UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SAMUEL BRISTOT LOLI

CODE SMELLS NO CONTEXTO DE MAPEAMENTO OBJETO-RELACIONAL EM PROJETOS JAVA: um catálogo e uma ferramenta de detecção

Recife
2020

SAMUEL BRISTOT LOLI

CODE SMELLS NO CONTEXTO DE MAPEAMENTO OBJETO-RELACIONAL EM PROJETOS JAVA: um catálogo e uma ferramenta de detecção

Este trabalho foi apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

Área de Concentração: Engenharia da Computação
Orientador: Prof. Dr. Leopoldo Motta Teixeira
Coorientador: Prof. Dr. Bruno Falcão de Souza Cartaxo

Recife
2020

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2105

L837c Loli, Samuel Bristot
Code smells no contexto de mapeamento objeto-relacional em projetos java: um catálogo e uma ferramenta de detecção/ Samuel Bristot Loli. – 2020.
111f.: il., fig., tab.

Orientador: Leopoldo Motta Teixeira.
Dissertação (Mestrado Profissional) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia da Computação. 2. ORM. 3. Code smells. 4. Java. I. Teixeira, Leopoldo Motta. (orientador) II. Título.

621.39 CDD (22. ed.) UFPE-CCEN 2020-180

Samuel Bristot Loli

CodeSmells no Contexto de Mapeamento Objeto-Relacional em Projetos Java: Um Catálogo e uma Ferramenta de Detecção

Dissertação apresentada ao Programa de Pós-Graduação Profissional em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre Profissional em 11 de setembro de 2020.

Aprovado em 11 de setembro de 2020.

BANCA EXAMINADORA

_____

Prof. Luciano de Andrade Barbosa
Centro de Informática / UFPE

_____

Prof. Gustavo Henrique Lima Pinto
Universidade Federal do Pará

_____

Prof. Leopoldo Motta Teixeira
Centro de Informática / UFPE
(Orientador)

AGRADECIMENTOS

Não seria possível a conclusão desta importante etapa em minha vida profissional sem o apoio de pessoas que por meio de incentivos, compreensão ou dicas me ajudaram a seguir até o fim desta etapa. Por isso agradeço:

Primeiramente a Deus, pelo dom da vida e por me dar força e saúde para seguir em frente;

A minha companheira Viviani, pelo amor incondicional, sempre me apoiando de todas as formas, suportando minhas ausências e não medindo esforços para que eu conseguisse concluir essa etapa;

Aos meus pais, Valmor e Cida, que sempre me incentivaram a estudar, batalhar, pelo amor sincero e exemplo de perseverança;

Aos meus irmãos, Bruno e Gabriel que por serem da mesma área de atuação me auxiliaram muito, seja em forma de apoio ou de troca ideias;

Aos colegas de mestrado Daniel e Eliandro, pela companhia nas aulas, viagens, trabalhos em grupo e apoio;

Aos colegas da DTIC do IFSC, pelos conselhos, sugestões, apoio e compromisso assumido durante minha ausência. Em especial aos colegas Shirlei e Victor por me auxiliarem de diversas formas nas etapas da pesquisa;

Ao Professor e Orientador Leopoldo Motta Teixeira, por acreditar na ideia, pela parceria e por estar sempre presente disponível em todas as etapas desta pesquisa;

Ao Professor e Coorientador Bruno Falcão de Souza Cartaxo, pela disponibilidade e pelos conselhos durante o desenvolvimento da pesquisa;

Ao IFSC por me oferecer esta grande oportunidade;

A todos que contribuíram, indiretamente, para a realização desta pesquisa.

RESUMO

Más escolhas durante o desenvolvimento de software podem causar futuros problemas de manutenção e desempenho. *Code smells* são tipicamente utilizados como indicadores de tais problemas. Vários *code smells* foram propostos anteriormente com foco no código genérico. Neste trabalho, focamos nas especificidades do código *Object Relational Mapping* (ORM) em Java. Desenvolvedores normalmente utilizam *frameworks* ORM para abstrair a complexidade de acesso ao banco de dados. No entanto, quando mal utilizados podem causar problemas que afetam a manutenção e o desempenho geral do sistema. Isto posto, apresentamos um catálogo contendo oito *code smells* ORM extraídos do estado de pesquisa e da prática, por meio de uma combinação de *Rapid Review* RR e *Grey Literature Review* (GLR). Para cada *smell*, também apresentamos uma solução sugerida e discussão a cerca do tema. Para avaliar o catálogo, realizamos uma pesquisa com 84 participantes. A maioria dos participantes concorda que os *code smells* ORM são um problema e que as soluções sugeridas são adequadas. Além do catálogo, apresentamos uma ferramenta para detectar e sugerir soluções em quatro *code smells* por meio de análise estática do código. Avaliamos a ferramenta quanto à precisão e revocação utilizando uma pequena aplicação de referência desenvolvida com exemplos de *smells* e posteriormente em quatro projetos reais extraídos do Github e do GitLab do Instituto Federal de Santa Catarina (IFSC). Aplicamos a ferramenta também em 379 projetos para análise da difusão geral dos *code smells* ORM, detectando 2829 *smells* em 75% dos projetos analisados. Em conclusão, este trabalho contribui com uma maneira sistemática de descrever os *code smells* ORM em um catálogo inicial, que pode ser útil tanto para pesquisadores quanto para profissionais, avaliado positivamente pelos nossos resultados iniciais, além de uma ferramenta para detecção automática por meio de análise estática do código de um subconjunto dos *smells* definidos.

Palavras-chave: ORM. Code smells. Java. Analise estática do código.

ABSTRACT

Bad choices during software development might lead to maintenance and performance issues. Code smells are typically used to indicate such problems. A number of smells have been proposed, usually focused on generic code problems. In this work, we focus on the specifics of Object-Relational Mapping (ORM) code in Java. Developers use ORM frameworks to abstract the complexity of accessing a database. However, when poorly used, frameworks can lead to problems that might affect maintenance and overall performance of the system. Therefore, we present a catalog of eight smells extracted from the state of research and practice, through a combination of rapid review and grey literature review. For each smell, we also present a suggested solution and rationale. To evaluate the catalog, we conducted a survey with 84 respondents. The majority of the respondents agree both that the code smells are a problem, as well as that the suggested solution is adequate. In addition to the catalog, we present a tool to detect and suggest solutions in four code smells through static code analysis. We evaluated the tool for precision and recall using a small reference application developed with some examples of smells and after using real projects extracted from Github and GitLab from the Federal Institute of Santa Catarina (IFSC). The tool will also be applied in 379 projects to analyze the general diffusion of ORM code smells detecting 2829 smells in 75% of the analyzed projects. In conclusion, this work contributes with a systematic way of describing ORM code smells and an initial catalog, which can be useful for researchers and practitioners, positively evaluated by our initial results. In addition, we present a tool that detects ORM code smells through static code analysis in an automated way.

Keywords: *ORM*. *Code smells*. *Java*. *Static Code Analysis*.

LISTA DE FIGURAS

Figura 1 – Exemplo de tabela no banco de dados relacional 22
Figura 2 – *Ranking* - Percentual de uso por modelo de banco de dados 22
Figura 3 – Comunicação JDBC 24
Figura 4 – Arquitetura JPA, ... 26
Figura 5 – Tecnologias utilizadas para acesso a base de dados em Java 30

Figura 6 – Resultado do procedimento de seleção 37
Figura 7 – Evidence Briefing 39

Figura 8 – Dados em Excesso 43
Figura 9 – N+1 54
Figura 10 – Refatoração 1 57
Figura 11 – Refatoração 2 58

Figura 12 – Qualificação 70
Figura 13 – Experiência de desenvolvimento Java 70
Figura 14 – Número de aplicações utilizando ORM 70
Figura 15 – Nível de concordância geral (média) 71
Figura 16 – Nível de concordância considerando desenvolvedores experientes (média) 72
Figura 17 – Nível de concordância considerando desenvolvedores iniciantes ou intermediários (média) 72
Figura 18 – Nível de concordância por *code smell* 73
Figura 19 – Nível de concordância por solução 73

Figura 20 – Teste de detecção - *EagerSmell* 79
Figura 21 – Teste de detecção - *N1JoinFetchSmell* 82
Figura 22 – Teste de detecção - *OneToManyCollectionSmell* 84
Figura 23 – Teste de detecção - *ProjectionSmell* 86

Figura 24 – Filtro para mostrar somente os *Code Smells ORM* 90
Figura 25 – Caso de teste - BadCaseEagerSmell 93
Figura 26 – Caso de teste - GoodCaseEagerSmell 93

LISTA DE EXEMPLOS

Exemplo 1 – Consulta ORM para obter uma lista de DiscentesGraduacao 17
Exemplo 2 – Entidades relacionadas 18

Exemplo 3 – Exemplo de DDL 23
Exemplo 4 – Exemplo de DML 23
Exemplo 5 – Mapeamento de classe de domínio 28
Exemplo 6 – Código duplicado Smell 32
Exemplo 7 – Agrupamento de Dados Smell 32

Exemplo 8 – Eager Smell 44
Exemplo 9 – Ajuste Eager Smell 44
Exemplo 10 – Projeção Smell 47
Exemplo 11 – Ajuste Projeção Smell 48
Exemplo 12 – Update Smell 50
Exemplo 13 – Ajuste Update Smell 50
Exemplo 14 – Paginação Smell 52
Exemplo 15 – Correção Paginação Smell 52
Exemplo 16 – *Join Fetch Smell* 55
Exemplo 17 – Ajuste *Join Fetch Smell* 56
Exemplo 18 – Acesso um por um Smell 59
Exemplo 19 – Ajuste: Acesso um por um Smell (`@BatchSize`) 60
Exemplo 20 – Ajuste: Acesso um por um Smell (JOIN FETCH) 60
Exemplo 21 – OneToMany Smell 62
Exemplo 22 – Ajuste OneToMany smell - transformando em bidirecional 63
Exemplo 23 – Ajuste oneToMany smell com `Set` 63
Exemplo 24 – Somente Leitura Smell 65
Exemplo 25 – Ajuste Somente Leitura Smell 66
Exemplo 26 – Utilização `@Imutable` 67
Exemplo 27 – Definição somente-leitura na sessão 67

Exemplo 28 – SQL utilizado para mineração no Stackoverflow 68

Exemplo 29 – Classe *EagerDetector* 78
Exemplo 30 – Classe *N1MissedJoinFetchDetector* 81
Exemplo 31 – Classe *OneToManyCollectionsDetector* 84
Exemplo 32 – Classe *ProjectionDetector* 86

LISTA DE TABELAS

Tabela 1 – Refatoração ORM 18

Tabela 2 – Estratégias de busca por relacionamento 30

Tabela 3 – Questões referentes a inclusão de literatura cinza nas revisões de engenharia de software 35

Tabela 4 – Padrão de Busca RR 36

Tabela 5 – Padrão de Busca GLR 37

Tabela 6 – Estudos selecionados RR e GLR 40

Tabela 7 – Catálogo de Code Smells ORM 42

Tabela 8 – Três possíveis semânticas com anotação `@OneToMany` em Java . . . 64

Tabela 9 – Número de instruções realizadas a partir de cada semântica utilizada . 64

Tabela 10 – *Code smells* ORM selecionados 77

Tabela 11 – Cálculo Precisão e Revocação 89

Tabela 12 – Padrão de busca - repositórios para avaliação manual 90

Tabela 13 – Padrão de busca - repositórios para análise de difusão 91

Tabela 14 – Analise de precisão - Aplicação de Referência 94

Tabela 15 – Análise de revocação - Aplicação de Referência 95

Tabela 16 – F1-Score - Aplicação de Referência 95

Tabela 17 – Análise de precisão - Aplicação em projetos reais 97

Tabela 18 – Analise de revocação - Aplicação em projetos reais 97

Tabela 19 – F1-Score - Aplicação em projetos reais 97

Tabela 20 – Difusão dos *smells* em projetos reais 99

LISTA DE SIGLAS

API	*Application Programming Interface*
CRUD	*Create, Read, Update and Delete*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
DTO	*Data Transfer Object*
GLR	*Grey Literature Review*
GUI	*Graphical User Interface*
HQL	*Hibernate Query Language*
IFSC	Instituto Federal de Santa Catarina
JAR	*Java Archive*
JDBC	*Java Database Connectivity*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
LOC	Lines of code
ORM	*Object Relational Mapping*
PAEVS	*Programa de Assistência Estudantil*
RR	*Rapid Review*
SGBD	Sistema de Gerenciamento de Banco de Dados
SIG	Sistema Integrado de Gestão
SIGAA	Sistema Integrado de Gestão Acadêmica
SLR	Systematic Literature Review
SQL	*Structured Query Language*
UFRN	Universidade Federal do Rio Grande do Norte
XML	*Extensible Markup Language*

SUMÁRIO

1	**INTRODUÇÃO**	14
1.1	CONTEXTUALIZAÇÃO	14
1.2	MOTIVAÇÃO	15
1.3	OBJETIVOS	19
1.3.1	**Objetivo Geral**	19
1.3.2	**Objetivo Específico**	19
1.4	ESTRUTURA DA DISSERTAÇÃO	20
2	**FUNDAMENTAÇÃO TEÓRICA**	21
2.1	PERSISTÊNCIA DE DADOS	21
2.1.1	**Banco de dados relacional**	21
2.1.2	**Linguagem SQL**	22
2.1.3	**Persistência em Java**	23
2.2	ORM	24
2.2.1	**JPA**	25
2.3	*FRAMEWORKS* ORM QUE IMPLEMENTAM JPA	29
2.4	*CODE SMELLS*	31
2.5	SPOTBUGS	33
3	**METODOLOGIA**	34
3.1	MÉTODO	34
3.2	PROBLEMA PRÁTICO	35
3.3	QUESTÕES DE PESQUISA PARA CONDUÇÃO DA RR E GLR	35
3.4	ESTRATÉGIA DE BUSCA	36
3.5	PROCEDIMENTO DE SELEÇÃO	36
3.5.1	**Resultado**	37
3.6	PROCEDIMENTO DE SíNTESE	38
3.7	RELATÓRIO	39
4	**CATÁLOGO DE CODE SMELLS ORM**	41
4.1	DADOS EM EXCESSO	41
4.1.1	**Eager como estratégia de busca nos relacionamentos a nível de classe (estático)**	43
4.1.1.1	*Descrição do Smell*	43
4.1.1.2	*Sugestão de Refatoração*	43
4.1.1.3	*Detalhamento e Discussão*	44

