UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

THYAGO HENRIQUE PACHER

CODICE-UNIO: UMA ABORDAGEM INTEGRADA DE MÉTODOS PARA DETECÇÃO E INSERÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE USANDO AGENTES

DISSERTAÇÃO

PONTA GROSSA
2020

THYAGO HENRIQUE PACHER

CODICE-UNIO: UMA ABORDAGEM INTEGRADA DE MÉTODOS PARA DETECÇÃO E INSERÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE USANDO AGENTES

Dissertação apresentada como requisito parcial à obtenção do título de Mestre em Ciência da Computação do programa de Pós-Graduação em Ciência da Computação da Universidade Tecnológica Federal do Paraná - Campus Ponta Grossa.

Área de Concentração: Sistemas e Métodos de Computação.

Orientadora: Prof. Dra. Simone N. Matos

Coorientadora: Prof. Dra. Eliana C. M. Ishikawa

PONTA GROSSA

2020

Ficha catalográfica elaborada pelo Departamento de Biblioteca
da Universidade Tecnológica Federal do Paraná, Campus Ponta Grossa
n.58/20

P116 Pacher, Thyago Henrique

Codice-unio: uma abordagem integrada de métodos para detecção e inserção de padrões de projeto em código-fonte usando agentes. / Thyago Henrique Pacher, 2020.
142 f. : il. ; 30 cm.

Orientadora: Profa. Dra. Simone Nasser Matos
Coorientadora: Profa. Dra. Eliana Cláudia Mayumi Ishikawa

Dissertação (Mestrado em Ciência da Computação) - Programa de Pós-Graduação em Ciência da Computação. Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2020.

1. Agentes inteligentes (Software). 2. Software - Refatoração. 3. Padrões de software. 4. Projeto de sistemas. I. Matos, Simone Nasser. II. Ishikawa, Eliana Cláudia Mayumi. III. Universidade Tecnológica Federaldo Paraná. IV. Título.

CDD 004

Elson Heraldo Ribeiro Junior. CRB-9/1413. 31/08/2020.

Ministério da Educação
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS PONTA GROSSA
Diretoria de Pesquisa e Pós-Graduação
Programa de Pós-Graduação em Ciência da Computação

FOLHA DE APROVAÇÃO

Título de Dissertação **Nº 22/2020**

CODICE-UNIO: UMA ABORDAGEM INTEGRADA DE MÉTODOS PARA DETECÇÃO E INSERÇÃO DE PADRÕES DE PROJETO EM CÓDIGO-FONTE USANDO AGENTES

Por

Thyago Henrique Pacher

Esta dissertação foi apresentada às **14 horas** de **03 agosto 2020**, à distância, por webconferência, como requisito parcial para a obtenção do título de MESTRE EM CIÊNCIA DA COMPUTAÇÃO, Programa de Pós-Graduação 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 do Rocio Senger de Souza (USP)

Prof. Dr. Gleifer Vaz Alves (UTFPR)

Prof. Drª. Simone Nasser Matos (UTFPR)
Orientadora e presidente da banca

Visto do Coordenador:

Prof. Dr. André Koscianski
Coordenador do PPGCC
UTFPR - Câmpus Ponta Grossa

- A Folha de Aprovação assinada encontra-se na Coordenação do Porgrama -

RESUMO

PACHER, Thyago Henrique. **Codice-Unio:** uma abordagem integrada de métodos para detecção e inserção de padrões de projeto em código-fonte usando agentes. 2020. 142 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2020.

O processo de refatoração garante uma qualidade maior no código-fonte aumentando a sua manutenibilidade, confiabilidade e flexibilidade. De acordo com a literatura cerca de 70% do custo do software é para manutenção e isto pode ser diminuído usando técnicas que permitem aumentar a qualidade do código-fonte tal como a refatoração baseada em padrões de projetos. Constatou-se por meio de um mapeamento sistemático que os trabalhos na literatura de detecção e inserção de padrões de projeto não são realizados de forma autônoma. Este trabalho criou a abordagem Codice-Unio para detectar pontos de inserção e aplicar padrões de projeto com agentes usando a arquitetura de *Belief-Desire-Intention* (BDI). A abordagem contempla em um mesmo ambiente três métodos da literatura capazes de detectar e aplicar padrões de projeto em código-fonte escrito em linguagem Java. A fim de comparar o processo de refatoração antes e depois da aplicação do padrão de projeto foi contemplado na abordagem a avaliação de métricas relacionadas aos atributos de qualidade tais como manutenibilidade, reusabilidade e confiabilidade. A Codice-Unio foi implementada em um framework para agentes que suporta a arquitetura BDI e usou ferramentas específicas para leitura de código e avaliação dos atributos de qualidade. A abordagem foi avaliada com cenários de testes providos pelos métodos da literatura e posteriormente por projetos *open-source* encontrados na web via *GitHub*. Como resultado, a Codice-Unio é capaz de identificar e aplicar padrões de projeto em classes candidatas automaticamente em cerca de aproximadamente 97% dos projetos que foram submetidos ao experimento.

Palavras-chave: Agentes. Refatoração de software. Padrões de projeto.

ABSTRACT

PACHER, Thyago Henrique. **Codice-Unio:** an integrated approach of *method*s for detecting and inserting design *patterns* in source *code* using agents. 2020. 142 p. Thesis (Master's Degree in Computer Science) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2020.

The refactoring process guarantees a higher quality in the source code, increasing its maintainability, readability and flexibility. According to the literature about 70% of the cost of the software is for maintenance and this can be reduced using techniques that allow to increase the quality of the source code such as refactoring based on design standards. It was found through a systematic mapping that the works in the literature for the detection and insertion of design patterns are not carried out autonomously. This work created the Codice-Unio approach to detect insertion points and apply design patterns with agents using the Belief-Desire-Intention (BDI) architecture. The approach contemplates in the same environment three methods from the literature capable of detecting and applying design patterns in source code written in Java language. In order to compare the refactoring process before and after the application of the design standard, the assessment of metrics related to quality attributes such as maintainability, reusability and readability was contemplated in the approach. Codice-Unio was implemented in an agent framework that supports the BDI architecture and used specific tools for reading code and evaluating quality attributes. The approach was evaluated using test scenarios provided by the literature methods and later by open-source projects found on the web via GitHub. As a result, Codice-Unio is able to automatically identify and apply design patterns in candidate classes in approximately 97% of the projects that have undergone the experiment.

Keywords: Agents. Software refactoring. Design patterns.

LISTA DE FIGURAS

Figura 1 - Passos para Criação da Codice-Unio 17
Figura 2 - Fluxograma do método Gaitani et al. (2015) 25
Figura 3 - Exemplo de Refatoração com *Null Object* 26
Figura 4 - Fluxograma de análise *Wei et al.* 27
Figura 5 - Exemplo refatoração *Factory Method* 28
Figura 6 - Exemplo de refatoração com *Strategy* 29
Figura 7 - Fluxograma análise *Singleton* 30
Figura 8 - Exemplo refatoração com Singleton 30
Figura 9 - Exemplo de Árvore de Sintaxe Abstrata simples 32
Figura 10 - AST - Implementação com *JavaParser* 33
Figura 11 - Exemplo de criação de AST básico a partir de string de código-fonte 34
Figura 12 - Exemplo do uso da classe CompilationUnit 34
Figura 13 - Funcionamento BDI 44
Figura 14 - Iniciar agente *Jadex* por meio da linguagem *Java* 48
Figura 15 - Declaração de planos via anotações - *Jadex*...49
Figura 16 - Criação de um plano com uma classe *Java*...50
Figura 17 - Objetivos via anotações *Jadex* 50
Figura 18 - Declaração de classe de agente em *Jadex* 51
Figura 19 - Exemplo de classe Agente com Crenças..51
Figura 20 - Exemplo de um agente em Jadex (Agente Tradutor).............................52
Figura 21 - Critérios de exclusão para seleção 60
Figura 22 - Critérios para inclusão de trabalhos...61
Figura 23 - Fluxograma para seleção de trabalhos 61
Figura 24 - Arquitetura ReLE..78
Figura 25 - Comunicação na Arquitetura ReLE 80
Figura 26 - Funcionamento da Refatoração Usando Peixe-Espada 81
Figura 27 - Agente de refatoração - Peixe Espada 82
Figura 28 - Gerente local de Agentes...82

Figura 29 - Detecção de *bad smells* e refatoração do estudo S11 83
Figura 30 - Arquitetura do sistema - S6..86
Figura 31 - Visão geral da Codice-Unio 91
Figura 32 - Agente proposto..93
Figura 33 - Pacotes e classes do Agente 97
Figura 34 - Pacotes e classes do Agente 101
Figura 35 - Declaração de desejos para o agente...105
Figura 36 - Declaração de desejo para uso na classe de agente *Jadex* 105
Figura 37 - Desejo DirectoryQualifier do Agente 106
Figura 38 - Plano DirectoryPlan do Agente 106
Figura 39 - Declaração de Crença - directoryExists - Agente 107
Figura 40 - Implementação método de análise Singleton 107
Figura 41 - Método java para aplicar Padrão de Projeto Singleton 108
Figura 42 - Apresentação de resultados do agente após o processo de refatoração 108
Figura 43 - Aviso de classes modificadas do agente ao usuário final 115
Figura 44 - Aviso do sistema operacional de arquivos modificados 116
Figura 45 - Fluxo de uma Nova Proposta para o Agente no Processo de Refatoração...119
Figura 46 - Adaptação ao agente para separar métricas 120

LISTA DE GRÁFICOS

Gráfico 1 - Arquiteturas para Agentes 68

Gráfico 2 - Quantidade de projetos em que se detectou e aplicou 110

Gráfico 3 - Quantidade de Classes candidatas encontradas para refatoração 111

Gráfico 4 - Relação de Padrões de Projeto Aplicados X Qtd. Total de Classes......112

Gráfico 5 - Relação antes e depois da refatoração para atributo Reusabilidade.....113

Gráfico 6 - Antes e depois da refatoração para o atributo Manutenibilidade 114

Gráfico 7 - Antes e depois da refatoração métrica para o atributo Confiabilidade...114

Gráfico 8 - Quantidade de Code Smells reduzidos por projeto com o uso da Codice-Unio..118

LISTA DE QUADROS

Quadro 1 - Método de Refatoração fundamentado em Padrões de Projeto (2018) ..23

Quadro 2 - Trabalhos relacionados a revisão ou mapeamento sobre refatoração 37

Quadro 3 - Trabalhos por ano de revisão e/ou mapeamento sobre refatoração de software 39

Quadro 4 - Implementação do BDI no framework *Jadex* 48

Quadro 5 - Entidades para modelagem TDF 54

Quadro 6 - Bases e quantidade de estudos 58

Quadro 7 - Estudos duplicados 62

Quadro 8 - Quantidade de citações por trabalho pesquisado 64

Quadro 9 - Autores X Quantidade de trabalhos 65

Quadro 10 - Quantidade de trabalhos por ano 66

Quadro 11 - Trabalhos que foram excluídos após sua análise 67

Quadro 12 - Separação de estudos por Arquitetura 68

Quadro 13 - Como a arquitetura foi aplicada 69

Quadro 14 - Ferramenta encontradas 71

Quadro 15 - Como o código-fonte foi extraído 72

Quadro 16 - Relação entre trabalhos dos pesquisadores 72

Quadro 17 - Como o experimento foi realizado 73

Quadro 18 - Trabalhos e suas IDE de programação 77

Quadro 19 - Ferramentas para produção de Agentes 77

Quadro 20 - Técnicas de Refatoração para Bad Smells (S10, S11) 84

Quadro 21 - Sumarização dos Trabalhos Relacionados 87

Quadro 22 - Métodos e seus padrões de projeto 95

Quadro 23 - Codice-Unio na arquitetura BDI 95

Quadro 24 - Conjunto de Planos do Agente 96

Quadro 25 - Exemplo de um subplano 98

Quadro 26 - Subplanos para aplicação de padrões de projeto 137

LISTA DE TABELAS

Tabela 1 - Iterações para escolha da String de busca ideal......................................59

Tabela 2 - Projetos realizados teste a refatoração - Github 109

Tabela 3 - Relação antes x depois de aplicar refatoração para métricas de qualidade..115

Tabela 4 - Quantidade de Code Smells - Antes e Depois do Agente.......................117

Tabela 5 - Relação dos Projetos para Realização do Experimento 141

LISTA DE SIGLAS

ACL	*Agent Communication Language*
AST	*Abstract Syntax Tree*
BCEL	*Byte Code Engineering Library*
BDI	*Belief-Desire-Intention*
CC	*Cyclomatic complexity*
CRUD	*Create-Read-Update-Delete*
DAI	*Distributed Artificial Intelligence*
DDG	*Data Derivation Graph*
DeLC	*Distributed e-Learning Center*
DIT	*Depth Inheritance Tree*
FIPA	*Foundation for Intelligent Physical Agents*
IDE	*Integrated Development Environment*
JADE	*Java Agent Development Environment*
JDT	*Java Development Tools*
LOC	*Line of Code*
LTK	*Language Toolkit*
MAS	*Multi-agent Systems*
MaSE	*Multi-agent System Engineering*
PDE	*Plug-in Development Environment*
RA	*Refactoring Agent*
ReLE	*Refactoring eLearning Environment*
RKB	*Refactoring Knowledge Base*
RMA	*Remote Agent Management*
RUP	*Rational Unified Process*
SOAP	*Simple Object Access Protocol*
SQL	*Structure Query Language*
TDF	*Tactics Development Framework*
WMC	*Weight Method Class*
WSDL	*Web Services Description Language*

SUMÁRIO

1 INTRODUÇÃO ... 13
1.1 JUSTIFICATIVA ... 15
1.2 OBJETIVOS ... 16
1.3 METODOLOGIA ... 16
1.4 ORGANIZAÇÃO DA DISSERTAÇÃO ... 19
2 REFATORAÇÃO DE SOFTWARE ... 20
2.1 A IMPORTÂNCIA DO PROCESSO DE REFATORAÇÃO ... 21
2.2 REFATORAÇÃO COM PADRÕES DE PROJETO ... 22
2.3 MÉTODOS DE REFATORAÇÃO PARA INSERÇÃO E DETECÇÃO DE PADRÕES DE PROJETO ... 23
2.3.1 Método Gaitani *et al.* ... 25
2.3.2 Método Wei *et al.* ... 27
2.3.3 Método Ouni ... 29
2.4 ABORDAGENS DE EXTRAÇÃO DE CÓDIGO-FONTE USADOS EM REFATORAÇÃO ... 31
2.4.1 Extração via AST ... 31
2.5 TRABALHOS RELACIONADOS A REFATORAÇÃO DE SOFTWARE ... 34
2.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO ... 39
3 AGENTES INTELIGENTES ... 41
3.1 DEFINIÇÃO E TIPOS DE AGENTES ... 41
3.2 ARQUITETURA PARA DESENVOLVIMENTO ... 42
3.2.1 Arquitetura BDI (*Belief, Desire e Intention*) ... 45
3.3 FERRAMENTAS PARA CRIAÇÃO DE AGENTES ... 46
3.3.1 Framework Jadex ... 47
3.4 METODOLOGIA PARA DESENVOLVER AGENTES ... 52
3.4.1 Metodologia Prometheus ... 53
3.5 CONSIDERAÇÕES FINAIS DO CAPITULO ... 55
4 ESTADO DA ARTE ... 56
4.1 METODOLOGIA DE PESQUISA ... 56
4.1.1 Questões de Pesquisa ... 57
4.1.2 Bases e Quantidade de Retorno ... 58
4.1.3 *Strings* de Busca ... 58
4.1.4 Seleção dos Estudos ... 59

4.2 RESULTADOS 63
4.2.1 Respostas as Questões 63
4.2.2 Lições Aprendidas 75
4.3 TRABALHOS RELACIONADOS 76
4.4 CONSIDERAÇÕES FINAIS DO CAPITULO 87
5 CODICE-UNIO: ABORDAGEM DE REFATORAÇÃO PROPOSTA 89
5.1 VISÃO GERAL 89
5.2 REQUISITOS PARA FUNCIONAMENTO DA CODICE-UNIO 92
5.3 PROCESSO DE FUNCIONAMENTO DA CODICE-UNIO 93
5.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO 99
6 RESULTADOS 100
6.1 IMPLEMENTAÇÃO DA CODICE-UNIO 100
6.2 AVALIAÇÃO DA ABORDAGEM PROPOSTA.......................................109
6.3 PROPOSTA PARA EXTENSÃO DA ABORDAGEM PROPOSTA 118
6.4 CONSIDERAÇÕES FINAIS DO CAPITULO 120
7 CONCLUSÃO 122
7.1 TRABALHOS FUTUROS 123
REFERÊNCIAS 125
APENDICE A - SUBPLANOS ENVOLVENDO PADRÕES DE PROJETO 136
APENDICE B - PROJETOS PARA O EXPERIMENTO DE P11 A P60 140

1 INTRODUÇÃO

Refatorar o código visa a criação de uma melhor estruturação sem alterar suas saídas, ou seja, seu comportamento final. O projeto com código-fonte de qualidade baixa irá exigir maior tempo de manutenção e torna a leitura mais difícil para os desenvolvedores. Os motivos para refatorar o código-fonte são: i) tornar mais fácil a adição de código novo; ii) melhorar o projeto de código existente; iii) obter melhor entendimento e iv) deixar a programação sem duplicações (KERIEVSKY, 2009).

Na literatura existem vários exemplos de autores que indicam como refatorar código-fonte, tal como a refatoração por *code smells* baseada em Fowler (1999) e a refatoração com padrões de projeto em Kerievsky (2008). A aplicação de padrões de projeto em geral visa melhorar a estruturação do código ao usar o conhecimento previamente registrado por pessoas mais experientes.

Ferramentas de refatoração são utilizadas para facilitar o processo mudança no código-fonte, dentre essas, muitas são pouco apreciadas por engenheiros de software, pois acabam alterando seu fluxo de trabalho e apresentam refatorações ditas como incorretas por alguns desenvolvedores. Estas ferramentas têm como objetivos em sua maioria, pequenas refatorações já definidas na literatura e não a implementação de padrões que provêm uma melhora nos requisitos de qualidade como: confiabilidade, manutenção, baixo acoplamento, alta coesão e reutilização (FOWLER, 1999; MENS; TOURWÉ, 2004).

Beluzzo (2018) propôs em seu trabalho a integração em um único ambiente os métodos de inserção e detecção de padrões de projeto, porém não agem de forma autônoma. Por isto, para se ter uma abrangência maior dos trabalhos já desenvolvidos sobre o assunto de detecção e inserção de padrões de projeto autônoma foi realizado um mapeamento sistemático abrangendo os anos de 1998 a 2020. O método de mapeamento adotado usou como base os métodos de Kitchenham e Charters (2007). Os trabalhos tais como Sandalski (2011), Stoyanov (2012), Lakshmi (2013), Santos (2015), Neto (2015), entre outros, abordam refatoração com agentes em nível de *code smells* e não padrões de projeto, em que utilizam técnicas como: As ferramentas da literatura que refatoração para *code smells* tais como *Replace Temp with Query*, *Extract Method*, *Preserve Whole Object*, entre outras.

Um agente é um sistema de computador que está situado em algum ambiente e é capaz de realizar ações de forma autônoma a fim de realizar seus objetivos e assim auxiliar o ser humano (WOOLDRIDGE, 2009; COPPIN, 2015). O agente pode ser implementado usando arquiteturas tais como: Subsunção, Reativa ou não-deliberativa, Hibrida e a BDI *(Belief, Desire, Intention)* (JUCHEM, 2001). A arquitetura de Subsunção (BROOKS, 1986) é voltada para agentes reativos e implementados com agentes inteligentes. A arquitetura reativa ou não-deliberativa é usada quando o agente tem percepções do ambiente e passa a tomar decisões a partir de informações obtidas por sensores (BASTOS, 1998). A arquitetura híbrida age como abordagem deliberativa e reativa ao mesmo tempo e permite ao agente ter a capacidade de reagir, raciocinar e planejar (BASTOS, 1998). A arquitetura *Belief-Desire-Intention* (BDI) (BRATMAN, 1988) cria um agente racional no qual sua capacidade de raciocínio é dividida em camadas: crença, desejo e intenção.

A Codice-Unio foi concebida utilizando métodos de refatoração em um único ambiente e de forma que o processo de refatoração ocorra de forma autônoma, tendo como finalidade ajudar a:

- Disponibilizar em um ambiente os métodos de refatoração de: Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017). Os métodos detectam e aplicam os padrões *Null Object*, *Strategy* e *Factory Method, Factory Method; Singleton* e *Strategy.*
- Retornar valores relacionados as métricas de atributos de qualidade de código: manutenibilidade, reusabilidade, e confiabilidade após o agente realizar o processo de verificação. As métricas foram usadas para avaliar o antes e o depois do processo de refatoração realizado pela Codice-Unio.
- Realizar a análise do código sem exigir que o usuário importe seu projeto e aplicar os padrões que forem considerados aplicáveis de acordo com raciocínio autônomo.

Dentre as arquiteturas para implementação de agentes, este trabalho utilizou a BDI em que foi analisado como o processo de refatoração pode ser distribuído nas camadas de crença, desejo e intenções. O desejo é realizar o processo de refatoração fundamentado em padrões de projeto, as crenças guardam informações sobre o projeto tais como classes candidatas, propriedades da classe, entre outras e as intenções são os procedimentos para a identificação de classes candidatas a

aplicação do padrão. O agente proposto realiza de forma autônoma o monitoramento do código-fonte e por meio dos métodos de refatoração que foram implementados, refatora o código-fonte e gera informações sobre os atributos de qualidade antes e depois do processo de refatoração.

A implementação da Codice-Unio foi realizada usando o framework *Jadex* (PIUNTI, 2008) e para os experimentos se utilizou exemplos da literatura e projetos disponíveis no GitHUB (2018). Foram analisados 60 (sessenta) projetos, totalizando 8.206 (oito mil e duzentos e seis) classes. Dos 60 (sessenta) projetos testados, 97% foram refatorados pela Codice-Unio.

1.1 JUSTIFICATIVA

O propósito geral da refatoração é prover uma boa estrutura do sistema sem mudar seu comportamento final. Algumas das possibilidades de mudanças são: i) fragmentar métodos; ii) reestruturar a hierarquia das classes; iii) redefinir nomes de métodos e classes; etc. A refatoração é um processo usado para eliminação de *bad smells* e fazer a possível melhoria de confiabilidade e manutenibilidade do código para facilitar futuras modificações (FOWLER, 1999).

As ferramentas voltadas para refatorar são usadas para desenvolver softwares, porém são pouco apreciadas por engenheiros de software pois podem apresentar refatorações ditas como incorretas por alguns desenvolvedores.

A criação de ferramentas que auxiliam no processo de refatoração e realizam o processo de forma autônoma, em um ambiente integrado com vários métodos de *design patterns* para detecção e inserção de padrões de projeto, torna algo relevante na produção de produtos de software de maior qualidade, pois realiza a refatoração enquanto o usuário continua a trabalhar na criação de seu código-fonte. Esta automatização pode ser realizada por um agente que é capaz de executar uma tarefa a fim de ajudar o desenvolvedor na produção de códigos do projeto que está desenvolvendo.

1.2 OBJETIVOS

O objetivo geral deste trabalho é criar uma abordagem fundamentada em agentes que possa detectar e inserir padrões de projeto em código-fonte orientado a Objetos usando os métodos da literatura que identificam e aplicam os padrões de projeto. Os objetivos específicos são:

- Realizar um mapeamento sistemático sobre o assunto de refatoração com Agentes e *Design Patterns.*
- Implementar e realizar experimentos com a Codice-Unio.
- Analisar os benefícios e dificuldades do uso da Codice-Unio.

1.3 METODOLOGIA

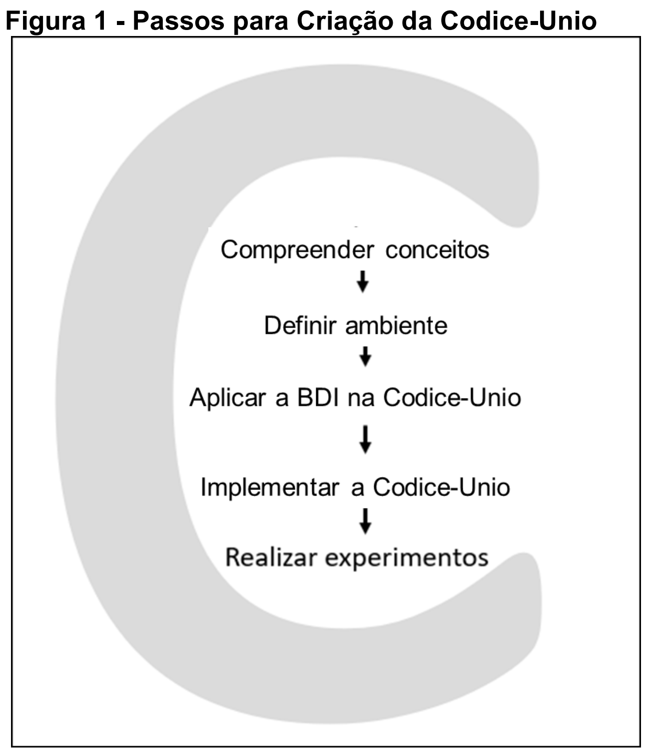
A metodologia para o desenvolvimento deste trabalho foi dividida em duas etapas: uma para revisão da literatura e outra para a criação da abordagem.

A revisão da literatura, para entender o estado da arte sobre o assunto de refatoração baseada em padrões de projeto de forma autônoma, foi realizada por meio de um mapeamento sistemático de 1998 a 2020 usando o método Kitchenham e Charters (2007). O ano de 1998 foi escolhido porque a partir desta data os trabalhos de refatoração começaram a ser divulgados. O mapeamento possibilitou identificar os assuntos que ainda não haviam sido propostos ou testados em relação ao tema de refatoração de software automatizada. Um destes assuntos é a criação de uma abordagem de refatoração de software com a aplicação de padrões de projeto e avaliação do código-fonte de forma autônoma.

A metodologia para a criação da abordagem proposta, Codice-Unio, seguiu os passos ilustrados na Figura 1. Inicialmente foram estudados os conceitos relacionados a agentes, métodos de refatoração fundamentados em padrões de projeto e atributos de qualidade. Em relação a agentes, estudou-se sobre agentes inteligentes, arquitetura para criação de agentes e metodologias de modelagem. Optou-se pela arquitetura BDI (*Belief, Desire e Intention*) (BRATMAN, 1988) porque permite separar e estruturar as funcionalidades do agente em camadas e são usadas em ambiente dinâmicos, no qual recebem estímulos do ambiente para realizar suas

intenções, fundamentadas em suas crenças e analisa seus desejos para atingir seus objetivos. A modelagem foi realizada pelo Prometheus (AMBIEL, 2010) porque permite desenvolver modelos de agentes com menos fases, porém contempla as funcionalidades internas que compõem o agente.

Figura 1 - Passos para Criação da Codice-Unio



Fonte: Autoria própria

Em relação aos métodos de refatoração que existem na literatura foram identificados no trabalho de Beluzzo (2018) um total de 21 (vinte e um) e na abordagem foram escolhidos 3 (três) inicialmente, mas outros podem ser incorporados futuramente. Os métodos de Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017) foram escolhidos porque contém o processo de detecção e aplicação de padrões de projeto comuns de serem usados em códigos-fonte. Os métodos apresentam refatoração de programas escritos em linguagem Java, por isto, é a linguagem suportada pela Codice-Unio a fim de realizar análise dos experimentos.

Considerando os atributos de qualidade foram estudadas as métricas para avaliar: manutenibilidade, reusabilidade e confiabilidade do código-fonte (SOMMERVILLE, 2011). Estes atributos foram selecionados por serem os mais importantes no processo de refatoração e permitem verificar a qualidade do código-fonte. Este trabalho utilizou as métricas para avaliar o código-fonte dos projetos antes

e depois do processo de refatoração a fim de identificar se a Codice-Unio contribui para melhorar estes atributos.

No passo de Definir o Ambiente foram escolhidos dois frameworks, um de implementação de agentes e outro para desenvolvimento de aplicação. O *Jadex* (PIUNTI, 2008) foi selecionado por ser um framework que suporta a arquitetura BDI de forma nativa sem necessidade de adaptações, possibilidade de inicialização por código-fonte Java e versões atualizadas estão sendo disponibilizadas. No Jadex não existe a implementação de intenções, mas sim de planos e subplanos. A *Eclipse IDE Compiler* (ECLIPSE IDE, 2019) foi o ambiente para desenvolvimento da aplicação porque suporte projetos com agentes e programas escritos em várias linguagens, entre elas Java.

A arquitetura BDI para o processo de refatoração foi aplicada e modelado na etapa Aplicar BDI na Codice-Unio, sendo que a crença é o conhecimento que o agente tem sobre o código-fonte, o seu desejo é realizar a refatoração e calcular os valores para os atributos de qualidade, e a sua intenção representa a aplicação da refatoração propriamente dita nas classes que foram identificadas como candidatas a aplicação pelos métodos da literatura. Nesta etapa utilizou-se ferramentas como AST (*JavaParser*, *ASTParser*) (JAVAPARSER, 2019) para extração de código-fonte e leitura das propriedades relacionadas a classe lida, *CkMetrics* (GITHUB, 2018) biblioteca Java encontrada para retornar as métricas Complexidade Ciclomática, Profundidade da Árvore de Herança, e Tamanho do Número de Linhas do Código, as quais são usados para medir os atributos de qualidade manutenibilidade, reusabilidade e confiabilidade.

A Codice-Unio foi implementada (passo Implementar a Codice-Unio) para que fosse possível a realização de sua avaliação. Sua implementação foi realizada em uma arquitetura de camadas conforme prevê a arquitetura BDI, utilizando as ferramentas que foram escolhidas na etapa Definir o Ambiente.

Por fim, o último passo Realizar Experimentos foi realizado em duas avaliações: a primeira foi executada por meio da obtenção de candidatos a refatoração em um cenário simples de aplicação usando como referências os exemplos de projetos descritos nos trabalhos de Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017). É importante ressaltar que os projetos que foram refatorados pelos métodos da literatura possuíam no máximo 40 (quarenta) classes. A segunda avaliação foi

executada aplicando a refatoração em 60 (sessenta) projetos *open-source* disponíveis na web no GitHUB.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Este trabalho está organizado em sete Capítulos. O Capítulo 2 apresenta o referencial teórico sobre refatoração e seus benefícios. O Capítulo 3 descreve sobre agentes inteligentes e como podem ser implementados. O Capítulo 4 detalha o processo de estado da arte com o mapeamento sistemático sobre ferramentas de refatoração de agentes voltadas para *code smells* e padrões de projeto. O Capítulo 5 apresenta o funcionamento da Codice-Unio. O Capítulo 6 descreve os resultados desta pesquisa. O Capítulo 7 expõe as conclusões sobre a abordagem Codice-Unio, bem como os possíveis trabalhos futuros.

2 REFATORAÇÃO DE SOFTWARE

O processo de refatoração é importante porque garante uma qualidade maior no código-fonte aumentando a sua manutenibilidade, confiabilidade, flexibilidade, entre outros atributos de qualidade. Segundo Fowler (1999), este é um processo disciplinado de alteração de código-fonte que minimiza a chance de introdução de erros e melhora a sua estrutura depois de ter sido escrito. Segundo Martin (2009), o processo de refatoração ajuda a manter o código-fonte limpo e aumenta a coesão e diminui o acoplamento.

A refatoração pode ser conduzida com técnicas em geral, ou padrões de projeto (*Design Patterns*). Fowler (1999) criou um catálogo de técnicas de refatoração separado nos seguintes grupos: *composing methods, moving features between objects, organizing data, simplifying conditional expressions* e *dealing with generalizations*. Cada grupo de técnicas definidas por Fowler tem mais de uma técnica envolvida, por exemplo o grupo *composing methods* tem as técnicas de *Extract Method, Inline Method, Replace Temp with Query,* entre outras. Já a refatoração baseada em padrões de projeto é fundamentada no catálogo de Gamma (1995) e existem na literatura métodos que ajudam a identificar e aplicar padrões de projeto tais como de Cinnéide e Nixon (1999), Kerievsky (2009), Ouni *et al.* (2017).

A Seção 2.1 aborda a importância de refatoração em código-fonte tanto de sistemas legados quanto em aqueles que estão em desenvolvimento. É apresentado como os desenvolvedores tem usado ferramentas semi-automatizadas para refatoração e descreve algumas das abordagens para extrair informações do código-fonte comuns da literatura. A Seção 2.2 descreve a refatoração baseada em padrões de projeto. Os métodos de detecção e inserção de padrões de projeto em código-fonte são explicados na Seção 2.3. As abordagens de extração de código-fonte são explicadas na Seção 2.4. A Seção 2.5 apresenta os trabalhos relacionados a refatoração de software. Por fim, as considerações finais deste capitulo foram descritas na Seção 2.6.

2.1 A IMPORTÂNCIA DO PROCESSO DE REFATORAÇÃO

Meyer (1997) comenta que 70% do custo de software é voltado para sua manutenção. Mall (2018), relata que o custo de manutenção é de 60% do custo total no ciclo de vida de um produto de software típico e salienta que estes custos variam muito de um domínio de aplicativo para outro, ou seja, em sistemas embarcados o custo pode ser de 2 (dois) a 4 (quatro) vezes o custo de desenvolvimento. Isto mostra que é essencial a redução dos custos de manutenção, e de acordo com Ferreira (2008), pode ser obtida quando o processo de realizar alterações no software ficar mais fácil.

De acordo com Trifu (2004), o software sofre manutenção e como consequência é observado que o seu projeto degrada continuamente, tornando a manutenção e extensões funcionais caras. Abordagens são utilizadas para ajudar no processo de manutenção tal como o uso de engenharia reserva por meio da refatoração.

Chikofsky (1990) afirma que a engenharia reversa pode reduzir o custo geral do software, evitando que se gaste tempo no entendimento da estrutura, tornando-o mais legível e compreensível. Para Fowler (1999) refatorar é importante pois:

- Melhora a qualidade da estrutura de software: ajuda a remover códigos que não estão em lugar certo. As pessoas mudam o código sem refatorar a estrutura e isto pode o tornar difícil de ser compreendido.
- O software fica mais fácil de ser compreendido: torna mais fácil o entendimento do código-fonte evitando ações desnecessárias para o mesmo.
- Ajuda a encontrar possíveis erros de código-fonte: para alguns programadores é mais complicado analisar o código e encontrar alguma falha. Executar a refatoração pode corrigir possíveis erros de programação.
- Auxilia o desenvolvimento de código mais rápido: a boa estruturação é essencial para manter a produtividade em questão de desenvolvimento de software. Ela ajuda porque o *design* do sistema fica melhor estruturado.

2.2 REFATORAÇÃO COM PADRÕES DE PROJETO

Alguns padrões de projeto são importantes de serem aplicados devido a evolução natural do software e adição de novas funcionalidades. De acordo com Fowler (2006), um padrão é um conjunto de boas práticas que foram aplicadas e validadas e que podem ser reusadas em outros projetos.

Para Kerievsky (2009), desenvolvedores podem não ter conhecimento do funcionamento de alguns padrões e com isso o projeto construído não apresenta atributos como manutenibilidade, confiabilidade, flexibilidade entre outros. Ao usar padrões de projeto ao invés de estar pensando no reuso de código-fonte, o desenvolvedor terá a reutilização de uma experiência já conhecida por outra pessoa naquele mesmo problema (FREEMAN *et al.*, 2009).

Alguns exemplos de padrões de projeto que são encontrados na literatura e no catálogo de Gamma *et al*. (1995) são de *Abtract Factory, Builder, Factory Method, Prototype, Singleton, Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor*.

De acordo com Gamma *et al*. (1995) aplicar padrões de projeto pode ajudar a resolver problemas do dia a dia que desenvolvedores de programas orientados a objetos enfrentam tais como:

- Encontrar objetos apropriados: melhora o encapsulamento de classes.
- Determinar granulidade do objeto: objetos muito grandes em tamanho e número podem ser divididos com *Facade* (criando subsistemas) ou *Flyweight* (que proporciona entrega em partes de objetos grandes, sendo estes: imagens, documentos ou coisas do gênero).
- Especificar interface do objeto: caracteriza um conjunto completo de solicitações que podem ser enviados a um objeto. Qualquer pedido que corresponda a assinatura para interface do objeto pode ser enviado para o objeto.
- Especificar implementação do objeto: especifica os dados internos do objeto e a representação definindo as operações que pode executar.

