados a seguir.

Refatoração de código-fonte. O processo de refatorar código visa melhorar e manter a qualidade interna do sistema sem afetar o seu funcionamento. Fowler et al. [1] definem a refatoração como uma alteração na forma estrutural do software para torná-lo mais compreensível e com custo baixo de modificação sem descaracterizar seu comportamento observável.

A refatoração consiste de técnicas pré-definidas que utilizando a ideia de "*bad smells*" para identificar trechos de código que indicam a necessidade de mudanças para aprimorar o seu funcionamento. Essas técnicas foram elencadas e dividas por Fowler et al. [1] em grupos que são: Compondo Métodos, Movendo Recursos Entre Objetos, Organizando Dados, Simplificando Expressões Condicionais, Tornando Mais Simples as Chamadas de Métodos, Lidando com Generalização e Refatorações Grandes.

Um catálogo na web [8], criado e mantido por Fowler, contém a descrição de técnicas como *Extract Class, Inline Function, Encapsulate Record*, que foram abordadas pelo autor em seus livros [1] [9].

Padrões de Projeto. Os padrões de projeto (*design patterns*) podem ser descritos como soluções abstratas para problemas recorrentes que aparecem durante o desenvolvimento de software. A aplicação de padrões de projeto ao sistema tem o propósito de torná-lo mais flexível a mudanças, ampliando a sua reutilização por meio das modelagens utilizadas. Os padrões de projeto são compostos por nome, solução para um problema, seu contexto, forma de aplicação e consequências. A seguir é apresentado as categorias encontradas nos padrões de projeto, sendo elas: criacionais, estruturais e comportamentais.

Os padrões criacionais são caracterizados por abstraírem ou adiarem o processo de criação dos objetos, permitindo assim, que o sistema funcione independentemente de como é criado, quem cria ou quando é criado o objeto. Esse tipo de padrão possibilita

configurar os objetos como "produtos" que variam de estrutura e funcionalidade. Alguns exemplos de padrões criacionais são: *Singleton, Abstract Factory* e *Factory Method.*

Os Padrões estruturais visam explicar como montar e organizar objetos e classes em grandes estruturas. Os padrões de classes utilizam herança na composição de interfaces ou implementações, enquanto, os padrões de objetos explicam como são compostos os objetos para receber novas funcionalidades, flexibilizando o sistema durante o tempo de execução. Nessa categoria se tem exemplos como o *Adapter, Bridge, Decorator* e *Facade.*

Os padrões de comportamento além de descrever padrões de classe e objeto, relatam padrões de comunicação entre objetos e mantém o foco no algoritmo e nas reponsabilidades entre objetos. Essa categoria de padrão de projeto possibilita ao desenvolver se concentrar em como os objetos estão intercalados. A herança é utilizada pelos padrões comportamentais de classe para atribuir o comportamento entre classes. Os padrões comportamentais de objeto utilizam da composição ao invés da herança. Alguns exemplos desse padrão são *Command Interpreter, Iterator, Observe, State Strategy* e *Template Method.*

Refatoração baseada em Padrões de Projeto. As refatorações baseadas em padrões de projeto tem como objetivo geral eliminar zonas de "*bad smell*" reestruturando o trecho a partir da inserção de padrões de projeto, tornando mais simples o processo de evolução do sistema. O processo de refatoração baseada em padrões pode ser incorporado a qualquer momento ao desenvolvimento do sistema independente do estágio que esteja, a dificuldade de atualização do software pode custar caro para o dono do projeto [10].

Um exemplo de refatoração baseada a padrões de projetos é o método apresentado por Cinnéide e Nixon [4] que segue o princípio de *minipatterns* e *minitrasformations*, em que um padrão é escolhido como objetivo da refatoração e a partir disso, pequenas transformações acontecem para que propósito seja alcançado.

Para identificar quais métodos fariam parte da RMT, Beluzzo [5] executou um mapeamento sistemático sobre o tema de refatoração de software baseado em padrões de projeto com o objetivo indentificá-los. Utilizando 7 (sete) repositórios digitais, um jornal e o Google Scholar, foram selecionados um total de 1149 trabalhos entre 1997 e 2017. Ao fim do processo de exclusão e inclusão foram selecionados 26 trabalhos para análise completa.

Após a finalização do mapeamento foi constatado que dentre os trabalhos mais citados na literatura está o livro de "Refatoração para Padrões" de Joshua Kerievisky [11] com 745 citações e Mel Ó Cinnéide sobressaiu com mais autorias entre os trabalhos. O padrão comportamental foi o mais abordado pelas publicações entre os padrões de projeto.

Ferramenta RMT. A ferramenta *Refactoring and Measurement Tool* (RMT) [5] possui a característica extensiva, que visa a inclusão de novos métodos de refatoração, métodos de detecção de candidatos a refatoração e métricas de avaliação. Estruturalmente a ferramenta possuí 4 (quatro) módulos que são responsáveis por interagir com o usuário, esses módulos são denominados *Client App*, *Interaction Service*, *Detection Methods Service* e *Metrics Service* [5].

Após cada incremento na ferramenta um "novo método" é gerado, o que caracteriza que uma nova versão.

METODOLOGIA

Para atingir o objetivo proposto de maneira satisfatória foi realizado um estudo da literatura sobre como efetuar refatoração de código-fonte e padrões de projeto, visando compreender o funcionamento de ambas e como utiliza-las. O estudo foi desenvolvido usando bibliotecas de pesquisa tais como o Portal Capes e o *Google Scholar*, dissertações, teses e artigos publicados em conferências e *journals*. Durante este estudo foi realizado a implementação de alguns padrões de projeto para melhor compreensão do conceito.

Também foi estudado e compreendido o método de refatoração de Gaitani *et al.* [6] analisando suas características conceituais e estruturais a fim de propor sua incorporação a ferramenta *Refactoring and Measurement Tool* (RMT) [5]. A proposta de incorporação foi realizada por meio de um estudo do diagrama de classe proposta por [5] e realizada a alteração do modelo.