4.1.2 Recuperação de dados sem projeção para somente leitura 46
4.1.2.1 Descrição do Smell 46
4.1.2.2 Sugestão de Refatoração 47
4.1.2.3 Detalhamento e Discussão 47
4.1.3 Atualização desnecessária de toda a entidade 49
4.1.3.1 Descrição do Smell 49
4.1.3.2 Sugestão de Refatoração 49
4.1.3.3 Detalhamento e Discussão 49
4.1.4 Falta de paginação quando não necessário todo o resultado 51
4.1.4.1 Descrição do Smell 51
4.1.4.2 Sugestão de Refatoração 51
4.1.4.3 Detalhamento e Discussão 52
4.2 PROBLEMA DO N+1 53
4.2.1 Falta de JOIN FETCH nas consultas de atributos do tipo EAGER . . . 54
4.2.1.1 Descrição do Smell 54
4.2.1.2 Sugestão de Refatoração 54
4.2.1.3 Detalhamento e Discussão 55
4.2.2 Acesso um por um: estrutura de repetição e estratégia de busca do tipo LAZY 58
4.2.2.1 Descrição do Smell 58
4.2.2.2 Sugestão de Refatoração 58
4.2.2.3 Detalhamento e Discussão 59
4.2.3 `@OneToMany` unilateral com uso inadequado de coleções 61
4.2.3.1 Descrição do Smell 61
4.2.3.2 Sugestão de Refatoração 61
4.2.3.3 Detalhamento e Discussão 64
4.3 OUTROS *CODE SMELLS* ORM 65
4.3.1 Não uso de consultas somente leitura 65
4.3.1.1 Descrição do Smell 65
4.3.1.2 Sugestão de Refatoração 65
4.3.1.3 Detalhamento e Discussão 66

5 AVALIAÇÃO DO CATÁLOGO DE CODE SMELLS ORM 68
5.1 METODOLOGIA DA APLICAÇÃO DO *SURVEY* 68
5.2 FORMA DE AVALIAÇÃO 69
5.3 RESULTADOS 69

6 FERRAMENTA PARA DETECÇÃO DE CODE SMELLS ORM 75
6.1 ARQUITETURA 75
6.1.1 Artefatos Principais 75

6.1.2 **Artefatos de Testes** . . . 75
6.1.3 **Artefatos de Recursos** . . . 76
6.2 DETECTORES . . . 76
6.2.1 **Metodologia de seleção** . . . 76
6.2.2 **EAGERSMELL** . . . 77
6.2.2.1 *Regras para Detecção* . . . 77
6.2.2.2 *Funcionamento do Detector* . . . 77
6.2.2.3 *Limitações conhecidas* . . . 79
6.2.3 **N1JOINFETCHSMELL** . . . 80
6.2.3.1 *Regras para Detecção* . . . 80
6.2.3.2 *Funcionamento do Detector* . . . 80
6.2.3.3 *Limitações conhecidas* . . . 82
6.2.4 **ONETOMANYCOLLECTIONSSMELL** . . . 82
6.2.4.1 *Regras para Detecção* . . . 83
6.2.4.2 *Funcionamento do Detector* . . . 83
6.2.4.3 *Limitações conhecidas* . . . 85
6.2.5 **PROJECTIONSMELL** . . . 85
6.2.5.1 *Regras para Detecção* . . . 85
6.2.5.2 *Funcionamento do Detector* . . . 85
6.2.5.3 *Limitações conhecidas* . . . 87

7 **AVALIAÇÃO DA FERRAMENTA DESENVOLVIDA** . . . 88
7.1 METODOLOGIA . . . 88
7.2 AVALIAÇÃO UTILIZANDO A APLICAÇÃO DE REFERÊNCIA . . . 92
7.2.1 **Aplicação de Referência** . . . 92
7.2.2 **Resultado da avaliação** . . . 94
7.3 AVALIAÇÃO UTILIZANDO PROJETOS REAIS . . . 95
7.3.1 **Aplicações Selecionadas** . . . 95
7.3.2 **Resultado** . . . 96
7.4 AVALIAÇÃO DA DIFUSÃO DOS *CODE SMELLS* ORM . . . 98

8 **CONCLUSÕES** . . . 100
8.1 TRABALHOS RELACIONADOS . . . 101
8.2 LIMITAÇÕES E AMEAÇAS À VALIDADE . . . 102
8.3 TRABALHOS FUTUROS . . . 104

REFERÊNCIAS . . . 105

1 INTRODUÇÃO

Neste capítulo é retratada uma visão geral do trabalho, contendo a contextualização do problema, a motivação da pesquisa, bem como os objetivos, gerais e específicos, a serem realizados neste trabalho.

1.1 CONTEXTUALIZAÇÃO

Desenvolver software de qualidade em uma arquitetura multicamadas que suporte grande volume de dados é uma constante necessidade na indústria, exigindo do desenvolvedor habilidades e técnicas em diversas áreas de conhecimento, além do uso de ferramentas de apoio adequadas para auxiliar no processo de desenvolvimento. Na fase de concepção de um software, escolhas incorretas podem trazer problemas de manutenção e desempenho. Segundo Fowler (1999), sintomas de implementação abaixo do ideal, que indiquem código de baixa qualidade são chamados de *code smells*. Na literatura, estudos anteriores mostram que *code smells* afetam a manutenibilidade do código (Yamashita & Moonen, 2013) e também indicam potencial para refatorações futuras (Khomh *et al.*, 2009). Embora essa definição seja bastante ampla e pareça sugerir diversos *code smells* que possam afetar os sistemas, geralmente são focados no código, sem considerar as características específicas de domínio dos sistemas e *frameworks* utilizados durante o desenvolvimento.

Frameworks Object Relational Mapping (ORM) são popularmente utilizados pelos desenvolvedores para facilitar o gerenciamento de dados no desenvolvimento de software, permitindo uma abstração conceitual entre objetos de sistemas orientado a objetos e dados armazenados em bancos de dados relacionais (Chen *et al.*, 2016a). Entretanto, essa abstração traz alguns riscos. Segundo Chen *et al.* (2014), os desenvolvedores costumam codificar ORM sem levar em consideração outros aspectos relevantes que não o funcionamento imediato do código, a exemplo do desempenho do banco de dados e manutenção do código, ocasionando problemas, como recuperar dados em excesso do banco de dados, desperdiçar recursos computacionais ou executar N consultas quando apenas uma seria suficiente, podendo ter transações que são canceladas por tempo limite ou até travamentos no sistema em grande escala. Pode-se concluir, portanto, que o uso incorreto de ORM durante a codificação não é considerado problema imediato por não gerar erros na hora da codificação, mas pode ser considerado como *code smell* uma vez que apresenta sintomas de implementação abaixo do ideal, representando risco de futuros problemas e necessidade de manutenção. Chen *et al.* (2016b) em seus estudos sobre manutenção do código ORM em Java, conclui que as alterações no código ORM são complexas e que estudos futuros devem analisar as causas dos problemas para ajudar o desenvolvedor a melhor projetar o código ORM.

No contexto de gerenciamento de dados na linguagem de desenvolvimento Java, este trabalho propõe um catálogo com oito *code smells* ORM de forma estruturada, incluindo

descrição, solução sugerida e discussão acerca do *code smell* verificado. O catálogo foi desenvolvido a partir do estudo de problemas, causas e soluções referentes ao ORM em Java extraídos do estado de pesquisa e da prática, por meio de uma combinação de *Rapid Review* (RR) e *Grey Literature Review* (GLR). O catálogo proposto foi avaliado por desenvolvedores familiarizados com ORM Java por meio de um *survey*. Eles responderam até que ponto concordam que os *smells* são sintomas de problemas reais e que as soluções sugeridas são de fato candidatas a corrigir esses *smells*. Obtivemos ao final 86 respostas com uma média de concordância de 75,44% entre *smells* e soluções apresentadas no catálogo.

Em adicional, apresentamos uma ferramenta para detecção automatizada através de análise estática de quatro *code smells* selecionados do catálogo, com objetivo de facilitar a detecção apresentando o local no código e sugerindo possíveis correções. A ferramenta foi avaliada quanto a eficácia e revocação em duas etapas. Para primeira etapa, desenvolvemos um projeto de referência com exemplos de *code smells* ORM para avaliação em um ambiente controlado. Posteriormente, a ferramenta foi aplicada em quatro projetos reais extraídos de sistemas de controle de versão GitLab (IFSC) e GitHub (Geral). Também verificamos a difusão dos *code smells* ORM através da aplicação da ferramenta em 379 projetos para observar quais *code smells* são mais e menos praticados.

1.2 MOTIVAÇÃO

A motivação para este estudo foi elaborada por meio de duas percepções: a primeira pela experiência do pesquisador em seu trabalho como analista de T.I no Instituto Federal de Santa Catarina (IFSC); a segunda fundamentada na literatura.

A primeira perspectiva motivacional aconteceu devido a repetidos incidentes relatados pelos usuários referente ao baixo desempenho do sistema de gestão da instituição que o pesquisador trabalha. Ao averiguar esses incidentes, foi identificado como causa o excesso e tamanho das instruções *Structured Query Language* (SQL) executadas no banco de dados. Investigando a causa, verificamos que a maior parte das ocorrências eram por conta de instruções SQL geradas pelo Hibernate (*framework* ORM do sistema da instituição) que causavam diversos problemas, sendo as maiores incidências as seguintes situações:

- Excesso de dados: instruções SQL trazendo mais dados que o necessário para os casos de uso analisados, recuperando colunas e tabelas relacionadas desnecessariamente;
- Excesso de instruções (N+1): N instruções SQL geradas a partir de uma instrução principal, que poderia ser otimizada para carregar todas as informações previamente.

Para exemplificar, exibimos um caso real do Sistema Integrado de Gestão (SIG) do IFSC. O caso de uso em questão é referente ao pré-processamento de matrículas da ins-

tituição, no qual o objetivo é preparar uma lista de discentes de graduação para matricular no próximo semestre com base nos dados acadêmicos. Os usuários relataram baixo desempenho e, após análise, verificou-se que o problema estava no trecho de código ORM com a consulta responsável por montar a lista inicial dos discentes a matricular. A consulta realizada para popular essa lista utiliza a linguagem de consulta *Hibernate Query Language* (HQL). Conforme Mihalcea *et al.* (2019), HQL é uma linguagem utilizada pelo framework ORM *Hibernate* com a sintaxe semelhante ao SQL, mas totalmente orientada a objetos, utilizada para transformar a instrução HQL em instruções SQL. O Exemplo 1 demonstra o método `findDiscentesPreProcessamento` com a consulta HQL original (antes da correção), com o objetivo de recuperar informações da tabela `discente_graduacao` do banco de dados para transformar em uma lista de objetos de `DiscenteGraduacao`.

Utilizando as estatísticas do Hibernate,[1] foi realizado um teste com um grupo de 6651 discentes para realizar o pré-processamento. Verificou-se que a partir desta única consulta HQL eram geradas 6767 consultas SQL adicionais, sendo este o principal motivo do baixo desempenho de execução. Esse problema é conhecido como *N+1*, em que uma consulta gera *N* outras, causada por um ou mais *code smells* que serão melhores detalhados no catálogo desta pesquisa. Observou-se que a maior parte das 6 mil consultas extras realizadas pelo *Framework* ORM eram referentes à entidade `Pessoa`. A entidade alvo da consulta, `DiscenteGraduacao`, possui um relacionamento muitos para um (anotação `@ManyToOne`) utilizando a estratégia de busca do tipo EAGER com a entidade `Discente` e que por sua vez possui o mesmo tipo de relacionamento com a entidade `Pessoa`. O Exemplo 2 apresenta as classes das entidades envolvidas, utilizamos reticências (`...`) nos exemplos para indicar que pode existir código antes e depois dos trechos apresentados. A estratégia de busca do tipo EAGER, melhor detalhada no catálogo, conforme Mihalcea (2019a) pode ser considerada um *code smell* por sempre recuperar as informações do banco de dados, mesmo quando não utilizadas e dificultar a manutenção do código quando necessário realizar correções, como alterar para LAZY. Dessa forma, toda consulta que utiliza a entidade `DiscenteGraduacao` receberá como resultado sempre as entidades `Discente` e `Pessoa`, mesmo sem utilizar essas informações.

Com base nos estudos realizados para criar o catálogo, foram realizadas duas refatorações distintas detalhadas na Seção 4.2.1 referentes ao *code smell*: FALTA DE JOIN FETCH NAS CONSULTAS DE ATRIBUTOS DO TIPO EAGER com objetivo de melhorar o desempenho e minimizar risco de falha:

1. Adicionado `JOIN FETCH` para a entidade `Pessoa` por estar como EAGER no relacionamento a nível de classe e ser complexa a alteração para LAZY devido a dependências no código.