- Programar com interfaces e não com implementações: permite que obtenha novas implementações herdando a maior parte do que precisa das classes existentes.
- Reforçar o reuso para as classes: com os padrões de projeto é feito a construção de software e projetos flexíveis e reutilizáveis.

O processo de detecção e inserção de padrões de projeto pode ser feito de forma manual ou automatizada, sendo necessário a aplicação de métodos específicos. Os métodos específicos encontrados na literatura são explorados na próxima Seção.

2.3 MÉTODOS DE REFATORAÇÃO PARA INSERÇÃO E DETECÇÃO DE PADRÕES DE PROJETO

Revisões sistemáticas foram realizadas sobre refatoração de software e estão apresentadas na Seção 2.5, porém elas não abordam métodos de inserção e detecção de padrões de projeto em código-fonte. Por isto, Beluzzo (2018) realizou um mapeamento sistemático sobre este tema e identificou 21 (vinte e um) estudos relacionados. Os métodos encontrados na literatura que refatoram para padrão de projeto estão apresentados no Quadro 1.

Quadro 1 - Método de Refatoração fundamentado em Padrões de Projeto (2018)

Autor	Título	Ano Publicação
Gaitani, M. A. G.; Zafeiris, V. E.; Diamantidis, N. A.; Giakoumakis, E. A.	*Automated refactoring to the Null Object design pattern*	2015
Christopoulou, A.; Giakoumakis, E. A.; Zafeiris, V. E.; Soukara, V.	*Automated refactoring to the Strategy design pattern*	2012
Zafeiris, V. E.; Poulias, S. H.; Diamantidis, N. A.; Giakoumakis, E. A.	*Automated refactoring of super-class method invocations to the Template Method design pattern*	2017
Ouni, A.; Kessentini, M.; Sahraoui, H.; Cinnéide, M. Ò.; Deb, K.; Inoue, K.A.	*A multi-objective refactoring approach to introduce design patterns and fix anti-patterns*	2015

Cinnéide, M. Ó.	*Automated application of design patterns: a refactoring approach*	2001
Cinnéide, M. Ó.; Nixon, P.	*A Methodology for the automated introduction of design patterns*	1999
Mens, T.; Tourwé, T.	*A declarative evolution framework for object-oriented design patterns*	2001
Jeon, S.U.; Lee, J.S.; Bae, D. H.	*An automated refactoring approach to design pattern-based program transformations in Java programs*	2002
Cinnéide, M.Ó.; Nixon, P.	*Automated Software Evolution Towards Design Patterns*	2001
Cinneide, M. Ó.	*Automated refactoring to introduce design patterns*	2000
Liu, W.; Hu, Z.; Liu, H.; Yang, L.	*Automated pattern-directed refactoring for complex conditional statements*	2014
Ram, D.J; Rajesh, J.	*Detecting Intent Aspects from Code to Apply Design Patterns in Refactoring: An Approach Towards a Refactoring Tool*	*2004*
Hotta, K.; Higo, Y.; Kussumoto, S.	*Identifying, tailoring and suggesting form template method refactoring Opportunities with program dependence graph.*	*2012*
Rajesh, J.; Janakiram, D.	*JIAD: A tool to infer design patterns in refactoring*	2004
Ouni, A.; Kessentini, M.; Ó Cinnéide, M.; Sahraoui, H.; Deb, K.; Inoue, K.	*MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells*	2017
Eden, A.H.; Gil, J.; Yehudai, A.	*Precise specification and automatic application of design patterns*	1997
Kerievsky, J.	*Refactoring to Patterns*	2008
Kim, J.; Batory, D.; Dig, D.	*Scripting parametric refactorings in java to retrofit design patterns*	2015
Kim, J.; Batory, D.; Dig, D.	*Scripting Refactorings in Java to Introduce Design Patterns*	2014
Jullerat, N.;Hirsbrunner, B.	*Toward an implementation of the ”Form Template Method” Refactoring*	2007
Ajouli, A.;Cohen, J.;Royer, J.-C.	*Transformations between composite and visitor implementations in Java*	2013

Fonte: Beluzzo (2018)

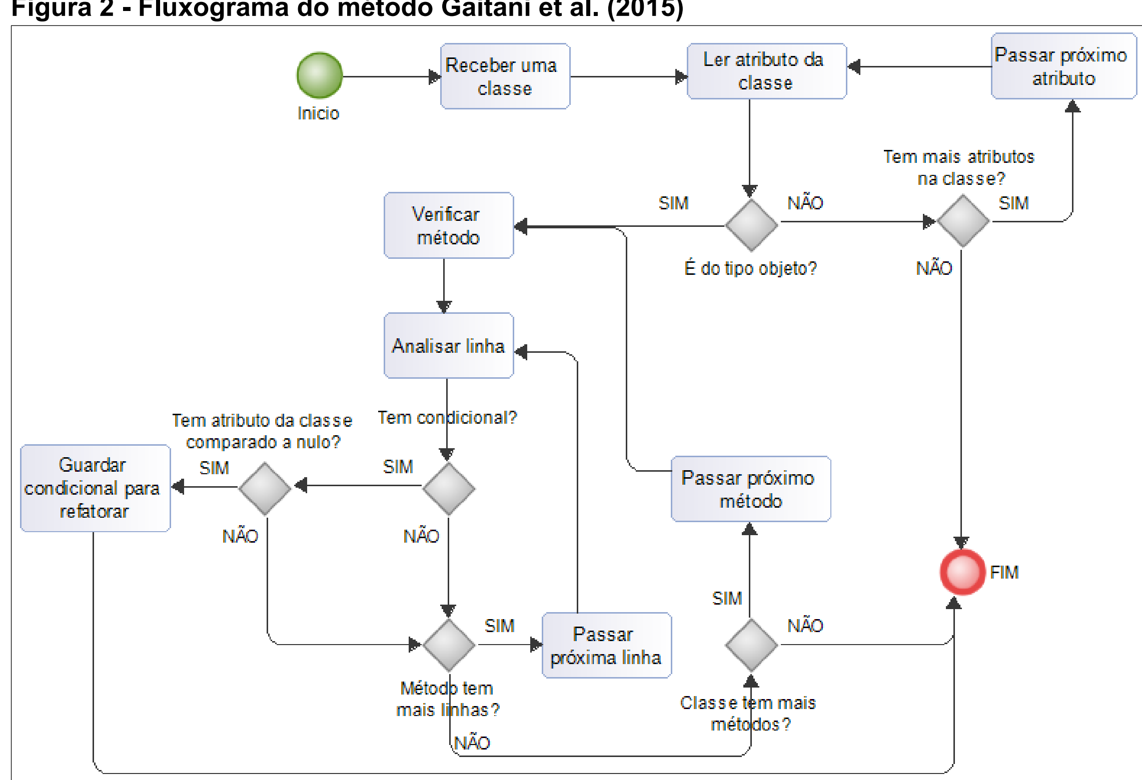
Dentre os métodos da literatura, este trabalho incluiu na Codice-Unio os de: Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017) porque apresentavam os algoritmos de execução do processo de refatoração, os quais são detalhados nas próximas subseções. Esses métodos utilizam como forma de leitura de código-fonte

JavaParser e aplicam os conceitos de AST (Árvore de Sintaxe Abstrata). *JavaParser* e a AST são explicados na Seção 2.4.1.

2.3.1 Método Gaitani *et al*.

O método de Gaitani *et al.* (2015) é apresentado na Figura 2. O processo de análise inicia recebendo uma classe escrita em linguagem Java e por meio desta é executado a sua leitura para detectar se os atributos da classe ou expressões condicionais recebem ou comparam o tipo *null*, respectivamente. Após a execução do processo ilustrado na Figura 2 é realizado a aplicação do padrão de projeto *Null Object*.

Figura 2 - Fluxograma do método Gaitani et al. (2015)

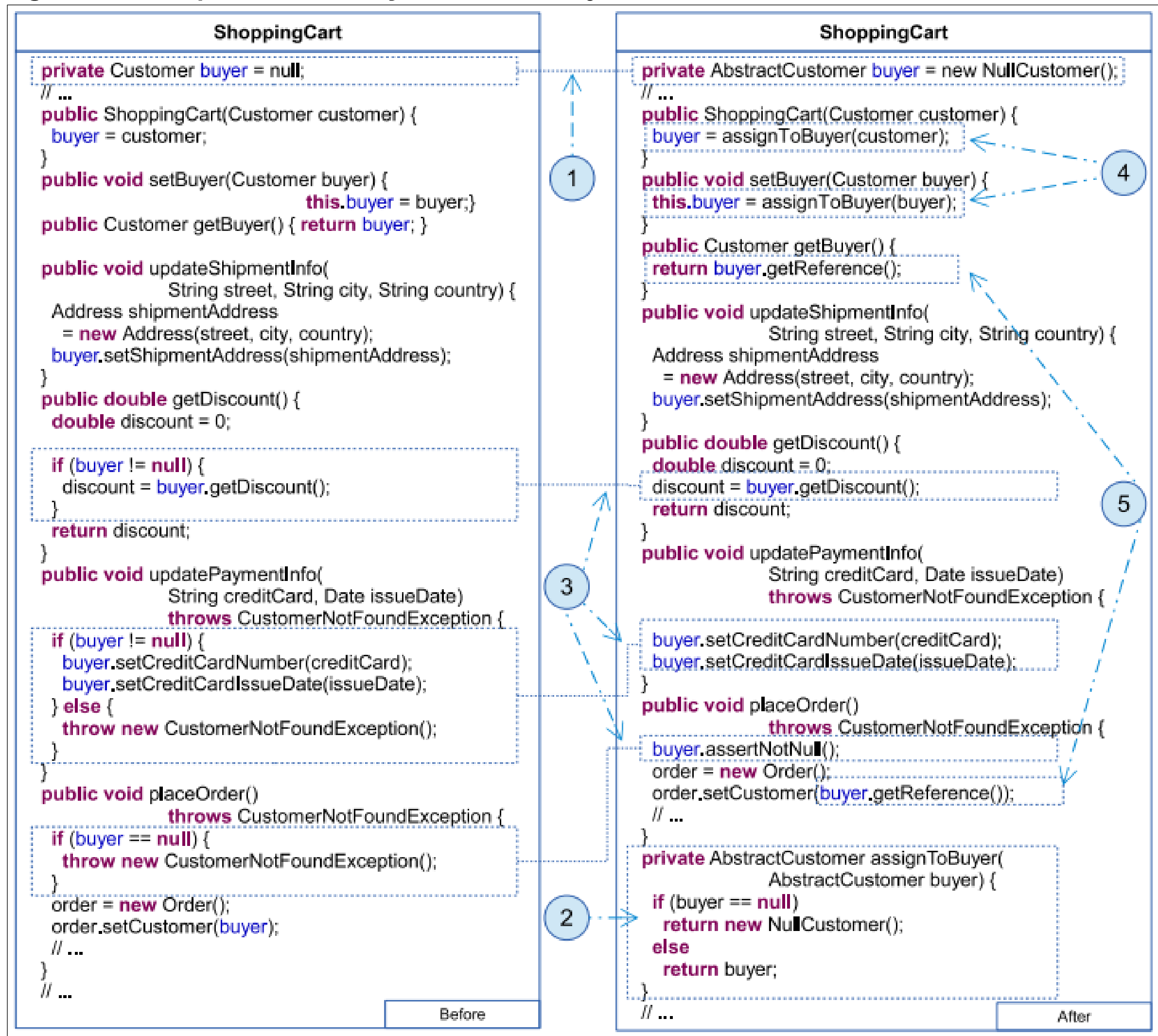


Fonte: Adaptado de *Gaitani et al.* (2015)

A Figura 3 apresenta um exemplo de refatoração com o padrão de projeto *Null Object* na classe "*ShoppingCart*". O lado da esquerda representa o código original e o da direta é o refatorado. Neste exemplo, observa-se que as condicionais e

incrementos foram substituídos. Na primeira linha a classe "*NullCustomer*" é responsável por trazer um objeto nulo do tipo "*Customer*" e foi criado um método na classe "*ShoppingCart*" após refatoração chamado "*assignToBoyer*" responsável por ter a condicional para avaliar se o campo da classe chamado "*buyer*" está ou não preenchido. Caso esteja nulo, ele faz o retorno da classe "*NullCustomer*", senão considera o que está preenchido no campo.

Figura 3 - Exemplo de Refatoração com *Null Object*



Fonte: ***Gaitani et al.* (2015)**

Os benefícios envolvendo o uso do padrão de projeto *Null Object* de acordo com Gaitani *et al*. (2015) são:

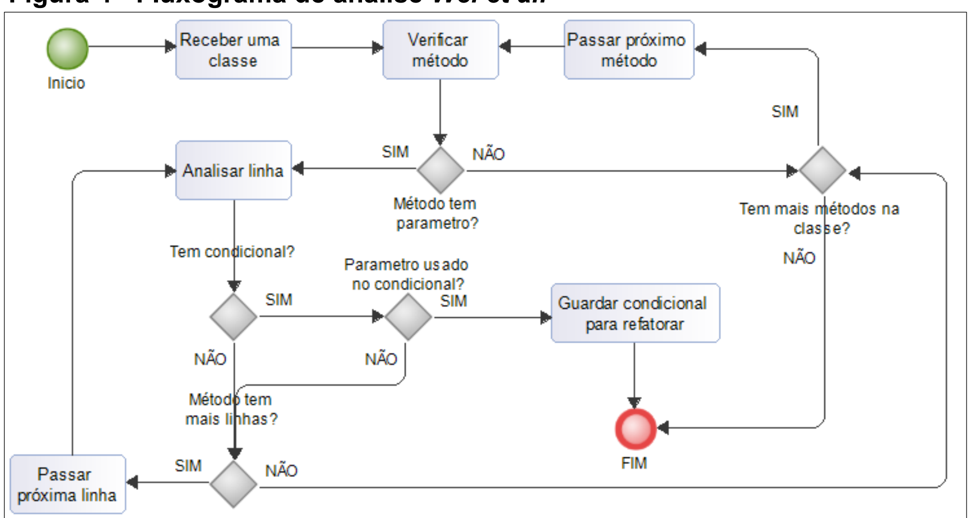
- Reuso do comportamento representado pela classe em diferentes campos opcionais.

- Introdução direcionada a alterações de comportamento padrão executado devido à ausência de um objeto da classe opcional.
- Extensão mais segura da classe contexto que requer chamadas adicionais no campo opcional. A presença da expressão condicional comparado a *null* fornece uma forte dica da opcionalidade do campo e permite que o desenvolvedor evite o código de verificação de nulo em suas invocações.

2.3.2 Método Wei *et al.*

O método de Wei *et al.* (2014) é apresentado na Figura 4 e método aplica os padrões de projeto *Factory Method* e *Strategy.*

Figura 4 - Fluxograma de análise *Wei et al.*

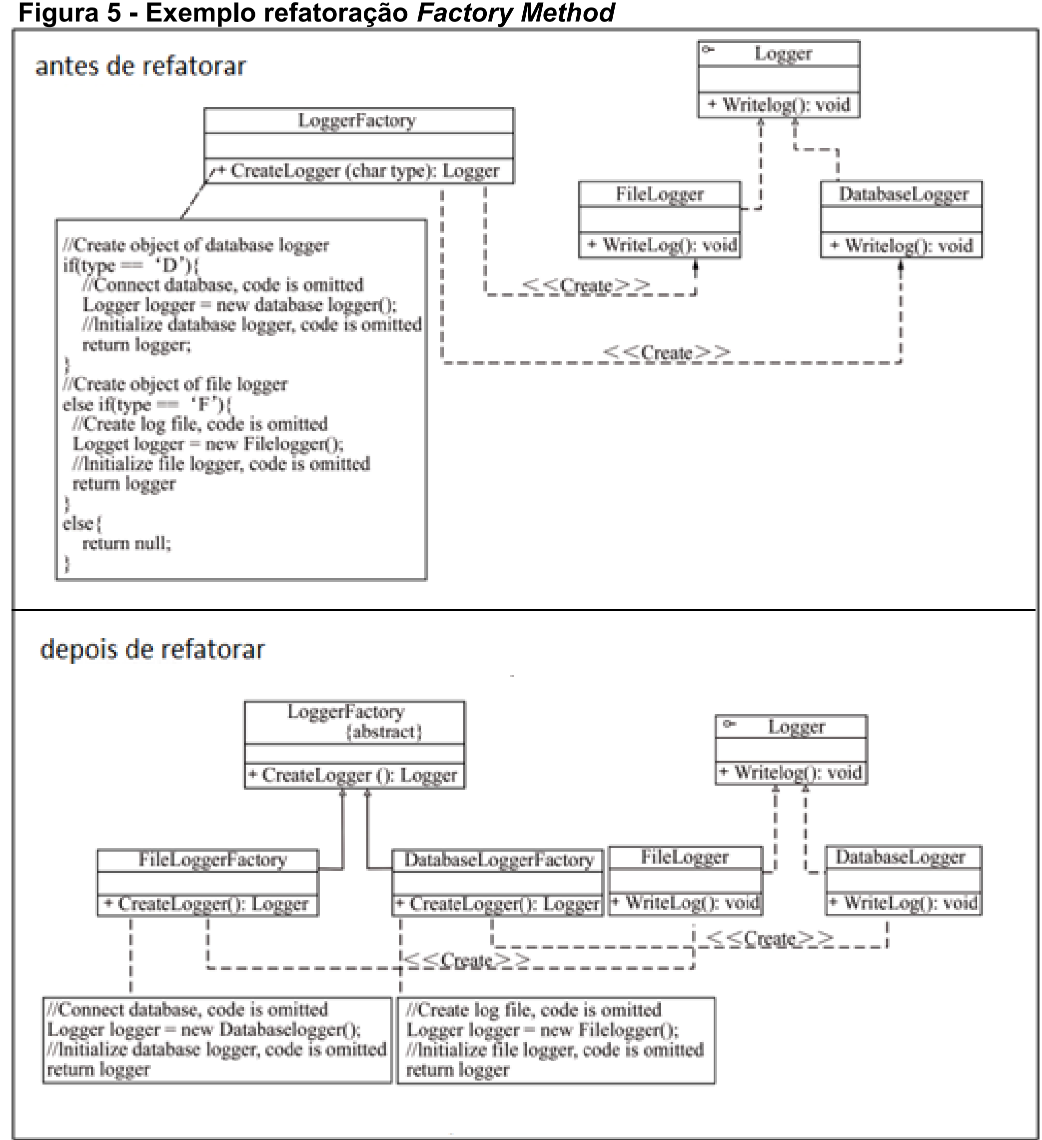


Fonte: Adaptado de Wei *et al.* (2014)

O processo de análise inicia recebendo uma classe escrita em linguagem Java e por meio desta é executado a sua leitura para analisar cada um de seus métodos. O método somente será analisado se apresentar obrigatoriamente ao menos um parâmetro e conter uma expressão condicional. Se o método internamente na condicional conter uma instrução referente a uma criação de um objeto, aplica-se o padrão de projeto *Factory Method,* do contrário, se houver um retorno para tipos de dados específicos o padrão *Strategy* é utilizado.

A Figura 5 apresenta a modificação realizada na estrutura de projeto aplicando o padrão de projeto *Factory Method*. Após a refatoração, foram criadas as classes *FileLoggerFactory, DatabaseLoggerFactory* e o método *CreateLogger* que está dentro da classe *LoggerFactory.* O código refatorado não tem toda a estrutura do método com condicionais.

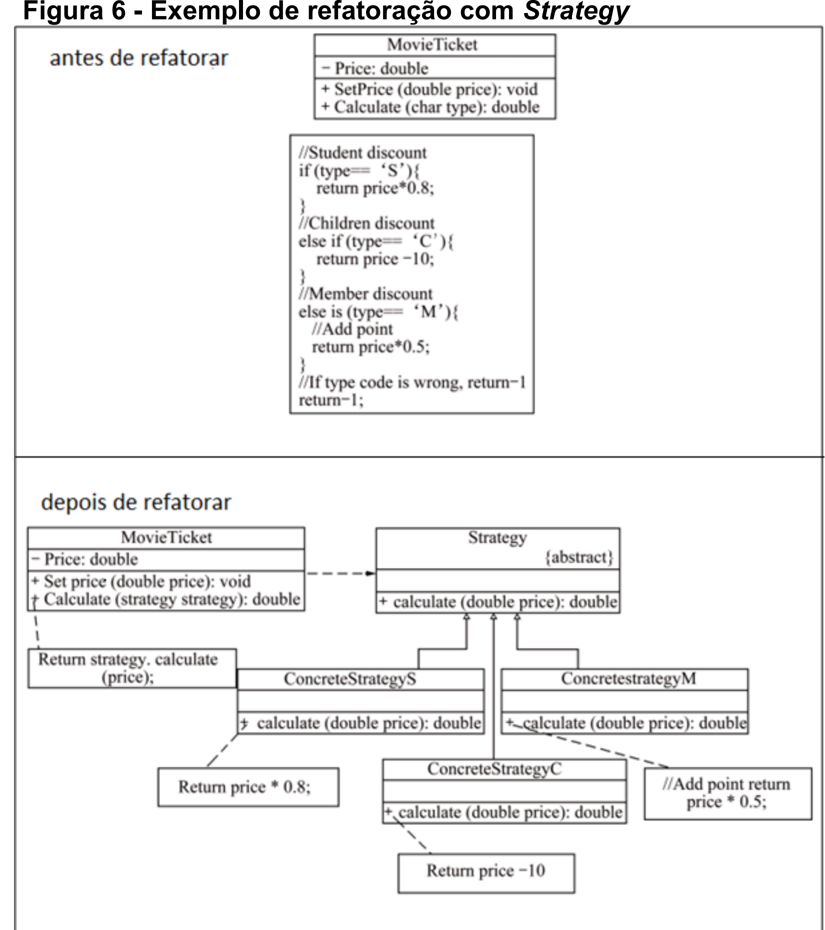
Figura 5 - Exemplo refatoração *Factory Method*



Fonte: *Wei et al.* (2014)

O outro padrão implementado pelo método de Wei *et al.* (2014) é *Strategy* apresentado na Figura 6. Foram criados após a refatoração as classes *Strategy, ConcreteStrategyS, ConcreteStrategyC, ConcreteStrategyM* com a intenção de evitar as condicionais que estavam antes da refatoração no método *Calculate* e dentro da classe *MovieTicket.*

Figura 6 - Exemplo de refatoração com *Strategy*



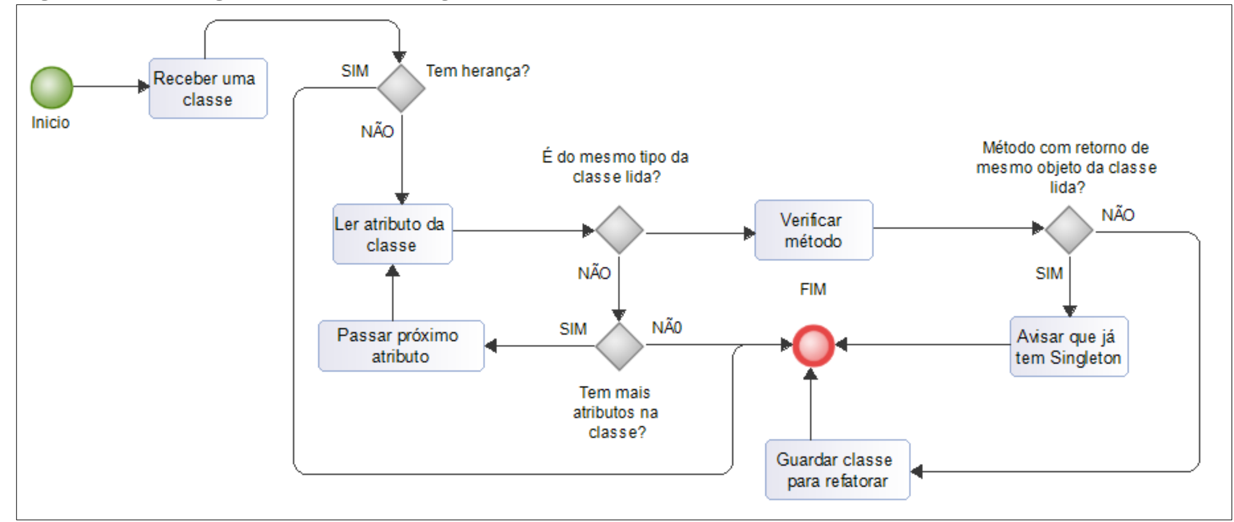
Fonte: ***Wei et al.*** **(2014)**

Para Wei *et al.* (2014) os testes finais com avaliação de qualidade mostram que pode reduzir o tamanho do código além da complexidade de classes.

2.3.3 Método Ouni

O método de Ouni (2017) aplica os padrões de projeto *Visitor, Factory Method, Singleton* e *Strategy.* Para identificar o padrão de projeto *Singleton* foi usado como referência o trabalho de Ajouli (2013). O processo de análise para este padrão é apresentado na Figura 7 e inicia recebendo uma classe escrita em linguagem Java e por meio desta é verificado se apresenta herança. Se sim, o processo é finalizado. Do contrário, cada atributo da classe é lido e verifica se o mesmo é do tipo da classe. Se sim, verifica os métodos da classe para analisar se seu retorno é do tipo da classe. Se for, o processo é finalizado, do contrário o padrão *Singleton* é aplicado.

Figura 7 - Fluxograma análise *Singleton*



Fonte: Adaptado de Ouni (2017)

Na Figura 8 é exibido um exemplo de como é feita a refatoração com padrão de projeto *Singleton.* Nesta o construtor está com o modificador público e após efetuar a refatoração foi alterado para privado e um atributo do tipo classe foi criado para controlar sua instanciação. A instanciação da classe ocorre por meio de um único método público *getInstance().*

Figura 8 - Exemplo refatoração com Singleton

Conexao
+ conexao : Conexao
+ Conexao() C()

Conexao
- conexao : Conexao
- Conexao() C()
+ getInstance(): Conexao

Description [html]

```
if(conexao == null){
    conexao = new Conexao();
}
return conexao;
```

Fonte: Adaptado de Ouni (2017)

Para identificar o padrão de projeto *Visitor* é usado como referência o trabalho de Heuzeroth (2003) e para o *Factory Method* foi usado como referência à análise de classes candidatas do trabalho de Cinnéide (1999). No padrão de projeto *Strategy* de Ouni (2017) quando um método tem uma grande instrução condicional com vários *if* e *else*, sugere que seja extraído cada ramo da instrução condicional em sua própria classe, onde está implementa uma interface comum.

Foram apresentados os 3 (três) métodos que serão utilizados pela abordagem Codice-Unio, a qual por integrar esses métodos é capaz de detectar e inserir os seguintes padrões de projeto: *Null Object*, *Factory Method*, *Strategy*, *Singleton* e *Visitor*. Devido a esses métodos utilizarem processos de extração para leitura de código, a próxima Seção aborda sobre este assunto.

2.4 ABORDAGENS DE EXTRAÇÃO DE CÓDIGO-FONTE USADOS EM REFATORAÇÃO

Beluzzo (2018) na realização do mapeamento sistemático de inserção e detecção de padrões de projeto identificou 5 (cinco) ferramentas para extrair código-fonte. As ferramentas encontradas são: JavaCC (JAVACC, 2018), Prolog Facts, AST, Eclipse IDE Compiler (ECLIPSE IDE, 2019) e Soot Framework. O uso da AST é feito por meio de várias ferramentas tais como *JavaParser*, *ASTParser* entre outras. Neste trabalho é detalhado o *JavaParser* o qual foi utilizado na abordagem Codice-Unio. A biblioteca *JavaParser* foi usada porque é uma biblioteca que implementa as propriedades da AST (BELUZZO, 2018).

2.4.1 Extração via AST

O conceito de AST (*Abstract Syntax Tree*), definido pelo OMG (2011), é uma representação formal da sintaxe em que a estrutura obtida permite expressar relacionamentos de composição e fornecem uma maneira de entender as propriedades diretas e derivadas dos construtores de linguagem. Ainda de acordo com OMG (2011), a AST serve para fornecer um formalismo adequado para derivação de propriedades necessárias à descoberta detalhada de conhecimento.

De acordo com Kozaczynski (1992), os elementos básicos de um programa são obtidos ao analisar a representação de uma AST. Itens usados para aumentar a confiabilidade em código-fonte tais como: recuo, palavras-chave, comentários não são representados na AST. A representação do código é a descrição de sintaxe abstrata em vez de somente uma *string* de texto contendo a linha do código.

A Figura 9 mostra o exemplo fornecido pelo autor Welty (1997) para a construir uma AST simplificada. Nesta figura cada elemento do programa é separado em expressão na linguagem de programação C: “a = a + 1”.

Figura 9 - Exemplo de Árvore de Sintaxe Abstrata simples

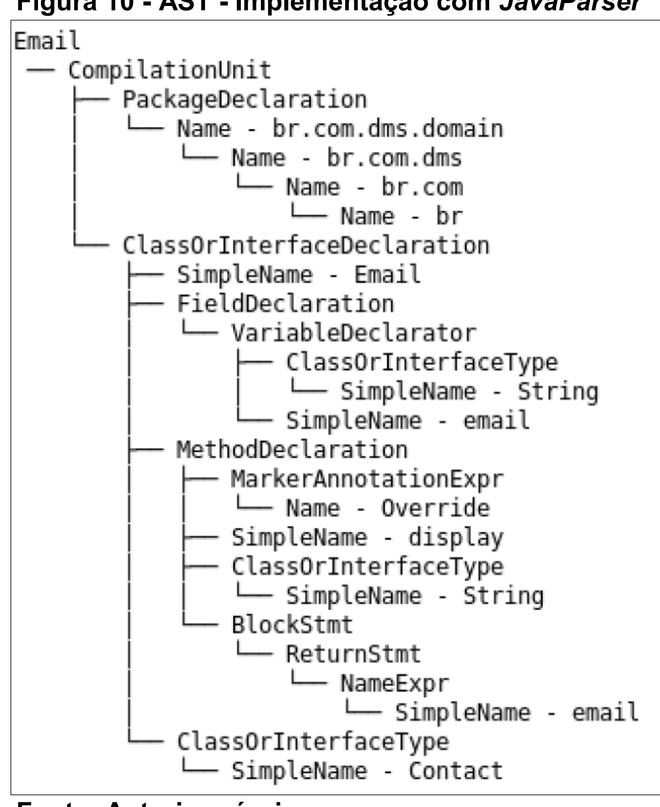
Fonte: WELTY (1997)

A AST estrutura o código-fonte em:

- No raiz da árvore é definido a expressão.
- Segundo nível da árvore é visto a declaração da expressão “a” do lado esquerdo e do lado direito é coloca o sinal de adição “+”.
- Terceiro nível tem 2 (duas) ramificações: o lado esquerdo sendo novamente dado a expressão da variável “a” e do lado direito é dado o número 1.

A Figura 10 demonstra como é a extração de código-fonte com o uso de *JavaParser*.

Figura 10 - AST - Implementação com *JavaParser*



Fonte: Autoria própria

Considerando a Figura 10, a leitura da classe “Email” tem os seguintes elementos:

- *SimpleName*: contém o nome da classe.
- *FieldDeclaration*: responsável por conter os campos da classe.
- *MethodDeclaration*: faz a representação dos métodos envolvidos na classe lida.
- *ClassOrInterfaceType*: tipo de retorno de um campo da classe ou método. Quando os atributos da classe ou do método são lidos o tipo deste atributo pode ser visto sendo: classes nativas do Java tais como *double*, *int* ou até classes de objetos.
- *BlockStmt*: responsável por conter o corpo do método Java que estiver sendo lido.
- *PackageDeclaration*: contém o nome do pacote ao qual a classe pertence.

Outro exemplo usando AST é o *ASTParser* que é um analisador de linguagem Java para criação de árvores de sintaxe abstrata (AST). Esta biblioteca está no núcleo do Eclipse (ASTPARSER, 2019). A Figura 11 exibe a estruturação base para geração

da AST via biblioteca ASTParser. Pode ser visto na primeira linha que ele recebe "..." em uma aplicação real, isto é substituído pela *string* contendo a leitura textual da classe lida.

Figura 11 - Exemplo de criação de AST básico a partir de string de código-fonte

```
1 char[] source = ...;
2 ASTParser parser = ASTParser.newParser(AST.JLS3);
3 parser.setSource(source);
4 // In order to parse 1.5 code, some compiler options need to be set to 1
5 Map options = JavaCore.getOptions();
6 JavaCore.setComplianceOptions(JavaCore.VERSION_1_5, options);
7 parser.setCompilerOptions(options);
8 CompilationUnit result = (CompilationUnit) parser.createAST(null);
```

Fonte: ASTPARSER (2019)

A terceira linha exibe o valor do código para ser realizado o *parser* e na última linha o *parser* feito com o código-fonte é criado a AST propriamente dito e atribuído a um o objeto do tipo "*CompilationUnit*". A definição de *CompilationUnit* serve como unidade de compilação do tipo de nó AST, ele é o tipo de raiz de uma AST onde o intervalo de origem para este tipo de nó é normalmente o arquivo de origem inteiro, incluindo espaços em branco iniciais e finais, além de comentários (COMPILATIONUNIT, 2019).

A classe *CompilationUnit* é usada para manipulação da estrutura da classe e um exemplo pode ser visto na Figura 12. Nesta figura na linha 1 é definido um nome para o pacote da classe, na linha 2 é definido um bloco de comentário ao arquivo, e por último, na linha 3 se tem um exemplo de como obter o conteúdo do arquivo para posteriores verificações que sejam necessárias ou até mesmo usado para regravar no arquivo original via classe "*FileWriter*".

Figura 12 - Exemplo do uso da classe CompilationUnit

```
1     result.setPackageDeclaration(nomePacote);
2     result.setBlockComment("Classe gerada pelo Factory Method");
3     String conteudoArquivo = result.toString();
```

Fonte: Autoria própria

2.5 TRABALHOS RELACIONADOS A REFATORAÇÃO DE SOFTWARE

Na literatura, encontra-se estudos sobre refatoração de software. O estudo de Mens (2004) prove uma revisão das tarefas de refatoração de software, artefatos

usados, formalismos, técnicas a serem aplicadas, questões essenciais a serem consideradas no desenvolvimento de ferramentas de refatoração de software.

O estudo de Wangberg (2010) obteve uma visão geral da pesquisa relacionada a cada uma das etapas do processo de refatoração: detecção de *code smells*, tomar decisões sobre quais refatorações escolher e como realizar esta refatoração. Neste estudo foi encontrado um crescimento significativo de publicações sobre o tema de *code smells* e refatorações, o que indicou um aumento da popularidade do tema dentro da comunidade de pesquisa. No entanto, a revisão encontrou uma falta de evidências sobre a escolha das melhores estratégias de refatoração para melhorar manutenibilidade e guiar os profissionais.

Pate (2013) avaliou se uma ferramenta pode ajudar os desenvolvedores a serem mais eficazes ao executar uma mudança na presença de clones de código. O estudo de Zhang (2011) identificou os estudos em relação aos *bad smells* e constatou que: 1) o nível atual de conhecimento sobre *bad smells* varia; 2) *duplicated code* tem atraído mais atenção; 3) *duplicated code* pode ser considerado mais fácil de entender; 4) alguns *bad smells* tem atraído poucos estudos tal como a técnica *Message Chains*.

Laguna (2013) delimitou seu estudo em investigar técnicas específicas de reengenharia e refatoração de código ou modelos. Constatou que alguns avanços na definição formal de refatoração foram alcançados, especialmente no contexto de modelagem de recursos e FOP (programação orientada a recursos), mas são limitados.