Por fim, foi realizado um estudo para verificar possíveis melhorias em termos de distribuição de processamento para a ferramenta *Refactoring and Measurement Tool* (RMT) [5] e métricas de software. Este estudo utilizou como base a teoria sobre balanceamento de carga [16] e métricas de software.

RESULTADOS E DISCUSSÕES

Os resultados são apresentados em relação a implementação de padrões de projeto e as melhorias para a ferramenta RMT (*Refactoring and Measurement Tool*) que foi dividida em: i) análise do padrão de projeto NULL Object, ii) incorporação do NULL Object a RMT, iii) uso de balanceamento de carga na RMT, iv) controle de versão e respositório para a ferramenta RMT e v) inserção de métricas e atributos de qualidade para a ferramenta RMT.

Implementação de Padrões de Projeto. A seguir são apresentados 4 (quatro) exemplos de implementação de padrões de projeto que foram desenvolvidos durante a pesquisa a fim de analisar na prática características específicas de cada padrão, que podem ser encontrados no catálogo *online* [8] que contém informações referentes a características e forma de implementação dos mesmos.

O padrão de projeto *Singleton* tem o propósito de garantir que uma classe tenha apenas uma instância, disponibilizando um acesso global para essa instância. Esse padrão desabilita todos os meios de criação de objeto e substitui por um método especial que cria o objeto, utilizando um construtor privado, caso a classe ainda não tenha sido instanciada ou retorna o objeto existente por meio da função de retorno [2].

A implementação do padrão necessita atender alguns requisitos, Gamma *et al* [2] elencaram 5 (cinco) passos que auxiliam o programador a compreender como realmente aplicar o padrão *Singleton* ao seu código. Abaixo estão listadas o mecanismo necessário para atender o padrão:

1. Adicione um campo privado estático na classe para o armazenamento da instância *Singleton*.
2. Declare um método de criação público estático para obter a instância *Singleton*.
3. Implemente a "inicialização preguiçosa" dentro do método estático. Ela deve criar um novo objeto na sua primeira chamada e colocá-lo no campo estático. O método deve sempre retornar aquela instância em todas as chamadas subsequentes.

4. Faça o construtor da classe ser privado. O método estático da classe vai ainda ser capaz de chamar o construtor, mas não os demais objetos.
5. Vá para o código cliente e substitua todas as chamadas diretas para o construtor do *Singleton* com chamadas para seu método de criação estático.

Para obter a instância *Singleton* é necessária efetuar uma "iniciação preguiçosa". A Figura 1 apresenta um trecho de código referente a função de criação e retorno da instância que atende os itens de mecanismo 2 e 3.

```
public static Singleton consulSingleton(String valor){
    if (instancia == null){
        instancia = new Singleton(valor);
    }
    return instancia;
}
```

Figura 1. Função de retorno do Singleton.

No padrão *Singleton* nenhuma outra classe pode substituir a instância salva em cache a não ser a própria classe, utilizando o comando "*private*" na função construtora se evita que a classe possa ser instanciada através do operador "*new*" no código do cliente.

O padrão *Factory Method* fornece uma interface de criação de objetos que utiliza de subclasses que são responsáveis de fato pela criação do objeto substituindo o operador "new" na classe cliente por chamadas de métodos fábrica. Esse padrão separa o código criação do produto do código que realmente utiliza esse produto [2].

Com a utilização desse padrão o código se torna mais flexível. Para melhorar o entendimento de como funciona o *Factory Method*, foi desenvolvido um exemplo em que o cenário de aplicação é um sistema de controle de pizzaria em regiões diferentes do Brasil, onde cada filial possui o seu processo de criação, o produto criado ainda é o mesmo, que no exemplo é a pizza.

A interface do padrão deve ser declarada como pública e abstrata assim como todos os métodos que são sobrescrito pelas subclasses [2], a Figura 2 apresenta um exemplo desse formato.

```
public abstract class PizzaFactory {

    protected Pizza pizza;

    public abstract void criarPizza(String tipo);

    public Pizza delivery(){
        return pizza;
    }
```

Figura 2. Classe abstrata.

Nesse padrão é aplicado o conceito de herança para que as subclasses herdem todas as variáveis e métodos da superclasse abstrata, para isso é utilizado o comando "*extends*" na declaração das subclasses.

O padrão *Abstract Factory* permite que você construa família de objetos relacionados através de interfaces, sem necessariamente especificar suas classes concretas de criação [2]. A ideia desse padrão é facilitar a adição ou remoção de interfaces ao sistema tornando-o mais flexível, fazendo valer o princípio de responsabilidade única.

As classes concretas de criação dos produtos devem ser implementadas de acordo com a sua interface, assim, garante que o objeto criado mantenha as características do produto abstrato. Como exemplo, é utilizado um sistema que lida com conjuntos de

carros, agrupando aqueles com comportamentos parecidos entre os tipos carro sedan e carro popular das fábricas Fiat e Ford.

A classe “FabricaDeCarro” representa a interface utilizada pelas fábricas concretas “FabricaFiat” e ”FabricaFord” onde serão implementados os métodos da interface, que está representada na Figura 3.

```
public interface FabricaDeCarro {

    CarroSedan criarCarroSedan();

    CarroPopular criarCarroPopular();
}
```

Figura 3. Interface “FabricaDeCarro”

O padrão de projeto *Facade* fornece uma interface simplificada para classes complexas abstraindo trechos do sistema por meio de fachadas escondendo o seu funcionamento interno, tonando mais fácil a comunicação entre classes [2].

A aplicação desse padrão é recomendada quando se deseja estruturar um subsistema em camadas, reduzindo a dependência entre subsistemas fazendo a comunicação ser realizada pelas fachadas que são responsáveis por iniciar e gerenciar o ciclo de vida das chamadas.