2. Alterado de EAGER para LAZY no mapeamento dos atributos relacionados às

[1] https://docs.jboss.org/hibernate/orm/3.5/Javadocs/org/hibernate/stat/Statistics.html

Exemplo 1 - Consulta ORM para obter uma lista de DiscentesGraduacao

```
1  public List<DiscenteGraduacao> findDiscentesPreProcessamento(FiltroPre
       filtro){
2   StringBuilder hql = new StringBuilder();
3   hql.append("SELECT discenteGrad FROM DiscenteGraduacao discenteGrad");
4   hql.append(" JOIN FETCH discenteGrad.matrizCurricular matriz ");
5   hql.append(" JOIN FETCH discenteGrad.discente discente ");
6   hql.append(" JOIN FETCH discente.curriculo curriculo ");
7   hql.append(" JOIN FETCH curriculo.curso c ");
8   hql.append(" JOIN FETCH c.tipoCursoAcademico tipoCurso");
9   hql.append(" JOIN FETCH c.ofertaCursoAcademico ofertaCurso WHERE");
10  hql.append(TemplatesSql.getFiltroPreProcessadoPeriodo(filtro));
11
12  Query query = getSession().createQuery(hql.toString());
13  List<DiscenteGraduacao> discentesGraduacao = query.list();
14  return discentesGraduacao; }
```

Instrução SQL gerada:

```
-- SQL Principal
SELECT * FROM discente_graduacao d
INNER JOIN matriz_curricular m on d.id_matriz=m.id_matriz
INNER JOIN discente d2 on d.id_discente_graduacao=d2.id_discente
INNER JOIN curriculo c on d2.id_curriculo=c.id_curriculo
INNER JOIN curso cur on c.id_curso=cur.id_curso
LEFT OUTER JOIN curso_tecnico.....
-- Consutas adicionais realizadas
SELECT * FROM pessoa where id_pessoa = ?;
SELECT * FROM pessoa where id_pessoa = ?;
....
SELECT * FROM grau_academico WHERE id_grau_academico = ?;
SELECT * FROM grau_academico WHERE id_grau_academico = ?;
....
SELECT * FROM polo WHERE id_polo = ?;
SELECT * FROM polo WHERE id_polo = ?;
....
```

SQL realmente necessário:

```
SELECT * FROM discente_graduacao d WHERE ...;
```

entidades `Polo`, `TipoCotaDiscente` e `TipoCancelamento` relacionados a entidade `DiscenteGraduacao`.

A Tabela 1 apresenta o resultado após as refatorações diminuindo o número de consultas ao fim para um.

Apesar do aumento de desempenho após as modificações, o *framework* ORM continua recuperando mais informações que o necessário, visto que não foi possível alterar o mapeamento de EAGER para LAZY devido aos riscos envolvidos, que são melhores detalhados na Seção 4.2.1, evidenciando também problemas de manutenção no código. Portanto, se na concepção do *software* o desenvolvedor soubesse que era um *code smell* deixar o relacionamento como EAGER em nível de classe, ou fosse alertado por uma ferramenta, seria recuperada

Exemplo 2 - Entidades relacionadas

```
1  @Entity @Table(name = "discente_graduacao")
2  public class DiscenteGraduacao{
3   @Id @Column(name="id_discente_graduacao")
4   private int id;
5
6   @ManyToOne(fetch=FetchType.EAGER)
7   @JoinColumn(name="id_discente_graduacao")
8   private Discente discente;
9   ...
10 }
11 @Entity @Table(name = "discente")
12 public class Discente {
13  @Id @Column(name = "id_discente")
14  private int id;
15
16  @ManyToOne(fetch = FetchType.EAGER)
17  @JoinColumn(name = "id_pessoa")
18  private Pessoa pessoa;
19  ...
20 }
```

Tabela 1 – Refatoração ORM

Estatística *Hibernate*	**Original**	**Refatoração1**	**Refatoração2**
Consultas geradas pelo *Hibernate*	6768	117	1

Fonte: Elaborada pelo autor (2020)

uma menor quantidade de informações do banco de dados, não causariam os problemas apresentados e não seriam necessárias modificações no código. As modificações executadas foram realizadas devido ao conhecimento adquirido do autor ao estudar o desenvolvimento do catálogo desta pesquisa, sendo que muitos desenvolvedores desconhecem os *code smells* que podem causar problemas de desempenho e manutenibilidade referente a ORM e como corrigi-los.

A segunda percepção motivacional foi buscar na literatura pesquisas referentes a *code smells* e ORM. Foram encontrados trabalhos referentes a *code smells* em diferentes contextos, como em (Grigera *et al.*, 2014; Cunha *et al.*, 2012), e pesquisas relacionadas a problemas relacionados a ORM, conforme (Chen *et al.*, 2014; Chen *et al.*, 2016a; Nazário *et al.*, 2019), todavia não foram encontradas pesquisas relacionadas a *code smells* ORM. No estudo realizado por Chen *et al.* (2016b) referente as práticas de manutenção de código em ORM, os autores sugerem que trabalhos futuros estudem as principais causas de problemas para auxiliar a projetar melhor o código ORM, principalmente em sistemas Java, sendo esse outro fator motivador para criar o catálogo de *code smells* ORM desta pesquisa. Os autores também diagnosticaram outros problemas que motivaram o desenvolvimento da ferramenta

de detecção: (i) Os detectores tradicionais de código estático precisam ser estendidos para capturar as especificidades do código ORM; (ii) Apesar do código ORM ser constantemente alterado no desenvolvimento de sistema, há falta de ferramentas que auxiliem o desenvolvedor a detectar possíveis problemas na alteração de um código ORM. Nazário *et al.* (2019) em seu estudo, também observaram que os problemas relacionados ao ORM são específicos do domínio e, portanto, ferramentas de detecção como o FindBugs não fornecem detectores nesse contexto, sendo outra evidência motivacional para o desenvolvimento de uma ferramenta para detecção de *code smells* ORM.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Identificar e detectar *code smells* ORM na linguagem de desenvolvimento Java com objetivo de auxiliar no processo de desenvolvimento, visando melhorar o desempenho, qualidade de código e otimizar as instruções SQL[2] geradas pelos *frameworks* ORM.

1.3.2 Objetivo Específico

- Criar um catálogo de *code smells* ORM de forma estruturada contendo descrição, sugestão de solução e discussão, a partir do estudo de problemas, causas e soluções referentes ao ORM em Java extraídos do estado de pesquisa e prática, por meio de uma combinação de RR e GLR. O catálogo proposto neste trabalho foi aceito para publicação no XXXIV Simpósio Brasileiro de Engenharia de Software (Loli *et al.*, 2020).
- Desenvolver ferramenta que, por meio de análise estática, detecte *code smells* relacionados ao catálogo de forma automatizada e apresente o local da ocorrência no código, descrição e possível sugestão para o *smell*.
- Realizar pesquisa para avaliação do catálogo de *code smells* ORM quanto ao nível de concordância dos *smells* e das soluções apresentadas com desenvolvedores Java por meio de *survey*, encaminhando para desenvolvedores com familiaridade referente a tecnologia ORM e Java.
- Desenvolver uma aplicação de referência com exemplos de *Smells* para testes em menor escala e avaliar a ferramenta quanto a precisão e revocação em um ambiente controlado, além de observar a existência de novas limitações.
- Avaliar a ferramenta considerando a eficácia e revocação utilizando sistemas reais Java, extraídos de sistemas de controle de versões, que utilizem ORM como forma de acesso a dados.

[2]por instruções, entende-se como seleção, inserção, atualização e remoção

- Verificar a difusão dos *code smells* implementados pela ferramenta através da aplicação em projetos reais, de forma a verificar quais *code smells* são mais e menos praticados.

1.4 ESTRUTURA DA DISSERTAÇÃO

Após apresentar o problema de forma introdutória, bem como a motivação, os objetos gerais e específicos, o restante da dissertação está organizado da seguinte maneira:

- Fundamentação teórica, abrangendo os conceitos bases abordados na dissertação, como conceitos de Persistência, ORM, *frameworks* JPA, *Code Smells* e SpotBugs.
- Metodologia, apresentando as revisões RR e GLR realizadas para desenvolver o catálogo de *code smells*
- Catálogo de *Code Smells* ORM, com oito *code smells* identificados a partir das revisões realizadas. Cada *code smell* é apresentado de maneira estruturada contendo: descrição, solução, exemplos práticos e discussões a cerca do *smell* relacionado.
- Avaliação do Catálogo de *Code Smells* ORM, com o resultado da avaliação realizada com 86 desenvolvedores a cerda dos smells e soluções propostas.
- Ferramenta Automatizada para Detecção de *Code Smells* ORM, apresentando a ferramenta desenvolvida como um *plugin* do SpotBugs para detecção de quatro *code smells* apresentados no catálogo.
- Avaliação da Ferramenta Desenvolvida, contendo a avaliação quanto a precisão e revocação, bem como a difusão dos *code smells* em projetos reais.
- Conclusões, com as conclusões do trabalho, além de apresentar os trabalhos relacionados, limitações e ameaças a validade e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo descrever os principais conceitos e tecnologias envolvidas no desenvolvimento dessa pesquisa.

2.1 PERSISTÊNCIA DE DADOS

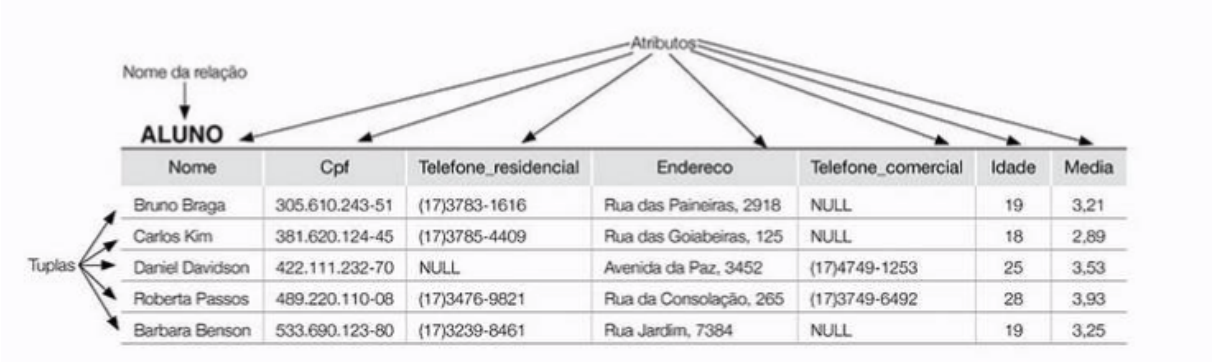
Em geral, a maioria das aplicações existentes precisam persistir de alguma forma seus dados. Segundo Bauer *et al.* (2016) um sistema de informação teria pouca utilidade se não preservasse seus dados quando desligado. Para a realização da persistência de dados, além da aplicação, é necessário: um banco de dados, que é conceituado por Elmari & Navathe (2010) como uma coleção de dados (fatos conhecidos que permitem ser registrados e tenha um significado implícito) inter-relacionados, e um Sistema de Gerenciamento de Banco de Dados (SGBD), que para Silberschatz *et al.* (2006) consiste em um conjunto de programas que definem estruturas para armazenamento dos dados relacionados e fornecem ferramentas para a manipulação desses dados.

Em se tratando de orientação a objetos, a persistência possibilita armazenar objetos sobrevivendo ao status da aplicação, permitindo recriá-los a partir do armazenamento. Em Java, a persistência da instância de objetos em um banco de dados é realizada utilizando a linguagem SQL (Bauer *et al.*, 2016). Nesta seção, será verificado o funcionamento do banco de dados relacional, da linguagem SQL e como é a utilização dessas tecnologias na persistência em Java.

2.1.1 Banco de dados relacional

A característica relacional do banco de dados tem como princípio o modelo relacional, que segundo Elmari & Navathe (2010) retrata o banco de dados como um conjunto de relações nas quais uma relação pode ser considerada uma tabela de valores e cada linha da tabela corresponde a um relacionamento ou entidade no mundo real. Silberschatz *et al.* (2006) conceitua de forma similar, definindo o modelo relacional como um conjunto de tabelas utilizadas para representar os dados e a relação entre os dados. Segundo o autor, a simplicidade conceitual do modelo fez com que fosse adotado pela maioria dos produtos de bancos de dados. No modelo relacional, cada linha é chamada de tupla, a tabela é chamada de relação e cada coluna representa um atributo. A Figura 1 demonstra um exemplo de tabela com suas terminologias em um banco de dados relacional. (Elmari & Navathe, 2010)

Figura 1 – Exemplo de tabela no banco de dados relacional



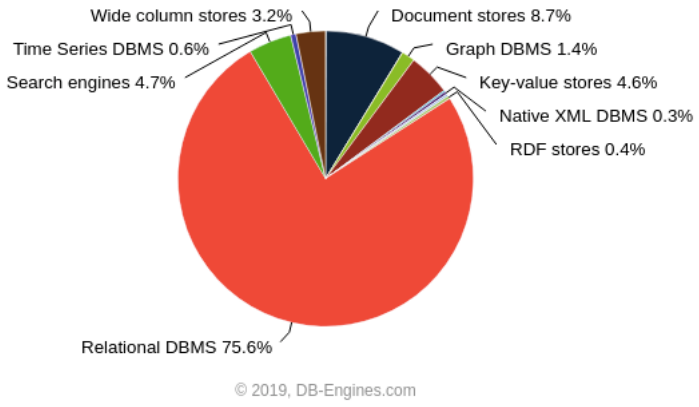
Nome	Cpf	Telefone_residencial	Endereco	Telefone_comercial	Idade	Media
Bruno Braga	305.610.243-51	(17)3783-1616	Rua das Paineiras, 2918	NULL	19	3,21
Carlos Kim	381.620.124-45	(17)3785-4409	Rua das Goiabeiras, 125	NULL	18	2,89
Daniel Davidson	422.111.232-70	NULL	Avenida da Paz, 3452	(17)4749-1253	25	3,53
Roberta Passos	489.220.110-08	(17)3476-9821	Rua da Consolação, 265	(17)3749-6492	28	3,93
Barbara Benson	533.690.123-80	(17)3239-8461	Rua Jardim, 7384	NULL	19	3,25

Fonte: Elmari & Navathe (2010)

Segundo Bauer *et al.* (2016), tanto banco de dados relacional como o SGBD não são de uso exclusivo de uma aplicação ou linguagem de programação. Isso devido ao princípio de independência dos dados, uma vez que os dados sobrevivem mais tempo que as aplicações. A tecnologia relacional é um denominador comum entre muitos *softwares* e linguagens de programação diferentes, sendo uma das razões de sua ampla utilização.