Vale (2014) examinou *bad smells* relacionados a linha de produção de software e com isto elaborou um catálogo de *bad smells* e métodos de refatoração. Foi possível notar que os *bad smells* no contexto Linha de Produtos de Software (SPL) é um tópico de estudo que surgiu a partir de 2007. Abebe (2014) constatou que desenvolver refatoração é um ato desejável para assegurar a qualidade do processo e do produto de software. Foi observado que os pesquisadores tem contribuído para o campo de refatoração de software, mas ainda tem uma grande quantidade de problemas não resolvidos.

Já no estudo de Rasool (2015) o objetivo foi classificar, comparar e avaliar técnicas e ferramentas usadas para detecção de *code smells*. Verificou que: 1) a maioria das técnicas/ferramentas realiza experimentos em sistemas *open-source* e/ou locais diferentes e não calculam a precisão de seus resultados por causa da definição de *code smells*; 2) técnicas diferentes renderizam resultados diferentes nos mesmos

sistemas por causa de diferentes definições; 3) há uma falta de sistema padrão de referência para avaliar as técnicas existentes.

Al (2015) identificou oportunidades para as atividades de refatoração de código. Embora a maioria dos pesquisadores sejam da área acadêmica, alguns dos participantes de estudos empíricos são da indústria o que indica que os pesquisadores procuram manter ao menos um contato com pessoas na indústria.

O estudo de Rochimah (2015) apresentou uma visão geral sobre as técnicas de refatoração de código não-fonte e analisou as vantagens e desvantagens de cada técnica. As conclusões obtidas são: 1) detectar a refatoração em nível de código-fonte pode ser feita no modelo de *design* de software, código-fonte com detecção não convencional e outros artefatos de software; 2) atividades de refatoração com métodos heurísticos são feitas com regras previamente definidas; 3) a vantagem do método heurístico é a velocidade e precisão.

Yusifog˘lu (2015) forneceu uma visão geral de abordagens para STCE (*Software Test-Code Engineering*), além de fazer a identificação das áreas que requerem maior atenção. Alves (2016) caracterizou os tipos de *technical debt*; identificou estratégias de gestão e levantou o nível de maturidade de cada proposta de *technical debt*. Singh (2017) realizou a revisão sobre desenvolvedores de software com a percepção dos *code smells*. As conclusões afirmam que a maior parte dos estudos primários são do mundo acadêmico, sendo 33,33% dos artigos provenientes de conferências, enquanto 3,1% dos estudos são livros e teses. Foram encontrados 6 tipos de abordagens de detecção de *code smells*: método tradicional, método automático, método semi-automático, estudos empíricos e método baseado em métricas.

Em relação ao estudo de Mariani (2017) o objetivo foi fornecer uma visão geral das abordagens de refatoração baseada em busca (SBR) existentes, apresentando suas características comuns e identificando tendências e oportunidades de pesquisa. Nunes (2017) coletou estudos relacionados a métricas de código-fonte, além de fornecer uma visão geral sobre o estado atual das métricas de código-fonte e suas tendências atuais. As conclusões obtidas foram que as métricas orientadas a objetos ganharam muita atenção, porém há uma necessidade atual de mais estudos sobre métricas orientadas a aspectos.

Sousa (2018) apresentou um mapeamento sistemático que investiga a relação entre *design patterns* e *bad smells*. As conclusões são que as análises

realizadas indicam que quando se tem implementação de padrão de projeto a mesma pode causar *bad smells* e consequentemente degrada a qualidade do software aumentando a sua complexidade e prejudica outros atributos externos tais como: modularidade, flexibilidade, testabilidade e outros. Porém, quando a aplicação do padrão de projeto é bem projetada, os impactos gerados são positivos.

Os resultados indicam que a relação de concorrência entre padrões de projeto e *bad smells* é um tópico atual e tem sido explorada desde 2013. O Quadro 2 mostra os estudos encontrados que abrangem revisão ou mapeamento sistemático na área de refatoração.

Quadro 2 - Trabalhos relacionados a revisão ou mapeamento sobre refatoração

Título Artigo	Autor	Ano	Qtd. Estudos	Revisão / mapeamento
A literature review on code smells and refactoring	Ruben Drøsdal Wangberg	2010	46	Revisão
Clone evolution: a systematic review	Jeremy R. Pate, Robert Tairas, Nicholas A. Kraft	2011	30	Revisão
Code Bad Smells: a review of current knowledge	Min Zhang, Tracy Hall, Nathan Baddoo	2011	39	Revisão
A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring	Miguel A. Laguna, Yania Crespo	2013	74	Mapeamento
Bad smells in software product lines: A systematic review	Gustavo Vale, Eduardo Figueiredo	2014	18	Revisão
Trends, opportunities and challenges of software refactoring: A systematic literature review	Mesfin Abebe, Cheol-Jung Yoo	2014	58	Revisão
A review of code smell mining techniques	Ghulam Rasool, Zeeshan Arshad	2015	46	Revisão
Identifying refactoring opportunities in object-oriented code: A systematic literature review	Jehad Al Dallal	2015	47	Revisão
Non-Source Code Refactoring: A Systematic Literature Review	Siti Rochimah, Siska Arifiani, Vika F. Insanittaqwa	2015	20	Revisão

Software test-code engineering: A systematic mapping	Vahid Garousi Yusifoglu, Yasaman Amannejad, Aysu Betin Can	2015	60	Mapeamento
Identification and management of technical debt: A systematic mapping study	Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, Carolyn Seaman	2016	100	Mapeamento
A systematic literature review: Refactoring for disclosing code smells in object oriented software	Satwinder Singh, Sharanpreet Kaur	2017	238	Revisão
A systematic review on search-based refactoring	Thainá Mariani, Silvia Regina Vergilio	2017	71	Mapeamento
Source code metrics: A systematic mapping study	Alberto S.Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, Carlos Soubervielle-Montalvo	2017	226	Revisão
A systematic literature mapping on the relationship between design patterns and bad smells	Bruno L. Sousa, Mariza A. S. Bigonha, Kecia A. M. Ferreira	2018	16	Mapeamento

Fonte: Autoria própria

O Quadro 3 mostra a quantidade de trabalhos relacionados sobre refatoração de 2010 a 2018. Observa-se que é fonte de estudos por pesquisadores, visto que a refatoração de software contribui para o desenvolvimento de produtos de software com mais qualidade.

Quadro 3 - Trabalhos por ano de revisão e/ou mapeamento sobre refatoração de software

Ano	Total de Estudos
2010	1
2011	2
2013	1
2014	2
2015	4
2016	1
2017	3
2018	1

Fonte: Autoria própria

Os trabalhos apresentados fornecem uma visão geral de muitos aspectos relacionados a refatoração de software, no entanto, não apresentam uma revisão sobre métodos ou abordagens de refatoração baseada em padrões de projetos, técnicas e ou refatoração usando agentes. Tal revisão permite a identificação de iniciativas bem-sucedidas, bem como lacunas na área de refatoração que podem apontar novas oportunidades e tendências de pesquisa em relação a abordagens e métodos de refatoração. Desta forma, este trabalho realizou um mapeamento sistemático de 1998 a 2020 e os resultados estão descritos no Capítulo 4.

2.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este Capítulo relatou a importância de um processo de refatoração que permite ao desenvolvedor maior assertividade em suas tarefas. Foi descrito a importância sobre refatorar código-fonte, refatoração baseada em padrões de projeto, métodos de refatoração fundamentados em padrões de projeto e abordagens de extração.

O tema de refatoração é importante porque, se corretamente aplicado ao sistema, melhora a manutenção do software, evitando gastos. Existem mais de uma maneira de verificar o ler o código-fonte, porém a AST é a que se mais se destaca em refatoração de software. A AST é empregada em bibliotecas tais como *JavaParser* ou *ASTParser* e isto foi constatado nos trabalhos de mapeamento sistemático que foram analisados.

Em Beluzzo (2018) foram encontrados 21 (vinte e um) métodos para inserção e detecção de padrões de projetos (apresentados no Quadro 1), porém nenhum deles contempla um processo autônomo de refatoração. Por isto, este trabalho realizou uma revisão sistemática sobre processos de refatoração com agentes, descritos no Capítulo 4.

3 AGENTES INTELIGENTES

De acordo com Wooldridge (2009), um agente inteligente trabalha de forma autônoma ou semiautônoma fazendo a comunicação com outros agentes, programas ou humanos. Coppin (2015) define agente, agente de software e agentes inteligentes. Um agente é uma entidade capaz de realizar alguma tarefa, geralmente para auxiliar um usuário humano. Um agente de software são programas de computador projetado para realizar tarefas em nome do ser humano. Por fim, um agente inteligente tem conhecimento adicional de dominio que os habilita a realizar as tarefas deles, mesmo quando os parametros da tarefa mudam ou surgem situações inesperadas.

Este Capítulo apresenta conceitos referentes a agentes usados para o desenvolvimento da Codice-Unio. A Seção 3.1 relata as definições, características e classificações de agentes. A Seção 3.2 descreve a arquitetura BDI *(Belief, Desire, Intention)* usada no desenvolvimento da Codice-Unio. A Seção 3.3 descreve resumidamente as ferramentas encontradas para a criação de agentes detalhando a *Jadex* (PIUNTI, 2008). A Seção 3.4 apresenta metodologias para modelagem de agentes. Por fim, a última Seção apresenta as considerações finais do Capítulo.

3.1 DEFINIÇÃO E TIPOS DE AGENTES

O agente é uma entidade com capacidade de realizar tarefas e por meio disto auxiliar o ser humano. A autonomia de um agente se entende por aquilo que é essencial em habilidades e requisitos. As propriedades relacionadas a agentes são essencialmente (COPPIN, 2015):

- Inteligência: tem um certo conhecimento que possibilita fazer suas tarefas mesmo quando os parâmetros iniciais da mesma mudam ou quando surgem situações que são consideradas inesperadas.
- Autonomia: se revela como a capacidade de o mesmo agir e tomar decisões independentes de uma intervenção do usuário ou programador. Este tipo de autonomia está vinculado as situações de inteligência artificial para tomada de decisões.

- Capacidade de aprender: quando está com novas informações ele as armazena para uso posterior com a possibilidade de melhorar seu desempenho.
- Cooperação: acontece em sistema multiagentes quando eles cooperam e conversam entre si, fazendo literalmente uma interação social entre os agentes. A cooperação é atrelada ao vínculo com humanos em que estes trabalham por meio de interface de entradas e instruções para indicar aonde o agente errou e quais passos melhorar.

As características ou propriedades dos agentes são que possuem autonomia, capacidade de aprendizagem, cooperação e que permitem classificar os principais tipos de agentes (WOOLDRIDGE, 2009). Para Wooldridge (2009), os três tipos de agentes são dados por:

- Agente inteligente: é dito como uma entidade autônoma capaz de observar o meio ao qual se encontra podendo usar sensores e atuar tomando decisões mediante os desejos que possui.
- Agente reativo: tem percepção do meio ambiente e reagem ao meio de acordo com suas características. Também é conhecido como agente reflexivo, eles reagem a eventos no ambiente em que estão, de acordo com regras anteriormente determinadas. Eles podem ser separados em: agentes baseados em objetivos, agentes baseados em utilidade e em funções de utilidade (COPPIN, 2015).
- Agente híbrido: são capazes de reagir e são proativos, criam subsistemas separados em camadas. Tem camadas na vertical e horizontal. Pode ser dito que um agente híbrido ao mesmo tempo que pensa pode reagir as noções do meio onde se encontra.

3.2 ARQUITETURA PARA DESENVOLVIMENTO

Uma arquitetura de agente descreve a organização, interação e a estrutura de como o agente trabalha. Juchem (2001), descreve as seguintes arquiteturas: Subsunção, Reativas ou não-deliberativas, Hibridas e a BDI *(Belief, Desire, Intention).*

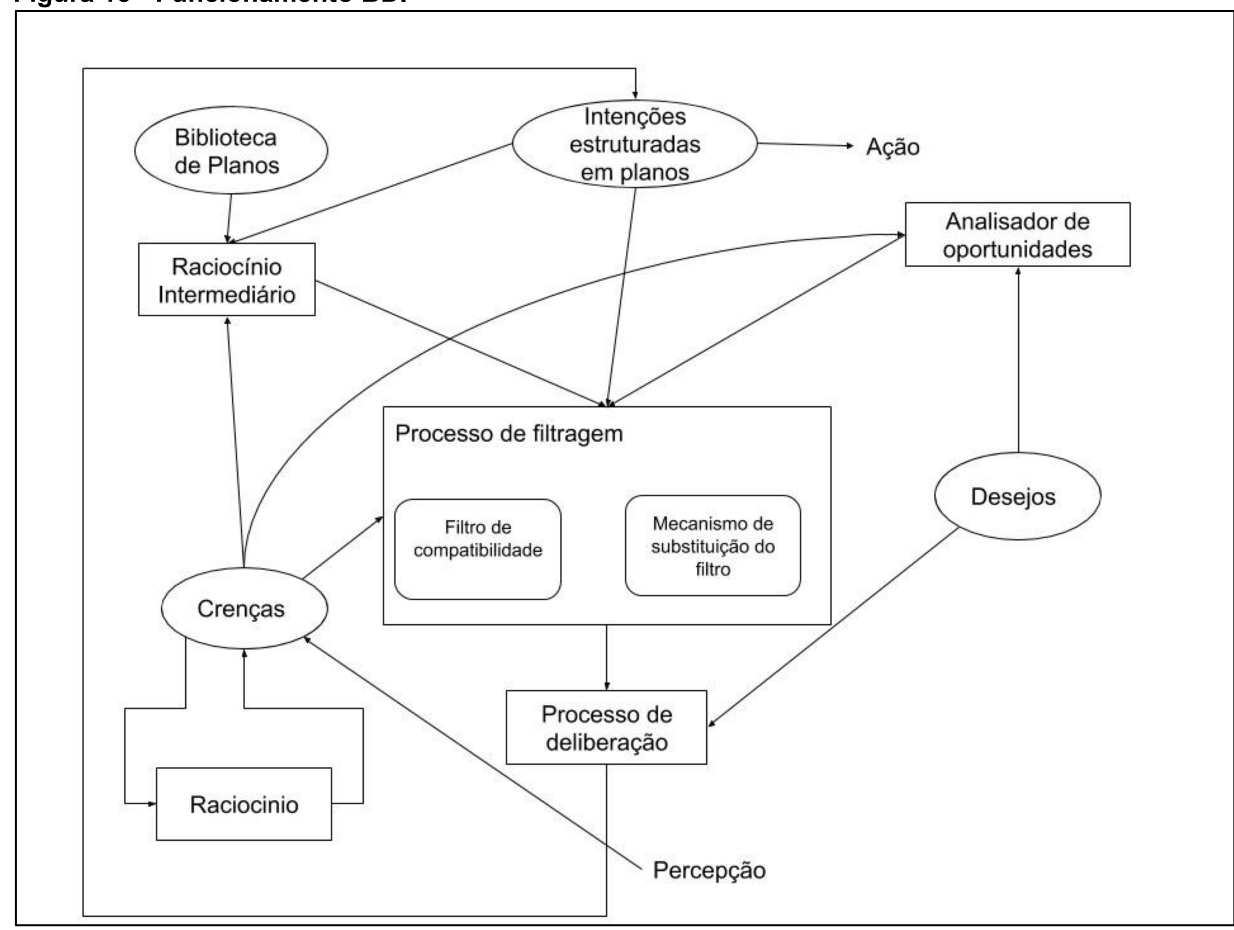
A arquitetura de Subsunção foi proposta por Brooks (1986) e é voltada para agentes reativos e implementados com agentes inteligentes. Trata-se de uma arquitetura em camadas e não envolve uma inteligência centralizada. Nesse tipo de arquitetura, tem-se um conjunto de entradas, possíveis ações e um conjunto de módulos em camadas (COPPIN, 2015).

Para Bastos (1998), a arquitetura reativa ou não-deliberativa são requeridas quando o agente tem percepções do meio e passa a tomar decisões a partir de informações obtidas por sensores. Ao contrário de arquiteturas que faz deliberação, este tipo de arquitetura não possui um modelo para simbolizar o mundo e nem planejam suas ações. A escolha de uma ação é feita de acordo com a situação do ambiente em que o agente se encontra, ou seja, executando uma determinada ação haverá uma reação correspondente sendo filtrada por uma condição (FROZZA, 1997).

Em se tratando das arquiteturas híbridas, essas agem como abordagem deliberativa e reativa ao mesmo tempo para tornar o sistema mais adequado a construção com agentes. A ideia é fazer a construção com 2 (dois) subsistemas sendo separados em deliberativo para tomada de decisões e reativo para ser capaz de reagir a determinados eventos do ambiente, mesmo sem possuir a capacidade de raciocínio (FROZZA, 1997). Para Bastos (1998), o uso de arquiteturas híbridas ajuda a tornar o agente com capacidade de reagir, raciocinar e também planejar.

A arquitetura BDI cria um agente racional no qual são descritos suas *Beliefs, Desires and Intentions* (BRATMAN, 1988). A Figura 13 ilustra como foi inicialmente projetado o funcionamento da arquitetura BDI *(Belief, Desire, Intention)*, e nele pode ser visto o fluxo de informações que é feito em ciclos e que não necessariamente tem um início. O princípio deste diagrama deve ser lido pelas intenções, ou seja, quando o agente tem intenções elas são analisadas por meio de deliberação para ver se podem ser executadas ou não.

Figura 13 - Funcionamento BDI



Fonte: Bratman (1988)

Considerando a Figura 13 as Crenças representam o conhecimento do agente relacionada com as situações do mundo a sua volta. Ela é um estado mental fundamental para as interações do agente com noção parecida a de conhecimento. Os Desejos estão relacionados ao que o agente vai realizar e decide quais metas devem ser alcançadas. A decisão das metas é realizada pelo processo de deliberação e o que é feito para alcança-las é chamado de Raciocínio Intermediário. Os desejos representam os estados desejáveis que o sistema pode apresentar ou seu estado motivacional do agente. O agente seleciona os desejos que são possíveis a partir de algum critério, incentivando a realizar as tarefas para o qual foi projetado. As intenções determinam as ações a serem realizadas. Ou seja, são as etapas a serem realizadas para executar o que foi definido nos planos. Os planos representam como o processo vai ser feito.

A arquitetura BDI permite realizar o desenvolvimento em camadas e aproxima o agente ao raciocínio humano, fazendo com que ele tenha várias características tais como: crer em algo, desejar algo e ter planos para seu futuro. Além disso, ele pensa

muito a respeito da situação e se realmente deseja e vai ter condições de concluir seus desejos. Por estes motivos foi a arquitetura escolhida para a Codice-Unio e será descrita na próxima seção.

3.2.1 Arquitetura BDI (*Belief, Desire e Intention*)

A arquitetura BDI pode ser considerada a base para criação de agentes, para tanto são considerados os atributos (RAO; GEORGEFF, 1995; WOOLDRIDGE, 2009):

- Crença (*Beliefs)*: são informações que um agente tem sobre o mundo no qual ele se encontra. As crenças são atualizadas após a percepção de cada ação.
- Desejo *(Desire*s): contém a informação sobre os objetivos a ser atingido.
- Intenção *(Intention)*: contém o atual plano de ação escolhido.

Os agentes BDI são usados em ambientes dinâmicos, onde estão recebendo os estímulos do ambiente para realizar ações (intenções), que são baseadas no seu conhecimento sobre como o mundo se encontra (crenças) e pôr fim a todo momento analisando suas opções disponíveis (desejos) para atingir seu objetivo inicial. O conjunto de intenções que o agente irá realizar são denominados de Planos (*Plans*) e fornecem um propósito concreto para o raciocínio. Eles restringem o escopo de deliberação a um determinado número de opções. Os planos são necessários para conquistar um objetivo (BRATMAN,1988).

De acordo com Fagundes (2007), BDI é composto por: i) estados mentais que de acordo com a psicologia pode ser explicado como crenças desejos e intenções, em que a ideia principal é que agentes com cognição têm estados relacionados ao estado do ambiente; ii) raciocínio, pois o processo de deliberação ocorre verificando quais são os possíveis desejos a se aplicar. Braubach (2003) comenta que ao usar o modelo de arquitetura BDI se pode visualizar um agente como uma entidade direcionada por objetivo que age de maneira racional.

Bordini *et al.* (2007) afirma que objetivos e desejos estão relacionados, e usa objetivos para se referenciar a um subconjunto consistente de desejos que o agente venha a ter. Segundo Bordini *et al.* (2005), o BDI é a arquitetura mais conhecida e usada por agentes cognitivos. Para Neto (2010) o uso da BDI pode ser realizado por

meio de *frameworks* voltados a linguagem de programação Java que são: *AgentBuilder*, *Jack*, *Jadex*, *JAM* e *Jason*, descritas na próxima Seção.

3.3 FERRAMENTAS PARA CRIAÇÃO DE AGENTES

Dentre as ferramentas usadas para criação de agentes se tem: *Jadex, Jade* e *Jason.* O *framework* Jade é uma estrutura de *software* que visa tornar mais fácil o desenvolvimento de aplicativos com agentes de acordo com as especificações de FIPA (*Foundation for Intelligent Physical Agents*), em que o objetivo é garantir a conformidade padronizada por meio de serviços e agentes oferecidos pelo sistema (BELLIFEMINE, 1999). As características do *framework Jade* são dadas (JADE, 2019)*:*

- Implementado em Java.
- Serve para simplificar a implementação de sistemas multiagentes por meio de um middleware atendendo as especificações do FIPA.
- Um sistema construído com JADE pode ser distribuído entre máquinas e a configuração é controlada por meio de uma GUI remota.
- Arquitetura baseada em comportamentos. Silva (2003) afirma que os comportamentos definem a maneira que os agentes agem ou reagem mediante tarefas definidas no ambiente. Para Smart (2014), uma arquitetura comportamental faz com que o sistema produzido tenha comportamentos similares a humanos que pode ser programável e adaptável.
- A configuração é possível de ser alterada mesmo em tempo de execução movendo agentes de uma máquina para outra, quando necessário.

Para a plataforma *Jason* a linguagem usada é uma extensão do *AgentSpeak*. De acordo com Bordini (2005), um *AgentSpeak* é definido como sendo um conjunto de crenças que servem para fornecer o estado inicial da base de crenças em que este é um conjunto de fórmulas básicas e um conjunto de planos que formam a sua biblioteca de planos.

A linguagem usada no interpretador Jason é baseada na arquitetura BDI e sendo assim um dos componentes da arquitetura de agentes para quem o usa é a base de crenças e o interprete faz constantemente a verificação do ambiente e consequentemente atualização das suas crenças.

Esta plataforma foi desenvolvida por Jomi F. Hübner e Rafael H. Bordini e suas características são (JASON, 2019):

- Open source.
- Distribuído sob licença GNU LGPL.
- Plataforma para desenvolvimento de sistema multi-agentes (MAS).
- Usa agentes BDI.
- Tem *plugin* para integração ao Eclipse.
- A linguagem interpretada pelo Jason é uma extensão da linguagem chamada *AgentSpeak (L)* criada por Anand Rao.
- Funções para seleção, confiança, e arquitetura em geral do agente (percepção, revisão das crenças, comunicação entre agentes e atuação) são personalizáveis em Java.

Dentre os frameworks citados, este trabalho utilizou o Jadex para implementação da abordagem Codice-Unio, pois permite a inicialização do agente por meio da linguagem Java, a separação das camadas da arquitetura BDI é feita com classes Java usando @*anotations*, não usa XML e não necessita de instalação de softwares adicionais para a inicialização do agente.

3.3.1 Framework Jadex

As características do *framework Jadex* são (PIUNTI, 2008):

- Open source.
- Usa a arquitetura BDI (*Belief Desire Intention*).
- Usa estados mentais.
- Melhora a interação agente/artefato em um nível cognitivo alto.

De acordo com Braubach (2003), o *Jadex* é uma extensão do Jade para fazer arquiteturas de agentes com a representação em estados mentais seguindo o modelo

BDI. Braubach (2004), comenta que este *framework* usa uma abordagem declarativa para definição dos componentes do agente, os corpos do plano devem ser implementados como classes Java onde estendem uma determinada classe de estrutura e com isto fornece acesso genérico as facilidades do BDI.

A configuração e implementação do agente em *Jadex* pode ser feito via *xml* ou por meio de implementações diretamente em linguagem de programação *Java,* conforme visto na Figura 14. Apresenta também como iniciar o agente implementado com *jadex* diretamente via código *java*, o que permite ao agente iniciar na máquina sem iteração do usuário diretamente. Na sintaxe desta figura se faz a configuração da plataforma de agente e sua instanciação sem precisar da criação de uma interface.

Figura 14 - Iniciar agente *Jadex* por meio da linguagem *Java*

```
public class Main {
    public static void main(String[] args) {
        PlatformConfiguration   config  = PlatformConfiguration.getDefaultNoGui();

        config.addComponent("a1.TranslationBDI.class");
        Starter.createPlatform(config).get();
    }
}
```

Fonte: ACTIVE COMPONENTS (2019)

O uso do *Jadex* com BDI é realizado de acordo com a estruturação interna apresentada no Quadro 4.

Quadro 4 - Implementação do BDI no framework *Jadex*

Camadas do BDI	Implementação no Jadex
Crença	*Belief*, são estruturados como atributos da classe agente.
Desejo	*Goal*, é implementado como classes e está sempre interligada a um conjunto de planos (*Plan*).
Intenção	São representadas por Planos (*Plan*). Os planos podem ser implementados como métodos da linguagem Java ou ainda ser classes com várias funcionalidades.

Fonte: Autoria própria

De acordo com ACTIVE COMPONENTS (2019), o Jadex pode ter a configuração do agente diretamente no código-fonte por meio de anotações e inicia o agente sem depender da plataforma de configuração. A Figura 15 mostra como é a implementação de planos via anotações em *Java* usando um método interno na classe. Nesta figura tem-se as seguintes anotações:

- @Plan: faz a declaração do plano, ou seja, na figura exemplo o método chamado “translate” é o plano do agente.
- @Trigger: acionador automático que vem ao final da execução do plano, e na figura está codificado que vai executar a classe “Translate.class”.

Figura 15 - Declaração de planos via anotações - *Jadex*

```
@Plan(trigger=@Trigger(goals=Translate.class))
protected void translate(String eword)
{
    System.out.println("Translated: "+eword+" "+wordtable.get(eword));
}
```

Fonte: ACTIVE COMPONENTS (2019)

Assim como se tem implementação de planos como métodos de uma classe, tem-se também a criação de planos como sendo classes. Um exemplo de um plano como sendo classe é mostrado na Figura 16, onde tem-se o primeiro ponto de importação que é o “@Plan” que informa ao agente instanciado que aquela classe é um dos seus planos, em sequência as 3 (três) anotações essenciais são:

- @PlanBody: é o corpo do plano em si, o desenvolvedor é livre para fazer outros métodos dentro da classe, porém é este método que será executado quando o plano for chamado.
- @PlanPassed: quer dizer que o plano foi executado e terminado com sucesso em toda sua tarefa.
- @PlanAborted: serve para personalizar mensagens de problemas ao agente, este método é executado quando o plano não consegue executar completamente o método com anotação “@PlanBody”.

Figura 16 - Criação de um plano com uma classe *Java*

```
@PlanBody
public void translateEnglishGerman()
{
  System.out.println("Plan started.");
  plan.waitFor(10000).get();
  System.out.println("Plan resumed.");

  System.out.println("Translated: dog - " + wordtable.get("dog"));
}
```

Fonte: ACTIVE COMPONENTS (2019)

A Figura 17 exibe como é feito a implementação de um objetivo (*Goal*) diretamente em *Java*, eles são considerados os desejos na arquitetura BDI. Nesta figura os pontos importantes são separados em:

- @Goal: é feito para dizer que aquela classe é um objetivo ao agente.
- @GoalParameter: diz que ao chamar este objetivo, este pode ser passado por parâmetro para a classe usando o atributo “eword”.
- @GoalResult: informa ao agente que ao final ele retorna o valor para quem o chamou. Neste caso, retornar o valor dentro da variável “gword”.

Figura 17 - Objetivos via anotações *Jadex*

```
@Goal
public class Translate
{
  @GoalParameter
  protected String eword;

  @GoalResult
  protected String gword;

  public Translate(String eword)
  {
    this.eword = eword;
  }
}
```

Fonte: ACTIVE COMPONENTS (2019)

A Figura 18 apresenta um exemplo de implementação básica de uma classe voltada a agente, contendo as seguintes anotações:

- @Agent: declara que a classe em específica é um agente.

- @Description: declara uma descrição geral sobre o agente e é considerada opcional.

Figura 18 - Declaração de classe de agente em *Jadex*

```
package a1;

import jadex.micro.annotation.Agent;
import jadex.micro.annotation.Description;

@Agent
@Description("The translation agent A1. <br> Empty agent that can be loaded and started.")
public class TranslationBDI
{
}
```

Fonte: ACTIVE COMPONENTS (2019)

A Figura 19 apresenta outra classe agente e é exibido a declaração de anotação para crença "@Belief". A instrução "(dynamic =true)" serve para informar ao agente que deve monitorar a esta crença porque ela vai mudar seu conhecimento interno em algum momento da execução.

Figura 19 - Exemplo de classe Agente com Crenças

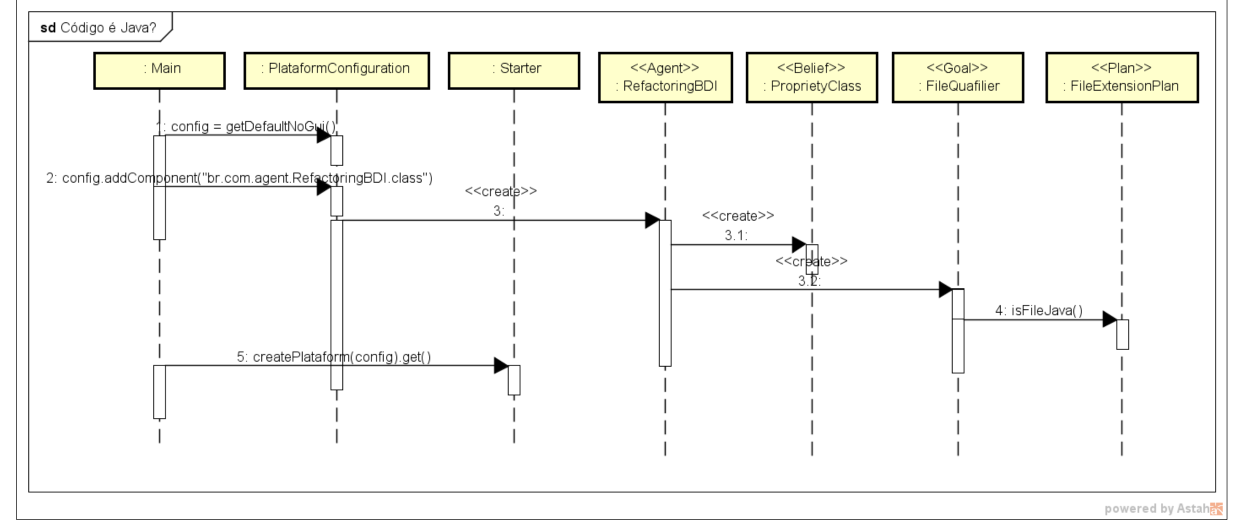
```
@Belief
protected Map<String, String> wordtable = new HashMap<String, String>();

@Belief(dynamic=true)
protected boolean alarm = wordtable.containsKey("bugger");
```

Fonte: ACTIVE COMPONENTS (2019)

A Figura 20 apresenta um exemplo de como um agente em Jadex funciona em termos chamadas de eventos. O exemplo utilizado é de um agente tradutor que realiza a tradução de uma palavra. Para isto, ele tem um plano interno composto por um conjunto de palavras que são pesquisadas de acordo com o idioma e depois oferece como saída a impressão da palavra traduzida na língua desejada. Neste exemplo, existem um objeto tipo *:Main* que realiza as configurações fazendo uma chamada para o *:PlataformConfiguration* o qual inicializa o agente *:TranslationBDI*.

Figura 20 - Exemplo de um agente em Jadex (Agente Tradutor)



Fonte: Autoria própria

O *:TranslationBDI* verifica suas crenças *:WordTable* (contém uma tabela de palavras traduzidas) e chama o objetivo *:Translate*, o qual requisita ao plano *:TranslationPlan* para realizar a tradução efetivamente e o resultado da tradução é mostrado no terminal de console.

3.4 METODOLOGIA PARA DESENVOLVER AGENTES

Existem várias metodologias para a implementação de agentes dentre as quais pode-se citar: Tropos, MaSE, ADELFE, Gaia e Prometheus (AMBIEL, 2010). A metodologia Tropos é feita para ter 4 (quatro) fases de desenvolvimento sendo separado em: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural, e Projeto Detalhado (SILVA, 2005). Esta metodologia também tem uma estrutura para testar o agente chamado eCAT e suporta derivação de casos de teste de forma semiautomática (AMBIEL, 2010).

Na metodologia MaSE (*Multi-agent System Engineering*) é independente de arquitetura, linguagem de programação ou qualquer estrutura de comunicação do agente e trata os agentes como um paradigma de orientação a objetos mais profundo, em que os mesmos são especializações de objetos (GAGO, 2009).

O ADELFE é liderado pelo RUP (*Rational Unified Process)*, porém é dedicado a engenharia de software para criação de MAS adaptável. Esta metodologia garante que o software seja desenvolvido de acordo com a teoria do AMAS1 e foca nos quatro primeiros fluxos do RUP (PICARD, 2004).

A Gaia permite que o usuário use sistematicamente uma declaração de requisitos para um *design* detalhado para que possa ser implementado diretamente. Na aplicação desta metodologia o analista usuário passa de conceitos abstratos para conceitos mais concretos, sendo que a análise e o *design* podem ser pensados como um processo de desenvolvimento de modelos cada vez mais detalhados do sistema que estiver sendo construído (WOOLDRIDGE *et al*., 2000).

A seguir será descrito a metodologia Prometheus que foi usada na estruturação da Codice-Unio. Esta metodologia foi escolhida por permitir criar modelos de agentes com menos fases, mas que contempla a identificação das funcionalidades e arquitetura do agente, bem como o funcionamento interno do agente.

3.4.1 Metodologia Prometheus

É uma metodologia composta por 3 (três) fases: i) especificação do sistema identificando as funções básicas, juntamente com entradas (percepções), saídas (as possíveis ações) e quaisquer outras fontes de dados compartilhadas; ii) *design* de arquitetura que usa as saídas da primeira fase para determinar quais agentes são parte do sistema e como interagem; iii) fase do projeto examina os componentes internos de cada agente e como ele executará as tarefas no sistema geral (PADGHAM *et al.,* 2002).

Existem algumas ferramentas que foram encontradas para estruturar diagramas com modelagem Prometheus: o JDE (*Jack Development Environment*), PDT (*Prometheus Design Tool*), e TDF (*Tatics Development Framework*). A base para entendimento das entidades do framework TDF foi retirada de Evertsz (2015) e pode ser visto no Quadro 5, exibindo os elementos que foram usados no modelo do agente da abordagem proposta neste trabalho.

Quadro 5 - Entidades para modelagem TDF

Elemento	Descrição
Agent	O agente
External Entity	Entidades que podem ser *plugins*, outros sistemas
Action	Representação de uma ação que o agente pode fazer
Activity	Atividades envolvendo o que o sistema de agente pode fazer
Percept	É uma percepção de entrada proveniente do ambiente onde o agente estaria
Goal	Objetivo que retrata onde o agente quer chegar
Role	Papeis (funções) que o sistema tem
Belief Set	Usado para guardar as crenças do agente, normalmente dados ou normas do ambiente
Message	Define uma mensagem (comunicação) que pode estar ocorrendo entre duas entidades
Plan	Representação de um plano que o agente pode fazer
Protocol	Um protocolo tem uma especificação de sequencias das interações do agente permitidas dentro de uma conversa.
Case Study	Estudo de caso, representa uma sequência linear dos eventos que ocorreram em uma situação histórica ou poderiam ocorrer em uma situação hipotética.
Generic	Genérico - elemento de uso geral para representar nós não categorizados no mapa conceitual
Note	Bloco de notas para o diagrama
Tactic	Tática - cumpre essencialmente 2 funções: i) como um recurso, permitindo comportamento a ser encapsulado de uma maneira que permita sua reutilização para criar soluções para problemas semelhantes; ii) agrupa a funcionalidade baseada em objetivos para uma tática, de modo que a tática pode ser usada para explicar o que o agente está fazendo em determinado momento.