A seguir é apresentado um exemplo de implementação do padrão de projeto Facade referente a um framework de conversor de vídeo que permite ao cliente converter um vídeo sem precisar conhecer todas as etapas do processo de conversão. A classe fachada desse exemplo é a “VideoConversionFacade” que fica responsável por controlar qual a ordem de execução do processo e está representada pela Figura 4.

```
public class VideoConversionFacade {
    public File convertVideo(String fileName, String format) {
        System.out.println("VideoConversionFacade: conversion started.");
        VideoFile file = new VideoFile(fileName);
        Codec sourceCodec = CodecFactory.extract(file);
        Codec destinationCodec;
        if (format.equals("mp4")) {
            destinationCodec = new OggCompressionCodec();
        } else {
            destinationCodec = new MPEG4CompressionCodec();
        }
        VideoFile buffer = BitrateReader.read(file, sourceCodec);
        VideoFile intermediateResult = BitrateReader.convert(buffer, destinationCodec);
        File result = (new AudioMixer()).fix(intermediateResult);
        System.out.println("VideoConversionFacade: conversion completed.");
        return result;
    }
}
```

Figura 4. Classe Fachada do Método.

O código do cliente é responsável por solicitar a conversão, mas não sabe como o processo é realizado, o cliente apenas instância a fachada e fornece as informações sobre o formato de conversão desejado.

Análise do Padrão de projeto NULL Object. No universo de desenvolvimento de softwares, os programas comumente lidam com referências nulas em sua execução para manter o fluxo correto de funcionamento, e para controlar tal fator é necessário aplicar verificações (*nullchecking*) que informam ao programa se o retorno obtido foi uma referência nula ou um objeto.

Em softwares extensos é provável que ocorram diversas verificações de nulos e isso acarreta na diminuição de desempenho. O padrão de projeto *NULL Object* utiliza herança e polimorfismo para substituir as verificações nulas repetitivas por chamadas de classes nulas que podem ter um comportamento padrão ou ter o corpo do(s) método(s) vazios [6]. A utilização do padrão contribui para a reusabilidade e manutenção do código-fonte possibilitando incrementos mais seguros para a evolução do sistema. Gaitani *et al*. [6] relata em seu trabalho três (3) formas estruturais de se aplicar o padrão *NULL Object* ao sistema e são elas:

1. Implementação simples, onde a classe nula herda da classe real do objeto todos os métodos e sobrescreve com o comportamento de "não fazer nada".
2. Implementação de ancestral comum, que cria uma classe abstrata com métodos não privados e faz com que tanto a classe real quanto a classe nula a implementem.
3. Implementação de interface comum, similar ao comportamento citado anteriormente, mas ao invés de utilizar uma classe abstrata para herdar os métodos é utilizado a implementação de uma interface.

Kevlin Henney frisa que "o objeto nulo não deve ser usado indiscriminadamente como um substituto para referencias nulas" [7]. A intenção do padrão é encapsular a ausência de objeto que não possui uma função significativa para o usuário do objeto real. A Figura 5 apresenta os tipos estruturais citados exemplificando de maneira sucinta e intuitiva a forma que as técnicas são aplicadas dada uma classe real denominada Component.

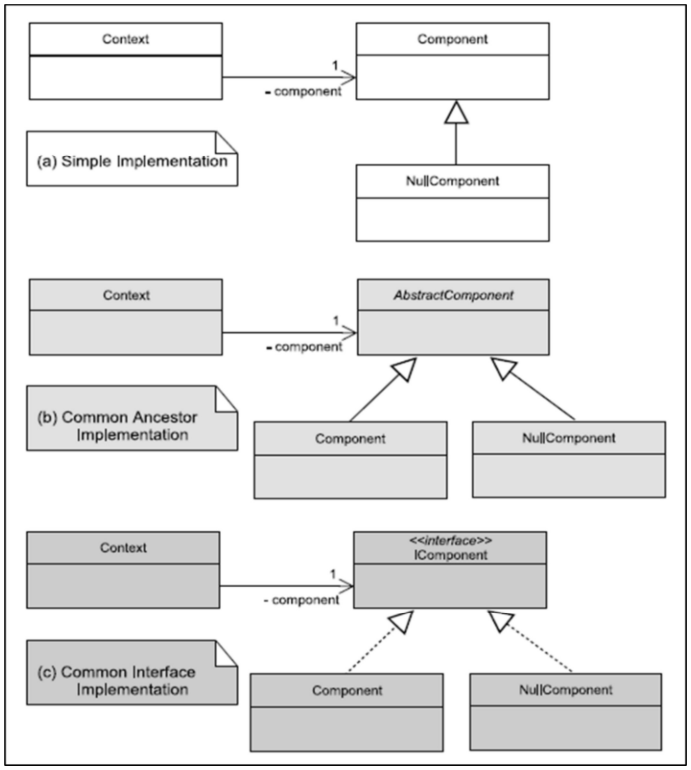

Figura 5. Alternativas de implementação NULL Object.

A refatoração para o padrão de projeto *NULL Object* é um tema não muito explorado na literatura, Fowler et al. descreve em seu livro [1] uma ideia introdutória de como efetuar essa refatoração, motivado pelo desejo de eliminar verificações de nulos com o intuito de melhorar o desempenho do sistema.

Baseando-se na ideia de Fowler et al. [1] sobre refatoração para o padrão de projeto *NULL Object*, Gaitani et al. [6] desenvolveram um método de refatoração que utiliza do princípio de ancestral comum, que percorre um trecho de código buscando o que é chamado pelos autores de "candidatos a refatoração". Para encontrar esses candidatos pré-condições devem ser atendidas, em alguns casos é necessário aplicar outros tipos de refatoração antes de avaliar se as pré-condições foram atendidas.