Embora existam novos modelos e tipos de banco de dados, como o modelo orientado a objetos *NoSql* e modelos orientados a documentos, o banco de dados relacional continua sendo o tipo mais utilizado, no qual a maioria dos dados corporativos estão armazenados. (Keith *et al.*, 2018) Na Figura 2, é apresentado um recente ranking com o percentual de uso por modelo de banco de dados, evidenciando o maior uso de bancos de dados relacionais.

Figura 2 – *Ranking* - Percentual de uso por modelo de banco de dados



Fonte: Engines (2019)

2.1.2 Linguagem SQL

Para realizar operações no banco de dados relacionais, é necessária uma linguagem amigável para manipulação dos dados. Com este objetivo, segundo Silberschatz *et al.* (2006),

a IBM desenvolveu na década de 70 a *Structured Query Language* (SQL). A linguagem utiliza uma combinação de álgebra e cálculo relacional para realizar manipulação e definição de dados, permitindo realizar operações como:

- *Data Definition Language* (DDL): utilizado para criar, alterar e remover estruturas, como esquemas e tabelas, através dos comandos: `CREATE`, `ALTER`, `DROP`. O Exemplo 3 apresenta um *script* SQL com o comando `CREATE`.

Exemplo 3 - Exemplo de DDL

```
CREATE TABLE comum.pessoa (
    id_pessoa serial NOT NULL, -- Chave Primária
    nome varchar(200) NOT NULL, -- nome da pessoa
    data_nascimento date NULL, -- data de nascimento
    sexo bpchar(1) NULL -- sexo da pessoa
    CONSTRAINT pessoa_pkey PRIMARY KEY (id_pessoa));
```

- *Data Manipulation Language* (DML): utilizado para inserir, alterar, consultar e remover dados através dos comandos: `INSERT`, `UPDATE`, `SELECT` e `DELETE`. O Exemplo 4 apresenta um *script* SQL com o comando `SELECT`.

Exemplo 4 - Exemplo de DML

```
SELECT data_nascimento
FROM comum.pessoa as p
WHERE p.nome = 'João de Souza';
```

A linguagem SQL pode ser considerada como uma das principais razões para o sucesso dos bancos de dados relacionais (Elmari & Navathe, 2010). Com a popularização do SQL, linguagens de orientação a objeto devem prover formas de acesso e utilização da linguagem para acesso aos dados relacionais.

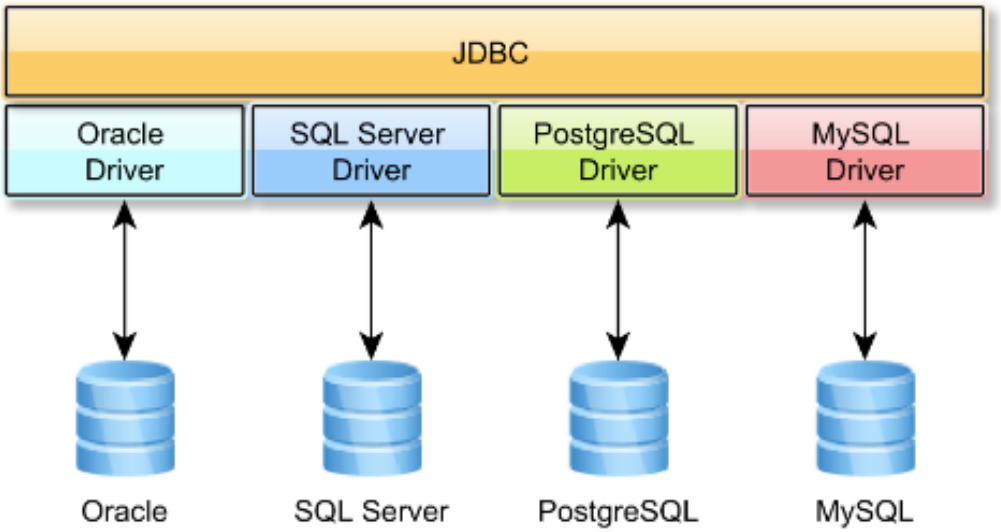
2.1.3 Persistência em Java

Em conjunto com a evolução dos complexos sistemas de informação baseados em arquitetura multicamada, surge frequentemente a necessidade de integrar dados relacionais com objetivo de facilitar o agrupamento de informações. Nesse sentido, os desenvolvedores de linguagens de programação orientada a objetos carecem de meios para tratar o armazenamento destes dados de maneira mais simples e eficiente, de modo que esses sejam persistidos em banco de dados. Barry & Stanienda (1998)

Em decorrência do modelo relacional ser distinto do modelo orientado a objetos - utilizado pelas linguagens de programação - são necessárias estratégias para realizar a comunicação entre objetos e dados em tabelas. Conforme Bauer *et al.* (2016), para manipular

dados de uma base de dados relacional em Java é necessário emitir instruções SQL para o SGBD do banco de dados utilizando a *Application Programming Interface* (API) *Java Database Connectivity* (JDBC). Independente se a consulta SQL for codificada de forma nativa (manual) ou gerada dinamicamente, o JDBC é o responsável por estabelecer a conexão, preparar parâmetros e percorrer o resultado da instrução. A API JDBC provê uma *interface* para comunicação com servidor de banco de dados. A lógica de comunicação de rede, bem como o protocolo de comunicação específico do banco de dados, são ocultos pela API JDBC (Mihalcea, 2016c). A Figura 3 demonstra a comunicação utilizando JDBC evidenciando a independência de fornecedor.

Figura 3 – Comunicação JDBC



Fonte: Mihalcea (2016c)

Pode-se inferir através da Figura 3, que o JDBC facilita a troca de fornecedor de banco de dados da aplicação. Porém, segundo Keith *et al.* (2018), apesar do JDBC ser portátil, a linguagem SQL não é, pois dependendo do fornecedor do banco de dados a sintaxe pode ser diferente, exigindo uma refatoração nas instruções existentes para adequar ao novo fornecedor.

2.2 ORM

Ao receber o resultado de uma consulta utilizando JDBC com SQL nativo, é necessário uma série de passos até por fim percorrer o resultado e realizar a transformação de forma manual para um objeto em Java. Esse é um processo considerado de baixo nível e, geralmente, desenvolvedores estão mais interessados na implementação da regra de negócio que utilizará esses dados. Consequentemente, preferem escrever um código que salva e recupera instâncias de classes sem a preocupação de acesso e conversão de dados (Bauer *et al.*, 2016).Segundo Keith *et al.* (2018) ao comparar uma classe utilizada para representar

um modelo de domínio com uma tabela do modelo relacional, observa-se que se assemelham em muitos aspectos, possibilitando uma forma de fazer com que ambas se comuniquem. Essa possibilidade deu origem ao ORM, conceituado por Bauer & King (2005) como uma tecnologia que, utilizando metadados que representam o mapeamento entre objetos e o modelo relacional do banco de dados, permite a persistência de objetos em tabelas de um banco de dados relacional de forma automatizada e transparente. Para Chen *et al.* (2014), o ORM fornece uma abstração conceitual aos desenvolvedores mapeando registros do banco de dados em objetos. Este mapeamento simplifica para o desenvolvedor acessar dados, sem se preocupar com detalhes de SQL. Por exemplo, é possível através do comando (`usuario.setNome("Joao")`) atualizar a tabela `usuario` que esta mapeada na classe do objeto instanciado. Para Bauer & King (2005), esta simplificação facilita principalmente o desenvolvimento em operações *Create, Read, Update and Delete* (CRUD). A tecnologia ORM é aplicada por meio de *frameworks* no desenvolvimento de software. Um *framework* ORM consiste em quatro partes: (Bauer & King, 2005)

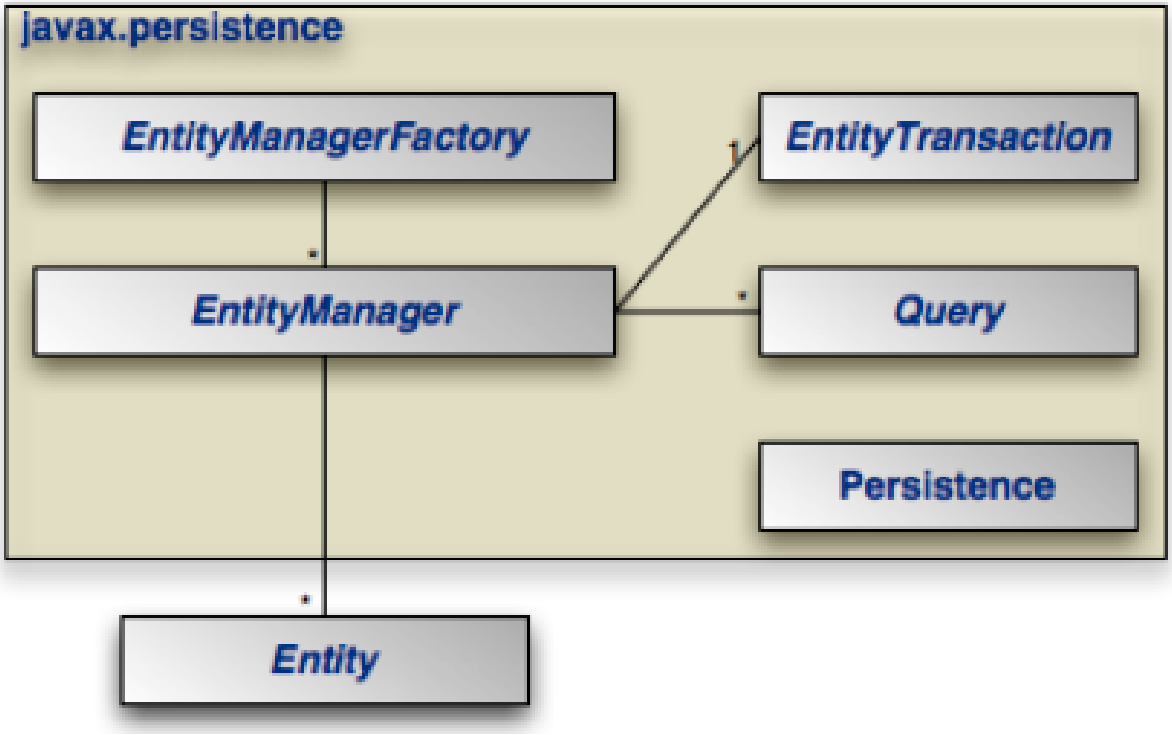
- Uma API responsável por executar as operações CRUD em objetos de classes que representam entidades da base de dados;
- Uma linguagem ou API para realizar instruções SQL acessando os atributos e propriedade de classes;
- Um recurso para especificar os metadados relacionando as classes as entidades, podendo ser anotações ou por *Extensible Markup Language* (XML);
- Técnicas para interação com objetos transacionais para executar otimizações automáticas como verificação suja, busca de associação lenta, entre outras.

Nesta seção, será verificado que em Java, os *frameworks* ORM seguem uma especificação padrão chamada JPA. Com base nesta especificação, será analisado como são realizados os mapeamentos ORM, as estratégias de busca possíveis através do mapeamento e quais os principais *frameworks* que realizam a implementação JPA.

2.2.1 JPA

Java Persistence API (JPA) é uma especificação Java com objetivo de estabelecer um padrão para uso de ORM para os diferentes *frameworks* Java (EclipseLink, 2019). Cabe salientar que o JPA não é um *framework* ORM. Conforme OpenJpa (2019), ele não pode ser utilizado diretamente por ser uma especificação, devendo ser implementado por algum *framework*. É como uma *interface* com um contrato a ser seguido.

Figura 4 – Arquitetura JPA,



Fonte: OpenJpa (2019)

A Figura 4 apresenta a arquitetura da API JPA, que conforme OpenJpa (2019), possui os seguintes componentes:

- *Persistence:* a classe `javax.persistence.Persistence` contém métodos auxiliares estáticos que permitem obter instâncias de `EntityManagerFactory`;
- *EntityManagerFactory:* `javax.persistence.EntityManagerFactory` é uma fábrica para a classe `EntityManager`;
- *EntityManager:* a classe `javax.persistence.EntityManager` é a principal interface JPA utilizada pelas aplicações. Cada `EntityManager` gerencia um conjunto de objetos persistentes. Também utilizado como fábrica para instância de consultas (*query*);
- *Entity:* são os objetos persistentes mapeados representando as tabelas do banco de dados;
- *EntityTransaction:* importante para manter a integridade de dados, os objetos do tipo `EntityTransaction` agrupam operações com dados persistentes de forma a garantir que sejam executadas todas as operações, na ocorrência de qualquer falha, nenhuma será executada. Cada `EntityManager` possui uma relação com um único `EntityTransaction`;

- *Query:* o JPA padroniza o suporte para consultas usando o *Java Persistence Query Language* (JPQL). É utilizada pelos *frameworks* que implementam JPA com o objetivo de localizar objetos persistentes que atendem a determinados critérios.

O JPQL é uma linguagem fornecida pelo JPA em nível de objeto com uma sintaxe próxima ao SQL, com utilização de objetos mapeados para entidades ao invés de tabelas. Linguagens como JPQL auxiliam a troca de fornecedor do banco de dados, evitando os problemas descrito na Seção 2.1.3 referente a diferenças na sintaxe SQL entre fornecedores. Isso porque o JPA converte o JPQL na sintaxe SQL do servidor configurado na aplicação. (EclipseLink, 2019). O JPA como uma especificação ORM, muda a percepção do desenvolvedor Java, preocupando-se menos com instruções SQL ao passo que eleva a importância sobre transições de estado da entidade. A entidade pode estar em um dos seguintes estados (Mihalcea, 2019b):

- *New (Transient)*: é o estado para um objeto de entidade recém-criado, que não é mapeado para nenhuma linha da tabela do banco de dados, sendo considerado um estado novo ou transitório. Quando o objeto for gerenciado (estado *managed*) será emitido uma instrução de inserção pelo contexto e persistência;

- *Managed (Persistent)*: neste estado, o objeto da entidade está persistente no banco, sendo gerenciado pelo atual contexto, ao passo que alterações em qualquer atributo serão detectadas pelo mecanismo de verificação suja do *framework* ORM utilizado e propagada através de instruções de atualização para o banco de dados no momento da liberação do contexto da persistência (*flush*);

- *Detached*: quando o contexto de persistência é encerrado, qualquer alteração nos objetos das entidades gerenciadas até então não serão mais propagadas ao banco, por estarem desanexadas (status *Detached*);

- *Removed*: é o estado de uma entidade removida. Fica no agendamento para exclusão do banco durante a liberação do contexto de persistência.