Fonte: Evertsz (2019)

A TDF foi escolhida porque fornece suporte a engenharia orientada a agentes e tem uma extensão do PDT com padrões de *design* táticos, uma representação processual de diagramas de alto nível, uma definição de missão e estruturas de objetivos mais ricas (EVERTSZ, 2014).

3.5 CONSIDERAÇÕES FINAIS DO CAPITULO

Este Capítulo apresentou o conceito de agente inteligente o qual possui inteligência, autonomia, capacidade de aprender e pode cooperar com outros agentes. Um agente pode ser modelado por meio de metodologias específicas, tal como a Prometheus que permite modelar o agente entendendo seu funcionamento interno e seus componentes por meio de três etapas.

Um agente após ser modelado pode ser implementado em um framework específico tal como o *Jadex* que suporta a arquitetura BDI em que se pode utilizar crença, desejos e um conjunto de planos. A BDI permite realizar o desenvolvimento em camadas e aproxima o agente próximo ao raciocínio humano.

Para compreender como agentes são aplicados no processo de refatoração de software foi realizado um mapeamento sistemático e é apresentado no próximo capítulo.

4 ESTADO DA ARTE

Com a intenção de separar e entender o que existe na literatura sobre a área de refatoração com agentes, a prática mais comum é a realização de revisões ou mapeamentos sistemáticos. Este trabalho realizou um mapeamento sistemático focado no uso de agente para o processo de refatoração de software.

O mapeamento proposto procurou por estudos de 1998 até 2020, usando 7 (sete) repositórios digitais e 7 (sete) questões para ajudar a identificação de trabalhos sobre o tema de inserção e detecção de padrões de projeto em código-fonte com agentes.

Este Capítulo está organizado em quatro seções. A Seção 4.1 apresenta uma descrição do método de pesquisa usado para realização do mapeamento sistemático. A Seção 4.2 exibe os resultados encontrados a partir da execução do mapeamento. A Seção 4.3 descreve os trabalhos relacionados ao tema desta pesquisa e como foram analisados. Por fim, a última Seção apresenta as considerações finais do Capítulo.

4.1 METODOLOGIA DE PESQUISA

Dentre os métodos encontrados na literatura para realizar o mapeamento sistemático, destaca-se o desenvolvido por Kitchenham e Charters (2007) e a sua relevância é constatada pela quantidade de citações a qual se aproxima a 3.283 (três mil duzentos e oitenta e três). Este método é usado por autores em suas revisões tais como Dyba e Dingsor (2008), Garousi e Mantyla (2016), Martins e Gorshek (2016) e vários outros. O artigo de Martins e Gorshek (2016) aponta que o trabalho de Kitchenham e Charters (2007) é uma abordagem que contempla passos diversificados para realização de uma pesquisa mais abrangente.

O método para o mapeamento sistemático usado neste trabalho foi o de Kitchenham e Charters (2007) em que os critérios de inclusão e exclusão foram voltados ao tema deste trabalho que foi a busca de estudos que abordassem refatoração de software usando agentes. O mapeamento foi feito pelo autor deste trabalho junto com duas professoras da área de Engenharia de Software. Ressalta-se

que um mapeamento sistemático sobre o assunto de refatoração de software para inserção e detecção de padrões de projeto já foi desenvolvido por Belluzzo (2018), porém este estudo não abordou refatorações com agentes, foco desta revisão.

O período de busca foi de 1998 a 2020. O ano 1998 foi considerado como data de início em que o assunto começou a ser mais abordado por pesquisadores contemplando abordagens que contemplam agentes no processo de refatoração.

Como previsto pelo método de mapeamento foi definido um protocolo pelos autores para trazer ao pesquisador informações sobre: i) autor e ano do trabalho, ii) relevância e sua real contribuição para comunidade, iii) arquitetura do agente para refatoração de software; iv) realização dos experimentos e v) identificação se o processo de refatoração é em nível de *code smells* ou inserção e detecção de padrões de projeto. Ainda foi definida as questões de pesquisa, bases de busca, palavras-chave e filtros que são apresentados nas próximas seções.

4.1.1 Questões de Pesquisa

O objetivo central da pesquisa foi realizar levantamento das principais ferramentas e métodos usados em refatoração que detectam e aplicam padrões de projeto em código-fonte e que usam a concepção de agentes. As seguintes questões foram elaboradas:

- Q1. Quais foram os pesquisadores que desenvolveram refatoração com agentes? Qual a quantidade de trabalhos de cada autor? Qual o número de citações de cada trabalho?
- Q2. Como é definido a arquitetura do método ou abordagem que utilizam agentes?
- Q3. Quais adaptações foram feitas ao modelo arquitetural para contemplar refatorações?
- Q4. Quais são as ferramentas desenvolvidas pelos pesquisadores encontrados em Q3?
- Q5. De qual forma é o código-fonte analisado?
- Q6. Existe uma relação entre os trabalhos dos pesquisadores do assunto? Por exemplo, Autor 1 usa o método do Autor 2?

- Q7. Como foram realizados os experimentos para validar o método de refatoração fundamentado em agentes?

4.1.2 Bases e Quantidade de Retorno

A extração dos estudos foi executada em 7 (sete) bibliotecas digitais e também na web (*Google Scholar*). As pesquisas foram feitas por frase exata, buscando em título, *Abstract* e palavras-chaves do texto. O Quadro 6 apresenta as bases e as quantidade de estudos que foram encontrados em relação ao objetivo do mapeamento sistemático.

Quadro 6 - Bases e quantidade de estudos

Base	Quantidade de resultados
Springer	5
ACM	1
Science Direct	1
Wiley	0
IEEEXplore	4
Google Scholar	30
Scopus	10

Fonte: Autoria própria

Observa-se que a maior quantidade de estudos foi encontrada no *Google Scholar*, porém nem todos os 30 (trinta) foram usados por não estarem dentro do escopo estabelecido pelo mapeamento sistemático. Dos 30 estudos selecionados, somente 16 (dezesseis) estavam de acordo com o objetivo proposto pelo mapeamento.

4.1.3 *Strings* de Busca

Vários termos de busca já haviam sido pensados e discutidos entre os pesquisadores envolvidos no processo de mapeamento sistemático, o que facilitou o processo de composição das *strings*.

Foram realizadas 7 (sete) iterações para identificar a melhor *string* de busca, conforme apresenta a Tabela 1. Houve a necessidade de realizar as iterações para

refinar as strings de busca, visto que na primeira iteração foi obtido mais de 52.000 (cinquenta e dois mil) estudos para análise. As *strings* de busca foram usadas somente em inglês por ser a língua adotada nos trabalhos publicados.

Tabela 1 - Iterações para escolha da String de busca ideal

Iter.	*String* de busca	Base						
		Springer	***ACM***	***Science Direct***	***Wiley***	***IEEE Xplore***	***Scopus***	***Google Scholar***
1	*"Agent" OR "BDI" OR "Designer Patterns" OR "Refactoring" OR "Refactore"*	52.070	55.343	358.521	814.512	55.814	311.269	34.190.000
2	*"Agent" OR "BDI"*	34.831	76.765	358.522	813.690	54.425	309.835	4.570.000
3	*"Refactoring"*	18	1.084	19	925	1.401	1.438	70.200
4	*"BDI"*	12	342	99	37	158	873	265.000
5	*"Refactoring Agent"*	5	1	1	0	4	4	32
6	*"Refactoring" AND "Agent"*	12	6	9.210	25	103	77	968
7	*"Refactoring Agent" OR "agent Refactoring Method"*	5	1	1	0	4	10	30

Fonte: Autoria própria

As iterações de 1ª (primeira) a 4ª (quarta) buscaram os estudos por tema usando como filtro de busca o título. A partir da 5ª (quinta) iteração as *strings* de busca utilizaram o relacionamento entre os assuntos sobre refatoração e agentes e a consulta foi realizada por título, resumo e palavras-chaves. A melhor combinação foi obtida pela *string* "*Refactoring* Agent" OR "agent *Refactoring Method*", representada pela 7ª (sétima) iteração.

4.1.4 Seleção dos Estudos

Após a busca dos estudos nas bases encontradas na 7ª (sétima) iteração, foi estabelecida a filtragem por meio dos critérios de inclusão e exclusão. Os critérios de

exclusão são exibidos na Figura 21 e foram executados em primeiro momento para eliminar situações tais como: não estar na língua definida e não disponível para leitura. Caso o estudo não estivesse disponível para leitura, foi entrado em contato com os autores para verificar a disponibilidade de envio do trabalho.

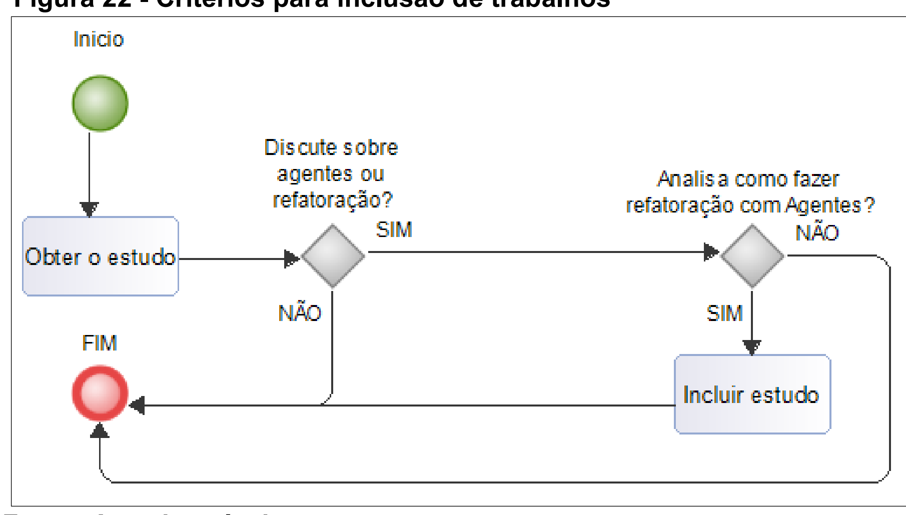
Figura 21 - Critérios de exclusão para seleção

Fonte: Autoria própria

Durante o mapeamento sistemático houve 2 (dois) casos de trabalhos não estavam disponíveis, então foi enviando um e-mail para os autores para obtê-los. Um dos autores do trabalho *Context-Aware E-Learning Infrastructure (*STOYANOV, 2016) respondeu e o do estudo *Computer and Information Sciences II* não foi obtido resposta (GELENBE, 2012).

A Figura 22 apresenta os critérios de inclusão para a seleção dos estudos. Nesta figura é visto que se o estudo é sobre agente ou refatoração, o mesmo foi analisado para verificar se contempla a refatoração com agentes. Se sim, o mesmo foi incluído para leitura completa, do contrário, foi eliminado. A análise foi realizada por meio da leitura do título, resumo e palavras-chaves.

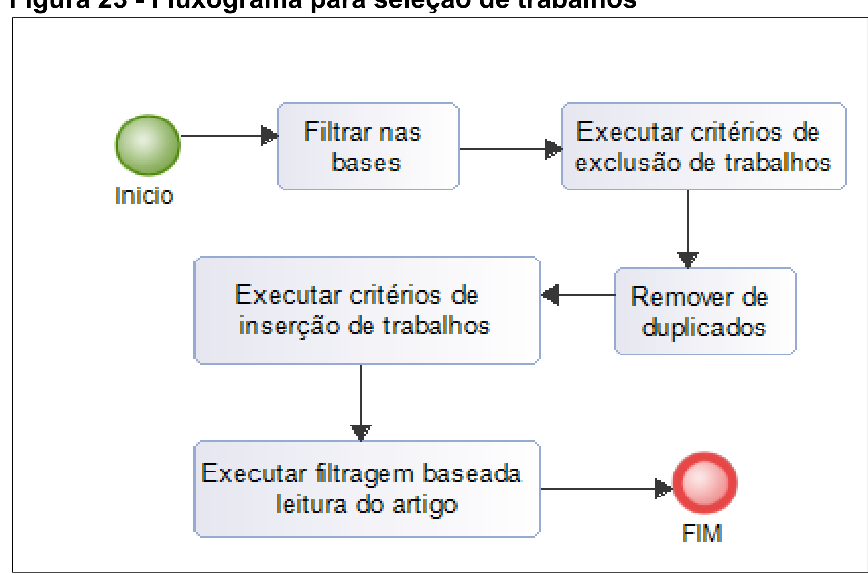
Figura 22 - Critérios para inclusão de trabalhos



Fonte: Autoria própria

A seleção dos estudos foi dividida em 4 (quatro) fases apresentadas na Figura 23: i) buscar nas bases, ii) remover os duplicados, iii) filtrar por título, resumo e palavras-chave e iv) filtrar por finalidade do mapeamento sistemático.

Figura 23 - Fluxograma para seleção de trabalhos



Fonte: Autoria própria

A primeira fase de Filtrar nas Bases retornou 51 (cinquenta e um) estudos, sendo que pelo critério do idioma foram removidos 2 (dois) porque estavam em Búlgaro sendo: *Среда за обучение по софтуерни технологии* e *Среда за доставка на електронни образователни услуги* os quais continham a *string* "*Refactoring* Agent" em seu conteúdo. Dos 49 (quarenta e nove) restantes, 3 (três) deles relacionavam-se a registros de patentes que continham informações de forma superficial sobre agentes e refatoração e por isto foram eliminados sendo: *Systems*

and Methods for image engagement analysis e *Systems and Methods for dynamic image amplification*, o último era o mesmo artigo.

Na segunda fase haviam 46 (quarenta e seis) artigos no qual 15 (quinze) foram eliminados por serem duplicados. Os artigos duplicados estão listados no Quadro 7. Portanto, ficaram 31 (trinta e um) estudos para leitura.

Quadro 7 - Estudos duplicados

Titulo	Duplicado em	Quantidade de duplicações
A Lightweight Approach for Detection of Code smells	*Springer e Scopus*	1
Agent based tool for topologically sorting badsmells and Refactoring by analyzing complexities in source code	*Google Scholar e IEEEExplore*	1
An approach to Modeling and supporting the rework process in Refactoring	*Google Scholar e IEEEExplore*	1
Applying Refactoring Techniques to UML/OCL Models	*Google Scholar e Springer*	1
AutoRefactoring: A platform to build Refactoring agentes	*Google Scholar, Science Direct, Scopus*	2
Development of a Refactoring Learning Environment	*Duplicado no próprio Google Scholar e Scopus*	2
Refactoring in Multi Agent System Development	*Google Scholar e Springer*	1
Refactoring object constraint language specifications	*Google Scholar e Springer*	1
ReLE-A Refactoring Supporting Tool	*Google Scholar e Scopus*	1
Research on self healing technology of smart distribution network based on multi Agent system	*Google Scholar e IEEEExplore*	1
Using Jason to Develop Refactoring Agents	*Google Scholar, ACM, IEEEExplore, Scopus*	3

Fonte: Autoria própria

O estudo "*Computer and Information Sciences II"* retornado pela pesquisa do *Google Scholar* foi removido porque havia conteúdo sobre Agente, mas não eram voltados a refatoração de Agentes. Portanto, com esta remoção ficaram 30 (trinta) trabalhos selecionados.

Na terceira etapa, executar critérios de inclusão, dos 30 estudos foi analisado se os artigos estavam focados em agentes para refatoração de código-fonte usando *code smells* ou *Patterns*. Dos 30 (trinta), somente 14 (quatorze) estudos estavam relacionados aos critérios estabelecidos. Um dos artigos selecionados "*Context-*

Aware and Adaptable eLearning Systems" era um Capítulo de livro e com custo para *download*. Neste caso, o artigo foi obtido por meio de contato com o autor pelo site "researchgate".

4.2 RESULTADOS

Esta Seção exibe as respostas as questões de pesquisa elaboradas na Seção anterior. Ao final desta Seção é apresentado também as possíveis lições aprendidas com o mapeamento sistemático.

4.2.1 Respostas as Questões

Os resultados da aplicação do mapeamento sistemático são apresentados com base em cada uma das perguntas que foram levantadas no início do processo. Os estudos selecionados tiveram seu conteúdo extraído por meio de um formulário de extração padronizado com atributos que respondessem as questões de pesquisa. Estes formulários contendo as extrações não estão presentes neste documento, isto porque os seus respectivos dados estão distribuídos em cada resposta para as perguntas levantadas.

4.2.1.1 Q1. Quais foram os pesquisadores que desenvolveram refatoração com agentes? Qual a quantidade de trabalhos de cada autor? Qual o número de citações de cada trabalho?

Esta pergunta busca avaliar quais são os principais autores na área de refatoração com agentes, bem como identificar a quantidade de trabalhos que já publicou, se seu trabalho é citado por outros estudos e qual a evolução de estudos de 1998 até 2020. O Quadro 8 apresenta as citações por estudos extraídos, cada estudo recebeu um código de identificação atribuído por ordem crescente de ano.

Quadro 8 - Quantidade de citações por trabalho pesquisado

Código	Titulo	Ano	N° Citações
S1	*Interaction and adaptation to the specificity of the subject domains in the system for e-Learning and distance training DeLC*	2008	7
S2	*Development of a Refactoring Learning Environment*	2011	10
S3	*ReLE (Refactoring eLearning Environment) - a Refactoring supporting tool*	2011	7
S4	*Education Cluster Supporting eTesting and eLearning in Software Engineering*	2012	2
S5	*An Agile Method for E-Service Composition*	2012	1
S6	*An Approach to Modeling and Supporting the Rework Process in Refactoring*	2012	10
S7	*Context-Aware and Adaptable eLearning Systems*	2012	21
S8	*Some Approaches for the Realization of an Adaptive Interactive System for Distance Learning*	2012	3
S9	*Using Jason to Develop Refactoring Agents*	2013	1
S10	*Agent Based Tool for Topologically sorting Badsmells and Refactoring by Analyzing Complexities in Source Code*	2013	2
S11	*Detecting and Scheduling Badsmells using Java Agent Development (JADE)*	2013	1
S12	*AutoRefactoring: A platform to build Refactoring agents*	2015	6
S13	*Software rejuvenation via a multi-agent approach*	2015	2
S14	*Context-Aware E-Learning Infrastructure*	2016	1

Fonte: Autoria própria

O trabalho com mais citação foi de S7 e constatou-se que este é usado por trabalhos mais atuais como: *A Model for Generation of Test Questions* (2017), *Ambient-Oriented Modeling of Intelligent Context-Aware Systems (2018)* e *Implementing an internet of things eLearning ecosystem* (2018).

O Quadro 9 apresenta a quantidade de trabalhos realizados por cada autor encontrado na extração de artigos. Os autores de um mesmo artigo foram separados para contabilizar a quantidade de estudos que eles participaram. Pode ser visto no quadro que a quantidade maior de trabalhos realizados por um autor é 6 (seis). Observou-se que estes autores continuaram sua pesquisa em artigos próprios ou indiretamente como coautores de outros pesquisadores.

Quadro 9 - Autores X Quantidade de trabalhos

Autor	Quantidade	Primeiro autor	Coautor
Asya Stoyanova-Doycheva	6	S9	S1, S2, S3, S4, S14
Stanimir N. Stoyanov	5	S3, S4, S7, S14	S2
Emil Doychev	3		S4, S8, S14
Viviane Torres da Silva	3		S9, S12, S13
Baldoino Fonseca dos Santos Neto	2	S9, S12	
Carlos José Pereira de Lucena	2		S9, S12
Ivan Popchev	2		S2, S3
Márcio Ribeiro	2		S9, S12
Mincho Sandalski	2	S2	S3
Ayshwarya Lakshmi	2	S10, S11	
S.Shanmuga Vadivu	2		S10, S11
Todorka Glushkova	2	S1	S8
Veselina Valkanova	2		S4, S14
Vladimir Valkanov	2		S4, S14
A.Ramachandran	1		S11
Christiano Braga	1		S14
Sahaaya Arul Mary	1		S11
Evandro de Barros Costa	1		S14
Heliomar Santos	1	S13	
Hussein Zedan	1		S14
João Felipe Pimentel	1		S13
Leon Joel Osterweil	1		S6
Leonardo Murta	1		S13
Luigi Benedicenti	1		S13
Pouya Fatehi	1	S5	
Seyyed Mohsen Hashemi	1		S5
Vanya Ivanova	1		S8

Fonte: Autoria própria

De acordo com o Quadro 9, 2 (dois) autores Asya Stoyanova-Doycheva e Stanimir N. Stoyanov se destacam em relação a quantidade de trabalhos. Asya foi o autor principal em 1 (um) trabalho e coautor em 5 (cinco) outros. Stanimir foi autor principal em 4 (quatro) trabalhos e 1 (um) com coautoria. Dos 28 autores, aproximadamente 46% eram autores ou coautores de somente 1 (um) trabalho, sendo 2 (três) como autores principais e 11 (onze) como coautores. Cerca de 34% dos

autores publicaram 2 trabalhos, sendo 4 como autores principais e 6 como coautores. Aproximadamente 6% dos autores publicaram 3 (três) trabalhos em que nenhum foi autor principal.

O Quadro 10 apresenta a quantidade de estudos que foram publicados por ano. O começo das publicações foi em 2008 onde teve-se 1 (um) estudo neste ano, seguindo-se no ano de 2011 com 2 (dois) estudos. Já o ano de 2012 foi o que mais teve publicação sobre refatoração com agentes totalizando 5 (cinco) estudos. Depois do ano de 2012 houve um decréscimo das publicações, no ano de 2013 teve 3 (três) publicações e no ano de 2015 foram publicados 2 (dois) estudos envolvendo o tema deste mapeamento sistemático. Em 2016, somente 1 (um) trabalho foi publicado e de 2017 até 2020 não houve publicação abrangendo refatoração e agentes.

Quadro 10 - Quantidade de trabalhos por ano

Código	**Ano**	**Quantidade de trabalhos**
S1	2008	1
S2, S3	2011	2
S4, S5, S6, S7, S8	2012	5
S9, S10, S11	2013	3
S12, S13	2015	2
S14	2016	1

Fonte: Autoria própria

Ao se buscar trabalhos de mapeamento sobre refatoração com padrões de projeto, encontrou-se o publicado por Beluzzo (2018) em sua dissertação quando relatou o estado da arte sobre refatoração baseada na inserção e detecção de padrões de projeto em código-fonte. Beluzzo (2018) encontrou um trabalho em 2017 "*MORE: A multi-objective Refactoring recommendation approach to introducing design Patterns and fixing code smells*" e publicou um artigo sobre "*A Refactoring architecture for measuring and identifying spots of design Patterns insertion in source code"*.

O Quadro 11 apresenta os estudos que foram encontrados de 2017 a 2020 durante o mapeamento em que os agentes são abordados, mas não estavam de acordo com o objetivo deste mapeamento após a análise ter sido realizada.

Quadro 11 - Trabalhos que foram excluídos após sua análise

Titulo	**Ano**	**Base**
A Lightweight Approach for Detection of Code Smells	2017	*Springer*
Software Design Smell Detection: a systematic mapping study	2018	*Springer*
A systematic literature review: *Refactoring* for disclosing *code smells* in object oriented software	2017	*Scopus*
Refactoring opportunity identification *Method*ology for removing long *Method smells* and improving *code* analyzability	2018	*Scopus*
Empirical Evaluation of the Impact of Object-Oriented *Code Refactoring* on Quality Attributes: A Systematic Literature Review	2018	*Scopus*
A systematic literature review on the detection of *smells* and their evolution in object-oriented and service-oriented systems	2019	*Scopus*

Fonte: Autoria própria

Portanto, a quantidade de estudos sobre refatoração envolvendo agentes são: 3 (três) em 2017, 4 (quatro) em 2018 e 1 (um) em 2019, totalizando 8 (oito) trabalhos o que indica que o assunto é alvo de pesquisas futuras.

4.2.1.2 Q2. Como é definida a arquitetura do método ou abordagem que utilizam agentes?

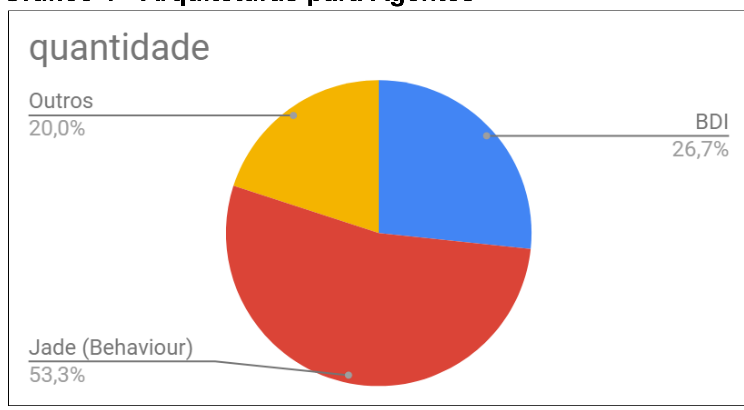
Em relação à arquitetura os trabalhos foram categorizados de 3 (três) formas:

- Uso do BDI (*Belief Desire Intention*): trabalhos que usam os conceitos de BDI na criação de sua abordagem. Dos 15 (quinze) trabalhos, 4 (quatro) foram classificados nesta categoria.
- Uso de arquiteturas fundamentadas em comportamentos: trabalhos que criaram abordagens usando como princípio comportamento implementados no framework Jade (2019). Dos 15 (quinze) trabalhos, 8 (oito) foram classificados nesta categoria.
- Uso de arquiteturas próprias: trabalhos que criaram suas próprias arquiteturas com agentes sem usar como fundamentação os conceitos de BDI e de comportamento. Nesta categoria foram classificados 4 (quatro) trabalhos.

O Gráfico 1 apresenta os estudos que contemplam arquiteturas dos métodos ou abordagens para refatoração. Nota-se que o uso de abordagens baseadas em

comportamentos é mais predominante representado aproximadamente 53% dos trabalhos.

Gráfico 1 - Arquiteturas para Agentes



Fonte: Autoria própria

O Quadro 12 exibe que 4 (quatro) dos estudos selecionados são voltados a arquitetura BDI, e outros 3 (três) abrangem as arquiteturas próprias. A arquitetura voltada a comportamentos construída usando Jade estava presente em 8 (oito) estudos. Ressalta-se que o trabalho S4 utiliza BDI e comportamento em uma mesma arquitetura.

Quadro 12 - Separação de estudos por Arquitetura

Arquitetura	Código
BDI	S4, S7, S9, S12
Comportamental	S1, S2, S3, S4, S8, S10, S11, S14
Arquiteturas próprias	S5, S6, S13

Fonte: Autoria própria

Notou-se que os trabalhos mais antigos aplicam arquiteturas usando comportamento ou constroem suas próprias arquiteturas e os mais recentes aplicam BDI.

Ressalta-se que o trabalho S14 não menciona o uso de BDI, porém ao analisá-lo observou-se que o autor evolui seu estudo de S7, no qual utiliza BDI dentro de sua arquitetura.

4.2.1.3 Q3. Quais adaptações foram feitas ao modelo arquitetural para contemplar refatorações?

O Quadro 13 apresenta possíveis adaptações na arquitetura para implementação do agente em refatoração. Em alguns casos, o estudo não explicitava a sua arquitetura e nem referenciava outros estudos dos quais foram extraídos, porém observou-se nas referências que o autor usava outros estudos na área.

No caso do estudo S2 e S3 foram incluídos na mesma linha e isto se deve ao fato que em ambos os casos o artigo apresentado foi muito parecido. De forma geral, os estudos são dados como continuidade entre ambos, além disto no S4 é observado que ele começa informando que o mesmo é uma sequência ao estudo S2 e S3.

Quadro 13 - Como a arquitetura foi aplicada

Código	Adaptações na Arquitetura
S1	Agentes inteligentes são voltados a dispositivos móveis. A interação entre Agentes e *WebServices* usa a especificação DAML-S (OWL-S). A arquitetura cliente-servidor.
S2, S3	Na IDE (*Integrated Development Environment*) de programação existem os sensores e reagentes que realizam o monitoramento código-fonte. O Agente é composto por uma central de controle possuindo um analisador e um *parser* para código-fonte. O Agente tem a base de conhecimento composta por regras e as classes de refatoração. Agentes são considerados inteligentes e essencialmente baseados em comportamentos por terem sido criados com JADE.
S4	A arquitetura possui as características que foram descritas para S2 e S3, porém a implementação foi realizada em BDI4Jade. A BDI4Jade é uma plataforma para construir Agentes com JADE e BDI, ou seja, além de comportamentos os agentes tem estados mentais. Foram criados 3 (três) assistentes para ajudar na plataforma: Assist. Avaliador de Testes Efetuados, Assist. de Fraudes para Evitar Trapaças de Alunos, Assist. Estatístico.
S5	Não faz uma descrição do agente, pois este artigo trabalha fazendo comparações e não descreve como a arquitetura foi construída.
S6	Separado em 3 (três) controles: responsável por checar o corpo do método, DDG (*Data Derivation Graph*) que serve para acompanhar como os valores dos artefatos mudam, e por fim, os artefatos que representam todas as classes do código-fonte.
S7	A arquitetura deste trabalho usa as características empregadas no estudo S4 e prevê a comunicação com o portal DeLC (Distributed e-Learning Center). A comunicação é realizada por meio de requisições SOAP (Simple Object Access Protocol) em que se criou o AVCallprocessor para comunicação entre agentes e o portal.
S8	Neste estudo a arquitetura desenvolvida contempla conceitos do trabalho de S7, porém existe uma Ontologia integrada e contém ferramentas administrativas.

Código	Adaptações na Arquitetura
S9	A arquitetura é fundamentada em BDI e contém as seguintes características: verifica o código-fonte por meio de percepções, a base de crenças é responsável por manter as regras necessárias a refatoração e ler as classes, tem biblioteca de Planos para aplicar refatorações. Quando um Plano é selecionado, ele conterá uma intenção que será analisada na base de crença para que a refatoração seja propriamente aplicada.
S10	Usa Jade, sua arquitetura tem *Behaviour* (Comportamento) para construir o agente e se comunicam com ACL (*Agent Communication Language*) definido pelo FIPA (*Foundation for Intelligent Physical Agents)*. Usa o agendamento do Jade para iniciar a refatoração e faz ordenação de *code smells* a fim de reduzir sua redundância durante a detecção.
S11	A arquitetura contém as características do estudo S10, além de indicar que seu agente inteligente é composto por sensores e percepções do meio em que está alocado.
S12	Usa BDI devido a implementação com Jason e utiliza JDT (*Java Development Tools*) para implementação da ferramenta e LTK (*Language Toolkit*).
S13	Arquitetura estruturada em, MAS (*Multi-agent Systems*) é aplicada e os agentes são divididos em: Agente Gerenciador, Agente Medidor (para ver a qualidade, ou seja, métricas), Agente Refatoração, e por fim, o Agente de Configuração de Gestão.
S14	A arquitetura foi criada com os conceitos aplicados no trabalho de S7, utilizando *clusters* de agentes chamado de *MyDeLC.*

Fonte: Autoria própria

4.2.1.4 Q4. Quais são as ferramentas desenvolvidas pelos pesquisadores encontrados em Q3?

O Quadro 14 exibe as ferramentas que foram desenvolvidas pelos pesquisadores com trabalhos na área de refatoração com agentes. Observa-se que no estudo S10 foi criada uma ferramenta que classifica os *bad smells* de acordo com tipo de refatoração necessária para melhorar o código. Em S6 a ferramenta de refatoração foi criada utilizando *Litle-JIL* e aceita somente refatorações em códigos Orientados a Objeto.

Quadro 14 - Ferramenta encontradas

Código	**Ferramenta**
S10	Ferramenta baseada em Agentes para classificação topológica de *bad smells* e refatoração analisando a complexidade no código-fonte.
S6	Ferramenta criada com *Litle-JIL* para apoiar a refatoração de programas Orientados a Objetos.
S2, S3, S4, S7, S14	Ferramenta ReLE (*Refactoring eLearning Environment*) construída para ser um *Plugin* para avaliar o código-fonte desenvolvido por alunos.
S9, S12	*AutoRefactoring* para construir agentes capazes de realizar a refatoração de código-fonte em Java.
S13	Sistema multi-agente chamado Peixe Espada para realizar tarefas de manutenção em software por meio de refatorações.

Fonte: Autoria própria

Em S9 e S12 utilizaram *AutoRefactoring* para realizar a refatoração de códigos em Java. No estudo S13 foi criado um sistema de multi-agente, denominado Peixe Espada, que realiza a manutenção em software por meio de refatorações, sendo está a única fundamentada em multi-agente.

4.2.1.5 Q5. De qual forma o código-fonte é analisado?

Nesta pergunta procurou-se responder qual a ferramenta foi usada para extração de dados do código-fonte, tais como: ASTParser que é feito com a técnica de AST (Abstract Syntax Tree), BCEL, RParser entre outras.

O Quadro 15 apresenta a ferramenta, a quantidade de vezes que foi usada nos estudos, tipo de ferramenta e em quais estudos. Neste quadro observa-se que a ferramenta mais usada para análise de código-fonte é o RParser sendo está contemplada em 6 (seis) estudos. As demais foram encontradas em um único trabalho. Em S12, os autores utilizam 3 (três) ferramentas para a análise de code smells, porém sugerem que podem ser substituídas por outras: DECOR (Método de detecção) (MOHA, 2009), *inFusion* (*Plugin*) (SRL, 2012), PMD (*Plugin*) (GRAF, 2013) e Stench Blossom (*Plugin*) (FRONT, 2014).

Quadro 15 - Como o código-fonte foi extraído

Nome da Ferramenta	Quantidade de estudos	Tipo de Ferramenta	Código
ASTParser	1	Biblioteca	S10
BCEL	1	Biblioteca	S13
RParser	6	Biblioteca	S2, S3, S4, S7, S8, S14
RefactorIT	1	*Plugin*	S6
JDeodorant	1	*Plugin*	S12
Checkstyle	1	Biblioteca	S12
inCODE	1	*Plugin*	S12

Fonte: Autoria própria

Vale lembrar que ferramentas do tipo de “Biblioteca” podem ser usadas dentro de novos projetos de ferramentas de refatoração. O tipo *Plugin* exige que o desenvolvedor a instale na IDE, no caso das ferramentas identificadas, todas tem suporte a Eclipse.

4.2.1.6 Q6. Existe alguma relação entre os trabalhos dos pesquisadores do assunto? Por exemplo, autor 1 usa o método do autor 2?

Neste contexto não foi encontrado autor que utiliza conceitos de outros autores. Identificou-se que em alguns trabalhos o autor continua sua pesquisa e por isso participa como autor ou coautor dos trabalhos posteriores. O Quadro 16 apresenta os estudos que respondem a Q6 com a relação entre trabalhos dos pesquisadores.

Quadro 16 - Relação entre trabalhos dos pesquisadores

Código	Agrupados por
S4	Continua o trabalho de S2, além de ter esta referência ao mesmo e alguns autores em comum
S7	Continua os trabalhos de S2, S3, e S4, além de ter esta referência ao mesmo e alguns autores em comum
S8	Continua o trabalho de S3, além de ter esta referência ao mesmo e um autor em comum nos trabalhos
S10	Continua o trabalho de S11 que foi publicado um pouco antes sendo 2 (dois) autores iguais em ambos os trabalhos

Código	Agrupados por
S12	Continua o trabalho de S9, não fazem referência entre si, porém são quase os mesmos autores além do assunto bem relacionado
S14	Continua o trabalho de S7, não fazem referência entre si, porém o autor principal e a ferramenta foco do estudo é a mesma

Fonte: Autoria própria

Os trabalhos S7 e S14 não se referenciam entre si, mas o autor principal do trabalho S7 é o mesmo do trabalho S14, isto também ocorreu em S9 e S12.

4.2.1.7 Q7. Como foram realizados os experimentos para validar o método de refatoração com agentes?

O Quadro 17 apresenta como foi realizado o experimento por meio da extração dos estudos selecionados para análise.