É importe ressaltar que a maior dificuldade da refatoração para o padrão de projeto NULL Object está na análise que determina se deve ou não incorporar esse padrão ao sistema, pois aplicar o padrão pode elevar a complexidade do código dificultando a sua manutenibilidade. Como um objeto nulo não possui valor significativo para o cliente, encarar tais mudanças no código pode não ser vantajoso para o sistema.

a) Identificação de candidatos a refatoração baseada em padrões de projeto

Refatorar é um processo complexo, Silva et al. [12] relatam que para refatorar é preciso identificar o que será refatorado, a operação que será aplicada e onde o código refatorado será alocado. A forma de execução dessas etapas pode sofrer variações dependendo do contexto, e para facilitar essa tarefa uma solução possível é a utilização de ferramentas de refatoração.

As refatorações baseadas em padrão de projeto descritas Gamma *et al* [13] comumente identificam classes como candidatas a refatoração, são elas que passam pelo processo de ser modificada para atender as características do padrão. Gaitani *et al* [6] utilizam uma abordagem baseada em regras para definir os predicados que devem ser atendidos para classificar um candidato como *optionalField* (campo opcional) de acordo com as especificações do método de identificação.

b) Detecção de ponto de inserção do padrão NULL Object

O algoritmo de detecção proposto por Gaitani et al. [6] é baseado na análise estática do código-fonte, utilizando a representação *Abstract Syntax Tree* (AST) para processar todas as classes do código. O candidato a refatoração nesse método é identificado como campo opcional que será inicializado com uma instância *NULL Object* após a refatoração.

Esse método visa eliminar verificações nulas repetidas adicionando um objeto nulo a classe que está sendo refatorada, com o objetivo de melhorar o desempenho do programa. O código-fonte é analisado pelo algoritmo buscando trechos que o valor do campo opcional é manipulado em instruções condicionais (if ou if/else) nulas, esses trechos são denominados *de Guarded Field Invocation Conditional* (GFI-Conditional). Foi utilizado uma abordagem de regras para definir as especificações dos predicados e relações necessárias que formam o vocabulário do método proposto.

Considera-se um campo como opcional quando é declarado em uma determinada classe e não possui inicialização ou é inicializado com o valor *null* e não possui um tipo primitivo. Para que a premissa seja satisfeita o campo não pode ter seu valor modificado em nenhum método construtor e a classe contexto deve possuir um método modificador não privado que modifica o valor do campo possivelmente opcional. Um método é considerado modificador se o campo estiver entre as variáveis definidas no *Program Dependence Graph* (PDG) que auxilia na análise de fluxo de dados.

Uma Invocação Condicional de Campo Protegido (*Guarded Field Invocation Conditional)* representa as instruções condicionais com no máximo duas ramificações que executam verificações nulas referente ao campo opcional (*optionalField*) que possuem a função de proteger o acesso e/ou a manipulação de campos e métodos. O trecho de código que executa essa função é denominado de *FieldInvocationFragment* que após a refatoração apresentará o comportamento real ou o comportamento padrão do *NULL Object* que é atribuído ao *optionalField* em vez da referência nula. O método aborda 4 (quatro) variantes de GFI-Conditional que estão representadas pela Figura 6.

```
/* Variant 1: Single branch GFI-Conditional,
              inequality comparison */
if (optionalField != null){
  /* field invocation fragment */
}

/* Variant 2: Two-branch GFI-Conditional,
              inequality comparison */
if (optionalField != null){
  /* field invocation fragment */
  /* other statements */
} else {
  /* raise of application specific exception */
}

/* Variant 3: Single branch GFI-Conditional,
              equality comparison */
if (optionalField == null){
  /* raise of application specific exception */
}

/* Variant 4: Two-branch GFI-Conditional,
              equality comparison  */
if (optionalField == null){
  /* raise of application specific exception */
} else {
  /* field invocation fragment */
  /* other statements */
}
```

Figura 6. Listagem de GFI-Conditional

- Variante 1: A primeira variante trabalha com a expressão condicional (if) que é satisfeita com uma comparação de não igualdade, onde o *optionalField* não é igual a nulo. Se o valor de retorno do método que é manipulado no corpo do fragmento tiver valor *void* o *FieldInvocationFragment* é marcado satisfazendo o predicado *emptyOn-Null* e terá uma implementação vazia do método no *NULL Object*. Caso o dado de retorno seja um valor primitivo ou *String* o trecho é marcado pelo predicado *returnConstantOnNull* e terá a implementação de "não fazer nada" no *NULL Object* com uma única instrução de retorno de valor constate.
- Variante 2: A segunda variante tem o mesmo funcionamento da variante 1, porém, inclui a ramificação *else* na análise do *FieldInvocationFragment* que é responsável por lançar uma exceção específica caso o resultado da comparação do *OptionalField* seja falso, essa exceção é chamada de *NullFieldException*. No processo de refatoração esse trecho é marcado atendendo o predicado *nullFieldExceptionOnNull* e terá a implementação da *NullFieldException* no *NULL Object*.
- Variante 3: A terceira variante trabalha com a expressão condicional (*if*) de ramo único que verifica a igualdade do *optionalField* com o valo *null*, onde o corpo da expressão possui uma instrução *throw NullFieldException*. No *NULL*

Object será feita a implementação da *throw NullFieldException* no *FieldInvocationFragment*.

- Variante 4: A quarta variante trabalha com a igualdade em relação ao optionalField em uma expressão condicional de ramo duplo (if/else). Nessa variante é lançada uma NullFieldException caso a expressão seja verdadeira e na ramificação else contém a manipulação do optionalField. Essa variante atente o predicado nullPointerExceptionOnNull.

c) Algoritmo de detecção de candidato

O algoritmo de detecção proposto por Gaitani *et al.* [6] processa individualmente todas as classes presentes no código base do sistema e produz como saída um conjunto de oportunidades de refatoração composto pelo campo opcional (*optionalField*), GFI-Condicional que podem ser eliminadas e um subconjunto que corresponde aos predicados que cada *FieldInvocationFragment* está associado.