Para que o JPA identifique um objeto como representante de uma instância de entidade no banco de dados, é necessário definir uma comunicação comum para que o *framework* ORM saiba que determinada classe, atributos, associações, tipos em Java correspondem a determinada tabela, colunas, relacionamentos e tipos SQL. Conforme EclipseLink (2019), o JPA padroniza duas formas de mapeamento: por meio de anotações Java ou por XML, gerando metadados referentes aos mapeamentos. Segundo Bauer *et al.* (2016), a maioria dos desenvolvedores preferem as anotações Java ao invés de arquivos XML como o principal mecanismo para declaração dos metadados.

Antes do Java 5, todos os mapeamentos ORM eram realizados por XML. Uma dificuldade desta abordagem se dava pelo fato do código e as informações referentes ao

mapeamento estarem em dois arquivos distintos, devendo as alterações estarem sempre em sincronia. Com a chegada do Java 5, foi introduzido o conceito de anotações. Isso possibilitou ao JPA utilizar anotações de mapeamento fazendo com que o código e as definições ficassem no mesmo arquivo Java (Janssen, 2019b). Em virtude de ser mais usual pela comunidade de desenvolvedores, iremos utilizar o mapeamento por anotações nos exemplos e definições da presente pesquisa.

Exemplo 5 - Mapeamento de classe de domínio

```
1  @Entity
2  @Table(name = "discente")
3  public class Discente {
4
5   @Id
6   @Column(name = "id_discente")
7   private int id;
8
9   @ManyToOne(fetch = FetchType.LAZY)
10  private Pessoa pessoa;
11  ...
12 }
```

O Exemplo 5 contém algumas das principais anotações referentes aos mapeamentos realizados em uma entidade do modelo de domínio. Para um melhor entendimento, o modelo de domínio da aplicação (também chamado de classes persistentes) é composto pelas classes que se deseja mapear, sendo o componente central no ORM (Mihalcea *et al.*, 2018). Segue definição de cada anotação utilizada no Exemplo 5:

- `@Entity`: informa que uma classe faz parte do contexto das classes persistentes e gerenciadas pelo JPA. Isto significa, segundo Bauer *et al.* (2016), que todos os seus atributos, a partir desta anotação, são automaticamente persistentes;
- `@Table`: é utilizada para fornecer explicitamente o nome da tabela ou especificar outras informações a respeito dela. Se não utilizada, por padrão o JPA identifica o nome da tabela como o nome da entidade (Mihalcea *et al.*, 2018);
- `@Id`: indica qual o atributo será mapeado como identificador único (chave primária) da entidade (EclipseLink, 2019);
- `@Column`: permite especificar explicitamente o nome da coluna ao atributo e adicionar outras informações. Por padrão, o JPA utiliza o nome do atributo, caso não informado (EclipseLink, 2019);
- `@ManyToOne`, `@OneToOne`, `@OneToMany`, `@ManyToMany`: anotações referentes aos relacionamentos entre entidades. Além de definir o relacionamento,

permite também escolher a estratégia de busca das entidades relacionadas (Bauer *et al.*, 2016). No exemplo, é utilizado o `@ManyToOne` para pessoa com a estratégia de busca do tipo LAZY, o que significa que um objeto `Pessoa` pode estar relacionado a um ou mais objetos `Discente` e a informação do relacionamento só será recuperada do banco quando acionada. Mais informações referente a estratégias de busca será visto em sequência.

Em um mapeamento ORM, cada atributo de uma entidade (sendo esse representativo de um relacionamento ou não) possui uma estratégia de busca para recuperação das informações no banco de dados. A estratégia de busca pode estar definida de forma implícita (padrão) ou explícita (definida no mapeamento). Segundo OpenJpa (2019), diversas anotações em JPA permitem definir a estratégia de busca de forma explícita, através da propriedade `fetch`. Esta propriedade pode assumir dois valores: `FetchType.EAGER` ou `FetchType.LAZY`.

A estratégia de busca do tipo EAGER faz com que o campo seja carregado pelo *framework* ORM antes de retornar o objeto persistente completo. Ao recuperar uma entidade de uma consulta ou do `EntityManager`, é garantido que todos os atributos do tipo EAGER sejam preenchidos com dados recuperados do banco de dados (Mihalcea *et al.*, 2018). No caso de um atributo cujo tipo seja um outro objeto (relacionamento), os dados de todos os atributos desse outro objeto serão carregados junto com o objeto alvo, independente se a informação carregada será utilizada ou não (Chen *et al.*, 2016a). Já a estratégia do tipo LAZY, ao contrário do tipo EAGER, traz a informação somente quando acionada, pois nem sempre todas as informações da tabela e relacionamentos serão utilizadas, permitindo que as informações sejam recuperadas do banco de dados somente no momento em que os dados serão necessários (Vial, 2018). O padrão da estratégia de busca difere de acordo com o tipo do atributo. Conforme Keith *et al.* (2018), por padrão os tipos básicos são todos EAGER. São considerados de tipos básicos, de acordo com OpenJpa (2019): `byte[]`, `Byte`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Calendar`, `char[]`, `Character`, `java.util.Date`, `java.sql.Date`, `Serializable`, `Enums`, `java.lang.String`, e `java.sql.Timestamp`.

Contudo, para os tipos que representam relacionamentos, depende do tipo de relacionamento mapeado. A Tabela 2 apresenta a estratégia padrão para cada tipo de relacionamento (Mihalcea, 2019a).

2.3 *FRAMEWORKS* ORM QUE IMPLEMENTAM JPA

Em concordância com o verificado na Seção 2.2.1, devido ao fato de ser uma especificação ORM, o JPA exige *frameworks* para implementá-lo. São exemplos de *frameworks*: EclipseLink (EclipseLink, 2019), Hibernate (Mihalcea *et al.*, 2018) e OpenJPA (OpenJpa, 2019). Atualmente, o *framework* ORM mais utilizado pela comunidade de desenvolvedores Java é o Hibernate, conforme se observa no *ranking* da Figura 5, seguido pelo EclipseLink (considerando

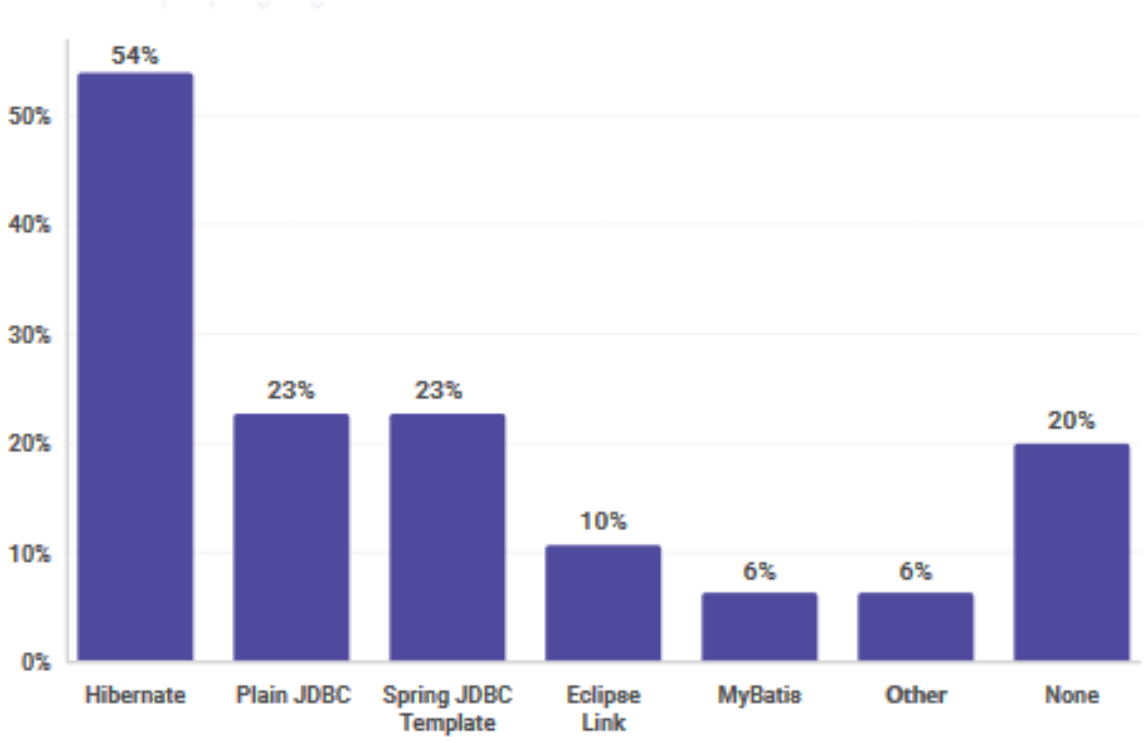
Tabela 2 – Estratégias de busca por relacionamento

Tipo de relacionamento	Estratégia de busca padrão
@OneToMany	LAZY
@ManyToMany	LAZY
@ManyToOne	EAGER
@OneToOne	EAGER

Fonte: Elaborada pelo autor (2020)

apenas *frameworks* ORM que implementam JPA contidos no *ranking*).

Figura 5 – Tecnologias utilizadas para acesso a base de dados em Java



Fonte: Maple & Binstoc (2018)

As diferenças entre os *frameworks* que implementam JPA consistem em configurações específicas, formas que realizam a conexão com o banco de dados, gerenciamento dos estados das entidades persistentes, utilização de cache, padrões de mapeamento, bibliotecas adicionais, entre outros. No estudo realizado por Dhingra *et al.* (2017), são apresentadas as diferenças em cada aspecto, concluindo que o Hibernate é o *framework* JPA que mais se destaca em termos de suporte de documentação e bibliotecas para gerenciamento de responsabilidades.

O Hibernate também disponibiliza uma linguagem própria para realizar instruções ao banco de dados utilizando objetos, chamada de HQL, semelhante ao JPQL do JPA. Para

criar uma consulta HQL é necessário uma *Session* que crie um objeto `Query` do pacote `org.hibernate` (Mihalcea *et al.*, 2019).

Conforme verificado, as implementações JPA reduzem a complexidade de acesso aos dados, permitindo ao desenvolvedor reutilizar o código para obter persistência de dados. Além disso, possibilita um alto nível de abstração na realização de recuperação, atualização e inserção de dados em um banco de dados relacional. A abstração de alto nível, no entanto, pode se tornar um grande problema para desenvolvedores que não possuam um conhecimento adequado do funcionamento do ORM. Um mapeamento ou instrução incorreta em um *framework* ORM, apesar de não representar um problema imediatado, pode gerar uma série de problemas ao sistema de informação. Um indicador dessas situações no código ORM pode ser caracterizado como *code smells*.

2.4 *CODE SMELLS*

O conceito de *Code Smells* (também conhecido por *bad smell* ou somente *smell*) foi introduzido por Martin Fowler a partir do livro *Refactoring: improving the design of existing code* de 1999 (Palomba *et al.*, 2013). O termo surgiu, segundo Fowler (1999), quando o autor estava tentando identificar uma forma de explicar “quando” refatorar determinado trecho de código. Explicar “o que” refatorar é menos complexo do que identificar “quando”, pois o “quando” não existe uma matemática exata e depende de cada contexto. A partir desta incógnita, surgiu o termo *code smell* como forma a utilizar uma analogia com cheiro ruim, explicando que se o código “cheira mal” então existe a possibilidade da necessidade de refatoração. Portanto, o termo *Code smells* pode ser definido como sintomas de más escolhas de design e implementação.

Alguns autores realizaram trabalhos para criar métricas como forma de identificação de *code smells*, como em Munro (2005), Simon *et al.* (2001) e Fontana *et al.* (2015). Para Fowler (2018), que introduziu o termo, nenhum conjunto de métricas rivaliza com a intuição humana. Com base na experiência, é possível procurar certas estruturas no código (*code smells*) que sugerem refatoração a fim de melhorar sua estrutura, manutenibilidade e evitar outros futuros problemas. São exemplos de *code smells*, segundo Fowler (2018):

- Código duplicado: trechos de código duplicados geram retrabalhos quando necessária manutenção, possibilitando erros devido a ausência de correção em alguma das duplicações. A solução neste caso é extrair o código duplicado para um método e efetuar a chamada para o mesmo método em ambos os lugares. O Exemplo 6 demonstra a ocorrência do *code smell* realizando operações semelhantes nos métodos `fazalgo1` e `fazalgo2`, sendo refatorados para um novo método, `fazalgo` com as operações internas extraídas também em novos métodos: `verificaValor(teste)` com objetivo de imprimir a mensagem caso valor falso;

`executaOperacao(operacao)` com objetivo de executar a operação encaminhada no parâmetro.

Exemplo 6 - Código duplicado Smell

```
1  public class CodigoDuplicadoSmell {
2   public void fazalgo1(Boolean teste) {
3    if (teste) executaOperacao1();
4    else System.out.println("Valor falso");
5   }
6   public void fazalgo2(Boolean teste) {
7    if (teste) executaOperacao2();
8    else System.out.println("Valor falso");
9   }
10 }
11
12 public class CodigoDuplicadoRefatorado {
13  public void fazalgo(Boolean teste, Operacao operacao) {
14     verificaValor(teste);
15     executaOperacao(operacao);
16  }
17 }
```

- Agrupamentos de Dados: grupo de dados que costumam ser vistos sempre juntos, seja em atributos de classes ou como parâmetros em assinaturas de métodos. Nessa situação, é recomendável convertê-los em um objeto. O Exemplo 7 demonstra este *code smell* através da classe `AguramentoDados` contendo métodos que utilizam o mesmo conjunto de parâmetros, sendo possível criar um objeto para os parâmetros (`ParamOperacao`) de forma a agrupar as informações em um objeto.