Quadro 17 - Como o experimento foi realizado

Código	Processo de Refatoração do Agente / Experimento
S1	Foi feito para analisar e avaliar o código escrito pelos alunos em tempo real. O agente fornece recomendações de mudanças em sua estrutura para melhorar sua qualidade. A análise e avaliação é feita por um assistente inteligente que representa o agente de refatoração em conformidade com regras de refatoração para linguagem de programação Java. O agente pode trabalhar de 3 (três) maneiras: 1) automaticamente depois de receber a confirmação do usuário nos casos que a refatoração é simples e os critérios para aplicação são bem claros; 2) exibir instruções detalhadas ao usuário para mostrar onde e como o código-fonte será alterado; 3) faz perguntas ao usuário afim de esclarecer condições atreladas a refatoração, ou seja, a escolha entre diferentes métodos de refatoração. O experimento realizado não especificou a quantidade de turmas e códigos analisados.
S2	É baseado no experimento do estudo S1, porém testado com o *plugin* Eclipse para casos de "*Inline Temp*", "*Replace conditional with polymorphism*", "*Typecode questionnaire behavior*".
S3	É baseado no experimento do estudo S1. O estudo fornece um exemplo de como foi usado para refatoração com "*Inline Temp*". No processo ele segue as etapas quando o botão do agente de refatoração é pressionado a primeira vez: 1) geração da árvore de sintaxe; 2) identificação das variáveis locais colocando as mesmas em destaque o que acontece normalmente com uma cor diferenciada na linha respectiva; 3) Ativação do diálogo para mostrar ao usuário a necessidade de refatoração; 4) quando o botão é pressionado novamente o agente de refatoração é desativado.

Código	Processo de Refatoração do Agente / Experimento
S4	É baseado no experimento do estudo S1, mas faz a validação de código-fonte por métricas sendo LOC (linhas de código) para avaliar classe ou método, número de métodos ou atributos por classe e assim sucessivamente. O Agente utiliza efeitos sonoros quando encontra código-fonte para refatorar. Os sensores e reagentes são coordenados pela função de LC (*Local Control*) do agente. O LC usa as informações dos sensores e as regras de refatoração que estão na RKB (*Refactoring Knowledge Base*). O código-fonte é analisado por etapas sendo: 1) *RParser* analisa o código-fonte e transforma em estrutura de arvore; 2) *RAnalyzer* analisa a estrutura e permite a filtragem de métodos adequados para refatorar aquele trecho.
S5	A automação do agente de refatoração cria solicitações para vincular o serviço necessário com base nos recursos selecionados e artefatos definidos pelo agente criador do teste de unidade. Utiliza BPEL para fazer a implementação do agente e é separado em uma estrutura de Cliente / servidor onde a comunicação do Cliente para o servidor ocorre com WSDL (Web Services Description Language). Não foi especificado no artigo a quantidade de códigos utilizados para avaliar o processo de refatoração.
S6	É apresentado um exemplo de refatoração chamado "*separating query from modifier*", em que é feito por meio de um processo em *Little-JIL*. A refatoração é realizada em uma classe exemplo chamada *CheckingAccount.java.* O processo é executado via terminal Linux e pede algumas informações de interação com usuário como por exemplo digitar o nome do método. O experimento foi realizado em uma única classe.
S7	É baseado nos estudos S2, S3, S4. Salienta que a análise não é feita somente em arquivos que estão sendo editados, mas também todos os outros de um projeto. A quantidade de projetos e arquivos analisados não foram especificados durante a realização do experimento.
S8	É baseado no experimento do estudo S3. Mas, neste estudo também foi verificado que as diferentes reações do agente levam a diferentes comportamentos dos estudantes que usavam a ferramenta. Foi observado que a ajuda do agente ou assistente inteligente ajuda os estudantes a serem mais criativos nos seus projetos e os motiva a tomarem decisões próprias.
S9	Foi feito analisando 2 (duas) classes chamadas *Contact* (com definições gerais para contato) e *Media* (com definições gerais para arquivos). A refatoração é medida por meio da fórmula de coesão do método que verifica quantas vezes o atributo é usado dentro do método ou na sua classe e quantas vezes ele é usado fora da classe.
S10	É baseado no estudo de S11 e neste estudo foram utilizados trechos de código-fonte que apresentavam mal e bons códigos. O experimento foi separado em 4 (quatro) etapas: 1) agente de análise: responsável por ler o código-fonte e gerar uma AST desta leitura; 2) identificar cheiros: recebe a sintaxe e analisa para verificar se há *bad smells*; 3) agente de agendamento: verifica os *bad smells* e os organiza para o próximo agente; 4) agente de refatoração: este é responsável pela refatoração em si.
S11	São feitos com trechos de código-fonte demonstrando o que é um trecho ruim ou bom. São usadas várias técnicas de refatoração: 1) *Extract Method*; 2) *Replace Method with Method Object*; 3) *Replace temp with Query*. O estudo não produziu em si nenhuma ferramenta, só esboça como fazer a detecção de forma manual ou semiautomática de bad *smells.*

Código	Processo de Refatoração do Agente / Experimento
S12	O experimento foi separado em etapa de Planejamento em que se formula questões essenciais para responder ao trabalho, tais como: se o projeto suporta *code smells*, se a plataforma diminuí o número de *code smells*, se melhora a qualidade do código-fonte, se preserva o comportamento, e se o desempenho é comparável a outras plataformas. Fazem a comparação de ferramentas de análise sendo separado em etapas: 1) *public fields*: comparou *JDeodorant* e *inCode*; 2) *Feature Envy*: comparou *JDeodorant* e *inCode*; 3) *Data Clumps*: comparou *JDeodorant* e *Checkstyle*. No procedimento geral do experimento foi avaliado 5 projetos de código aberto variando de 166 linhas até 711 classes, procurou-se selecionar projetos *Java* bem conhecidos e usados na prática para acoplar o máximo de ambientes possíveis tais como: ferramenta de modelagem UML, mecanismo de banco de dados, estrutura de multi-agentes, editor de textos, software de desenho e aplicações para design de interiores.
S13	O experimento foi testado em 4 (quatro) projetos, além disto a abordagem foi feita usando métricas tal como a LOC (Linhas de código). Foram considerados 5 (cinco) tipos de refatorações e 3 (três) atributos de qualidade. Durante o estudo podem ser vistos pequenos trechos de código-fonte para explicar como a refatoração está funcionando. Utilizou na quarta versão do sistema para extração de código-fonte à biblioteca BCEL para ler código binário Java, já na sua versão atual o Peixe-espada usa a extração de código via AST. Os experimentos foram restritos pela atual implementação da ferramenta Peixe-Espada: criado via *Maven* e para ser versionado usando o *Subversion.*
S14	É baseado no experimento do estudo S7. Comenta também que os sensores fornecem métricas básicas ao agente que é usado para a filtragem inicial de possíveis métodos de refatoração que podem ser melhor avaliados.

Fonte: Autoria própria

Observando os dados apresentados no quadro constatou-se que somente 2 (dois) estudos S12 e S13 realizaram experimentos com projetos e os outros utilizaram trechos de códigos ou códigos de alunos não especificando as quantidades avaliadas.

4.2.2 Lições Aprendidas

Durante o processo de obtenção, ordenação e extração dos trabalhos, foram observadas várias situações que merecem ser relatadas como lições aprendidas. Mediante as quais podem citar:

- Vários trabalhos são continuidade de outros, por exemplo, S9 e S12.
- BDI é a arquitetura aplicada na criação de agentes de refatoração nos estudos mais recentes.
- Nem todo o trabalho procura demonstrar resultados, alguns só mostram a arquitetura.
- Ao realizar a pesquisa no Google Scholar um estudo (S14) estava com nome diferente: “Context-Aware E-Learning Infrastructure Stanimir Stoynov, Hussein Zedan, Emil Doychev, Veselina Valkanova” e deveria

estar como nome sendo: “Context-Aware E-Learning Infrastructure”. Neste caso, acessou-se o artigo e para ele ser exportado ao Mendeley o seu nome ficou como especificado no segundo título.

- Nem todo o estudo obtido na pesquisa está de acordo com objetivo, por exemplo, na pesquisa no *Google Scholar* surgiu o resultado “*Computer and Information Sciences II*” que é um livro com uma coletânea de artigos e não havia um artigo que abordasse agentes e refatoração. Os assuntos dentro deste livro eram contemplados de forma isolada. Neste caso, o livro foi excluído para leitura completa.
- Em relação ao processo de experimento das ferramentas já existentes na literatura e são que são baseadas em agentes, utilizam pouco projetos para sua avaliação.

4.3 TRABALHOS RELACIONADOS

Durante o processo de leitura e extração de dados, observou-se algumas características comuns entre os trabalhos que abordam o tema de refatoração baseado em padrões de projeto tais como: IDE de programação, ferramentas para criação de agentes e arquitetura para sua implementação.

O Quadro 18 apresenta os estudos que foram encontrados com a indicação de qual IDE de programação usavam. É interessante perceber que há uma repetição do estudo S6 que faz uso das IDE’s Eclipse, JBuilder e IntelliJ IDEA. Neste quadro tem 7 (sete) estudos que usaram Eclipse e 7 (sete) que não documentaram o que foi usado.

Quadro 18 - Trabalhos e suas IDE de programação

IDE	Código
Eclipse	S2, S3, S4, S6, S7, S12, S14
JBuilder	S6
IntelliJ IDEA	S6
Não Informado	S1, S5, S8, S9, S10, S11, S13

Fonte: Autoria própria

A metade dos trabalhos usa a IDE Eclipse para desenvolvimento do projeto com exceção de S1, S5, S8, S9, S10, S11, S13 que não mencionam sobre a IDE de implementação.

No Quadro 19 apresenta as ferramentas encontradas para a construção de agentes, sendo que o único caso em que não se considerou é o estudo S1 pois foi estruturado com a ferramenta *eLSEBuilder Application.* Esta não foi incluída na tabela porque é uma ferramenta para criar programas fazendo a apresentação dinâmicas de fluxo de processos de software. O uso desta ferramenta explica o portal de alunos DeLC, e não é criada para agentes.

Quadro 19 - Ferramentas para produção de Agentes

Nome	Código
Jade	S2, S3, S4, S7, S10, S11, S14
Juliette Interpreter	S6
Jason	S9, S11, S12
Não Informado	S5, S8, S13

Fonte: Autoria própria

Em relação a criação de agentes, S1 não contém a descrição da implementação, somente relata o que agente de refatoração faz. A criação do agente foi relatada em S2, sendo este uma continuação do trabalho de S1 em que os autores utilizam Jade para concepção do agente.

A Figura 24 exibe a arquitetura da criação do agente definida nos trabalhos S2 e S3. A arquitetura proposta pelos autores é composta de 2 (dois) módulos: BE (*Back-end*) e FE (*Front-end*). O módulo BE é responsável pela parte do servidor que tem o agente composto por *Controle Local*, *RAnalyzer* e *RParser*.

Figura 24 - Arquitetura ReLE

BE (JADE)
FE(IDE)
Agente de refatoração
Sensores
Controle Local
Reagentes
RAnalyzer
RParser
Base de conhecimento para refatoração
Regras
Classes de refatoração

Fonte: Sandalski (2011)

O *Controle Local* é um componente que serve para coordenar sensores e reagentes. Os sensores fornecem informações básicas de métricas ao agente, que é usado para filtragem inicial de possíveis métodos de refatoração que podem ser melhor avaliados. Já o papel dos reagentes é desencadear diferentes eventos que podem auxiliar os alunos durante a realização de suas tarefas no FE, em que eles estão trabalhando. Os eventos dos reagentes podem ser:

- Exibir mensagens em janelas de diálogo, balões, etc.
- Emitir sinais sonoros, mensagens vocais.
- Representar o agente na forma de animação para exaltar o efeito.

O *Controle Local* se comunica com os *Behaviours* (comportamentos) do Jade e com o *RAnalyzer* que usa entendimento da base de conhecimento do agente para saber as regras e como aplicá-las. O *RParser* transforma o código-fonte da classe lida em uma estrutura de árvore chamada de AST. Após ter a árvore criada, é chamado o *RAnalyzer* que é responsável por encontrar os *bad smells* e filtrar os métodos de refatoração necessários. Na próxima fase ocorre a interação com o usuário fornecendo 3 (três) opções: 1) refatoração automática, ocorre depois de ter a confirmação do usuário sobre o que vai ser aplicado ou modificado; 2) proposta de

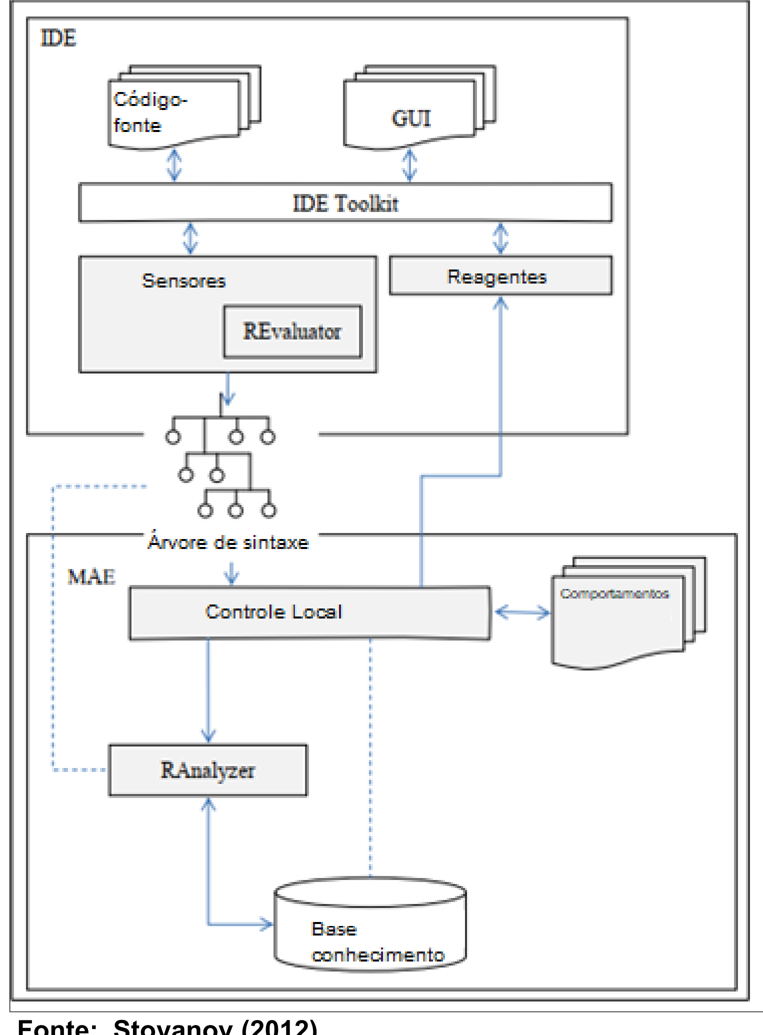
refatoração, fornece ao usuário trechos de código-fonte que podem ser refatorados e uma explicação de qual método empregar; 3) refatoração com questionamento, ocorre quando o sistema precisa fazer perguntas ao usuário para confirmar a aplicar as refatorações que foram detectadas.

O componente *Base de Conhecimento para Refatoração (RKB)* consiste em um conjunto de regras que são unidas a um conjunto de classes para construir a base de conhecimento consistente. Cada regra desta base é usada para descrever uma forma comum de condições que permitem que um método particular de refatoração entre na lista tendo como base algumas métricas.

A Figura 25 apresenta a comunicação da arquitetura ReLE. As requisições do *plugin* ao agente de refatoração são feitos pelo uso de requisições SOAP (*Simple Object Access Protocol*) para enviar a mensagem de comunicação ao agente. Esta arquitetura de agentes foi testada com técnicas de refatoração tais como: *Move Method, Move Field, Extract Class, Extract Method* entre outros. A maneira como o agente age pode exemplificado pela técnica *Move Method*:

- Se baseando nas regras do RKB o agente descobre que há um método da classe A que usa recursos principalmente da classe B, o que implica na aplicação do *Move Method.*
- O agente mostra uma mensagem na qual ele oferece ao usuário a opção de mover o método para a classe B.
- Caso o usuário venha a concordar, o agente move todo o método da classe A para a classe B por:
- Corrigir todas as referências aos recursos da classe A, para que sejam acessíveis à classe B e adiciona os parâmetros necessários ao método, se necessário.
- Substituir todas as invocações de método, de modo que usem sua nova posição na classe B.

Figura 25 - Comunicação na Arquitetura ReLE



Fonte: Stoyanov (2012)

Muitas vezes duas ou mais técnicas são possíveis para refatorar um pedaço de código-fonte, neste caso:

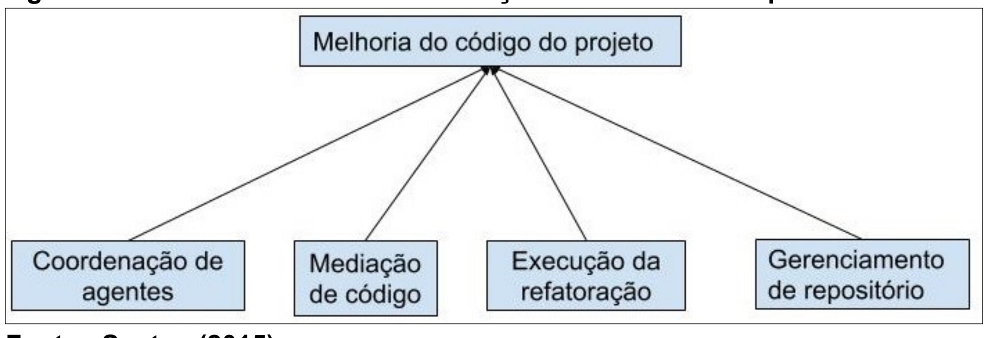
- O agente descobre que uma classe contém um "*type code*" numérico, mas não pode determinar se este código altera o comportamento da classe selecionada.
- O agente exibe uma pergunta ao usuário para saber se este "*type code*" pode influenciar no comportamento da classe.
- Se a resposta do usuário for "não", o agente oferece a ele a possibilidade de trocar o código de tipo com o método da classe.
- Se a resposta for "sim", o agente pergunta ao usuário se o atributo "*type code*" pode mudar durante o ciclo de vida do objeto.

- Se a resposta do usuário for "não", o agente oferece ao usuário a possibilidade de fazer a substituição do método do "*type code*" com uma subclasse.
- Caso contrário, o agente oferece a substituição do "*type code*" usando *State* ou *Strategy.*

Sobre a ferramenta ReLE (*Refactoring eLearning Environment*) ela é um *Plugin* para *Eclipse,* criado com integrações diretas ao *JADE* que é um framework para agentes e possibilita a avaliação por métricas: número de métodos ou atributos por classe, LOC (Linhas por Código) que são verificados por classe e método. Além disto, a arquitetura ReLE é apresentada nos estudos em S1, S2, S3, S4, S7, S8 e S14.

No estudo S13, na quarta versão do sistema, ele utiliza para a extração de dados o BCEL (faz leitura do código binário Java, ou seja, o arquivo .class), mas na versão atual usa o AST (*Abtract Syntax Tree*). Esta evolução foi necessária porque a AST é mais fácil iterar entre as informações da classe extraída. O Peixe-espada em geral trabalha de acordo com Figura 26 que apresenta suas principais funcionalidades. Pode ser visto nesta figura as funções de: coordenação do agente que é essencial porque em um sistema multi-agente deve haver comunicação entre as várias réplicas do agente de refatoração, medição de código usada para saber o quanto o código-fonte melhorou ou não, execução da refatoração para selecionar os candidatos a refatoração e fazer a aplicação e modificação do código-fonte e por último o gerenciamento de repositório.

Figura 26 - Funcionamento da Refatoração Usando Peixe-Espada



Fonte: Santos (2015)

A Figura 27 mostra como o agente de refatoração trabalha na arquitetura do peixe-espada mostrando os passos base do mesmo. O agente faz 2 (duas) ações

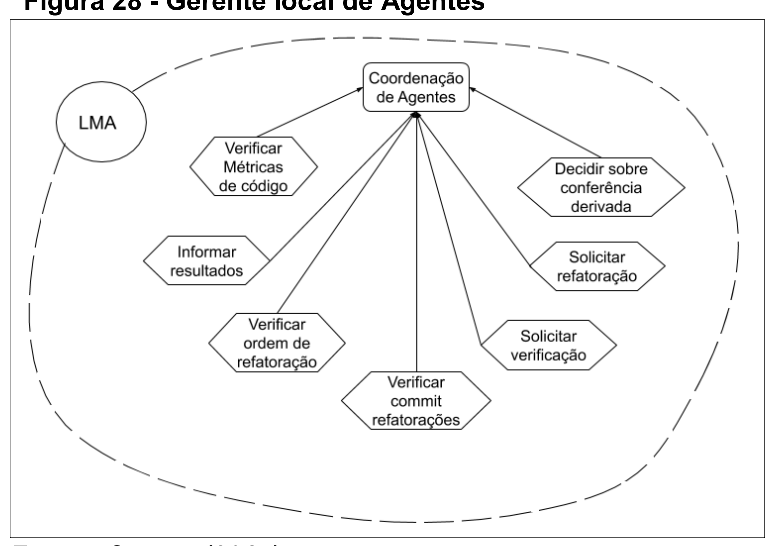
principais que são: busca por sintomas e aplicar a refatoração que são estruturadas no *Refactoring Agent* (RA).

Figura 27 - Agente de refatoração - Peixe Espada

Fonte: Santos (2015)

Devido ao sistema Peixe-Espada ser multi-agente ele precisa de uma gestão do trabalho dos vários agentes. A Figura 28 exibe como é o gerente local de agentes e suas funções são: perguntar por métricas de exportação, informar resultados, perguntar sobre a ordem de refatoração, perguntar sobre *commit* de transações de refatoração, solicitação de *checkout,* solicitar refatoração e decidir sobre conferência derivada.

Figura 28 - Gerente local de Agentes



Fonte: Santos (2015)

O *ASTParser* (desenvolvido com AST, faz leitura direto do arquivo .java e traz separado as propriedades da classe lida) do estudo S10. A Figura 29 mostra como o agente de refatoração age:

1. A inserção de código-fonte pelo usuário.

2. O processo de detecção começa de acordo com as regras de refatoração existentes, ou seja, ele busca no código-fonte pode ser aplicado alguma técnica de *bad smells* que esteja gravada anteriormente.
3. Ordena os *bad smells* com o intuito principal de evitar a redundância, ou seja, se uma técnica A cobre o *bad smells* X e Y, então não precisa aplicar a técnica de refatoração B que contempla só o *bad smell* Y.
4. Agente de refatoração começa seu trabalho fazendo a aplicação de técnicas de refatoração ao código-fonte que foram estipuladas no processo nas etapas anteriores.
5. Por último, tem a saída do código-fonte de novo para a classe.

Figura 29 - Detecção de *bad smells* e refatoração do estudo S11

Fonte: Lakshmi (2013)

Os estudos S10 e S11 prevêm o uso do agente com os tipos de *bad smells:*

- *Long Method:* Se o método do código-fonte for grande em questão de linhas internas.
- *Large class:* Uma classe que tem grande número de linhas e tem mais responsabilidades.
- *Long Parameter List:* Muitos parâmetros sendo passados no método
- *Feature Envy:* método ou fragmento de um método interessado no recurso envolvido em outra classe.
- *Primitive Obsession:* Uso de muitos objetos primitivos.
- *Useless field, Class Method:* Muitos campos, classes e métodos podem ser definidos e não usados, este tipo de coisa é chamado de campos, classes ou métodos inúteis.

São considerados somente em S10:

- *Data class*: Uma classe que contém variáveis e seus métodos de *getter* e *setter* é uma classe de dados. Outra classe pode usá-la para seus próprios comportamentos.
- *Middleman Smell*: quando uma solicitação para o cliente é realizada por outro método delegado então é chamado de *Middleman Smell*.

O Quadro 20 mostra uma coluna para *bad smell* e na terceira coluna lista as técnicas de refatoração que podem ser usadas para resolver um determinado *bad smell.* Pode ser visto que para um mesmo *bad smell* tem 2 (duas) ou mais técnicas de refatoração que podem ser usadas, ou seja, se em uma classe tem o *bad smell* 1 (um) e 4 (quatro) não é necessário aplicar todas as técnicas de refatoração relacionadas, é provável que o melhor caminho seja aplicar o *Extract Method* na classe que é a técnica de refatoração que intercala ambos *bad smells* 1 (um) e 4 (quatro) e depois refazer a análise para ver se eles foram corrigidos na classe.

Quadro 20 - Técnicas de Refatoração para Bad Smells (S10, S11)

Número	*Bad Smell*	Técnica de Refatoração
1	*Long Method*	*Extract Method* *Replace Temp with Query* *Preserve Whole Object* *Replace Method with Method Object*
2	*Large Class*	*Extract Class* *Extract Subclass*
3	*Long Parameter List*	*Replace Parameter Method* *Preserve Whole Object* *Introduce Parameter Object*
4	*Duplicate Code*	*Extract Method* *Pull Up Field* *Extract Class*
5	*Primitive Obsession*	*Replace Type code with class* *Replace Data value with object* *Replace type code with subclass*
6	*Feature Envy*	*Move Method* *Extract Method*

Fonte: Lakshmi (2013)

Um exemplo de como a técnica de refatoração é aplicada nos estudos S10 e S11 para o *Extract Method,* se tem os seguintes passos que foram feitos com base em Fowler (1999)*:*

- Identificar a função do método e seu nome.
- Obter o código do método de origem e copiar para o método de destino.
- Percorrer o método extraído para capturar as referências às variáveis que são locais no escopo do código do método. Tratar estas como variáveis locais e parâmetros do método.
- Procurar por variáveis temporárias usadas apenas dentro do código extraído. Se houver, manter como variáveis temporárias no método destino.
- Verificar se alguma modificação foi feita nas variáveis locais que precisam ser retornadas. Se o número de tais variáveis é limitado a uma, devolve-se ela. Se forem duas ou mais, divida o método novamente ou torne-as definitivas as variáveis.
- Faça uma chamada para o método de destino no lugar do código extraído.
- Compilar e testar.

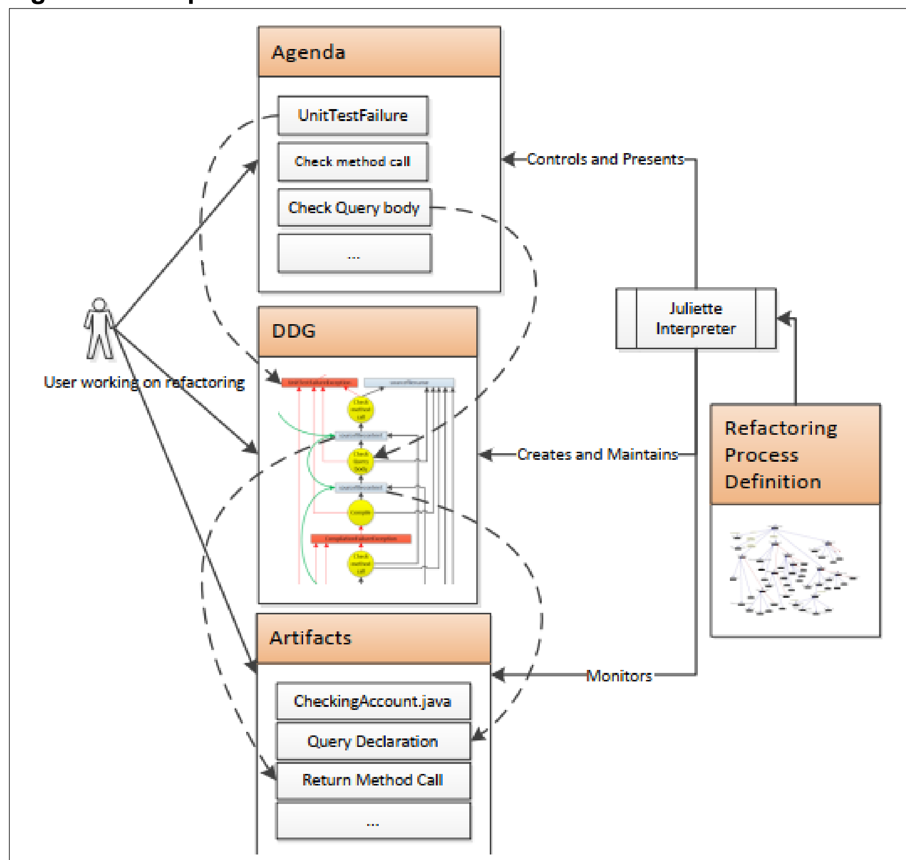
Os trabalhos S9 e S12 são dos mesmos autores, estão voltados a criação de uma plataforma baseada em agentes que permite a implementação de agente capazes de lidar autonomamente com os problemas de refatoração chamada *AutoRefactoring*.

O estudo S12 fornece exemplos de vários *plugins* ou ferramentas que podem ser usados para análise e extração de informações do código-fonte. Nos estudos S9 e S12 não é fornecido qual ferramenta é usado para extrair informações do código-fonte. O estudo S12 aponta que foram usadas as técnicas de refatoração no código-fonte de: *Extract Class*, *Encapsulate Fields* e *Move Method* sendo implementadas de acordo com Fowler (1999), mas não especifica como foi realizado a extração.

O estudo S5 apresenta que foi feito um Método Ágil baseado em XP e SCRUM para trazer agilidade ao processo de composição de serviços. Assim, o processo de produção de serviços se torna mais rápido e flexível. Em geral, pode ser visto que o estudo S5 usa uma comunicação WSDL (*Web Services Description Language*), ele não apresenta uma arquitetura propriamente dita, porém apresenta uma boa ideia de aplicativo de serviço para refatoração de código-fonte. Este estudo usa um servidor separado que recebe as requisições de refatoração onde está o agente que é responsável por verificar se refatorações podem ser realizadas no código.

A Figura 30 mostra a arquitetura do estudo S6, em que pode ser visto uma separação em *Agenda, DDG, Artifacts, Juliette Interpreter, Refactoring Process Definition*.

Figura 30 - Arquitetura do sistema - S6



Fonte: Zhao (2012)

Os elementos que compõe a arquitetura são:

- *DDG* (*Data Derivation Graph*): é um veículo para rastrear os valores dos artefatos à medida que um processo é executado.
- *Juliette Interpreter*: é construído sobre um substrato de objetos distribuídos que permite que as partes do interpretador sejam distribuídas próximas aos agentes com os quais devem interagir (CASS, 2000).
- *Agenda:* usado para atribuir trabalho a agentes possivelmente móveis (CASS, 2000).
- *Artifacts:* apresenta o conhecimento do programa propriamente dito, mediante as várias classes e métodos internos a elas.

O estudo S6 mostra como exemplo a refatoração chamada "*Separating query from modeling*" que é baseado na técnica de Fowler (1999). Nesta técnica se tem um método usado para consulta e para alterar o estado de um objeto. Em primeiro momento pode parecer uma boa ideia combinar os dois recursos, porém os efeitos colaterais são inadequados (ZHAO, 2012). Além disto, vale comentar que tem várias situações em que o usuário tem que interagir e isto foi feito via terminal Linux. Neste caso, o sistema de refatoração pergunta informações tais: como nome do método modificador de consulta.

Considerando as categorias que foram explicadas anteriormente para cada trabalho relacionado, o Quadro 21 sumariza as informações.

Quadro 21 - Sumarização dos Trabalhos Relacionados

Categorias	**Síntese**
IDE de Programação	A IDE mais utilizada foi Eclipse e encontradas nos estudos de: S2, S3, S4, S6, S7, S12, S14.
Ferramenta para criação do Agente	A ferramenta mais usada foi o Jade (S2, S3, S4, S7, S10, S11, S14) e o trabalho S8 implementa com BDI4Jade para usar conceitos de BDI e Jade para agentes.
Extrator de código	Encontradas bibliotecas tais como: ASTParser (S10), JavaParser, mas todas implementam o conceito de AST (S2 e S3).
Arquitetura	Arquiteturas próprias são encontradas em trabalhos mais antigos normalmente usando Jade (S2, S3, S4, S7, S10, S11, S14) e voltado a uma arquitetura comportamental do agente. Trabalhos atuais usam a BDI (S15).
Experimentos	São utilizadas com poucas classes e o projeto que tinha mais classes continha aproximadamente 40 arquivos foi o de S15.

Fonte: Autoria própria

4.4 CONSIDERAÇÕES FINAIS DO CAPITULO

Este Capítulo apresentou um mapeamento sistemático usando como base o que foi proposto por Kitchenham e Charters (2007). Os critérios de inclusão e exclusão do mapeamento foram adaptados à realidade de Agentes e Refatoração. O objetivo do mapeamento proposto foi o levantamento de trabalhos relacionados ao tema de refatoração com agentes. O mapeamento encontrado neste trabalho foi aplicado pelo autor deste trabalho e duas professoras da área de engenharia de software.

Foram identificados um total de 14 (quatorze) estudos que abordam o assunto deste mapeamento e o período realizado para pesquisa foi de 1998 até 2020 conforme pode ser visto no Quadro 8. Constatou-se que os estudos abordam agentes com técnicas de refatoração tais como: *Consoliate Conditional Expression*, *Decompose Conditional, Extract Method*, *Extract Surrounding Method, Extract Variable*, *Form Template Method*, porém não contemplam o tema de refatoração com padrões de projeto e agentes. Por isto, a abordagem Codice-Unio foi criada e é descrita no próximo capítulo.

5 CODICE-UNIO: ABORDAGEM DE REFATORAÇÃO PROPOSTA

Como mencionado no Capítulo 4, os estudos não abordam refatoração de código-fonte usando padrões de projeto com agentes. O uso de agente permite criar autonomia no processo de detecção e inserção de padrões de projeto. As refatorações que usam agente na literatura estão voltadas a identificação de *bad smells* usando técnicas de refatoração tal como *Extract Method*.

Este Capítulo apresenta a Codice-Unio para detectar pontos de inserção e aplicação de padrões em código-fonte por meio do uso de agentes em um código orientado a objetos em linguagem de programação Java. A abordagem proposta usa como referência os métodos da literatura de Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017). Escolheu-se estes métodos porque os autores disponibilizavam os algoritmos ou etapas de seu funcionamento.

O detalhamento da Codice-Unio é feito nas seções deste Capítulo. A Seção 5.1 fornece uma visão geral de como é feita a refatoração e estrutura do agente usando BDI. A Seção 5.2 apresenta os requisitos necessários para o funcionamento da abordagem proposta. A Seção 5.3 relata como o agente funciona internamente e como foi separado o BDI *(Belief, Desire, Intention)* na detecção e inserção de padrões de projeto. E, finalmente, a Seção 5.4 segue com as conclusões do Capítulo.