Para que um candidato se transforme em *optionalField* é necessário atender as pré-condições de refatoração automatizada para o padrão *NULL Object* com o objetivo de evitar que a reestruturação do código seja errônea inserindo novos problemas ou modificando o funcionamento externo do sistema. Essas pré-condições de refatoração são compostas pelas pré-condições de campo opcional e pré-condições de GFI-Conditional.

As pré-condições de campo opcional tem como intuito garantir que modificações que serão feitas no *optionalField* não alterem o funcionamento do sistema. As pré-condições de GFI-Conditional tem como objetivo garantir que as verificações nulas selecionadas possam ser eliminadas com segurança durante a refatoração para o padrão *NULL Object*.

O fluxo do processo de detecção é representado na forma de um diagrama de atividade desenvolvido na ferramenta *StarUML* e está exibido pela Figura 7. O algoritmo executa a detecção de campos opcionais a partir de uma classe de entrada pertencente ao código base e produz como saída um conjunto de oportunidades de refatoração [R = U (f; G; P empty; P const; P nfe; P npe)] [6], onde:

i. F é um campo opcional.
ii. G é o conjunto de GFI-Conditional a serem eliminadas.
iii. P empy; P const; P npe correspondem aos predicados satisfeitos.

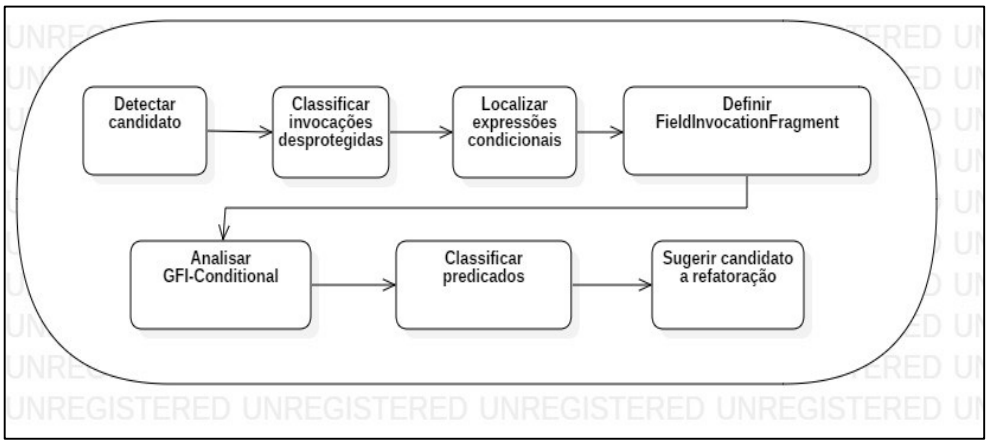

Figura 7. Diagrama do processo de dectecção.

Inicialmente o algoritmo analisa as declarações de variáveis feita na classe base em busca de candidatos a *optionalField*, que será utilizado no decorrer da execução para a identificação dos *FieldInvocationFragment*. Após a seleção de um candidato, é

realizada uma busca por métodos que modificam o valor do *optionalField* de forma desprotegida, tais métodos são classificados por atender o predicado *nullPointerExceptionOn-Null*.

A próxima etapa é composta por analisar o corpo do método em busca das expressões condicionais que satisfazem as características de alguma das variantes GFI-Conditional e, posteriormente, avaliar *FieldInvocationFragment* para identificar as invocações de métodos que manipulam o *optionalField* e associa-las aos predicados apropriados do *NULL Object*. Cada invocação de método que satisfaz um predicado do *NULL Object* é acrescentado a um conjunto temporário se atender as pré-condições de refatoração.

d) Aplicação da refatoração para o padrão NULL Object

A aplicação da transformação do código-fonte para o padrão de projeto NULL Object é constituída pela criação de classes extras e a modificação das classes existentes. O processo de refatoração pode ser decomposto em uma série de 4 (quatro) transformações com enfoque nos tipos de candidatos a refatoração possíveis. Para facilitar a compreensão de como a aplicação funciona usaremos uma classe denominada *Customer*.

Um dos objetivos do algoritmo de refatoração proposto por Gaitani *et al* [6] é tornar seu uso transparente para as classes presentes no contexto que manipulam a classe que foi refatorada. Este padrão permite alterar o tipo de declaração de um campo opcional atribuindo uma instância nula (*NullCustomer*) durante sua inicialização.

Para não ocorrer problemas caso o campo opcional refatorado forneça um valor de retorno ou parâmetro para outra classe, é utilizado um método específico *getReference*() que retorna o *null* quando invocado de uma instância *NullCustomer* ou retorna o valor referente ao objeto atual quando invocado na instancia *Customer*.

e) Etapas de refatoração do campo opcional

A execução das etapas de refatoração visa transformar a declaração e atribuição do campo opcional para atender as características propostas pelo método baseado no padrão *Null Object*, a seguir são apresentadas as etapas de transformação explicando o funcionamento de cada uma.

1. Criação da classe AbstractCustomer: O procedimento de criação define a classe abstractCustomer como superclasse abstrata de Customer e implementa todas as suas interfaces, além disso, os métodos não privados e não estáticos da classe Customer são declarados como métodos abstratos na superclasse com os mesmos argumentos. Os métodos que satisfazem o predicado nullFieldExceptionOnNull são declarados na classe abstrata lançando exceções do tipo NullFieldException.

   Para lidar com a ausência de referências nulas do optionalField, como por exemplo, instruções GFI-Conditional que não podem ser eliminada e tornar o objeto nulo transparente para o cliente do contexto, a interface AbstractCustomer é expandida introduzindo métodos que são chamados de Null Object Utility Methods. As declarações que fazem parte desse grupo de método são apresentadas na Figura 8.

```
public abstract boolean isNull();

public abstract Component getReference();

public abstract void assertNotNull() throws NullFieldException;
```

Figura 8. Declarações do NULL Object Utility Methods.