Exemplo 7 - Agrupamento de Dados Smell

```
1  public class AgrupamentoDados {
2   ...
3   executaOperacao1(Boolean teste, String descricao, Integer valor);
4   executaOperacao2(Boolean teste, String descricao, Integer valor);
5  }
6
7  public class AgrupamentoDadosRefatorado {
8   ...
9   ParamOperacao param = New ParamOperacao(teste, descricao, valor);
10  executaOperacao1(param);
11  executaOperacao2(param);
12 }
```

O conceito de *code smells* foi utilizado em vários contextos diferentes. Grigera *et al.* (2014) utilizou o conceito para tratar *code smells* em usabilidade; Elssamadisy & Schalliol

(2002) aborda *code smells* dentro da metodologia *extreme programming*; Cunha *et al.* (2012) desenvolveu um catálogo de *code smells* referente a planilhas; Fard & Mesbah (2013) verificou *code smells* na linguagem Javascript. A presente pesquisa irá focar em *code smells* dentro do contexto ORM em Java, com o objetivo de criar um catálogo e uma ferramenta que será a aplicação deste catálogo. Esta terá o objetivo de detectar *code smells* ORM no código com base em análise estática. A ferramenta proposta foi desenvolvida como um *plugin* para o detector SpotBugs o qual será descrito a seguir.

2.5 SPOTBUGS

No desenvolvimento de software, existem diversas ferramentas e processos que visam auxiliar o desenvolvedor a descobrir problemas antes do sistema entrar em produção. O uso de ferramentas de análise estática é uma delas, sendo uma poderosa técnica para encontrar erros durante o desenvolvimento de software (Tomassi, 2018). As ferramentas de análise estática utilizam técnicas que de forma automatizada varrem o código fonte do software procurando padrões de problemas previamente programados (Terra & Bigonha, 2008). Uma das ferramentas de análise estática de maior utilização pelos desenvolvedores de software, conforme White (2014), é o SpotBugs.

O SpotBugs (antigo FindBugs) é um software livre, distribuído sob os termos da *GNU Lesser General Public License*, utilizado para encontrar padrões de erros no código fonte através de análise estática do código (SpotBugs, 2018b). Segundo Ayewah *et al.* (2008), o projeto *FindBugs* iniciou através de observação, passou a experimento e evoluiu para uma ferramenta amplamente utilizada. Na fase de observação, foi verificado que erros encontrados nos softwares desenvolvidos em Java eram detectáveis com técnicas de análise corriqueiras. Foram realizados experimentos mostrando que até mesmo softwares em produção criados por desenvolvedores experientes continham estes erros.

Atualmente, é possível utilizar o SpotBugs como uma *Graphical User Interface* (GUI), por linha de comando, como uma tarefa do *ANT* ou como um *plugin* para: Eclipse, Maven ou Graddle. O SpotBugs verifica *bytecode* (arquivos `.class`) gerado após a compilação. Atualmente é homologado para execução em projetos com Java 8 ou superior, já o funcionamento em projetos com Java 11 ou mais recente ainda é experimental. O SpotBugs permite ainda desenvolver *plugins* com novos padrões de erros personalizados para detecção utilizando a arquitetura padrão[1] (SpotBugs, 2018b). Utilizaremos esta arquitetura inicial para estender e a partir das definições do catálogo de *code smells* ORM, desenvolver uma ferramenta como *plugin* para o SpotBugs de forma a detectar automaticamente code smells ORM.

[1] https://github.com/spotbugs/spotbugs-archetype

3 METODOLOGIA

Com base nos objetivos da pesquisa, coletamos evidências através de uma combinação de RR e GLR para criar o catálogo de ORM *code smells*. Utilizamos RR para fornecer evidências rapidamente através de revisão sistemática sobre problemas ou anti-padrões, causas e soluções relacionados a ORM na literatura acadêmica, enquanto a GLR foi realizada para obter evidências relacionadas a literatura cinza de forma a abranger também documentações oficiais dos *frameworks* ORM, *blogs* de autores das documentações entre outras publicações não científicas. O resultado das evidências de ambas as técnicas estabelece nosso catálogo de ORM *code smells* em Java. As seções a seguir fornecem mais detalhes sobre os protocolos da RR e da GLR, como método, problema prático, questões de pesquisa para condução das revisões, estratégia de pesquisa, procedimentos de seleção, resultado e síntese.

3.1 MÉTODO

Seguimos o procedimento descrito por Cartaxo *et al.* (2018) para condução da RR. Os autores definem RRs como estudos secundários a partir de adaptações de revisões sistemáticas regulares feitas para atender às restrições dos profissionais, como tempo e custos, e utilizados de forma a obter uma conexão maior com a prática da engenharia de software. A realização da RR tem como principal objetivo fornecer evidências de forma a auxiliar a tomada de decisão referente a problemas que profissionais enfrentam habitualmente. Na Seção 1.2, apresentamos um exemplo de problema real causado por *code smells ORM* utilizados na instituição de atuação do pesquisador, sendo que a configuração prática da pesquisa justifica a utilização da RR para verificar evidências na literatura. Segundo Cartaxo *et al.* (2018), apesar de ser uma forma mais rápida para transferir conhecimento comparada a Systematic Literature Review (SLR), a utilização de RRs possui rigor científico, deve ser conduzida de forma sistemática, seguindo um protocolo definido, não sendo consideradas revisões de literatura *ad-hoc*.

Segundo Garousi *et al.* (2019), dependendo da natureza da pesquisa pode ser desejável expandir a revisão de forma a utilizar literatura cinza. Uma revisão GLR é um tipo particular de SLR que permite incluir literatura cinza como fonte primária, além da literatura (formal) publicada. Permite incluir fontes, como *white papers*, *blogs*, documentações, entre outras fontes não científicas, de forma a complementar lacunas da literatura formal fornecendo perspectivas “atuais”. A Tabela 3 apresenta questões para verificar a necessidade de incluir ou não literatura cinza em revisões de engenharia de software. Se uma ou mais respostas forem positivas, é relevante a inclusão de literatura cinza na revisão da pesquisa de engenharia de software.

Diante da natureza prática da pesquisa, grande parte das questões referentes a Tabela 3 podem ser respondidas como afirmativas, tais como em `Q1` conforme observado por Chen

Tabela 3 – Questões referentes a inclusão de literatura cinza nas revisões de engenharia de software

Ref	Alternativas
Q1	O assunto é complexo e não solucionável, considerando apenas a literatura formal?
Q2	Existe falta de volume ou qualidade de evidência ou falta de consenso na literatura formal?
Q3	As informações contextuais são importantes para o sujeito em estudo?
Q4	O objetivo é validar ou corroborar resultados científicos com experiências práticas?
Q5	O objetivo é desafiar suposições ou falsificar resultados da prática?
Q6	Uma síntese de ideias ou evidências da comunidade industrial e acadêmica seria útil para ambas?
Q7	Existe um grande volume de fontes de profissionais que indicam alto interesse por um tópico?

Fonte: Garousi *et al.* (2019)

et al. (2016b) expondo a falta de estudos acadêmicos sobre as principais causas de problemas ORM, e em Q4 conforme objetivos citados na Seção 1.3.1. Deste modo, justifica-se a inclusão de fontes encontradas em literatura cinza utilizando GLR.

3.2 PROBLEMA PRÁTICO

Conforme apresentado na Seção 1.2, foi identificado que o sistema de gestão desenvolvido em Java da instituição de ensino que o pesquisador trabalha apresenta problemas relacionados a codificação ORM. Esses problemas poderiam ter sido evitados, caso fosse de conhecimento dos desenvolvedores quais são os *code smells*, ou práticas que devem ser evitadas ao realizar a codificação ORM. Para identificar esses *code smells*, é necessário investigar quais os problemas, causas e soluções referente a ORM.

3.3 QUESTÕES DE PESQUISA PARA CONDUÇÃO DA RR E GLR

Com base no objetivo da pequisa de criar um catálogo de *code smells* ORM em Java, formulamos questões para encontrar evidências de problemas, causas e soluções referentes a ORM de forma a orientar tanto a RR quanto a GLR:

- RQ1: Quais são os problemas, anti-padrões relacionados a ORM na linguagem Java?
- RQ2: Quais são as causas, *code smells* para os problemas relacionados a ORM em Java?
- RQ3: Quais são as soluções sugeridas para problemas relacionados a ORM em Java?

RQ1 tem por objetivo coletar evidências de problemas relacionados a ORM. **RQ2** tem o propósito de verificar as causas desses problemas para auxiliar a identificar *code smells*

a serem relacionados para o catálogo. O **RQ3** busca identificar a solução para ser utilizada como forma de sugestão para correção dos *code smells* relacionados.

3.4 ESTRATÉGIA DE BUSCA

Para a revisão RR, utilizamos o mecanismo de pesquisa de Scopus[1] conforme recomendado por Cartaxo *et al.* (2020), de forma a abreviar o tempo de busca por estudos primários e conduzir mais rapidamente o processo de encontrar as evidências. Apesar de ser apenas um mecanismo, o Scopus realiza a pesquisa em diversas bibliotecas digitais relevantes de engenharia de software de forma a abranger um maior número de estudos. Foram realizados testes com diferentes padrões de busca até encontrar um conjunto que resultasse em fontes relevantes em relações aos `RQs`. Por fim, a RR foi conduzida utilizando o seguinte padrão de busca:

Tabela 4 – Padrão de Busca RR

(”ORM” OR ”hibernate” OR ”JPA” OR ”eclipseLink” OR ”openJPA”)
AND
(problem* OR *smell OR anti-pattern* OR ”performance” OR ”maintainability”)
AND
(”software develop*” OR ”software engineering” OR ”software project” OR ”developer”)

Fonte: Elaborada pelo autor (2020)

Conforme Garousi *et al.* (2019), para cobrir a parte de literatura cinza com GLR, uma das formas recomendadas como estratégia de busca possíveis para engenharia de software é utilizar um mecanismo de pesquisa geral na web, sendo assim utilizamos o Google. Após testes, o padrão de busca foi modificado de forma que o algoritmo de busca do Google apresentasse os resultados mais relevantes de acordo com os RQs. Alguns coringas utilizados na busca foram modificados ou adaptados. A última parte do padrão de busca *(”software develop*” OR ”software engineering” OR ”software project” OR ”developer”)* utilizada na RR foi removida pelo fato do algoritmo do Google conseguir identificar pelas primeiras partes que se trata de uma busca referente a engenharia ou desenvolvimento de software, apresentando nos testes resultados mais relevantes aos RQs sem esta parte. A GLR foi realizada utilizando o seguinte padrão de busca:

3.5 PROCEDIMENTO DE SELEÇÃO

O procedimento de seleção utilizou os seguintes critérios de inclusão tanto para a RR quanto para a GLR:

[1] `https://scopus.com`

Tabela 5 – Padrão de Busca GLR

(ORM | hibernate | JPA | eclipseLink | openJPA) AND (smell | problem | ”anti-pattern” | performance | maintainability)

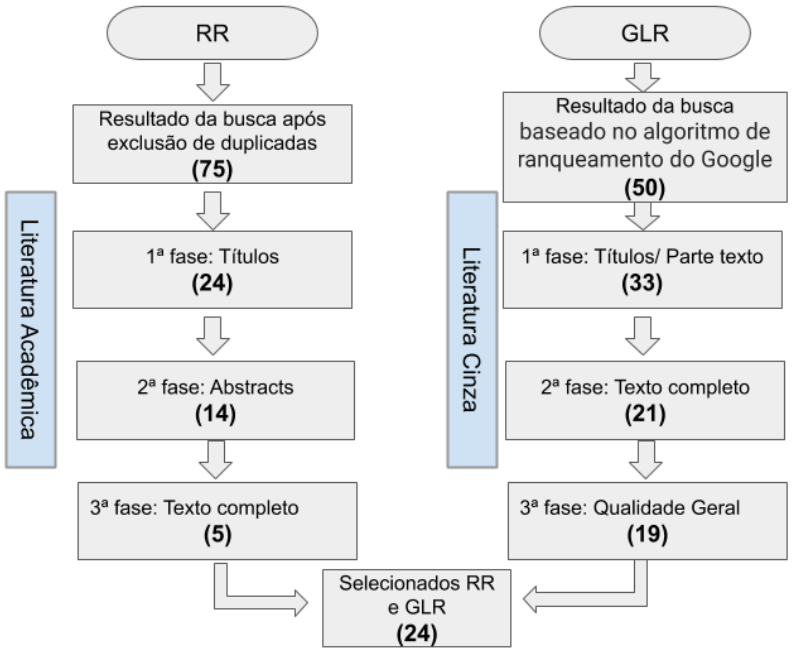
Fonte: Elaborada pelo autor (2020)

1. Fontes diretamente relacionadas ao ORM, no contexto de engenharia de software.
2. Fontes com problemas ou anti-padrões ORM que não são detectados durante tempo de compilação.
3. Fontes que forneçam respostas para pelo menos um dos `RQs`.
4. Fontes a partir de 2010 em diante.

O critério 2 traz uma restrição para selecionar somente evidências que as soluções sejam candidatas a *code smells*. O critério 4 foi adicionado para evitar evidências antigas, anteriores a versão 2.1 da especificação JPA lançada em dezembro de 2009.