5.1 VISÃO GERAL

A Codice-Unio tem como objetivo trazer melhorias para o processo de refatoração de código-fonte voltado a padrões de projeto usando agentes (PACHER, MATOS, 2019). As contribuições da abordagem proposta são:

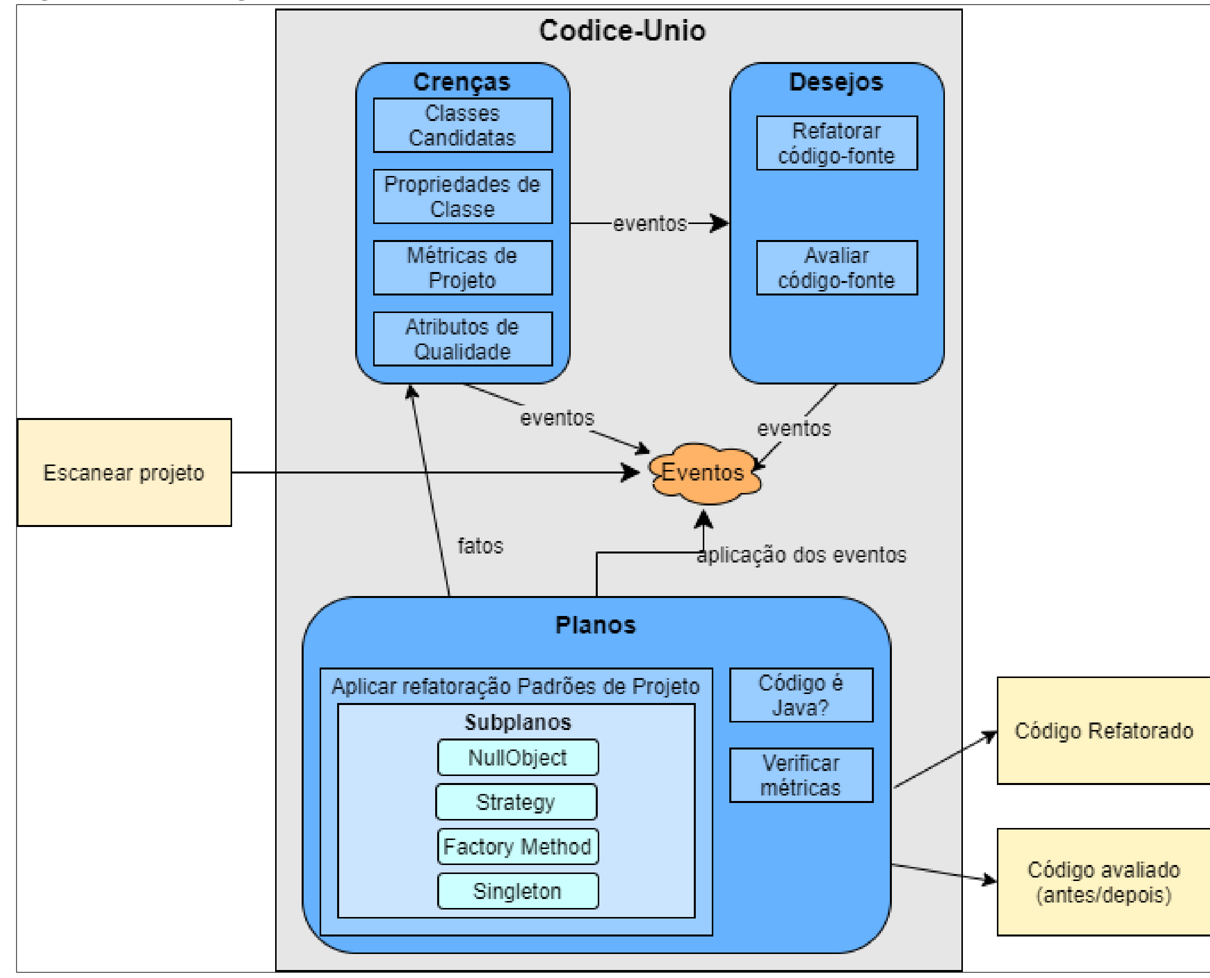
- Unir em um ambiente os métodos de refatoração de: Gaitani *et al.* (2015) que detecta e aplica o padrão de projeto *Null Object*, Wei *et al.* (2014) que identifica e insere os padrões *Strategy* e *Factory Method* e Ouni (2017) que detecta e aplica os seguintes padrões *Visitor; Factory Method; Singleton* e *Strategy.*
- Identificar quais padrões podem ser aplicados e em quais classes, analisando o conflito de identificação. O conflito pode ocorrer entre os

métodos de refatoração e os padrões de projeto. Entre os métodos ocorre quando dois ou mais métodos aplicam o mesmo padrão para a mesma classe e na mesma linha de código. Em relação aos conflitos entre padrões de projeto ocorre quando os métodos sugerem a aplicação de padrões diferentes que são conflitantes entre si. As soluções para os conflitos são detalhadas na Seção 5.2.

- Avaliar as classes sem exigir que o usuário importe seu projeto, ou seja, o usuário pode estar produzindo seu código e o agente o analisa de forma a identificar se os padrões podem ser aplicados. A ideia é que o agente após inicializado trate de todas as informações e tenha autonomia para evitar que o usuário faça qualquer ação. As informações detalhadas do processo de avaliação são apresentadas na Seção 5.3.
- Alterar o código do usuário de forma autônoma quando um padrão for detectado. O processo de alteração é detalhado na Seção 5.4.
- Informar ao usuário sobre os valores dos atributos de qualidade (manutenibilidade, reusabilidade e confiabilidade) após a aplicação dos padrões de projeto feitos pelo agente.

A visão geral da Codice-Unio é exibida naFigura 31 é está fundamentada na na BDI.

Figura 31 - Visão geral da Codice-Unio



Fonte: Autoria própria

A Codice-Unio é formada pelos componentes descritos a seguir:

- Escanear Projeto: permite visualizar a estrutura do projeto e encontrar os arquivos ".java" e representa a entrada no modelo.
- Planos: Representa a forma como o agente de refatoração atua em seu ambiente. Eles contêm uma biblioteca de ações que o agente de refatoração pode executar e são selecionados em resposta à ocorrência de eventos. Um plano é a classe responsável por aplicar a refatoração, identificar se o código é Java e verificar as métricas de qualidade. Os planos contêm subplanos que representam a implementação dos métodos de refatoração para detectar e aplicar os padrões de projeto (*NullObject*, *Strategy*, *Factory Method* e *Singleton*) nas classes do projeto que estão sendo analisadas.
- Crenças: Representa o conhecimento que o agente de refatoração tem sobre seu mundo e como as coisas internas a ele se encontram tais

como propriedades da classe, métricas de projeto, atributos de qualidade e classes candidatas.

- Desejos: São responsáveis por analisar todas as classes e verificar se é viável a aplicação de um padrão de projeto, bem como avaliar o código-fonte em termos de atributos de qualidade.
- Eventos: É a capacidade do agente reagir e foi definido por meio de eventos internos dentro do agente usando gatilhos (*triggers*).
- Código refatorado e Código avaliado (antes/depois): Representam as saídas da Codice-Unio. O Código refatorado faz a representação da gravação propriamente dita das informações alteradas na classe quando o padrão de projeto foi aplicado. O Código avaliado exibe informações sobre os valores dos atributos de qualidade (manutenibilidade, reusabilidade e confiabilidade) do código-fontes antes e depois da Codice-Unio.

As seções posteriores explicam os requisitos e detalhes do funcionamento da Codice-Unio.

5.2 REQUISITOS PARA FUNCIONAMENTO DA CODICE-UNIO

A Codice-Unio precisa de um código em Java, além da IDE (*Integrated Development Environment*) de programação Eclipse. Existem alguns sistemas de software que são separados em mais de um projeto, por isto um ponto importante a ser observado é que a abordagem proposta analisa os diretórios e subdiretórios da URL passada ao agente para verificação.

Os requisitos para a concepção da abordagem proposta são: i) identificação de métodos da literatura para detectar e inserir de padrões de projeto; ii) criação de um agente autônomo que realize a inserção e detecção de padrões de projeto usando uma arquitetura BDI e iii) aplicação de métricas para medir os atributos de qualidade. O primeiro dos requisitos foi feito para se ter conhecimento de métodos da literatura que foram implementados e que poderiam fazer parte da abordagem proposta. O segundo torna o processo de refatoração com padrões de projeto autônomo. O

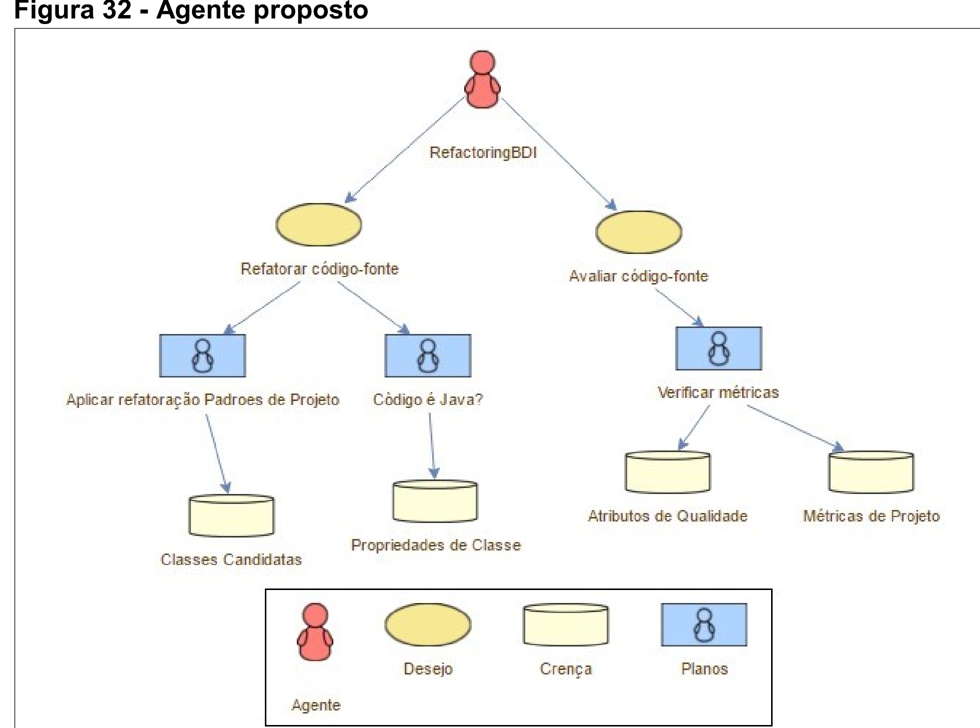
terceiro requisito permite ao agente avaliar o projeto após a aplicação dos padrões de projeto.

Os requisitos de saída são os códigos refatorados usando padrões de projeto que foram escolhidos pelo agente e a avaliação do projeto em termos de atributos de qualidade.

5.3 PROCESSO DE FUNCIONAMENTO DA CODICE-UNIO

De acordo com a Figura 31 cada componente da abordagem desempenha seu papel de forma a centralizar o uso de métodos de refatoração voltados à detecção e aplicação de padrões de projeto, provendo a análise destes no código-fonte antes mesmo de sua aplicação final. A Codice-Unio inicia a partir da leitura do projeto aberto que está sendo monitorado pelo agente (Figura 32).

Figura 32 - Agente proposto



Fonte: Autoria própria

Este agente foi separado em 2 (dois) desejos que são o ato de Refatorar o código-fonte e avaliar o código-fonte. Para execução desses dois desejos é necessário realizar a intenção de leitura do código-fonte e está irá extrair as informações da classe via AST (*Abstract Syntax Tree*) e defini-las em sua devida

propriedade dentro das crenças. Em sequência, é feito a execução dos planos que irá primeiramente ver se o código é .java, irá aplicar os padrões de projeto onde for possível e finalmente verifica as métricas dos atributos de qualidade desse código-fonte refatorado. Nesta versão da abordagem um código Java não se torna uma crença, isto pode ser realizado em um trabalho futuro.

O agente verifica o diretório do projeto realmente existe e posteriormente avalia o código original por meio de métricas que medem a manutenibilidade, reusabilidade e confiabilidade. As métricas escolhidas representam atributos de qualidade fundamentais no processo de refatoração. Para cada atributo de qualidade duas ou mais métricas são usadas para seu cálculo. A relação entre os atributos de qualidade e quais métricas são usadas para o cálculo foi obtida por do trabalho de Sommerville (2011). As métricas usadas na abordagem Codice-Unio são: DIT (*Depth Inheritance Tree*), LOC (*Lines of code*), CC (C*yclomatic Complexity*). O cálculo do atributo de reusabilidade é dado pela fórmula (1):

$$(DIT + LOC)/2 \qquad (1)$$

Já para o cálculo do atributo de confiabilidade é utilizado a fórmula apresentada em (2):

$$(CC + LOC)/2 \qquad (2)$$

Por fim, para o atributo de manutenibilidade a fórmula (3) foi usada:

$$(DIT + LOC + CC)/3 \qquad (3)$$

Após o cálculo das métricas, são percorridos todos os arquivos do projeto para verificar se eles podem ser refatorados. Durante o percorrimento dos arquivos foi verificado se o mesmo não contém no nome (*Abstract*, *Test* ou *Strategy*), pois constatou-se que quando continham esses nomes eram classes de teste ou já tinham sido refatoradas. Outro teste realizado nos arquivos foi em relação à extensão para ver se o mesmo é um (.java), pois é a implementação suportada pela abordagem proposta. Não houve a necessidade de verificar o conteúdo do arquivo, pois a leitura do conteúdo é feito via AST e se o conteúdo não for compatível nada será realizado.

Sabendo que o arquivo é adequado para análise, o agente faz a extração de informação via AST com a biblioteca *JavaParser,* guardando essas informações para serem usadas na análise de identificação de possíveis candidatos à refatoração com padrões de projeto. Nesta etapa o agente executa sua intenção chamada "IdentifyCandidates" que irá percorrer cada subplano de padrão de projeto analisando se a aplicação dele é viável ou não e realiza a sua aplicação ao código-fonte.

Para identificação dos candidatos a refatoração, foram implementados os métodos de Gaitani *et al.* (2015), Wei *et al*. (2014), Ouni (2017) que estão detalhados na Seção 2.3. Esses métodos são capazes de detectar os padrões exibidos no Quadro 22.

Quadro 22 - Métodos e seus padrões de projeto

Método de Detecção e Inserção de Padrões de Projeto	**Padrões detectados pelo método**
Gaitani *et al.* (2015)	*Null Object*
Wei *et al.* (2014)	*Strategy; Factory Method*
Ouni (2017)	*Visitor; Factory Method; Singleton; Strategy*

Fonte: Autoria própria

O uso da arquitetura BDI para a abordagem proposta foi definida de acordo com Quadro 23 em que se apresenta seu: desejo, crença e seu plano. As intenções não foram colocadas porque no framework escolhido para a implementação são usados plano e subplanos.

Quadro 23 - Codice-Unio na arquitetura BDI

<table>
<tr><th>Desejo (Goal)</th><th>Crença</th><th>Plano</th></tr>
<tr><td rowspan="2">Refatorar o código-fonte</td><td>Classes Candidatas</td><td>Código é Java?</td></tr>
<tr><td>Propriedades da Classe</td><td>Aplicar refatoração do Padrão de Projeto</td></tr>
<tr><td rowspan="2">Avaliar o código-fonte</td><td>Métricas de Projeto</td><td>Verificar métricas de qualidade</td></tr>
<tr><td>Atributos de Qualidade</td><td></td></tr>
</table>

Fonte: Autoria própria

O agente Codice-Unio usando arquitetura BDI possui 2 desejos para executar a refatoração:

- Refatorar código-fonte: procura verificar se é viável aplicar novos padrões de projeto a um determinado código.

- Avaliar código-fonte: faz a análise das métricas de atributos de qualidade de código (manutenibilidade, reusabilidade e confiabilidade) ante e depois do processo de refatoração.

Os desejos se comunicam diretamente com uma biblioteca de planos e retorna o resultado do processo de execução efetuado pelo plano que é a classe do agente. Terminando a leitura do arquivo atual, o processo é repetido para todos os códigos-fontes que estão no projeto.

A crença é responsável por guardar informações do projeto enquanto o agente está sendo executado. Dentro da implementação do agente se tem como crenças: classes candidatas, métricas de projeto, propriedades da classe que são guardadas via AST e atributos de qualidade da classe que são calculados após refatoração.

Os planos necessários para o agente realizar efetivamente a refatoração estão listados no Quadro 24.

Quadro 24 - Conjunto de Planos do Agente

Plano	Passos	Subplano
Código é Java?	1. Verificar extensão 2. Verifica se parte do nome do arquivo não é "Abstract" 3. Verifica se parte do nome do arquivo não é "Test" 4. Verifica se parte do nome do arquivo não é "Strategy"	
Aplicar refatoração do Padrão de Projeto	1. Verificar se tem padrões de projeto implementados no agente. 2. Iterar sobre cada padrão de projeto (subplanos) implementado no agente e verificar se ele é aplicável a classe lida que está sendo analisada. 3. Se a classe que estiver sendo analisada for candidata a algum padrão de projeto, o mesmo é guardado em uma lista. 4. Se a classe que estiver sendo analisada não for aplicável naquele padrão de projeto da iteração o mesmo é registrado em log. 5. Quando o agente verificar que algum padrão de projeto é viável a ser aplicado o mesmo já é aplicado a classe lida. 6. Retorna um novo objeto instanciado chamado "Candidate" que representa a classe lida e seus possíveis padrões de projeto que poderão ser aplicados.	S1 - *FactoryMethod* (ver Apêndice A) S2 - *NullObject* (ver Apêndice A) S3 - *Singleton* S4 - *Strategy* (ver Apêndice A)
Verificar métricas de qualidade	1. Verifica mediante uma URL (path) de arquivo quais seriam as os atributos de qualidade relacionados: (Complexidade Ciclomática, Profundidade da arvore de herança e Número de linhas do código-fonte) 2. Através dos atributos de qualidade obtidos o agente faz o cálculo das métricas de qualidade que seriam manutenibilidade, reusabilidade e confiabilidade	

Fonte: Autoria própria

Um exemplo de execução do Codice-Unio é apresentado na Figura 34 para o plano Código é Java?. Os nomes utilizados no diagrama correspondem a

implementação do agente no Jadex. O agente *:RefactoringBDI*, contém a crença *:ProprietyClass* a qual possui as propriedades da classe dentre elas a sua extensão, o objetivo *:FileQualifier* verifica se o arquivo lido no diretório é qualificado para refatoração e o plano *:FileExtensionPlan* avalia se o arquivo da classe lida realmente pode ser refatorado, visto que a Codice-Unio em sua versão inicial aceita somente arquivos .java.

Figura 33 - Pacotes e classes do Agente

Dentro da execução do plano chamado ele sabe quais padrões de projeto estão implementados, sendo que os padrões foram criados como sendo subplanos. Um exemplo de subplano está listado no Quadro 25. As classes de subplanos para padrões de projeto tem no mínimo dois métodos: “isApplicable” (método que verifica se o padrão de projeto é aplicável a classe) e o segundo chamado “applyMethod” (método que é responsável por alterar a classe da forma que for necessária para aplicar o padrão de projeto).

Quadro 25 - Exemplo de um subplano

Subplano	Sequências de passos
S3 - *Singleton*	1. Verifica se a classe lida usa herança de alguma outra. 2. Verifica se a classe lida é interface ou abstract. Caso seja, retorna que não é possível aplicar *Singleton*. 3. Realiza uma iteração entre os métodos da classe lida verificando quais são construtores. 4. Caso exista construtor como privado ele retorna que não é possível aplicar *Singleton* 5. Caso o construtor tenha parâmetros ele retorna que não é possível aplicar *Singleton.* 6. Caso o construtor seja diferente de privado, *abstract*, e o mesmo não tenha parâmetros e também dentro da classe não exista métodos chamados “getInstance” ele retorna que é possível aplicar o padrão de projeto *Singleton*. 7. Para aplicar o padrão de projeto ele lança um comentário no topo da classe fazendo referência que a mesma foi modificada para ter o Padrão *Singleton*. 8. Declara um atributo de nome *singleton* que tem o mesmo tipo da classe lida, definindo-o como privado e estático. 9. Itera sobre o(s) construtor(es) da classe lida definindo-os agora com modificador privado. 10.Faz a criação do método chamado “getInstance” onde ele será responsável por gerenciar a instância única da classe evitando assim que seja realizado novas instanciações.

Fonte: Autoria própria

Ao final da aplicação da refatoração o agente informa ao desenvolvedor o que foi alterado, contendo informações tais como: a classe que foi refatorada e os seus respectivos atributos de qualidade (manutenibilidade, confiabilidade e reusabilidade). O usuário não tem nenhuma interação no processo de refatoração, o agente decide quais padrões de projeto aplicar com base nos métodos encontrados na literatura.

Portanto, o agente proposto: i) tem conhecimento sobre os métodos de detecção e inserção de padrões de projeto; ii) define como analisar certos aspectos do código-fonte e verifica se a classe pode ser candidata a refatoração; iii) autonomia para aplicação dos padrões de projeto para avaliação do código-fonte; iv) calcular as métricas de qualidade (manutenibilidade, reusabilidade e confiabilidade) por meio de atributos de qualidade.

5.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este Capítulo apresentou uma abordagem para detecção de pontos de inserção e aplicação de padrões de projeto em código-fonte visando cobrir aspectos de melhoramento em relação aos trabalhos identificados no mapeamento sistemático.

A abordagem Codice-Unio difere das outras existentes na literatura pois: i) integra métodos de inserção e detecção de padrões de projeto em uma arquitetura BDI; ii) criação das crenças, desejos e planos para refatorar usando padrões de projeto; iii) cálculo das métricas dos atributos de qualidade de código-fonte: manutenibilidade, reusabilidade e confiabilidade ante e depois da refatoração. Os experimentos realizados com a Codice-Unio são abordados no próximo Capítulo.

6 RESULTADOS

Durante a execução do mapeamento sistemático dos trabalhos relacionados, os experimentos com ferramentas de refatoração são realizados com projetos de terceiros obtidos pelo *GitHub*, código de acadêmicos para testar os *bad smells* ou testes com exemplos desenvolvidos pelos próprios autores. A aplicação da Codice-Unio foi realizada utilizando códigos com maior popularidade no *GitHub* e os resultados são descritos neste Capítulo.

A Seção 6.1 relata as tecnologias usadas na implementação da Codice-Unio. A Seção 6.2 relata os resultados atingidos pela abordagem. Na Seção 6.3, é descrita uma extensão para a abordagem e finalmente é apresentado na Seção 6.4 as considerações finais deste Capítulo.

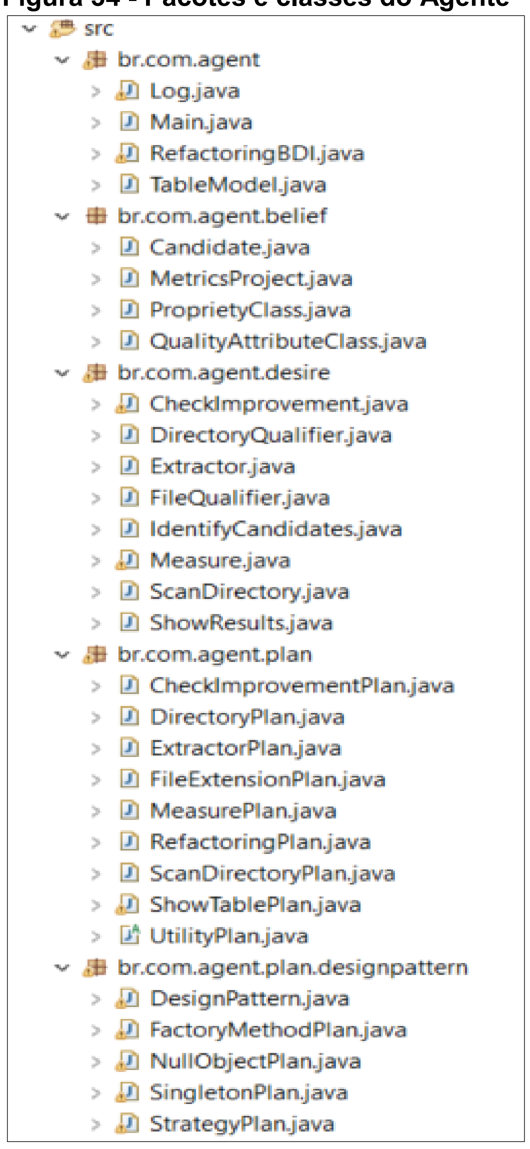
6.1 IMPLEMENTAÇÃO DA CODICE-UNIO

A Codice-Unio foi desenvolvida na IDE Eclipse (2020-03) usando um projeto Maven (MAVEN, 2020) em sua versão (3.6.3), em relação ao agente foi usado o *framework Jadex* por ser voltado a BDI, para a leitura das classes a biblioteca *JavaParser* (JAVAPARSER, 2019) porque contempla AST (*Abstract Syntax Tree*) e para avaliação das métricas foi usado a biblioteca *ck Metrics* (CK GITHUB, 2018).

O agente é iniciado via classe Java, pois permite que ele seja executado por qualquer outro programa de mesma linguagem de programação ou *plug-in*. Foi escolhida a refatoração para linguagem Java porque tem um domínio no meio acadêmico visto em 11(onze) trabalhos dos 14 (quatorze) que foram apresentados no mapeamento sistemático.

A Figura 34 mostra os pacotes para criação do agente: i) *agent*; ii) *agent.belief* - contém as classes usadas como crenças; iii) *agent.desire* - possui os desejos do agente; iv) *agent.plan* - composta por planos que serão executados pelo agente após seu desejo ser chamado; v) *agente.plan.designpattern* - são os subplanos e contém a análise que permite verificar se existe uma classe candidata a refatoração e tem o conhecimento de como modificá-la.

Figura 34 - Pacotes e classes do Agente



Fonte: Autoria própria

O pacote principal *br.com.agent* contém as classes gerais que são:

- *Log*: responsável por guardar informações gerais da execução do agente.
- *Main*: responsável por executar o agente, neste caso, é uma classe Java, porém poderia ser outro programa ou até um *plug-in*.
- *TableModel*: é usado como base para a criação da tabela final ao usuário com classe e atributos de qualidade que foram encontrados.
- *RefactoringBDI*: é a classe usada como base para o agente. Considerado como o “Interpretador” de ações do agente.

O pacote *br.com.belief* é responsável por guardar as classes que foram instanciadas como crenças na classe do agente a “RefactoringBDI”, as classes deste pacote são:

- *Candidate*: responsável por guardar o conhecimento do agente de todas as classes que foram consideradas candidatas a refatoração. Contém internamente os atributos *ProprietyClass* responsável por saber como esta classe é lida via AST e também uma listagem de padrões de projeto que serão empregados.
- *MetricsProject*: responsável por guardar as informações de métricas de atributos antes e depois da refatoração.
- *ProprìertyClass*: responsável por guardar as informações de leitura da classe via propriedades da AST.
- *QualityAttributeClass*: responsável por guardar as métricas de qualidade lidas da classe (confiabilidade, manutenibilidade e confiabilidade) e também nome da classe e o *path* onde se encontra no sistema.

O pacote *br.com.desire* é responsável por guardar os desejos que o agente quer fazer enquanto esta em execução e contém as seguintes classes:

- *CheckImprovement*: tem o objetivo de fazer o cálculo das métricas de qualidade.
- *DirectoryQualifier*: tem como objetivo verificar se um determinado diretório existe no disco rígido e se o *path* não está errado.
- *Extractor*: tem como objetivo extrair informações da classe lida para posterior verificação de como ela se encontra.
- *FileQualifier*: tem como objetivo verificar se o arquivo lido no diretório é qualificado para refatoração.
- *IdentifyCandidates*: tem como objetivo verificar as classes que são candidatas para uma refatoração.
- *Measure*: tem como objetivo avaliar o código em questão de atributos de qualidade.
- *ScanDirectory*: tem como objetivo listar todos os arquivos pertencentes aquele diretório e possíveis subdiretórios.

- *ShowResults*: tem como objetivo mostrar os resultados da refatoração ao usuário final.

O pacote é *br.com.agent.plan* é responsável por definir as classes que são os planos de execução do agente e neste pacote podem ser vistos:

- *CheckImprovementPlan*: responsável por calcular as métricas de manutenibilidade, confiabilidade e reusabilidade para o código-fonte. Este plano faz uma iteração entre os atributos de qualidade antes e depois da refatoração para fazer o cálculo das métricas. Ele retorna as métricas calculadas para seu desejo *CheckImprovement*.
- *DirectoryPlan*: para cada *path* de diretório ele retorna se o mesmo existe ou não para seu desejo *DirectoryQualifier*.
- *ExtractorPlan*: responsável por extrair informações da classe lida para refatoração e guardar as mesmas dentro do objeto dito como uma crença chamado de *PropriertyClass*, e irá retornar esta crença para o seu desejo chamado *Extractor*.
- *FileExtensionPlan*: responsável por avaliar se o arquivo da classe lida realmente pode ser refatorado, fazendo uma análise se o arquivo da classe lida tem: i) extensão .java; ii) não contém nome do arquivo como “Test” pois foi entendido que normalmente estes arquivos são voltados a testes unitários e não seriam uteis para refatoração; iii) não contém no nome do arquivo “Abstract” e nem “Strategy” pois notou-se que normalmente estes já foram refatorados.
- *MeasurePlan*: responsável por calcular os atributos de qualidade de um determinado diretório que foi passado, sendo que ele analisa a Complexidade Ciclomática, profundidade da árvore de herança, e por fim números de linhas do código. Ele retorna esta listagem de atributos avaliados para seu desejo chamado *Measure*.
- *RefactoringPlan*: responsável por analisar se a classe lida é possível integrar algum padrão de projeto e também já faz a refatoração da mesma caso possa ser implementado. Ele retorna para seu desejo *IdentifyCandidates* o objeto de classe candidata podendo com isto visualizar a classe lida pelo agente e também quais os possíveis padrões de projeto que foram encontrados.

- *ScanDirectoryPlan*: responsável por pegar um determinado path de arquivo e verificar quais todos os possíveis arquivos atrelados a este diretório e possíveis subdiretórios. Ele retorna estas informações para seu desejo chamado ScanDirectory.
- *ShowTablePlan*: responsável por mostrar para o usuário final todas as classes que foram visualizadas pelo agente e também as métricas atreladas ao código-fonte que foi modificado.
- *UtilityPlan*: não é um plano propriamente dito do agente, mas foi criado para servir como classe "pai" de todos os outros planos pois implementa dentro de si a instanciação da classe "Log". Ao usar a classe *UtilityPlan* como pai dos planos do agente, foi possível chamar nestes planos a classe *Log* sem necessidade de instanciá-la, pois já estava previamente instanciada em sua classe pai.

Como adaptação um último pacote foi criado chamado *br.com.agent.plan.designpattern* responsável por conter classes com métodos para analisar se é aplicável e aplicar o método de refatoração Java. As classes deste pacote são:

- *DesignPattern*: serve como classe "pai" as outras classes deste pacote. Esta classe implementa os métodos de salvar conteúdo da classe e listar quais padrões de projeto são conhecidos pelo agente que foram implementados em sua estruturação BDI.
- *FactoryMethodPlan*: serve para analisar se é aplicável ou refatorar com o padrão de projeto *FactoryMethod*.
- *NullObjectPlan*: analisa se é aplicável ou refatora com o padrão de projeto *NullObject*.
- *SingletonPlan*: analisa se é aplicável ou refatora com o padrão de projeto *Singleton*.
- *StrategyPlan*: analisa se é aplicável ou refatora com o padrão de projeto *Strategy*.

O fluxo do desejo "DirectoryQualifier" exibido na Figura 35, tem a sintaxe @Plan que é responsável por dizer que para aquela linha está sendo declarado um

plano específico e dentro desta mesma linha tem a sintaxe “goals = “, que é responsável por informar os desejos atrelados aquele plano.

Figura 35 - Declaração de desejos para o agente

```
44 @Plans({ @Plan(trigger = @Trigger(goals = FileQualifier.class), body = @Body(FileExtensionPlan.class)),
45         @Plan(trigger = @Trigger(goals = DirectoryQualifier.class), body = @Body(DirectoryPlan.class)),
46         @Plan(trigger = @Trigger(goals = ScanDirectory.class), body = @Body(ScanDirectoryPlan.class)),
47         @Plan(trigger = @Trigger(goals = Extractor.class), body = @Body(ExtractorPlan.class)),
48         @Plan(trigger = @Trigger(goals = Measure.class), body = @Body(MeasurePlan.class)),
49         @Plan(trigger = @Trigger(goals = CheckImprovement.class), body = @Body(CheckImprovementPlan.class)),
50         @Plan(trigger = @Trigger(goals = ShowResults.class), body = @Body(ShowTablePlan.class)),
51         @Plan(trigger = @Trigger(goals = IdentifyCandidates.class), body = @Body(RefactoringPlan.class)) })
52 @Description("Refactoring Agent of projects with Design Patterns")
53 @Arguments(@Argument(name = "pathName",
54     description = "path to be analyzed",
55     clazz = String.class, defaultvalue = ""))
56 @Agent
57 public class RefactoringBDI {
```

Fonte: Autoria própria

Dentro da classe de agente é necessário declarar que este chama o desejo para ser usado pelo agente tal como pode ser visto na Figura 36.

Figura 36 - Declaração de desejo para uso na classe de agente *Jadex*

```
96    @AgentBody
97    public void body() {
98        try {
99            this.directoryExists = (Boolean) bdiFeature.dispatchTopLevelGoal(new DirectoryQualifier(pathName)).get();
100           if (!this.directoryExists) {
101               throw new IllegalStateException("Directory not exists: " + pathName);
102           }
```

Fonte: Autoria própria

O próximo passo foi a criação da classe desejo implementada com o *Jadex* (Figura 37). A linha 7 tem a instrução @Goal necessária para dizer ao agente que esta classe é um desejo, em sequência na linha 10 é usado a instrução @GoalParameter para informar ao agente que este campo será usado e passado como parâmetro ao plano, e na linha 13, tem a instrução @GoalResult que é utilizado para informar ao agente que o retorno do plano *DirectoryPlan* será alocado neste campo, chamado na linha 14 de “exists”. Neste caso, retorna-se um booleano da execução do plano *DirectoryPlan* para o campo e este ao final será retornado para o agente.

Figura 37 - Desejo DirectoryQualifier do Agente

```
 7 @Goal
 8 public class DirectoryQualifier {
 9
10     @GoalParameter
11     public String pathName;
12
13     @GoalResult
14     public boolean exists;
15
16     public DirectoryQualifier(String pathName) {
17         this.pathName = pathName;
18     }
19
20
21
22 }
23
```

Fonte: Autoria própria

A análise da classe de plano é visualizada na Figura 38, onde a linha 10 tem a declaração @Plan que serve para dizer ao agente que esta classe é um plano, na linha 19 @PlanBody é responsável por dizer ao agente que ao entrar neste plano execute o respectivo método, posteriormente na linha 24 @PlanPassed é executado caso a execução do método em @PlanBody seja bem sucedida e por último tem a linha 29 @PlanAborted que é executada caso o método @PlanBody tenha uma execução mal sucedida.

Figura 38 - Plano DirectoryPlan do Agente

```
10 @Plan
11 public class DirectoryPlan extends UtilityPlan{
12
13     private String pathName;
14
15     public DirectoryPlan(String pathName) {
16         this.pathName = pathName;
17     }
18
19     @PlanBody
20     public boolean isFileJava() {
21         return new File(this.pathName).exists();
22     }
23
24     @PlanPassed
25     public void passed() {
26         super.logSystem.saveContent("Plan finished successfully DirectoryPlan");
27     }
28
29     @PlanAborted
30     public void aborted() {
31         super.logSystem.saveContent("Severe - Plan aborted DirectoryPlan");
32     }
33
34 }
```

Fonte: Autoria própria

Após a execução do plano *DirectoryPlan,* o mesmo retorna uma condição booleana verificando se o diretório que foi analisado existe ou não para o desejo

DirectoryQualifier alocado no seu parâmetro *exists* da classe e por fim retorna para a classe do agente para ser alocado a sua crença chamada de nome “directoryExists” que pode ser visto na linha 66 da Figura 39.

Figura 39 - Declaração de Crença - directoryExists - Agente

```
65⊖     @Belief
66      public boolean directoryExists;
```

Fonte: Autoria própria

A Figura 40 apresenta a implementação do método responsável por analisar se é aplicável um padrão de projeto em uma determinada classe lida. A implementação é composta da verificação com várias condicionais usados com a leitura de código-fonte extraído via *JavaParser.* Os condicionais foram elaborados de acordo com o método da literatura para ter como base o que deveria ser dito como classe candidata e o que não deveria ser.