O método *isNull()* tem o propósito de substituir as verificações de campo opcional que não podem ser eliminadas, retornando *true* quando o objeto é nulo. Já o método *getReference()* ajuda a classe a delimitar o uso da instância nula dentro da classe contexto retornando *null* quando implementado na classe *NullCustomer* ou retornando o objeto de referência atual quando utilizado dentro da classe *Customer*.

A utilização do método *assertNotNull()* tem como intuito substituir a variante 3 (três) que compõe as GFI-Conditionals, na classe *NullCustomer* o método *assertNotNull()* pode lançar uma exceção única chamada *NullFieldException* e na classe *Customer* possui sua implementação vazia. Caso a substituição no contexto inclua mais de um tipo de exceção, é necessário criar métodos *assertNotNull()* para cada exceção.

2. Refatoração da classe Customer: A classe *Customer* que simboliza o tipo do campo opcional é transformada em uma subclasse de *AbstractCustomer* e remove todas as declarações que implementam a classe como interface. As interfaces implementadas em *Customer* agora estão declaradas na sua classe pai, juntamente com os métodos padrões *isNull()*, *getReference()* e *assertNotNull()*.
3. Criação da classe NullCustomer: Assim, como a classe *Customer* a classe *NullCustomer* é declara como uma subclasse de *AbstractCustomer*, implementando os métodos patrões que estão relacionados com os predicados atendidos durante o processo de identificação de candidatos a refatoração. Caso o candidato não se enquadre em nenhum dos predicados, sua implementação padrão é lançar uma exceção *UnsupportedOperationException.*
4. Refatoração do contexto e eliminação das GFI-Conditional: A refatoração da classe contexto se concentra na eliminação de todas as verificações nulas que correspondem às GFI-Conditionals, juntamente com a instrução condicional. O restante dos casos são substituídos pela invocação do *is.Null()* no campo opcional.

   O processo de refatoração da classe contexto é resumido por Gaitani et al. [6] em 5 (cinco) procedimentos de transformação, sendo eles:

   i. Alteração do tipo de dados do campo opcional (*Customer*) para a nova instância nula (*AbstractCustomer*), inicializando-o com a atribuição do *NullCutomer*.
   ii. Criação do método unitário *assignToOptionalCampo*().
   iii. Eliminação das ramificações de verificações nulas referentes ao campo opcional começando pelas variantes 1 e 2, posteriormente a variante 3 é substituída pela invocação do método *assertNotNull*(). A variante 4 tem seu corpo de ramificação alterado para o novo modelo, as comparações de igualdades nulas restantes são alteradas para o método *optionalFieldisNull*() e as comparações de desigualdade nulas para o método *!optionalFieldisNull*()
   iv. Limitar o uso do *NullCustomer* dentro da classe contexto, substituindo as invocações do *optionalField* por *optionalFieldgetReference*() no caso de declarações que utilizam o valor do campo opcional como retorno ou parâmetro de um método fora da classe contexto.
   v. Substituir qualquer expressão (*exp*) de atribuição ao campo opcional por uma invocação do método *assignToOptionalField*(*exp*).

f) Avaliação do método baseado no padrão NULL Object

O método de refatoração elaborado por Gaitani et al. [6] foi avaliada de acordo com os atributos de qualidade de software solidez, eficácia e praticidade.

De maneira empírica a solidez foi avaliada por meio da aplicação do método de refatoração em um conjunto de projetos de *bechmark* e a execução de conjuntos de testes, com base na correção sintática da classe refatorada e na preservação do comportamento externo do projeto refatorado. Esta avaliação passa pelas etapas de refatoração para o método que identifica os candidatos a refatoração e efetua a respectiva refatoração da classe de acordo com o padrão de maneira automatizada. Como resultado, os autores não obtiveram erros na compilação dos projetos refatorados no conjunto de teste.

A eficácia e a praticidade foram avaliadas paralelamente em termos do número de candidatos a refatoração identificados no código analisado e no impacto da refatoração em relação a complexidade ciclomática dos métodos com verificações nulas eliminadas. Como base de avaliação foi utilizado os campos opcionais que foram refatorados automaticamente e manualmente. Os campos opcionais refatorados manualmente são apresentados ao usuário que está aplicando a refatoração como um bom candidato a refatoração, mas que acarretará em erros caso a refatoração ocorra de maneira automatizada, deixando assim, ao usuário a opção de realizar a refatoração ou não. Os campos refatoratos formam um novo conjunto, que também é avaliado.

O último subconjunto de campos opcionais analisados são os candidatos que possuem grande potencial a refatoração, porém, esse conjunto não pode ser refatorado pois a classe não pertence a base do código ou não apresenta nenhuma condição de verificação nula de campo opcional. Portanto, a refatoração seria custosa devido a necessidade de atender os pré-requisitos e não agregaria melhorias ao código.

Foi utilizado na composição dos conjuntos de testes para a avaliação projetos que atendem 3 (três) requisitos básicos, sendo eles: (a) o código-fonte do projeto deve estar disponível publicamente para uso de terceiros, (b) serem implementados na linguagem de programação Java e (c) possuírem tamanhos e complexidades variadas. É obtido de cada projeto o número total de linhas do código sem contar as em branco, o número total de classes, o número total de métodos de classe e a proporção de instruções executadas durante o teste em relação ao total de instruções do código-fonte, essas características compõem as métricas analisadas em cada projeto.

Como resultado da avaliação foi constatado que em média 65,7% das verificações nulas presentes nos projetos puderam ser removidas com a inserção do *NULL Object*. O cálculo da complexidade ciclomática segue o princípio de McCabe [14], que é baseado no Gráfico de Fluxo de Controle de Programa. A redução percentual da métrica de complexidade ciclomática variou entre 5% a 66,67% nos projetos refatorados, essa redução contribui para que o código-fonte fique mais legível proporcionando uma manutenibilidade mais intuitiva do código.