3.5.1 Resultado

Figura 6 – Resultado do procedimento de seleção



Fonte: Elaborada pelo autor (2020)

A Figura 6 apresenta o resultado das revisões realizadas através da RR e GLR. Na RR, a busca utilizando o Scopus retornou 75 fontes já desconsiderando repetidos. Fontes cujo título claramente não estava relacionado aos RQs ou que não estava de acordo com os critérios inclusão, como fontes com data inferior a 2010, foram removidos nessa primeira fase permanecendo 24 fontes. Na segunda fase, após análise dos resumos (*abstracts*), as fontes foram reduzidos para um total de 14. Ao fim, com a verificação do estudo completo, foram selecionados 5 fontes. O procedimento de seleção está disponível *online*.[2] Na GLR a consulta com o padrão de busca definido retornou mais de 71 milhões de registros, sendo que definimos como critério de parada da busca o esforço limitado, conforme recomendado por Garousi *et al.* (2019). Utilizamos as primeiras 50 fontes relevantes com base no algoritmo de ranqueamento do Google para análise subsequente. Na primeira fase utilizamos título e texto parcial disponível nos resultados da pesquisa. Assim como na RR, foram verificados os critérios de inclusão e o relacionamento com os RQs, após esta etapa, 33 fontes foram selecionados. Na segunda fase foi analisado o texto completo em cada fonte sendo 21 delas selecionadas. Para aumentar a confiabilidade das fontes, na terceira fase de seleção as fontes foram avaliadas quanto a qualidade geral em termos de confiabilidade da fonte, clareza, detalhe, consistência, plausibilidade e alinhamento com os RQs, a exemplo de avaliação de qualidade referente a literatura cinza na revisão realizado em Tom *et al.* (2013). As fontes que não atingiram um limite mínimo de qualidade foram excluídas, por exemplo, fontes sem detalhes sobre o assunto ou com explicações pouco claras sendo que esse limite não foi pré-determinado devido à natureza diversa da GLR necessitando de avaliação e consideração caso a caso. Nesta fase, excluímos uma fonte por falta de plausibilidade (Fabrício, 2015). Ele alegou ter descoberto uma maneira não documentada de melhorar o desempenho da JPA, mas alguns dos exemplos citados já estavam na documentação do Hibernate (Mihalcea *et al.*, 2018). Outro foi excluído por falta de detalhes (Mat, 2011) referente a um questionamento do StackOverflow, cuja resposta apontou para uma fonte encontrada pela GLR (Mihalcea, 2016a). Igualmente a RR, o procedimento de seleção está disponível *online*.[3] As fontes selecionadas das revisões realizadas podem ser verificados na Tabela 6.

3.6 PROCEDIMENTO DE SÍNTESE

Apesar de os critérios de inclusão utilizados na RR incluírem fontes a partir de 2010, a primeira fonte selecionada foi publicado em 2013 (Węgrzynowicz, 2013), sendo 2016 o ano com mais fontes no total de dois (Chen *et al.*, 2016a,b). Quatro fontes tem o mesmo autor como principal (Chen *et al.*, 2014; Chen, 2015; Chen *et al.*, 2016a,b) sendo que este último indica uma falta de estudos referente a soluções de problemas ORM, comprovado a partir da RR realizada. Referentes aos problemas ORM, dados em excesso são discutidos em (Chen

[2] `https://bit.ly/2uwn6rv`
[3] `http://bit.ly/39Amjom`

et al., 2014; Chen, 2015; Chen *et al.*, 2016a,b; Węgrzynowicz, 2013) sendo o problema mais citado na RR. O segundo problema mais citado foi o N+1, sendo discutidos nas fontes (Chen *et al.*, 2014; Chen, 2015; Chen *et al.*, 2016a).

Na GLR, foram selecionadas 19 fontes, sendo três documentações oficiais dos *frameworks* ORM (Mihalcea, 2017; Mihalcea *et al.*, 2018; Oracle, 2015), 14 postagens de blogs, sendo oito delas (Mihalcea, 2016a, 2019a; Janssen, 2015; Mihalcea, 2019b, 2020b; Janssen, 2017b, 2018, 2020) relacionadas a autores de documentações oficiais dos *frameworks* ORM, um fórum de desenvolvedores Java (Ahmed, 2018) e uma referente a apresentação de um dos autores da documentação oficial do *framework* Hibernate (Mihalcea, 2016b). O problema mais citado foi o N+1 com 18 fontes, apenas uma fonte (Janssen, 2020) não comentou sobre o tema. O segundo, foi o excesso de dados com 16 citações, apenas (Janssen, 2017b, 2018, 2015) não abordaram o tema. Outros problemas foram relacionados, alguns de forma descritiva, outros de forma categorizada.

3.7 RELATÓRIO

Para apresentar os resultados das revisões realizadas (RR e GLR) com respostas aos RQs, foi utilizado *Evidence Briefing*, que é um documento relatando em uma página as principais descobertas de uma revisão, especialmente no caso das RRs (Cartaxo *et al.*, 2016). A Figura 7 apresenta o *Evidence Briefing* realizado para relatar a RR e GLR. O documento também está disponível *online* [4]

Figura 7 – Evidence Briefing

Fonte: Elaborada pelo autor (2020)

[4] https://bit.ly/31iEKfF

Tabela 6 – Estudos selecionados RR e GLR

Ref.	Estudo	Tipo
RR01	Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks (Chen *et al.*, 2016a)	Artigo científico
RR02	An empirical study on the practice of maintaining object-relational mapping code in Java systems (Chen *et al.*, 2016b)	Artigo científico
RR03	Improving the quality of large-scale database-centric software systems by analyzing database access code (Chen, 2015)	Artigo científico
RR04	Detecting performance anti-patterns for applications developed using object-relational mapping (Chen *et al.*, 2014)	Artigo científico
RR05	Performance antipatterns of one to many association in hibernate (Węgrzynowicz, 2013)	Artigo científico
GLR01	Performance Features | EclipseLink 2.7 (Mihalcea, 2017)	Documentação Oficial
GLR02	Hibernate performance tuning tips (Mihalcea, 2019b)	Blog
GLR03	7 Tips to boost your Hibernate performance (Janssen, 2020)	Blog
GLR04	The best way to prevent JPA and Hibernate performance issues (Mihalcea, 2020b)	Blog
GLR05	Performance Improvement in Java Applications: ORM / JPA (Gomes, 2016)	Blog
GLR06	Solve Hibernate Performance Issues in Development (Janssen, 2017b)	Blog
GLR07	Hibernate Performance Tuning and Best Practices (Mihalcea, 2016a)	Blog
GLR08	3 Common Hibernate Performance Issues in Your Log (Janssen, 2018)	Blog
GLR09	9 Oracle TopLink (EclipseLink) JPA Performance Tuning (Oracle, 2015)	Documentação Oficial
GLR10	JPA: N+1 SELECT problem [Spring & JPA pitfalls series] (Marciniec, 2019)	Blog
GLR11	Avoiding JPA Performance Pitfalls | Veracode (Hut, 2015)	Blog
GLR12	Boost the performance of your Spring Data JPA application (Durix, 2020)	Blog
GLR13	Java Persistence API: Otimizando a performance das aplicações (Medeiros, 2015)	Blog
GLR14	Hibernate and the n+1 selections problem (Munhoz, 2019)	Blog
GLR15	Documentation - 5.4 - Hibernate ORM (Mihalcea *et al.*, 2018)	Documentação Oficial
GLR16	JPA Tips: Avoiding the N + 1 select problem - Java Code Geeks (Ahmed, 2018)	Forum
GLR17	Mastering JPA Performance - RainFocus (Mihalcea, 2016b)	Apresentação
GLR18	EAGER fetching is a code smell when using JPA and Hibernate (Mihalcea, 2019a)	Blog
GLR19	How to Improve JPA Performance (Janssen, 2015)	Blog

Fonte: Elaborada pelo autor (2020)

4 CATÁLOGO DE CODE SMELLS ORM

A tecnologia *ORM* apresenta diversos benefícios, conforme averiguado no Capítulo 2, com abstração das comunicações pertinentes a camada de dados e conversão entre objetos e o banco de dados relacional, permitindo ao desenvolvedor um foco maior no desenvolvimento das regras de negócio. Todavia essa abstração pode trazer problemas de desempenho e manutenibilidade se as configurações, mapeamentos e instruções ORM forem realizadas de forma inadequada. Nesta seção, apresentamos o catálogo de *code smells* ORM em Java.

Conforme verificado na Seção 2.4, a definição do que representa um *code smell* é decorrente de uma análise subjetiva baseada na experiência e na intuição humana. Para que uma prática seja considerada um *code smell* considerando o domínio específico de ORM em Java, ela deve indicar uma má escolha de implementação e sugerir sintomas que podem ser indicativos de algo errado no código, indicando a necessidade de refatoração. Portanto, com base nos resultados das revisões (RR e GLR) discutidas no Capítulo 3, extraímos más práticas relacionadas a ORM em Java. O catálogo possui o foco em Java devido à linguagem possuir uma especificação padrão para ORM através da API JPA que permite categorizar de uma mesma maneira *code smells* ORM em diferentes *frameworks*. Foram selecionados *code smells* cujo conteúdo possui pelo menos três citações em fontes diferentes nas revisões realizadas, resultando em oito *code smells* ORM.

Este capítulo apresenta os *code smells* selecionados agrupados por tipos. Os tipos foram definidos como os possíveis problemas causados pelo conjunto de *code smells* relacionados. Para uniformização, cada *code smell* ORM está estruturado em: *descrição do smell*, com uma descrição resumida do *code smell* ORM bem como exemplo prático; sugestão de refatoração, apresentando as melhores práticas encontradas nas revisões realizadas e um exemplo prático corrigindo o apresentado na descrição do *smell*; por fim, uma discussão e detalhamento referente ao *code smell* em questão. A Tabela 7 apresenta os oito *code smells* ORM selecionados para o catálogo, contendo um breve resumo e categorizados por tipo de problema, seguidos pelas referências das evidências do RR e GLR (Tabela 6) a partir dos quais os extraímos.

4.1 DADOS EM EXCESSO

Segundo Mihalcea *et al.* (2018), trazer dados em excesso do banco de dados para a aplicação pode ser considerado o problema de desempenho número um na maioria das aplicações que utilizam JPA. Chen *et al.* (2016a) relata que o fato do ORM operar em um nível inferior (acesso a dados), não permite identificar quais dados serão utilizados no retorno e assim não fornece uma abordagem ideal de recuperação de dados para todos os casos, podendo gerar problemas de desempenho ao recuperar dados em excesso do banco de dados. A Figura 8 apresenta um exemplo ao requerer apenas a matricula do objeto `discente`, a

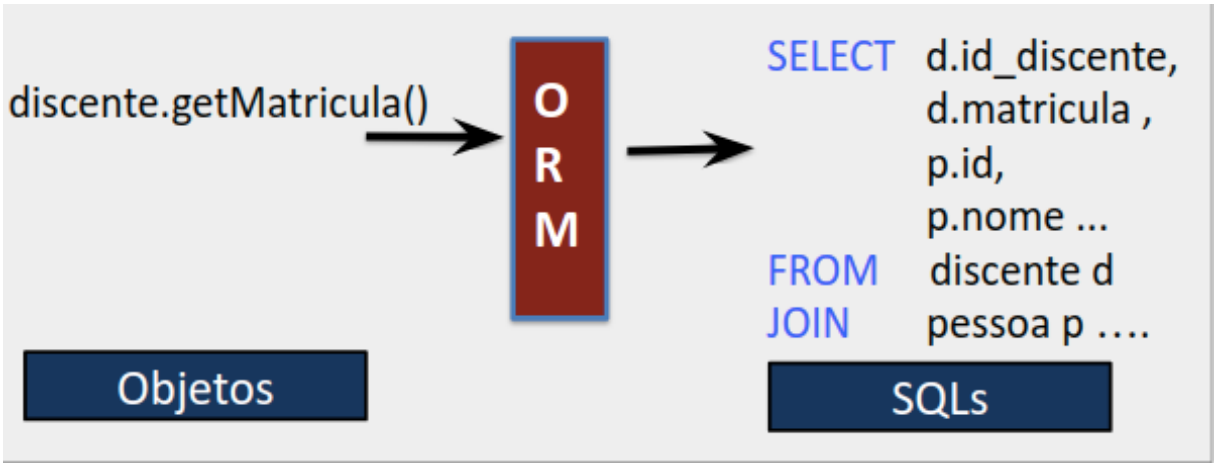
Tabela 7 – Catálogo de Code Smells ORM

Tipo	Code Smell ORM	Referências da Revisão	Resumo
Dados em excesso	1. EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE	[RR01], [RR03], [RR04], [GLR02], [GLR03], [GLR04], [GLR05], [GLR07], [GLR09], [GLR10], [GLR11], [GLR12], [GLR13], [GLR14], [GLR15], [GLR16], [GLR17], [GLR18].	Atributos que representam relacionamento mapeados com estratégia de busca EAGER a nível de classe (estático) faz o objeto relacionado sempre ser carregado pelo ORM, mesmo quando não utilizado, sem poder alterar o comportamento a nível de consulta (dinâmico)
	2. RECUPERAÇÃO DE DADOS SEM PROJEÇÃO PARA SOMENTE LEITURA	[RR01], [GLR02], [GLR03], [GLR05], [GLR17].	Não utilização de projeção ou DTOs para recuperar somente os atributos desejados do banco de dados quando o resultado for utilizado apenas para leitura, faz com que sejam recuperados dados em excesso.
	3. ATUALIZAÇÃO DESNECESSÁRIA DE TODA A ENTIDADE	[RR01], [RR02], [RR03].	Qualquer alteração de atributos na classe faz com que todas as colunas da entidade mapeada sejam atualizadas no banco de dados, quando seria necessário atualizar apenas as colunas dos atributos alterados.
	4. FALTA DE PAGINAÇÃO QUANDO NÃO NECESSÁRIO TODO O RESULTADO	[GLR01], [GLR02], [GLR05], [GLR13], [GLR17].	Consultas que retornam coleção de dados sem a utilização dos parâmetros de paginação para limitar os resultados quando não utilizado todos os registros.
N+1	5. FALTA DE JOIN FETCH: ESTRATÉGIA DE BUSCA EAGER	[RR01], [RR04], [GLR02], [GLR04], [GLR05], [GLR06], [GLR07], [GLR09], [GLR10], [GLR11], [GLR12], [GLR13], [GLR14], [GLR15], [GLR16], [GLR17], [GLR18].	Consultas ORM sem utilização de JOIN FETCH para os atributos que estão mapeados como EAGER a nível de classe, gerando N consultas adicionais para carregar os objetos relacionados.
	6. ACESSO UM POR UM: ESTRUTURA DE REPETIÇÃO E ESTRATÉGIA DE BUSCA LAZY	[RR03], [RR04], [GLR01], [GLR05], [GLR08], [GLR10], [GLR11], [GLR12], [GLR13], [GLR14], [GLR16], [GLR19].	Ao recuperar um atributo de um objeto referente a uma entidade dentro de uma estrutura de repetição, se a estratégia de busca for LAZY, consultas adicionais serão executadas para cada iteração.
	7. @ONETOMANY UNILATERAL COM USO INADEQUADO DE COLEÇÕES	[RR05], [GLR07], [GLR15].	`@OneToMany` unilateral com uso inadequando de coleções Java faz com que a cada inserção/remoção de um elemento seja modificado todos os registros da coleção no banco de dados.
Outros	8. NÃO USO DE CONSULTAS SOMENTE LEITURA	[GLR01], [GLR02], [GLR15].	Objetos recuperados da base de dados para fins únicos de consultas sem que possuam configurações para somente leitura, faz com que seja gerenciado pelo contexto de persistência desnecessariamente.