Figura 40 - Implementação método de análise Singleton

```
27⊖     public boolean isApplicable(ProprietyClass object) {
28          if (object.getType().getExtendedTypes().size() > 0) {
29              return false;
30          } else if (object.getType().isInterface()
31                  || object.getType().isAbstract()) {
32              return false;
33          }
34
35          List<Statement> constructors = new ArrayList<>();
36          try {
37              List<?> membros = object.getType().getMembers().stream()
38                      .filter(linha -> linha instanceof ConstructorDeclaration).collect(Collectors.toList());
39              if (!membros.isEmpty()) {
40                  for (Object method : membros) {
41                      if (method instanceof ConstructorDeclaration) {
42                          ConstructorDeclaration constructor = (ConstructorDeclaration) method;
43                          if(constructor.isPrivate()) {
44                              return false;
45                          }
46                          if(constructor.isAbstract() || !object.getType().getMethodsByName("getInstance").isEmpty()) {
47                              continue;//se ja houver método getInstance ou o construtor é abstrato
48                          }
49
50                          if (constructor.getParameters().size() > 0) {
51                              return false;
52                          } else {
53                              constructors.add(new ConstructorDeclaration().getBody());
54                          }
55                      }
56                  }
57              }
58              return true;
59          } catch (BDIFailureException ex) {
60              throw new IllegalStateException("Error caused by: " + ex.getMessage());
61          }
62      }
```

Fonte: Autoria própria

Na Figura 41 apresenta o método para implementar o padrão de projeto *Singleton*. O mesmo é simples em que se usou a sequência de passos do método

encontrado na literatura. Este padrão faz a modificação da classe lida e cria todos os possíveis atributos que foram necessários para a classe fazendo a escrita em tempo de execução.

Figura 41 - Método java para aplicar Padrão de Projeto Singleton

```
64    public void applyMethod(ProprietyClass object) {
65        try {
66            object.getType().setBlockComment("Classe modified for have Design Pattern Singleton");
67
68            VariableDeclarator variableSingleton = new VariableDeclarator();
69            variableSingleton.setName("singleton");
70            variableSingleton.setType(object.getType().getNameAsString());
71
72            FieldDeclaration fieldSingleton = new FieldDeclaration();
73            fieldSingleton.addVariable(variableSingleton);
74            fieldSingleton.setPrivate(true);
75            fieldSingleton.setStatic(true);
76            object.getType().getMembers().add(fieldSingleton);
77
78            List<?> membros = object.getType().getMembers().stream()
79                    .filter(linha -> linha instanceof ConstructorDeclaration).collect(Collectors.toList());
80            if (!membros.isEmpty()) {
81                for (Object method : membros) {
82                    ((ConstructorDeclaration) method).setModifiers(Modifier.PRIVATE.toEnumSet());
83                }
84            } else {
85                object.getType().addConstructor(Modifier.PRIVATE);
86            }
87
88            BlockStmt block = new BlockStmt();
89            block.addStatement("if(singleton == null){singleton = new " + object.getType().getNameAsString() + "();}")
90                    .addStatement("return singleton;");
91
92
93            MethodDeclaration method = new MethodDeclaration(EnumSet.of(Modifier.PUBLIC, Modifier.STATIC), object.getTypeClass(), "getInstance");
94            method.setBody(block).setBlockComment("Method Singleton for return one instance unique");
95            object.getType().addMember(method);
96            saveContent(object.getAbsolutePath(), object.getCu().toString());
97            super.setApplied(true);
98        } catch (Exception ex) {
99            super.setApplied(false);
100           throw new IllegalStateException("Error caused by: " + ex.getMessage());
101       }
102   }
103
```

Fonte: Autoria própria

A Figura 42 apresenta a indicação de como a abordagem informa os valores das métricas em relação as atributos de qualidade para cada classe do projeto após a aplicação dos padrões de projeto e também exibe uma média por atributo de qualidade para ver como foi o processo de refatoração do projeto como um todo.

Figura 42 - Apresentação de resultados do agente após o processo de refatoração

Refatorador de código-fonte

Reusabilidade: 52.3 - Manutenção:38.25 - Legibilidade: 56.89

Classe	Arquivo	Reusabilidade	Manutenção	Legibilidade
io.opencensus.trace.MessageEventTest	C:\codigo-fonte\progra...	38	27	40
io.opencensus.metrics.export.Value	C:\codigo-fonte\progra...	111	81	119
io.opencensus.metrics.DerivedDoubleGa...	C:\codigo-fonte\progra...	39	28	42
io.opencensus.common.NonThrowingClo...	C:\codigo-fonte\progra...	20	13	20
io.opencensus.trace.MessageEvent	C:\codigo-fonte\progra...	69	50	74
io.opencensus.impl.metrics.MetricsTest	C:\codigo-fonte\progra...	18	13	19
io.opencensus.implcore.trace.internal.Con	C:\codigo-fonte\progra...	76	56	83
io.opencensus.implcore.tags.TagContext...	C:\codigo-fonte\progra...	4	6	9
io.opencensus.impl.internal.DisruptorEven	C:\codigo-fonte\progra...	46	35	52
io.opencensus.metrics.LabelValue	C:\codigo-fonte\progra...	26	18	27
io.opencensus.trace.config.TraceConfigTe	C:\codigo-fonte\progra...	23	16	23
io.opencensus.common.TimeUtilsTest	C:\codigo-fonte\progra...	26	18	27
io.opencensus.implcore.metrics.export.Me	C:\codigo-fonte\progra...	50	36	54
io.opencensus.stats.MeasureMap	C:\codigo-fonte\progra...	51	36	53

Fonte: Autoria própria

6.2 AVALIAÇÃO DA ABORDAGEM PROPOSTA

Além dos cenários contemplados dos artigos de Gaitani *et al.* (2015), Wei *et al.* (2014) foram selecionados mais projetos para avaliar a abordagem e escolheu-se os projetos do *GitHub* (GITHUB, 2019), que são usados por empresas como IBM, *Paypal*, entre outros, os quais apresentam uma quantidade maior de classes. A seleção de quais seriam utilizados foi realizado de forma aleatória.

Os testes foram separados em 2 (duas) etapas: análise do projeto antes do agente aplicar os padrões de projeto e outra depois da refatoração realizada pelo agente. Foram usados no experimento 60 (sessenta) projetos e em ambos os testes foram avaliados avaliado os atributos de qualidade: manutenibilidade, reusabilidade e confiabilidade. A Tabela 2 mostra a relação de projetos que foram utilizados para os testes da Codice-Unio, esses foram retirados do GitHub com quantidade de classes diferentes, variando de 3 (três) a 694 (seiscentos e noventa e quatro). O quadro apresenta os projetos de P1 a P10 (demais estão no Apêndice B) e foram testados no total 8.206 (oito mil e duzentos e seis) classes.

Tabela 2 - Projetos realizados teste a refatoração - Github

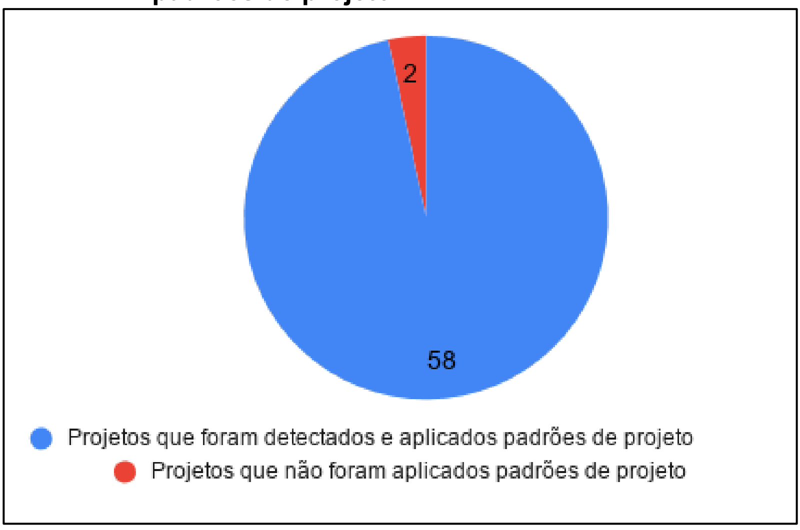
Código	**Nome do Projeto**	**Fonte**	**Qtde. de Classes**
P1	A whole bunch	https://github.com/Glank/Java-Games	74
P2	Change Metrics	https://github.com/mauricioaniche/change-metrics	6
P3	SpringLint	https://github.com/mauricioaniche/springlint	115
P4	RepoDriller	https://github.com/mauricioaniche/repodriller	92
P5	Smelly Repos	https://github.com/mauricioaniche/smellyrepos	15
P6	Logging Analyzer	https://github.com/mauricioaniche/logging-analyzer	28
P7	Prop-Comparator	https://github.com/mauricioaniche/prop-comparator	3
P8	techjobs mvc	https://github.com/LaunchCodeEducation/techjobs-mvc	6
P9	cheese-mvc-persistent	https://github.com/LaunchCodeEducation/cheese-mvc-persistent	6
P10	Java Swing MVC	https://github.com/LionByol/Java-Swing-MVC	37
Total			**382**

Fonte: Autoria própria

Dos 60 (sessenta) projetos utilizados para o experimento, a Codice-Unio detectou e aplicou padrões de projeto em 58 (cinquenta e oito) deles, sendo que

ficaram apenas 2 (dois) sem nenhum padrão de projeto correspondente, conforme apresentado no Gráfico 2.

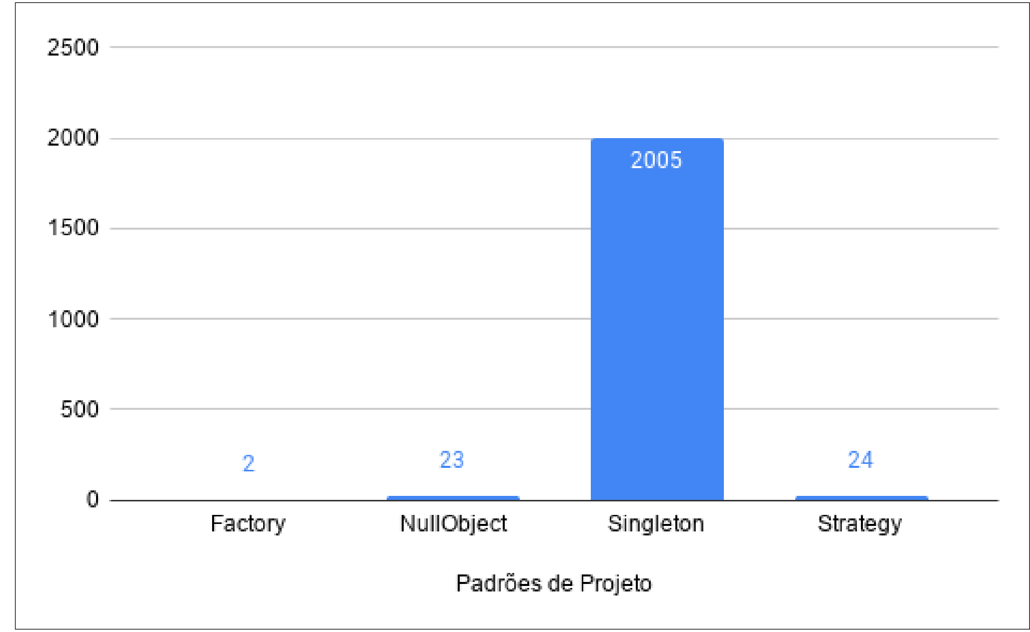
Gráfico 2 - Quantidade de projetos em que se detectou e aplicou padrões de projeto



Fonte: Autoria própria

Os projetos que foram detectados e aplicados padrões de projetos pela Codice-Unio tinham de 3 (três) a 694 (seiscentos e noventa e quatro) classes. O Gráfico 3 apresenta a quantidade de classes candidatas para os projetos com mais refatorações.

Gráfico 3 - Quantidade de Classes candidatas encontradas para refatoração

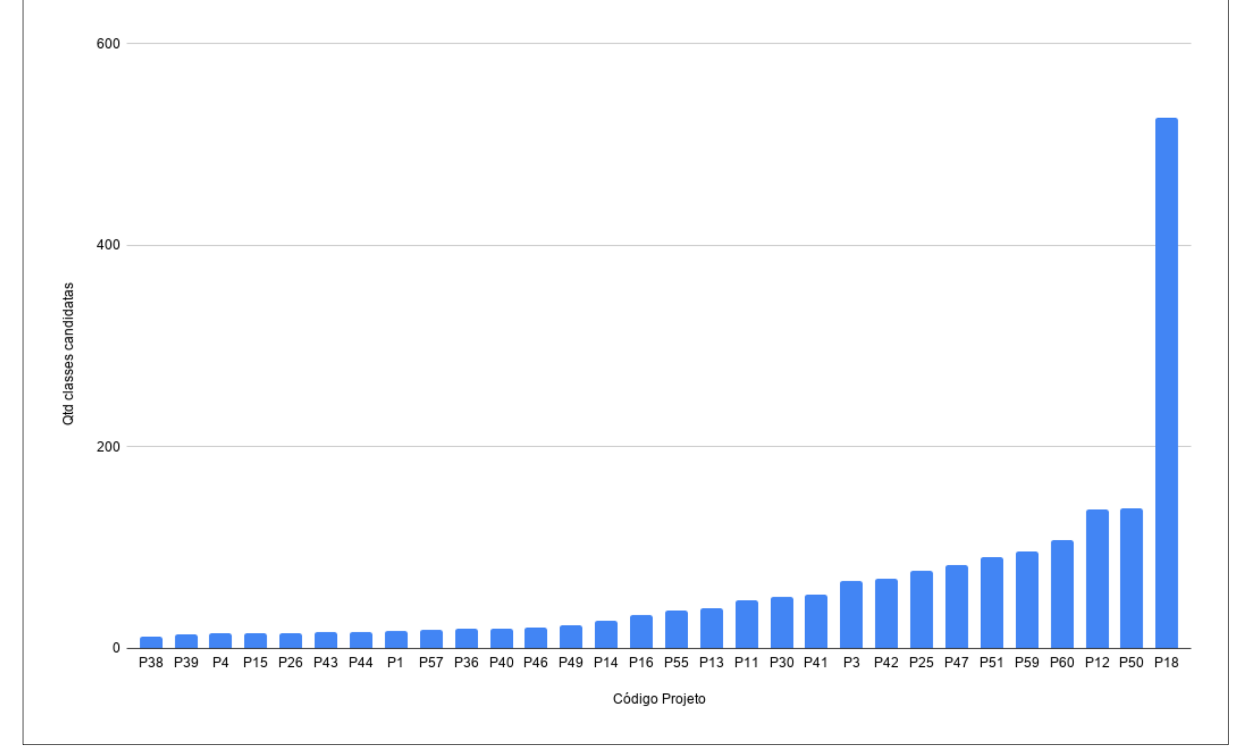


Fonte: Autoria própria

O padrão de projeto mais usado foi o *Singleton* em 97,6% das classes candidatas, totalizando sua aplicação em 2005 (duas mil e cinco) classes. O de menor porcentagem foi o *Factory Method* pois ele só foi encontrado para 2 (duas) classes candidatas. Já os padrões de projeto *NullObject* e *Strategy* ficaram parecidos em porcentagem sendo 1,1% (23 classes) e 1,2% (24 classes), respectivamente.

O Gráfico 4 ilustra a relação dos padrões de projeto que foram aplicados em alguns dos projetos do experimento. O projeto que teve menos classes candidatas foi o P17 (nenhuma classe candidata) e a com maior número de classes candidatas foi o P18 (com 527 classes candidatas das 694 classes originais).

Gráfico 4 - Relação de Padrões de Projeto Aplicados X Qtd. Total de Classes

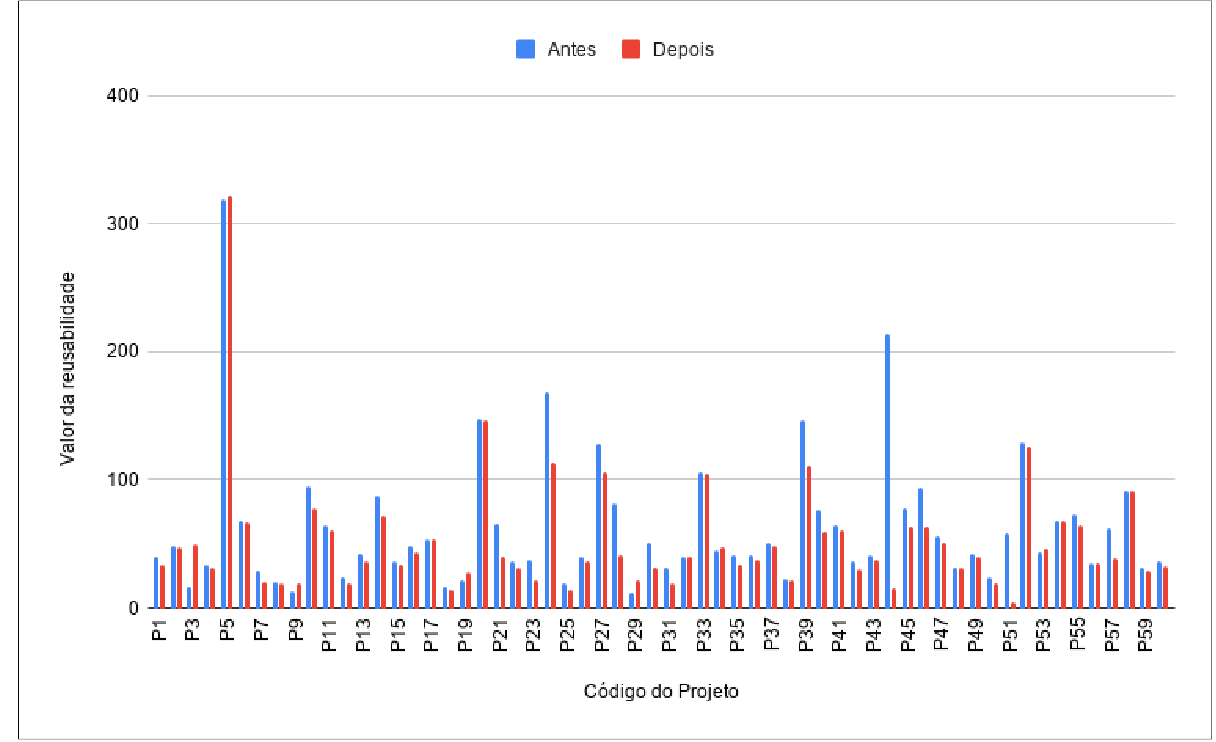


Fonte: Autoria própria

É importante considerar que a implementação do agente é voltada à detectar possíveis padrões de projeto a serem integrados a uma classe. Observou-se que a análise para identificar se o padrão de projeto é viável ou não, evita a aplicação do padrão em uma classe que já o contenha.

O Gráfico 5 exibe a relação entre o antes e depois da aplicação da refatoração para métrica de reusabilidade. Em 50 (cinquenta) projetos o valor da reusabilidade diminuiu em relação ao projeto original e para os 10 (dez) restantes houve um aumento da reusabilidade. A reusabilidade é calculada pela média dos valores das métricas: profundidade na árvore de herança (DIT) e Linhas de Código (LOC). Se os valores forem altos para DIT não se sabe todos os comportamentos da classe e a análise se torna complexa. Assim como, se LOC for alto a classe é mais complexa para ser analisada (CHIDAMBER; KEMERER, 1994; FILO, 2003). Portanto, quanto menor for o valor do atributo reusabilidade melhor é o código-fonte. Considerando o processo de refatoração da Codice-Unio houve uma melhora na reusabilidade em 83,33% dos projetos refatorados.

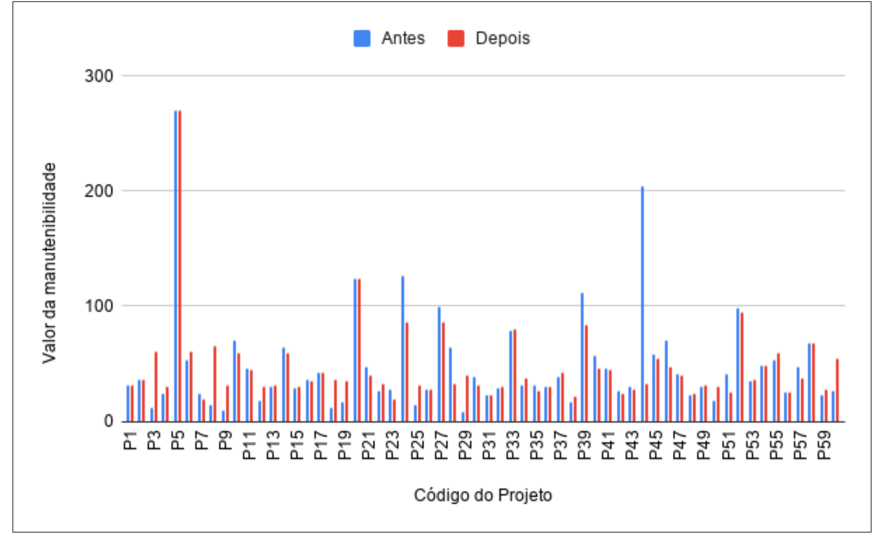
Gráfico 5 - Relação antes e depois da refatoração para atributo Reusabilidade



Fonte: Autoria própria

No Gráfico 6 mostra a relação entre o antes e depois para os atributos de manutenibilidade, em que em 27 (vinte e sete) projetos houve uma diminuição da manutenibilidade e em 33 (trinta e três) houve aumento. A manutenibilidade é calculada pela média dos valores das métricas: profundidade na árvore de herança (DIT); Linhas de Código (LOC) e Complexidade Ciclomática (CC). A CC mede a quantidade de caminhos de execução independentes a partir de um código-fonte (MCCABE, 1976), portanto quanto menos caminho houverem, mais fácil de ser testado (FILO, 2003). Portanto, se os valores de DIT, LOC e CC forem baixo, melhor é a manutenibilicade do código-fonte. Considerando o processo de refatoração da Codice-Unio houve uma piora no valor da manutebilidade em 55% dos projetos refatorados.

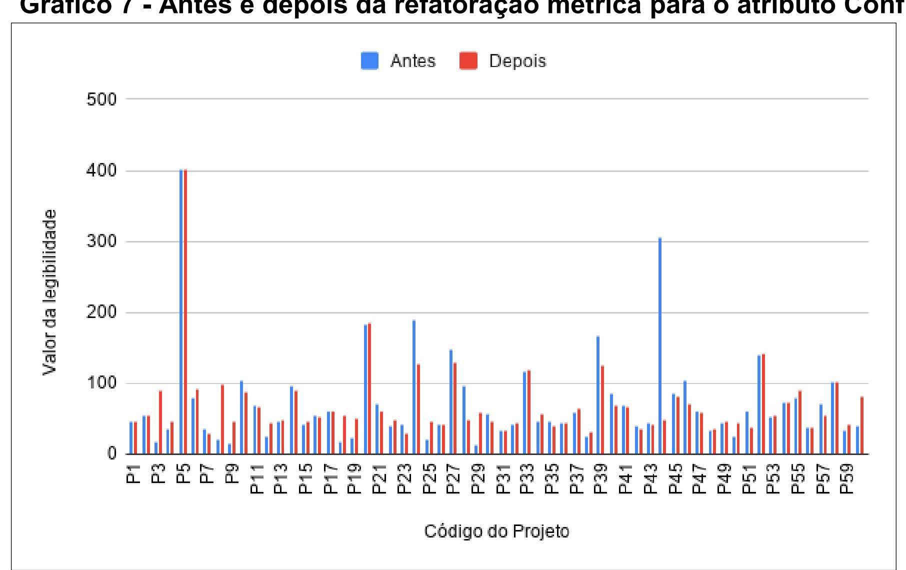
Gráfico 6 - Antes e depois da refatoração para o atributo Manutenibilidade



Fonte: Autoria própria

No Gráfico 7 é apresentado o antes e depois da refatoração por cada projeto para análise da métrica de qualidade para confiabilidade, em que nos 34 (trinta e quatro) projetos houve aumento do valor deste atributo e em 26 (vinte e seis) houve uma diminuição. A confiabilidade é calculada pela média dos valores das métricas: Linhas de Código (LOC) e Complexidade Ciclomática (CC). Portanto, se os valores de LOC e CC forem baixo, melhor é a confiabilidade do código-fonte. Considerando o processo de refatoração da Codice-Unio houve uma piora no valor da confiabilidade em 56,66% dos projetos refatorados.

Gráfico 7 - Antes e depois da refatoração métrica para o atributo Confiabilidade



Fonte: Autoria própria

A Tabela 3 apresenta os resultados obtidos como média das métricas dos atributos de qualidade, para antes e depois da refatoração. Apesar da manutenibilidade e da confiabilidade atingirem um aumento por projeto, (conforme ilustrado nos Gráficos 6 e 7), esses apresentaram um valor inferior aos demais, o que ocasionou uma queda na média geral. A análise mais criteriosa sobre a os benefícios da variação dos projetos não foi realizada, uma vez que foge ao escopo do presente trabalho, sendo esse um tema a ser realizado em estudos futuros.

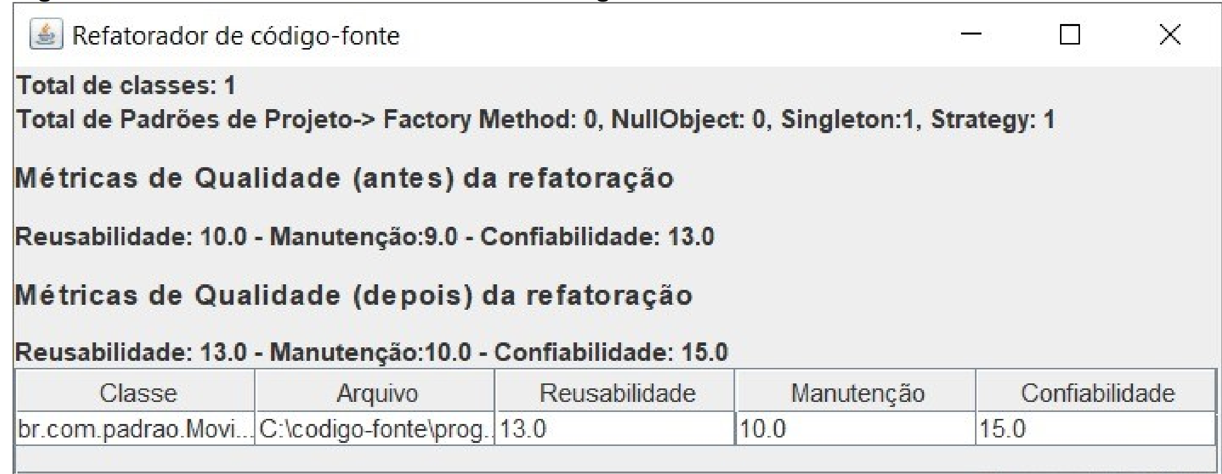
Tabela 3 - Relação antes x depois de aplicar refatoração para métricas de qualidade

	Manutenibilidade	**Reusabilidade**	**Confiabilidade**
Antes	2924,02	3767,05	4345,11
Depois	2839,31	3115,13	4222,88

Fonte: Autoria própria

Na Figura 43 pode ser visto como é apresentado ao usuário final o aviso d
o agente das classes que foram refatoradas e o antes e depois em relação as métricas de qualidade para o projeto.

Figura 43 - Aviso de classes modificadas do agente ao usuário final



Fonte: Autoria própria

A Figura 44 apresenta o aviso fornecido pelo sistema operacional quando o agente modifica alguma classe.

Figura 44 - Aviso do sistema operacional de arquivos modificados

```
package br.com.padrao;

public class MovieTicket {

    private double price;

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public double calculate(char type) {
        if(type == 'S'){
            return price * 0.8;
        }else if(type == 'C'){
            return price -10;
        }else if(type == 'M'){
            return price * 0.5;
        }
        return -1;
    }

}
```

Recarregar

"C:\codigo-fonte\programa-java\exemplo-cavado-Liu-strategy\src\br\com\padrao\MovieTicket.java"

Este arquivo foi modificado por outro programa.
Deseja recarregá-lo?

Sim | Não

Fonte: Autoria própria

Com a realização dos experimentos foi possível constatar que o processo de refatoração baseado em padrões de projeto pode ser realizado de forma autônoma usando agente. Os projetos que já haviam sido refatorados pela Codice-Unio foram submetidos a uma nova avaliação e o agente não realizou nenhuma mudança nas classes, ou seja, foi capaz de verificar que o código estava refatorado.

Foi realizado também uma avaliação em termos de *code smells* para 31 (trinta e um) dos projetos que foram refatorados pela Codice-Unio a fim de verificar se o agente não incorpora mais *code smells* ao código-fonte do projeto refatorado. A verificação foi realizada usando o SonarCube (SONARQUBE, 2020). A Tabela 4 ilustra os resultados obtidos.

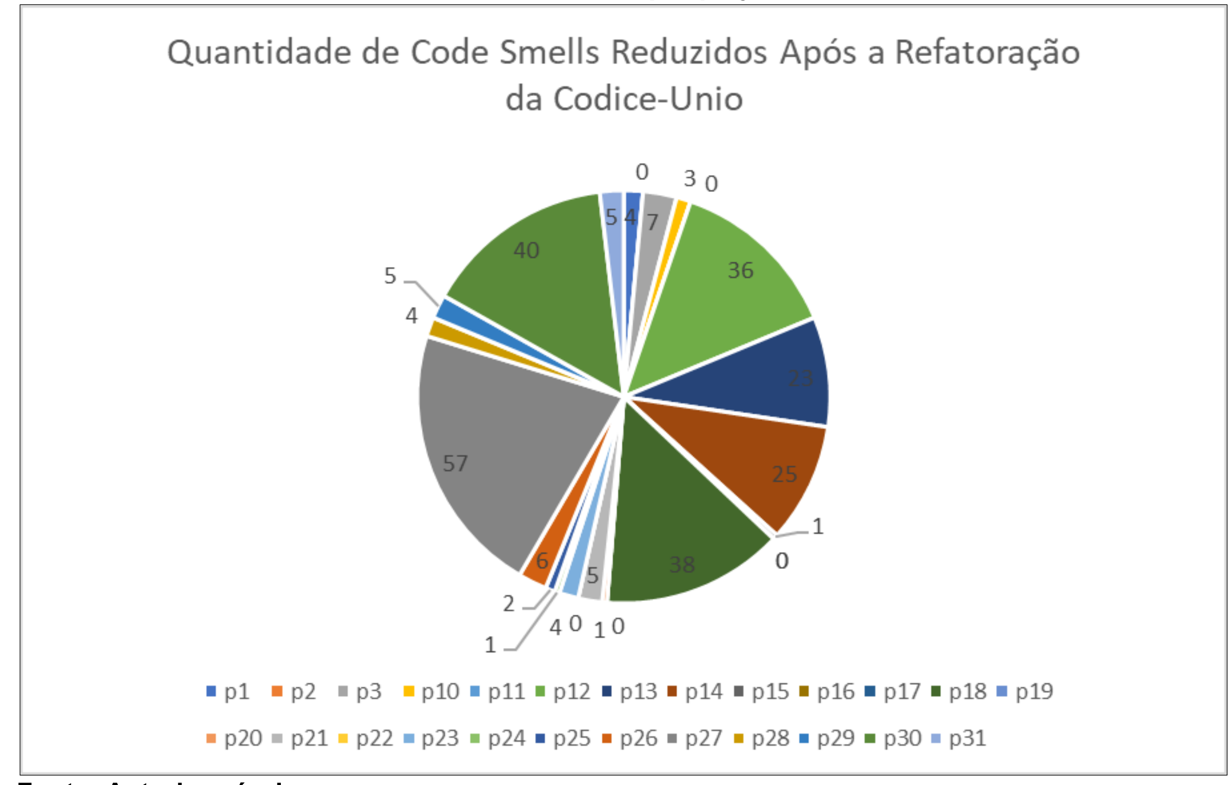
Tabela 4 - Quantidade de Code Smells - Antes e Depois do Agente

Projeto	***Code Smells* - Antes**	***Code Smells* - Depois**
p1	639	635
p2	12	12
p3	241	234
p10	241	238
p11	2216	2216
p12	803	767
p13	431	408
p14	333	308
p15	278	277
p16	526	526
p17	27	27
p18	1933	1895
p19	47	47
p20	541	540
p21	179	174
p22	86	86
p23	96	92
p24	90	89
p25	1119	1117
p26	186	180
p27	878	821
p28	624	620
p29	16	11
p30	507	467
p31	171	166

Fonte: Autoria própria

Observou-se que em 6 (seis) projetos (p2, p 11, p16, p17, p19, p22) a quantidade de *code smells* permaneceu a mesma e para 25 (vinte e cinco) houve a redução da quantidade de *code smells*. Os valores que foram reduzidos por projetos estão melhores representados no Gráfico 8.

Gráfico 8 - Quantidade de Code Smells reduzidos por projeto com o uso da Codice-Unio



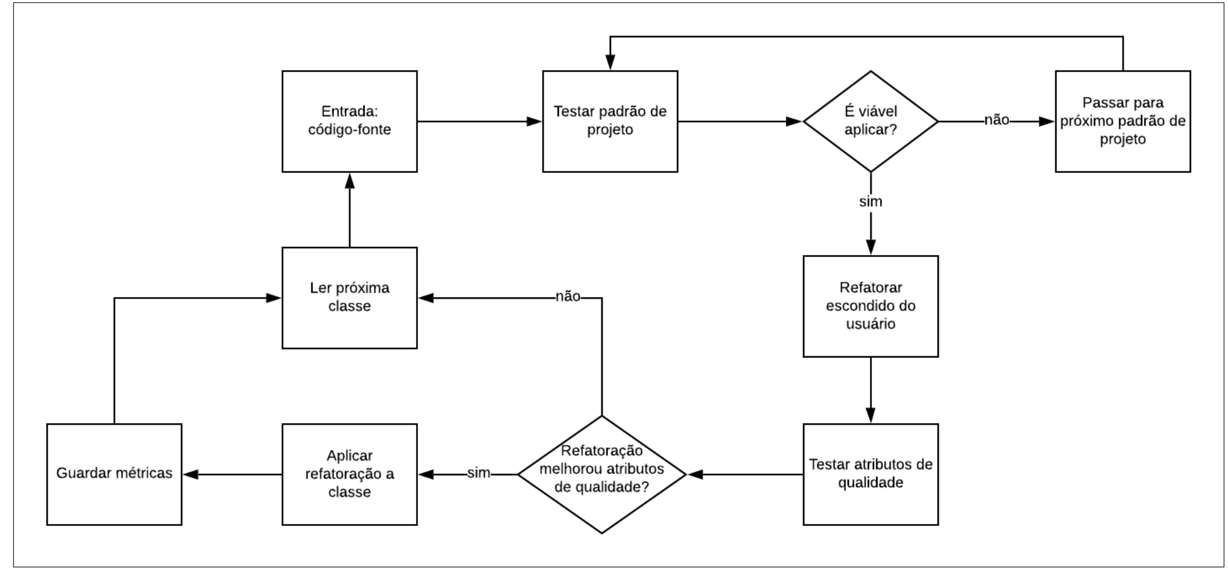
Fonte: Autoria própria

Após os experimentos, observou-se que uma proposta de extensão poderia ser sugerida e é detalhada na próxima seção.

6.3 PROPOSTA PARA EXTENSÃO DA ABORDAGEM PROPOSTA

O agente proposto é capaz de detectar e aplicar padrões de projeto fundamentado nos métodos da literatura de forma autônoma, porém observou-se pelos experimentos que para se obter melhores resultados seria necessário que houvesse mais um agente capaz de avaliar se a aplicação do padrão realmente proporciona ganhos em termos de atributos de qualidade ao projeto. A Figura 45 apresenta uma proposta de extensão que pode ser desenvolvida em trabalhos futuros para que o agente possa trazer mais ganhos ao processo de refatoração por meio da análise dos atributos de qualidade (manutenibilidade, flexibilidade e confiabilidade).