A última forma de avaliação está relacionada com o tempo utilizado para efetuar a refatoração usando a implementação do método, esse tempo variou de 30s a 4,5min que evidenciou uma sobrecarga de processamento, deixando claro, que o método possui limitações que recomendam a sua aplicação em código de rotina do programador.

Inserção do Padrão NULL Object a RMT. Para a introdução do método *NULL Object* [6] a nível de modelo na RMT é necessário incrementar o diagrama de classe do "*Detection Methods Service*" que representa os métodos que estão presentes na ferramenta juntamente com a sua forma de extração de candidatos a refatoração.

A ferramenta já possuía os métodos de Zafeiris *et al* [20] e Wei *et al* [21], após a inclusão, o método de Gaitani *et al.* [6] também passa a fazer parte dos métodos presentes na ferramenta de refatoração como se pode observar na Figura 3, que foi criado a partir da ferramenta *StarUML* (*Unregistered*). Outra mudança necessária no diagrama do "*Detection Methods Service*" é a inclusão da classe responsável por identificar os possíveis candidatos a refatoração que podem ser substituídos durante o processo de refatoração. O método de Gaitani *et al.* [6] tem como dependência o "*OptionalFieldFork*", que encontra campos opcionais que podem ser substituídos por instâncias nulas.

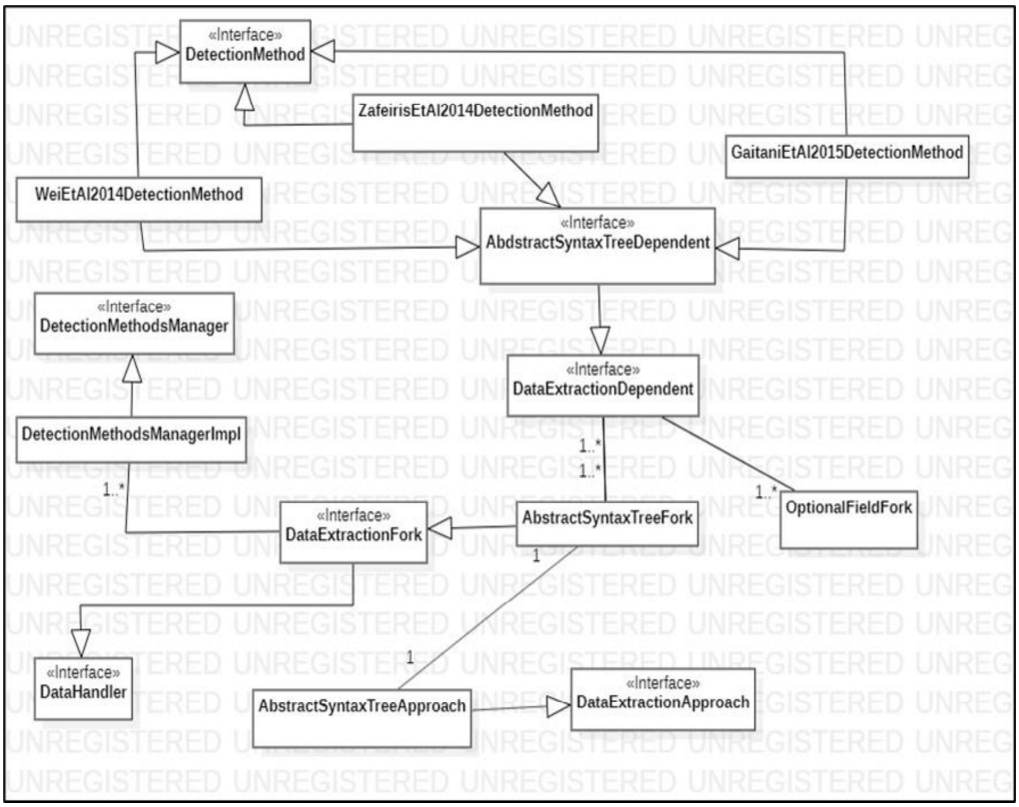

Figura 9. Diagrama de classe Detection Methods Service Atualizado.

Uso do Balanceamento de Carga na ferramenta RMT. Como princípio básico de funcionamento o balanceamento de carga utiliza o compartilhamento de recursos de hardware e software de uma determinada rede de computadores visando elevar a escalabilidade de processamento das requisições da aplicação. Utilizar o balanceamento de carga de maneira uniforme em vários processadores tende a melhorar o desempenho de computação [16]. Dentre as abordagens da literatura as mais comumente utilizadas estão as estratégias de balanceamento de carga estático e balanceamento de carga dinâmico.

O balanceamento de carga estático é definido a partir de informações do comportamento médio do sistema, onde as decisões de transferências são independes em relação ao sistema como um todo, sendo recomendada para aplicações que possuem um comportamento consistente e previsível [17]. A abordagem dinâmica possui um melhor desempenho em relação a estática e sua implementação é considerada complexa, o que dificulta a utilização desta forma de balanceamento [18].

Gurka Júnior [19] realizou um levantamento de algoritmos comuns na literatura que realizam o balanceamento de carga estático e dinâmico. Dentre os principais algoritmos para balanceamento estáticos estão: i) *Round Robin* e algoritmos randômicos,

que divide as tarefas de forma igualitária entre os processadores; ii) Algoritmo de gerenciamento central, onde um ponto central do sistema é responsável por dividir as tarefas entre os processadores levando em consideração para a escolha o processador com a menor carga no momento da distribuição; iii) Algoritmo limiar, onde os processadores são identificados em 3 (três) categorias sendo elas subcarregado, médio e sobrecarregado. Os algoritmos de balanceamento de carga dinâmicos mais comuns são: i) Algoritmo de fila central, que gerência as tarefas a serem processadas por meio de uma fila onde a primeira tarefa a entrar é a primeira a sair; ii) Algoritmo de fila local, onde a fila de requisições é alocada estaticamente até atingir um nível mínimo definido pelo usuário. Esse algoritmo possui alta complexidade e custo computacional, o que torna sua implementação proibitiva.