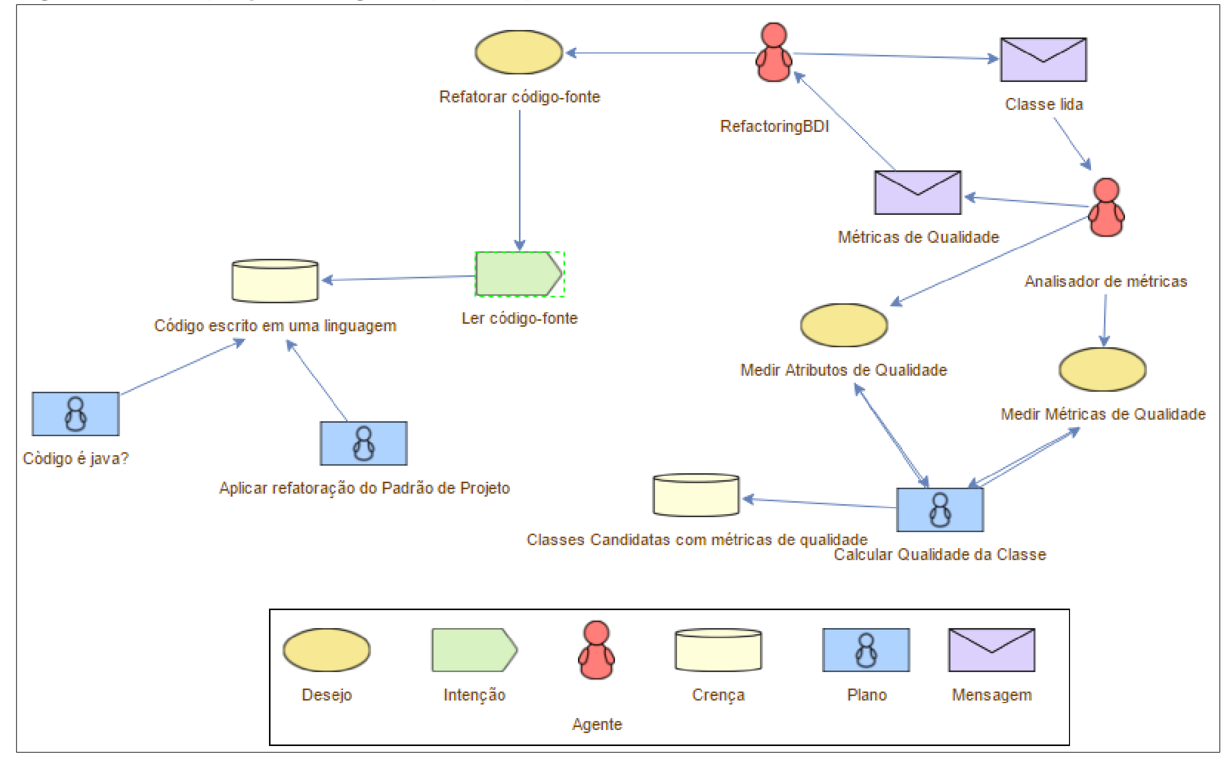
Figura 45 - Fluxo de uma Nova Proposta para o Agente no Processo de Refatoração



Fonte: Autoria própria

A nova proposta apresentada na Figura 46 contém 2 (dois) agentes: o primeiro responsável somente por refatorar o código-fonte da classe e o segundo por analisar as métricas. A ideia principal é que o agente que contém a ação de detectar classes candidatas possa solicitar ao agente de métricas a viabilidade ou não da aplicação do padrão. A viabilidade verifica se o padrão para a classe pode trazer ganhos em relação aos atributos de qualidade. Se sim, o agente refatorador efetiva a refatoração do padrão de projeto detectado na classe analisada e atualiza o projeto do desenvolvedor com as modificações. Caso contrário, o desenvolvedor continua com seu código-fonte original.

Figura 46 - Adaptação ao agente para separar métricas



Fonte: Autoria própria

Outra sugestão em relação ao agente proposto é que novas métricas e novos métodos de detecção e aplicação de padrões de projeto possam ser inseridos a fim de torná-lo mais completo em relação quantidade de padrões que podem ser encontrados e quantidade de atributos de qualidade que podem ser medidos. Essa extensão é possível por meio da criação dos planos dos novos padrões e métricas, mas sua estrutura de funcionamento não será alterada.

6.4 CONSIDERAÇÕES FINAIS DO CAPITULO

Este capítulo apresentou a implementação da abordagem Codice-Unio e sua avaliação, implementada utilizando a IDE de programação Eclipse na versão 2020-03 (ECLIPSE IDE, 2020). A programação do agente foi feita no *framework Jadex* na versão 3.0.43 (JADEX, 2020), escolhido por ser voltado a implementação com BDI. A extração dos dados das classes foi realizada por meio da biblioteca *JavaParser* na versão 3.6.0 (JAVAPARSER, 2020) implementada com Maven em sua versão 3.6.3 (MAVEN, 2020) pois fornece suporte a várias propriedades da classe por implementar a AST.

A avaliação da Codice-Unio foi feita com projetos do repositório *GitHub* escolhidos de forma aleatória. Foram analisados 60 (sessenta) projetos, e pode-se concluir que é possível que o agente encontre e aplique os padrões de projeto que foram identificados por meio da implementação dos Gaitani *et al (*2015), Wei *et al* (2014), e Ouni (2017) métodos da literatura.

Constatou-se com os experimentos que para ter uma melhor aplicação de padrões em projetos é necessário que a Codice-Unio seja capaz de refatorar mediante uma análise de métricas de qualidade, garantindo assim que a aplicação trará ganhos em relação aos atributos de qualidade.

7 CONCLUSÃO

Este trabalho criou uma abordagem denominada de Codice-Unio que é capaz de identificar e aplicar padrões de projeto de forma autônoma em códigos-fontes escritos em linguagem Java. Esta abordagem foi concebida depois da execução de um mapeamento sistemático de 1998 a 2020 para analisar o estado da arte sobre processos automáticos de refatoração. Neste mapeamento foi identificado que existem agentes para refatoração usando técnicas tais como *Extract Method*, *Replace Temp with Query*, *Preserve Whole Object*, entre outras, mas não para padrões de projeto. Além disto, outras informações sobre os estudos foram identificadas tais como: linguagens de programação mais usadas, abordagem para extrair dados dos projetos, formas de realização dos experimentos, arquiteturas dos agentes de refatoração baseado em técnicas, entre outras.

A Codice-Unio foi modelada usando a arquitetura BDI (*Belief - Desire - Intention*) em que se identificou os desejos, crenças, planos e subplanos para o processo de refatoração fundamentado em padrões de projeto. A refatoração e a avaliação constitui-se a camada de desejo. A crença teve como finalidade guardar informações do projeto tais como: classes candidatas, métricas de projeto, propriedade da classe. Os planos necessários para realizar efetivamente a refatoração são: Código é Java?, Aplicar refatoração, Verificar métricas de qualidade. Para execução destes planos se estabeleceu um conjunto de passos, sendo que para alguns deles foram criados subpassos. O subpasso ficou responsável por todos os passos para aplicação dos padrões de projeto *Singleton* (Quadro 24), *FactoryMethodPlan*, *NullObjectPlan*, *SingletonPlan* e *StrategyPlan* (Apêndice A). Estes padrões são contemplados nos métodos de Gaitani *et al*. (2015), Wei *et al*. (2014) e Ouni (2017) e que foram implementados na Codice-Unio porque apresentam os algoritmos de execução e o funcionamento do processo de refatoração.

A arquitetura BDI permitiu separar e estruturar o processo em camadas, facilitando a sua implementação no framework *Jadex* (PIUNTI, 2008). O framework permitiu a implementação do modelo sem necessidades de adaptações. O ambiente de desenvolvimento para o Codice-Unio foi a Eclipse IDE Compiler (ECLIPSE IDE, 2019).

Além de realizar o processo de refatoração, foi incorporado na Codice-Unio a avaliação do código-fonte considerando os atributos de manutenibilidade, reusabilidade e confiabilidade, os quais são os mais importantes de serem medidos em um processo de refatoração. Optou-se por incorporar a avaliação para que o usuário recebesse informações sobre como seu código ficou em termos de atributos de qualidade.

A avaliação da Codice-Unio foi realizada em duas etapas, sendo a primeira para verificar se era capaz de refatoração aplicando os padrões de forma automática e para isto usou os exemplos que estavam contemplados nos métodos de Gaitani *et al.* (2015), Wei *et al.* (2014) e Ouni (2017). Em outra etapa, utilizou-se 60 (sessenta) projetos do GitHub em que foram avaliadas um total de 8.206 (oito mil duzentos e seis)classes, selecionados de forma aleatória.

Com a relação dos experimentos notou-se que a Codice-Unio é capaz de detectar e aplicar os padrões de projeto de forma autônoma, sem a intervenção do usuário. Considerando a aplicação dos padrões o mais utilizado foi o *Singleton*, pois é o mais simples de ser encontrado, seguido pelo *NullObject* e *Strategy*. Já para o padrão *Factory Method* a aplicação foi mais difícil de ser encontrada. Isto se deve ao fato do método de refatoração implementado ter muitas restrições para sua aplicabilidade.

Em relação aos atributos de qualidade notou-se uma melhora nos projetos em relação a reusabilidade, porém perdeu-se em manutenibilidade e confiabilidade porque os valores das métricas (DIT, LOC, CC) que são usados para calcular estes atributos aumentaram. Uma proposta de melhoria para a Codice-Unio foi apresentada a fim de que o processo de avaliação do código ocorra antes da aplicação do padrão.

7.1 TRABALHOS FUTUROS

Como sugestão para trabalhos futuros têm-se:

- Incorporar ao agente de refatoração mais métodos de refatoração voltados a padrões de projeto
- Testar e integrar mais abordagens de extração de dados para o agente, pois a abordagem atual contempla apenas uma abordagem da extração da literatura que é a AST.

- Integrar mais métricas capazes de avaliar outros atributos de qualidade tal como flexibilidade.
- Criar um novo agente capaz de verificar se a aplicação efetiva do padrão pode trazer ganhos relacionados a atributos de qualidade para o projeto.
- Avaliar o desempenho da Codice-Unio na realização do processo de refatoração.

REFERÊNCIAS

ABEBE, M; YOO, C. Trends, opportunities and challenges of software refactoring: A systematic literature review. **International Journal of Software Engineering and Its Applications**, v. 8, n. 6, p. 299-318, 2014.

ACTIVE COMPONENTS. Disponível em < https://download.actoron.com/docs/releases/latest/jadex-mkdocs/tutorials/bdiv3/05%20Using%20Goals/>. Acessado em: 05 Agosto de 2019.

AJOULI, A. **Vues et Transformations de Programmes pour la Modularité des Évolutions**. Tese de Doutorado. École des mines de Nantes. 2013.

AL DALLAL, J. Identifying refactoring opportunities in object-oriented code: A systematic literature review. **Information and software Technology**, v. 58, p. 231-249, 2015.

ALVES, N SR et al. Identification and management of technical debt: A systematic mapping study. **Information and Software Technology**, v. 70, p. 100-121, 2016.

AMBIEL, L. Tópicos de Engenharia de Software Orientada a Agentes. 2010.

ASTPARSER. Disponível em: < https://help.eclipse.org/kepler/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html> Acessado em: 18 de junho de 2019.

BASTOS, R. **O planejamento de alocação de recursos baseado em sistemas multiagentes**. 1998. 267 f. Monografia (Doutorado em Ciência da Computação) - Universidade Federal do Rio Grande do Sul, Porto Alegre, 1998.

BCEL. Disponível em: <https://commons.apache.org/proper/commons-bcel/> Acessado em: 18 de junho de 2019.

BELLIFEMINE, F; POGGI, A; RIMASSA, G. JADE-A FIPA-compliant agent framework. In: **Proceedings of PAAM**. 1999. p. 33.

BELUZZO, L. B**. Abordagem para avaliar e detectar pontos de inserção e aplicação de padrões de projeto em código-fonte.** 2018. 100 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2018

BELUZZO, L.B; MATOS, S.N; PACHER, T.H. A Refactoring architecture for measuring and identifying spots of design *Patterns* insertion in source code. In: ICSOFT. **Proceedings of the 13th International Conference on Software Technologies**. [S.l.], 2018. p. 632-639.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object-oriented design, **IEEE Transactions on Software Engineering**, v. 20, n° 6, p. 476-493, 1994.

CINNÉIDE, M. Ó.; NIXON, P. A **Methodology for the automated introduction of design Patterns.** 1999. 463-472.

BORDINI, R H.; HÜBNER, J. F. BDI agent programming in AgentSpeak using Jason. In: **International Workshop on Computational Logic in Multi-Agent Systems**. Springer, Berlin, Heidelberg, 2005. p. 143-164.

BORDINI, R H.; HÜBNER, J; WOOLDRIDGE, M. **Programming multi-agent systems in AgentSpeak using Jason**. John Wiley & Sons, 2007.

BRATMAN, M E.; ISRAEL, D J.; POLLACK, M E. Plans and resource-bounded practical reasoning. **Computational intelligence**, v. 4, n. 3, p. 349-355, 1988.

BRAUBACH, L; LAMERSDORF, W; POKAHR, A. Jadex: Implementing a BDI-infrastructure for JADE agents. 2003.

BRAUBACH, L; POKAHR, A; LAMERSDORF, W. Jadex: A short overview. In: **Main Conference Net. ObjectDays**. 2004. p. 195-207.

BROOKS, R. A robust layered control system for a mobile robot. **IEEE journal on robotics and automation**, v. 2, n. 1, p. 14-23, 1986.

CASS, A. Little-JIL/Juliette: a process definition language and interpreter. In: **Proceedings of the 22nd international conference on Software engineering**. ACM, 2000. p. 754-757.

CHIKOFSKY, E. J..; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. **IEEE software**, v. 7, n. 1, p. 13-17, 1990.

CINNÉIDE, M; NIXON, P. A methodology for the automated introduction of design patterns. In: **Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)**. IEEE, 1999. p. 463-472.

CINNÉIDE, M. Ó. **Automated application of design patterns:** a refactoring approach. 2000. 242 f. Thesis (Doutorado), Programa de Pós-Graduação University of Dublin, Trinity College Dublin, 2000.

CK GITHUB. Disponível em < https://github.com/mauricioaniche/ck/>. Acessado em: 21 junho de 2020.

COMPILATIONUNIT. Disponível em: < https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FCompilationUnit.html> Acessado em: 18 de junho de 2019.

COPPIN, B. **Inteligência artificial**. 1ª. ed. Rio de Janeiro: Grupo Editorial Nacional, 2015.

COSER, A**. Utilização de Agentes Inteligentes no Trabalho Colaborativo via Internet.** 1999. 147 f. Dissertação (Mestrado em Engenharia de Produção) - Universidade Federal de Santa Catarina, Ponta Grossa, 1999

DYBA, T.; DINGSOR, T. Empirical studies of agile software development: A systematic review. **Information and software technology**, v. 50, p. 833-859, 2008.

ECLIPSE IDE. Disponível em <http://eclipse.org>. Acessado em: 21 Junho.de 2020.

EVERTSZ, R. Tactics development framework. In: **Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems**. International Foundation for Autonomous Agents and Multiagent Systems, 2014. p. 1639-1640.

EVERTSZ, R. A framework for modelling tactical decision-making in autonomous systems. **Journal of Systems and Software**, v. 110, p. 222-238, 2015.

EVERTSZ, R; THANGARAJAH, J; LY, T. **Practical Modelling of Dynamic Decision Making**. Springer International Publishing, 2019.

FAGUNDES, M. **Integrating BDI model and Bayesian networks**. 2007. 104f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federado do Rio Grande do Sul. Porto Alegre, 2007.

FERREIRA, K; BIGONHA, M; BIGONHA, R. Reestruturação de software dirigida por conectividade para redução de custo de manutenção. **Revista de Informática Teórica e Aplicada**, v. 15, n. 2, p. 155-180, 2008.

FILO, T. G. S. **Identificação de valores referência para métricas de softwares orientados por objetos**. 2014. 197 f. Dissertação — Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2003.

FOWLER, M. **Refactoring:** improving the design of existing code. [S.l.]: Pearson Education India, 1999.

FOWLER, M. **Padrões de Arquitetura de Aplicações Corporativas**. São Paulo: ARTMED, 2006.

FREEMAN, E. *et al.* **Use a Cabeça! Padrões de Projetos (Design *Patterns*)**. 2°. ed. [S.l.]: [s.n.], 2009.

FRONT, Developer Liberation. Stench Blossom. 2014. Disponivel em: < https://github.com/DeveloperLiberationFront/refactoring-tools/wiki/Stench-Blossom>. Acesso em: 06 maio 2019.

FROZZA, R. **SIMULA:** Ambiente para desenvolvimento de sistemas multiagentes reativos. 1997. 117f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal do Rio Grande do Sul. Porto Alegre, 1997.

GAGO, I; WERNECK, V; COSTA, R. Modeling an educational multi-agent system in maSE. In: **International Conference on Active Media Technology**. Springer, Berlin, Heidelberg, 2009. p. 335-346.

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

GAMMA, E. **Design *Patterns*:** elements of reusable object-oriented software. [S.l.]: Pearson Education India, 1995.

GAROUSI, V.; MÄNTYLÄ, M. V. A systematic literature review of literature reviews in software testing. **Information and Software Technology**, v. 80, p. 1915-216, 2016.

GELENBE, E; LENT, R; SAKELLARI, G. **Computer and Information Sciences II**. Springer, 2012.

GITHUB Disponivel em: <https://github.com>. Acesso em: 09 Maio 2019.

GRAF, P. eclipse-pmd. 2013. Disponivel em: < https://marketplace.eclipse.org/content/eclipse-pmd>. Acesso em: 06 maio 2019.

HEUZEROTH, D. et al. Automatic design pattern detection. In: **11th IEEE International Workshop on Program Comprehension, 2003.** IEEE, 2003. p. 94-103.

JAVACC. Disponível em <http://javacc.org>. Acessado em: 06 Agosto.de 2019.

JADE. Perguntas frequentes. **Jade**, 2019. Disponivel em: <https://jade.tilab.com/>. Acesso em: 06 maio 2019.

JADE. Introduction to the RMA. **Jade**, 2019. Disponivel em: < https://jade.tilab.com/documentation/tutorials-guides/jade-rma/introduction-to-the-rma/>. Acesso em: 06 maio 2019.

JADEX. Disponivel em: < https://www.activecomponents.org/index.html#/download>. Acesso em: 21 Junho 2020.

JASON. Disponivel em: <http://jason.sourceforge.net/wp/>. Acesso em: 09 Maio 2019.

JAVAPARSER. Disponivel em: < https://javaparser.org/>. Acesso em: 09 Maio 2019.

JUCHEM, M; BASTOS, Ricardo M. Arquitetura de Agentes. **Thecnical Report Series. Porto Alegre**, n. 013, 2001.

KERIEVSKY, J. **Refatoração para padrões**. [S.l.]: Bookman Editora, 2008.

KOZACZYNSKI, W; NING, J; ENGBERTS, A. Program concept recognition and transformation. **IEEE Transactions on Software Engineering**, v. 18, n. 12, p. 1065-1075, 1992.

KITCHENHAM, B. C., S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. [S.l.], 2007.

LAGUNA, M. A.; CRESPO, Y. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. **Science of Computer Programming**, v. 78, n. 8, p. 1010-1034, 2013.

LAKSHMI, S. A; VADIVU, S. S; RAMACHANDRAN, A. Detecting and Scheduling Badsmells using Java Agent Development (JADE). **International Journal of Computer Applications**, v. 67, n. 10, 2013.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering** . v. 4, n. 1, p.308-320, 1976.

MALL, R. **Fundamentals of software engineering**. PHI Learning Pvt. Ltd., 2018.

MARIANI, T; VERGILIO, S. R. A systematic review on search-based refactoring. **Information and Software Technology**, v. 83, p. 14-34, 2017.

MARTIN, R. C. **Clean code:** a handbook of agile software craftsmanship. [S.l.]: Pearson Education, 2009.

MARTINS, L. E. G.; GORSCHEK, T. Requirements engineering for safety-critical systems: A systematic literature review. **Information and software technology**, v. 75, p. 71-89, 2016.

MAVEN.Disponivel em: < http://maven.apache.org/>. Acesso em: 06 junho 2020.

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.

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

MEYER, B. **Object-oriented software construction**. New York: Prentice hall, 1997.

MOHA, N. E. A. Decor: A *Method* for the specification and detection of *code* and design *smells*. **IEEE Transactions on Software Engineering**, v. 36, n. 1, p. 20-36, 2009.

NETO, B. F. S. *et al.* AutoRefactoring: A platform to build refactoring agents. **Expert Systems with Applications**, v. 42, n. 3, p. 1652-1664, 2015.

NETO, A. B. **Uma arquitetura para agentes inteligentes com personalidade e emoção**. 2010. 127f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de São Paulo. São Paulo. 2010.

NETO, M. L. Sistemas Multi-Agentes Inteligentes & Personalização da Informação. **Engenharia de Sistemas Eletrônicos-Escola Politécnica da USP**, 2003.

NUÑEZ-VARELA, A. S. et al. Source code metrics: A systematic mapping study. **Journal of Systems and Software**, v. 128, p. 164-197, 2017.

OMG. Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM). 2011. Disponivel em: < https://www.omg.org/spec/ASTM/1.0/PDF>. Acesso em: 06 julho 2019.

OUNI, A. *et al.* MORE: A multi-objective refactoring recommendation approach to introducing design *Patterns* and fixing *code smells*. **Journal of Software: Evolution**, v. 29, 2017.

PACHER, T; MATOS, S. **Refatoração Baseada em Padrões de Projeto Usando Agentes**. In: **WPCCG - Workshop de Pesquisas em Computação dos Campos Gerais**. Paraná, 2019.

PADGHAM, L; WINIKOFF, M. Prometheus: A methodology for developing intelligent agents. In: **International Workshop on Agent-Oriented Software Engineering**. Springer, Berlin, Heidelberg, 2002. p. 174-185.

PATE, J. R.; TAIRAS, R; KRAFT, N. A. Clone evolution: a systematic review. **Journal of software: Evolution and Process**, v. 25, n. 3, p. 261-283, 2013.

PDE.Disponivel em: < https://www.eclipse.org/pde/>. Acesso em: 06 maio 2019.

PICARD, G; GLEIZES, M. The ADELFE methodology. In: **Methodologies and Software Engineering for Agent Systems**. Springer, Boston, MA, 2004. p. 157-175.

PIUNTI, M. *et al.* Goal-directed interactions in artifact-based mas: Jadex agents playing in CARTAGO environments. In: **Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology-Volume 02**. IEEE Computer Society, 2008. p. 207-213.

RAM, J. D. RAJESH, J. Detecting Intent Aspects from *Code* to Apply Design Patterns in Refactoring: An Approach Towards a Refactoring Tool. In: **Proceedings of 2nd Workshop of Software Design and Architecture**. [S.l.], v.4, 2004.

RAO, A. S; GEORGEFF, M. P. Bdi agents: From theory to practice. In: **ICMAS**. [S.l.: s.n.], 1995. v. 95, p. 312-319.

RASOOL, G; ARSHAD, Z. A review of code smell mining techniques. **Journal of Software: Evolution and Process**, v. 27, n. 11, p. 867-895, 2015.

ROCHIMAH, S; ARIFIANI, S; INSANITTAQWA, V. F. Non-source code refactoring: A systematic literature review. **International Journal of Software Engineering and Its Applications**, v. 9, n. 6, p. 197-214, 2015.

SANTOS, H. Software rejuvenation via a multi-agent approach. **Journal of Systems and Software**, v. 104, p. 41-59, 2015.

SILVA, C. A**. Modalegam comportamental para agentes autônomos em ambientes reais.** 2015. 139 f. Qualificação de Dissertação (Mestrado em Ciência da Computação) - Pontifícia Universidade Católica de São Paulo, São Paulo, Capital, 2015

SILVA, D**. Arquitetura de agentes para a geração automática de roteiros OCC-RDD.** 2018. 100 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2018

SILVA, I**. Projeto e Implementação de Sistemas Multi-Agentes: O Caso Tropos.** 2005. 126 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Pernambuco, Recife, 2005

SINGH, S; KAUR, S. A systematic literature review: Refactoring for disclosing *code* smells in object oriented software. **Ain Shams Engineering Journal**, 2017.

SOMMERVILLE, I. **Engenharia de Software**, São Paulo, 2011.

SONARQUBE. Disponível em: https://software.com.br/p/sonarqube. Acesso em: ago/2020.

SOUSA, B. L. A systematic literature mapping on the relationship between design patterns and bad smells. In: **Proceedings of the 33rd Annual ACM Symposium on Applied Computing**. ACM, 2018. p. 1528-1535.

SRL, I.. inFusion Hydrogen. 2012. Disponivel em: < https://marketplace.eclipse.org/content/infusion-hydrogen>. Acesso em: 06 maio 2019.

SMART, J. F. **BDD in Action**: Behaviour-Driven Development for the whole software lifecycle. Manning Publications, 2014.

STEIL, A. V.; BARCIA, R. M. Aspectos estruturais das organizações virtuais. **Submetido ao ENANPAD99, Foz do Iguaçu**, v. 19, 1999.

STOYANOV, S. **Context-Aware and Adaptable eLearning** Systems. 2012.

STOYANOV, S. et al. Context-Aware E-Learning Infrastructure, The ICT Age. 2016.

STOYANOV, S. *et al*. ReLE-A Refactoring Supporting Tool. **Compt. Rend. Acad. Bulg. Sci**, v. 64, n. 7, p. 1017-1026, 2011.

TRIFU, A; SENG, O; GENSSLER, T. Automated design flaw correction in object-oriented systems. In: **Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.** IEEE, 2004. p. 174-183.

VALE, G. *et al.* Bad smells in software product lines: A systematic review. In: **2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse**. IEEE, 2014. p. 84-94.

WANGBERG, R. **A literature review on *code* smells and refactoring**. 2010. Dissertação de Mestrado.

WEI, L; *et al.* Automated pattern-directed refactoring for complex conditional *State*ments. **Journal of Central South University**, Springer, v. 21, n. 5, 2014.

WELTY, C. A. Augmenting abstract syntax trees for program understanding. In: **Proceedings 12th IEEE International Conference Automated Software Engineering**. IEEE, 1997. p. 126-133.

WOOLDRIDGE, M; JENNINGS, N R.; KINNY, D. The Gaia methodology for agent-oriented analysis and design. **Autonomous Agents and multi-agent systems**, v. 3, n. 3, p. 285-312, 2000.

WOOLDRIDGE, M. **An introduction to multiagent systems**. [S.l.]: John Wiley & Sons, 2009.

YUSIFOGLU, V; AMANNEJAD, Y; CAN, Aysu B. Software test-code engineering: A systematic mapping. **Information and Software Technology**, v. 58, p. 123-147, 2015.

ZHANG, M; HALL, T; BADDOO, N. Code bad smells: a review of current knowledge. **Journal of Software Maintenance and Evolution: research and practice**, v. 23, n. 3, p. 179-202, 2011.

ZHAO, X; OSTERWEIL, Leon J. An approach to modeling and supporting the rework process in refactoring. In: **2012 International Conference on Software and System Process (ICSSP)**. IEEE, 2012. p. 110-119.

APENDICE A - SUBPLANOS ENVOLVENDO PADRÕES DE PROJETO

Quadro 26 - Subplanos para aplicação de padrões de projeto

Subplano	**Sequências de passos**
S1 - *FactoryMethod*	1. Itera sobre os possíveis métodos da classe lida 2. Se encontrar algum método como "void" ou *abstract* ele salta a iteração e passa para a leitura do próximo método 3. Dentro da iteração dos métodos ele faz a verificação se aquele método tem parâmetros. Caso não tenha, ele salta a iteração e passa para leitura do próximo método 4. Verifica se o método tem estrutura de corpo. 5. Verifica se o método tem condicionais "IF" dentro do mesmo. 6. Verifica se algum parâmetro que tenha no método também é usado na condicional. 7. Verifica se internamente ao corpo do condicional tem alguma instanciação de novo objeto. 8. Verifica se a instanciação do novo objeto tem algum retorno deste objeto ainda dentro do condicional. 9. Caso as verificações acima sejam verdadeiras ele grava o condicional analisado em uma lista. 10. Ao final da leitura do método, caso tenha condicionais propícias a refatoração, ele grava método + condicionais em uma lista de métodos a serem vistos pela refatoração. 11.Retorna a lista de possíveis métodos e suas condicionais para refatoração. 12.Para aplicar a refatoração do padrão de projeto *FactoryMethod* ele primeiramente lança o comentário no topo da classe dizendo que a mesma foi modificada para conter o padrão de projeto *Factory Method* 13.Faz a iteração sobre os métodos Java que foram achados propícios a refatoração. 14.Retira o corpo desde método e remove os parâmetros encontrados. 15.Faz a iteração sobre os condicionais encontrados envolvendo o método da primeira iteração. 16.Encontra no condicional dentro de seu corpo qual é a instanciação do novo objeto. 17.Cria uma nova classe com nome desse objeto + "Factory" ao final dela para a mesma servir como uma *Factory* daquele objeto. 18.Adiciona dentro desta *Factory* um método para fazer o retorno do objeto primeiramente lido. 19.Define este método com mesmo nome do método lido na iteração. 20.Define o tipo igual a do método lido na iteração. 21.Define os modificadores do novo método como público 22.Define a classe *Factory* criada para conter herança da primeira classe lida. 23.Adiciona a classe lida com modificador abstrato.

Subplano	Sequências de passos
S2 - *NullObject*	1. Itera sobre os campos declarados da classe lida. 2. Itera sobre os métodos da classe lida. 3. Verifica se o método é um construtor, abstrato ou é privado e salta para próxima iteração. 4. Verifica se o método tem corpo. 5. Verifica os condicionais dentro do método para ver se tem alguma comparação a nulo. 6. Caso o atributo da classe seja usado no condicional cuja comparação é nula, ele guarda em uma lista de condicionais boas para refatoração. 7. Terminando a iteração do método, se ele verificar que ele teve condicionais com comparação a nulo, ele guarda em uma listagem de métodos para refatoração. 8. Para aplicar a refatoração na classe lida logo no começo da mesma, ele coloca o comentário de classe modificada para conter o padrão *NullObject.* 9. Faz a iteração sobre os possíveis métodos encontrados na análise. 10. Faz a criação da classe Null + "nome da classe comparada a nulo" 11. Faz a criação da classe Abstract + "nome da classe comparada a nulo" 12. Faz a criação dentro da classe originalmente lida do método *assign* + "nome da classe comparada a nulo" com retorno do mesmo tipo da classe abstrata criada. Dentro deste método é colocado uma decisão, ele cria uma nova instância da classe *NullObject.* Do contrário, ele retorna a instância anteriormente criada. 13. Dentro da classe objeto original comparada a nulo, da classe abstrata e também da classe *NullObject,* são criados os métodos i*sNull*, *getReference* e *assertNotNull*. 14. Com a estruturação dispara os condicionais da classe lida são alterados para usarem o método *assignObject* evitando vários condicionais com comparação a nulos espalhados na classe.

S4 - *Strategy*	1. Itera sobre os métodos da classe lida 2. Se o método for abstrato, ou um construtor privado ele pula e passa para próxima iteração. 3. Caso o método lido não tenha parâmetros, ele salta e passa para a próxima iteração de método. 4. Verifica se o método tem corpo. 5. Verifica se o método tem condicionais com "IF". 6. Verifica se o condicional tem uma estrutura de comparação usando "==". 7. Verifica se o parâmetro do método também é usado no condicional. 8. Verifica se o corpo do condicional não tem nenhuma instanciação de novo objeto criado. 9. Verifica se o corpo do condicional faz retorno de alguma expressão. 10.Caso passe nas verificações ele armazena o condicional em uma listagem de possíveis condicionais viáveis a refatoração. 11.Terminando a iteração do método ele verifica se houve condicionais viáveis a refatoração. Caso tenha, ele guarda aquele método e condicionais em listagem de métodos para serem vistos com a refatoração. 12.Para aplicar a refatoração do padrão *Strategy* logo no começo da classe lida ele coloca um comentário no topo dizendo que a classe foi modificada para conter o padrão *Strategy*. 13.Faz a criação da classe *Strategy* que é do tipo abstrata, responsável por conter a declaração o método originalmente lido para refatoração. 14.Faz a criação das classes Concretas com nome *ConcreteStrategy* + "valor comparado no condicional" que internamente será declarado o método original, porém cada uma com sua forma de fazer o retorno da expressão. 15.Faz com que a classe lida original use a modificada em seu método refatorado, assim retirando os condicionais que originalmente estavam ali.

APENDICE B - PROJETOS PARA O EXPERIMENTO DE P11 A P60

Tabela 5 - Relação dos Projetos para Realização do Experimento

(continua)

Código	**Nome do Projeto**	**Fonte**	**Quantidade de Classes**
P11	OpenCensus	https://github.com/census-instrumentation/opencensus-java	645
P12	Data Structure And Algorithms Made Easy In Java	https://github.com/careermonk/DataStructureAndAlgorithmsMadeEasyInJava	163
P13	Java Game Server	https://github.com/menacher/java-game-server	202
P14	2D Java Game Framework	https://github.com/pacampbell/Game	47
P15	JavaGame	https://github.com/redomar/JavaGame	59
P16	Baritone	https://github.com/cabaletta/baritone	314
P17	Java-Snake-Game	https://github.com/janbodnar/Java-Snake-Game	2
P18	JavaRush	https://github.com/alexilyenko/JavaRush	694
P19	The Snake	https://github.com/hexadeciman/Snake	7
P20	ExeploreGame	https://github.com/srbcheema1/ExeploreGame	16
P21	Game On	https://github.com/gameontext/sample-room-java	16
P22	Java FX	https://github.com/dmpe/JavaFX.git	32
P23	Java FX Chat	https://github.com/DomHeal/JavaFX-Chat.git	32
P24	Java Simple Connector	https://github.com/scream3r/java-simple-serial-connector.git	7
P25	JCip	https://github.com/jcip/jcip.github.com.git	144
P26	Jetcd	https://github.com/etcd-io/jetcd.git	153
P27	Jsonld	https://github.com/jsonld-java/jsonld-java.git	41
P28	Liblinear	https://github.com/bwaldvogel/liblinear-java.git	34
P29	Nuwa	https://github.com/jasonross/Nuwa.git	8
P30	Redmine	https://github.com/taskadapter/redmine-java-api.git	126
P31	Scrypt	https://github.com/wg/scrypt.git	19
P32	Stateless4j	https://github.com/oxo42/stateless4j.git	44
P33	Fastdfs	https://github.com/happyfish100/fastdfs-client-java.git	39
P34	Retrolambda	https://github.com/evant/gradle-retrolambda.git	22
P35	JAdventure	https://github.com/Progether/JAdventure.git	63
P36	Java Game Server	https://github.com/menacher/java-game-server.git	202
P37	Swagger	https://github.com/swagger-api/swagger-parser.git	24
P38	JFairy	https://github.com/Devskiller/jfairy.git	100
P39	Java API wrapper	https://github.com/soundcloud/java-api-wrapper.git	64
P40	LevelDB	https://github.com/dain/leveldb.git	105
P41	OpenCensus	https://github.com/census-instrumentation/opencensus-java.git	645
P42	Raml	https://github.com/raml-org/raml-java-parser	511

Tabela 5 - Relação dos Projetos para Realização do Experimento

(conclusão)

Código	**Nome do Projeto**	**Fonte**	**Quantidade de Classes**
P43	Javacord	https://github.com/Javacord/Javacord.git	569
P44	GeoIp API	https://github.com/maxmind/geoip-api-java.git	28
P45	Jnative Hook	https://github.com/kwhat/jnativehook.git	42
P46	Joinery	https://github.com/cardillo/joinery.git	52
P47	Kurento	https://github.com/Kurento/kurento-java.git	506
P48	Play Authenticate	https://github.com/joscha/play-authenticate.git	122
P49	Vlcj	https://github.com/caprica/vlcj.git	299
P50	Data Structure	https://github.com/careermonk/DataStructureAndAlgorithmsMadeEasyInJava.git	163
P51	Intro to RX	https://github.com/Froussios/Intro-To-RxJava.git	93
P52	J2V8	https://github.com/eclipsesource/J2V8.git	98
P53	JsBridge	https://github.com/lzyzsd/JsBridge.git	12
P54	Just Java	https://github.com/udacity/Just-Java.git	1
P55	MySQL BinLog	https://github.com/shyiko/mysql-binlog-connector-java.git	101
P56	Open tracing	https://github.com/opentracing/opentracing-java.git	86
P57	Docker Java	https://github.com/kpelykh/docker-java.git	31
P58	Firebase admin java	https://github.com/firebase/firebase-admin-java.git	408
P59	Selenide	https://github.com/codeborne/selenide.git	528
P60	Send Grid Java	https://github.com/sendgrid/sendgrid-java.git	105
Total			**7.824**