Como a ferramenta RMT trabalha com mais de um método de refatoração que detecta o candidato a refatoração, avalia seu impacto e depois refatora (caso seja a escolha do usuário), é possível introduzir o balanceamento de carga dentro dos módulos "*Metrics Service*" e "*Detection Methods Service*" com o objetivo de paralelizar as etapas de cada método, gerando assim, um tempo de resposta menor ao usuário.

Controle de versão e repositório para a ferramenta RMT. O processo de Controle de Versões (CV) consiste em efetuar o gerenciamento adequado das versões diferentes dos artefatos criados durante o processo de execução software, isso permite que os stakeholders acompanhem de maneira direta a evolução completa do sistema. O CV acelera e simplifica o processo de desenvolvimento de software, controlando os arquivos e seu histórico e disponibiliza um modelo de acesso rápido [15].

Para gerenciar de maneira adequada a evolução do software e de sua documentação, recomenda-se utilizar um sistema de controle de versões adequado para a ferramenta. Visando atender esse objetivo foi criado um repositório no GitHub que contém os arquivos da ferramenta para que desenvolvedores possam acessar e contribuir com a evolução do projeto. O repositório da ferramenta pode ser acessado por meio do link (https://github.com/Lesic-UTFPR/Ferramenta-RMT.git).

Inserção de Métricas e Atributos de Qualidade para a Ferramenta RMT. Os atributos de qualidade utilizados até o momento pela ferramenta são: Manutenibilidade, Confiabilidade e Reusabilidade. Dentre as métricas de softwares utilizadas na RMT se pode elencar a Profundidade da Árvore de Herança (PAH), Complexidade Ciclomática (CC) e Tamanho do Programa em Linhas de Código (TPLC).

Como outra proposta de melhoria a ferramenta RMT é possível realizar a inclusão de novos atributos de qualidade de softwares como usabilidade, funcionalidade, eficiência, solidez e praticidade. Utilizar mais atributos de qualidade tornará a avaliação da refatoração do código-fonte mais refinada, resultando em um feedback mais adequado para o usuário.

Como a ferramenta é baseada na correlação entre atributos e métricas, é necessário relacionar esses novos atributos com novas métricas de software. Dentre as métricas ainda não implementadas na ferramenta se tem número de mensagens de erro e extensão do manual do usuário, tais métricas possuem relação com o funcionamento da ferramenta e com o objetivo de melhorar seu gerenciamento.

CONCLUSÕES

Um sistema sem atualizações tende a não acompanhar o avanço da tecnologia. Como resultado desta pesquisa foi possível identificar 4 (quatro) melhorais importantes

que agregariam qualidade a ferramenta RMT. O balanceamento de carga vai permitir melhorar atributo eficiência, introduzir novas métricas de softwares e atributos de qualidade visa aumentar a confiabilidade nos resultados antes da ferramenta aplicar os padrões de projeto. Efetuar o controle de versões juntamente com um repositório, torna a manutenibilidade da ferramenta mais intuitiva. Por fim, introduzir um novo método de refatoração além de aumentar o conjunto de opção agrega em melhorias dos atributos usabilidade e funcionalidades da ferramenta RMT.

REFERÊNCIAS

[1] FOWLER, et al. Refactoring: improving the design of existing code. [S.l.]: Addison-Wesley Professional, 1999.

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

[3] MENS, T.; TOURWÉ, T. A Declarative evolution framework for object-oriented design patterns. Proceedings on the IEEE International Conference on Software Maintenance (ICMS), IEEE 2001.

[4] CINNÈIDE, M. Ó.; NIXON, P. A methodology for the automated introduction of design patterns. In: IEEE. Software Maintenance, 1999. p. 463-472.

[5] 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.

[6] GAITANI, M. A. G. et al. Automated refactoring to the null object design pattern. Information and Software Technology, v. 59, p. 33-52, 2015.

[7] HENNEY, K. Null object, Proceedings of the 7th European Conference on Pattern Languages of Programs, 2002.

[8] Catálogo on-line. Disponível em: <https://www.refactoring.com/catalog/>. Acesso: 18 jan. 2020.

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

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

[11] KERIEVISKY, J. Refatoração para Padrões, Porto Alegre, Bookman, 2008.

[12] SILVA, C. et al. Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review. Proceedings on the XXIII Congresso Ibero-Americano em Engenharia de Software, CIbSE, 2020.

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

[14] McCABE, T. J. A Complexity Measure. In: IEEE Transactions on Software Engineering Addison, v. SE-2, n. 4, p. 308-320, 1976.

[15] OTTE, S. Version control systems. Computer Systems and Telematics, p. 11-13, 2009.

[16] RATHORE, N; CHANA, I. Load balancing and job migration techniques in grid: a survey of recent trends. Wireless personal communications, v. 79, n. 3, p. 2089-2125, 2014.

[17] PADUA, D. Encyclopedia of Parallel Computing. Springer Science & Business Media, 2011.

[18] KAMEDA, H et al. A performance comparison of dynamic vs. static load balancing policies in a mainframe-personal computer network model. IEEE Conference on Decision and Control. v. 2, p. 1415-1420, 2000.

[19] GURKA JÚNIOR, Mácio José. miRQuest 2: solução computacional para integração de ferramentas de predição de micro RNA utilizando balanceamento de carga. 2019. 56f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2019.

[20] ZAFEIRIS, V. E. et al. Automated refactoring of super-class method invocations to the template method design patterns. Information and Software Technology. p.19-35. 2017.

[21] WEI, L. et al. Automated pattern-directed refactoring for complex conditional statements. Journal of Central South University, v. 21, n. 5, p. 1935-1945, 2014.

Simone Nasser Matos
Orientadora

Carlos Eduardo Rodrigues Cardoso
Orientando