Fonte: Elaborada pelo autor (2020)

consulta gerada pelo *framework* ORM busca mais informações que a desejada no banco de dados. A seguir são apresentados os *code smells* relacionados a este problema.

Figura 8 – Dados em Excesso



Fonte: Elaborada pelo autor (2020)

4.1.1 Eager como estratégia de busca nos relacionamentos a nível de classe (estático)

4.1.1.1 Descrição do Smell

Utilização da estratégia de busca EAGER (anotação `FetchType.EAGER`) nos atributos que representam relacionamentos a nível de classe, ou a falta de anotação `FetchType.LAZY` para relacionamentos com anotação `@ManyToOne` e `@OneToOne` que por padrão utilizam a estratégia de busca EAGER, faz com que o objeto relacionado sempre seja recuperado do banco de dados mesmo quando não utilizado, podendo causar desperdício de processamento computacional, problemas de desempenho e manutenção pois não permite a alteração da estratégia de busca em nível de consulta (dinâmico). O Exemplo 8 demonstra a ocorrência do *smell* utilizando a anotação `@ManyToOne` com a estratégia de busca do tipo EAGER. O objetivo do código é recuperar apenas a matrícula de um discente, porém recupera também a entidade `Pessoa` sem necessidade através de JOIN.

4.1.1.2 Sugestão de Refatoração

A estratégia de busca definida para relacionamentos a nível de classe deve ser do tipo LAZY, conforme recomendações encontradas nas revisões realizadas, evitando assim carregar o objeto relacionado para todas as regras de negócio que utilizam o objeto da classe em questão. Com a utilização de LAZY, o ORM irá recuperar o objeto relacionado somente quando alguma informação do mesmo for solicitada. Se for necessário que o objeto do relacionamento seja recuperado com o comportamento EAGER para alguma regra de negócio de forma a evitar

Exemplo 8 - Eager Smell

```
1  @Entity
2  class Discente{
3   @ManyToOne(fetch = FetchType.EAGER)
4   private Pessoa pessoa;
5   ...
6  }
7
8  public static void main(String[] args) {
9   ...
10  Discente d = findDiscenteById(1);
11  d.getMatricula();
12 }
```

Instrução SQL gerada:

```
SELECT * FROM Discente d LEFT OUTER JOIN Pessoa p
ON     p.id_pessoa = d.id_pessoa WHERE   d.id=1;
```

consultas adicionais, deve ser realizado o tratamento em nível de consulta, realizando JOIN FETCH na consulta para a regra de negócio ou através da anotação `@NamedEntityGraph` introduzida no JPA 2.1[1] que permite definir um grafo de entidades a serem recuperadas no banco de dados. O Exemplo 9 apresenta a refatoração do Exemplo 8, alterando a estratégia de busca para LAZY, evitando assim o JOIN desnecessário com a entidade `Pessoa`.

Exemplo 9 - Ajuste Eager Smell

```
1  @Entity
2  class Discente{
3   @ManyToOne(fetch = FetchType.LAZY)
4   private Pessoa pessoa;
5   ...
6  }
7  public static void main(String[] args) {
8   ...
9   Discente d = findDiscenteById(1);
10  d.getMatricula();
11 }
```

Instrução SQL gerada:

```
SELECT * FROM Discente d WHERE d.id=1;
```

4.1.1.3 Detalhamento e Discussão

Uma das causas de excesso de dados em *frameworks* ORM é a utilização indevida da estratégia de busca do tipo EAGER. Essa estratégia é utilizada para carregar previamente as

[1] `https://docs.jboss.org/hibernate/jpa/2.1/api/`

informações do banco de dados de um objeto relacionado. O mapeamento na forma estática utiliza uma anotação a nível de classe no atributo que representa outro objeto, de forma a sempre carregar o objeto relacionado. Por exemplo: objeto A possui um atributo relacionado ao objeto B. Utilizando EAGER, sempre que qualquer atributo do objeto A for carregado, o objeto B também será, independente se alguma informação do objeto B será utilizada. (Bauer & King, 2005)

Segundo a documentação oficial do Hibernate, a utilização da estratégia de busca do tipo EAGER de forma estática (definida a nível de classe) é quase sempre uma má escolha (Mihalcea *et al.*, 2018). Esta estratégia é considerando um *code smell* ORM devido aos seguintes riscos:

- Os dados do objeto relacionado sempre serão carregados por completo, ainda que não sejam utilizados. Assim, poderá acarretar em um processamento desnecessário ao banco de dados, carregando sempre o objeto relacionado. O problema pode se agravar caso o objeto relacionado também esteja com outros objetos associados utilizando EAGER, gerando mais consultas desnecessárias. (Chen *et al.*, 2014);
- Se durante uma consulta ORM não for realizado o JOIN FETCH para todas os relacionamentos anotados como EAGER, será realizada uma sub-consulta para cada relacionamento, podendo gerar o problema do $N + 1$ (Mihalcea *et al.*, 2018).
- Ao definir um relacionamento na classe para EAGER não poderá sobrescrever o comportamento para LAZY em nível dinâmico (nível de consulta). Já o contrário (de LAZY para EAGER) é possível utilizando JOIN FETCH na consulta (Mihalcea, 2017).

Chen *et al.* (2014) relata que após realizada a refatoração alterando a estratégia de busca de EAGER para LAZY em uma classe com 300 registros associada a outra com 10 registros, houve um aumento de 71% de desempenho. Para um dos autores responsáveis pela documentação do Hibernate, Mihalcea (2019b) a estratégia de busca EAGER definida de forma estática é um *code smell*. Na maioria das vezes é utilizado pelo desenvolvedor para simplificar o desenvolvimento, porém não são considerados os problemas de desempenho e manutenibilidade a longo prazo.

Apesar da documentação oficial da versão 5 do Hibernate (Mihalcea *et al.*, 2018) e da documentação oficial do EclipseLink (Oracle, 2015) recomendar a não utilização, atualmente EAGER é a estratégia de busca padrão para mapeamentos do tipo `ManyToOne` e `OneToOne` nos *frameworks* ORM em Java. O motivo, segundo Mihalcea *et al.* (2018), é a implementação do JPA. Antes da especificação JPA, o Hibernate por exemplo definia todos os seus mapeamentos com a estratégia de busca LAZY por padrão. Com o surgimento da especificação JPA 1.0, pensava-se que nem todos os provedores que iriam implementar JPA usariam *proxies*, podendo causar erros ao buscar uma informação de forma LAZY em uma

conexão fechada, lançando a exceção `LazyInitializationException`. Como são *frameworks* ORM que implementam o JPA, seguem a especificação, porém recomendam em suas documentações oficiais que os mapeamentos do tipo `ManyToOne` e `OneToOne` devem ser explicitamente definidos como LAZY. As demais estratégias de busca são LAZY por padrão. Chen *et al.* (2016a) relata que relacionamentos `ManyToOne` e `OneToOne` apesar de representar um relacionamento de associação única, podem gerar grandes problemas de desempenho utilizando EAGER quando o objeto recuperado contém grandes dados (por exemplo, binários ou imagens) ou o objeto recuperado também recupera de forma EAGER muitos outros objetos relacionados). O problema se agrava quando associações do tipo `OneToMany` e `ManyToMany`, que representam coleções e por padrão são do tipo LAZY, são definidas explicitamente com a estratégia de busca do tipo EAGER. Com essa estratégia, a consulta pode exigir um produto cartesiano limitando a velocidade da consulta a quantidade de registros associados, ou gerando o problema de $N + 1$ (o qual será visto na próxima seção) com N consultas adicionais (Mihalcea, 2016c).

Com base no exposto, consideramos um *code smell* a utilização da estratégia de busca EAGER a nível de classe (forma estática). Seu uso pode não gerar problemas caso a classe sempre necessite do objeto relacionado e este possua dados pequenos na base de dados. Mesmo nestes casos, é considerado um *code smell* pois é complexo ter certeza no momento do desenvolvimento que todos os casos de usos futuros precisarão do objeto do relacionamento e que a base de dados se manterá da mesma forma, sendo recomendado utilizar LAZY a nível de classe e utilizar o comportamento EAGER a nível de consulta através de JOIN FETCH para os que necessitarem. Fazendo uma relação com o apresentado na Seção 2.4 é possível concluir que é uma indicação de refatoração, uma má escolha de implementação e indica possíveis problemas futuros relativos a desempenho e manutenibilidade. Portanto, qualquer atributo em uma classe que representa uma associação entre entidades contendo qualquer anotação de relacionamento (`@ManyToMany`, `@OneToMany`, `@ManytoOne`, `@OneToOne`) com o `FetchType.EAGER` definido, ou, em específico para relacionamentos MANYTOONE e ONETOONE que não estejam explicitamente com `FetchType.LAZY` anotados, são considerados *code smells* para o contexto deste trabalho. Os casos dos relacionamentos MANYTOMANY e ONETOMANY são considerados como mais graves, devido ao *Many* (Muitos) ao fim do relacionamento, o que faz com que a consulta de um objeto apenas, trará sempre N outros objetos do relacionamento.

4.1.2 Recuperação de dados sem projeção para somente leitura

4.1.2.1 Descrição do Smell

Utilizar consultas ORM sem a utilização de projeção quando a finalidade for somente leitura (quando nenhum gerenciamento do contexto de persistência é necessário) pode ocasionar problemas de desempenho, especialmente ao recuperar informações grandes como

colunas do tipo BLOB ou junções com outras entidades de forma desnecessária, gerando incompatibilidade entre dados necessários e dados recuperados. O Exemplo 10 demonstra a ocorrência do *smell*, onde o HQL utilizado recupera todas as colunas da entidade `Discente`, incluindo dados grandes (coluna `arquivo`), sendo apenas a informação da matrícula necessária para o caso de uso.

Exemplo 10 - Projeção Smell

```
1  public Discente findDiscenteById(Integer id){
2   String hql = "FROM Discente d WHERE id = :id";
3   Query q = entityManager.createQuery(hql.toString());
4   q.setInteger("id", id);
5   return (Discente) q.uniqueResult();
6  }
7
8  public static void main(String[] args) {
9   ...
10  Discente d = findDiscenteById(1);
11  d.getMatricula();
12 }
```

Instrução SQL gerada:

```
SELECT d.id, d.matricula, d.arquivo, d.data_cadastro, d...
FROM   Discente d WHERE d.id=1;
```

4.1.2.2 Sugestão de Refatoração

Para evitar trazer todos os dados da entidade, a recomendação é utilizar projeção nas consultas. Uma das formas de realizar projeção em consultas ORM é utilizar o operador `NEW`. Esse operador permite declarar apenas as colunas ou atributos que são necessários carregar do banco de dados através do construtor da classe consultada ou de um *Data Transfer Object* (DTO). O Exemplo 11 demonstra um ajuste no Exemplo 10 utilizando o operador `NEW` para trazer o próprio objeto apenas com a informação necessária para o caso de uso.

4.1.2.3 Detalhamento e Discussão

Realizar consultas sem projeção pode ocasionar problemas de desempenho e manutenção, trazendo mais informação do que necessário. Quando realizada uma consulta sem projeção, o *framework* ORM não sabe quais dados serão necessários posteriormente, trazendo todas as colunas da entidade e seus relacionamentos (dependendo da estratégia de busca). Essa prática pode recuperar colunas que contenham informações grandes do banco de dados (como uma imagem), ou muitos relacionamentos gerando um processamento desnecessário pelo banco e pela aplicação (Chen *et al.*, 2016a). Se for necessário atualizar os dados posterior a recuperação, é importante buscar a entidade completa para ser gerenciada pelo *framework*

Exemplo 11 - Ajuste Projeção Smell

```
1  public Discente findDiscenteById(Integer id){
2   ...
3   hql.append("SELECT new Discente(matricula) ")
4   hpl.append("FROM Discente WHERE id = :id");
5   Query q = entityManager.createQuery(hql.toString());
6   q.setInteger("id", id);
7   return (Discente) q.uniqueResult();
8  }
9
10 public static void main(String[] args) {
11  ...
12  Discente d = findDiscenteById(1);
13  d.getMatricula();
14 }
```

Instrução SQL gerada:

```
SELECT d.matricula FROM Discente d WHERE d.id=1;
```

ORM, isso devido ao chamado *mecanismo de verificação suja*, conforme discutido na Seção 2.2.1 (Mihalcea *et al.*, 2018).

No entendimento de Mihalcea *et al.* (2018), se o objetivo for recuperar informações somente leitura, é recomendável utilizar DTO manipulando somente os atributos desejados para o caso de uso. Essa estratégia traz alguns benefícios, como reduzir a carga no contexto de persistência em execução devido a projeções realizadas em DTOs não precisarem ser gerenciadas pelo *framework* ORM, além de evitar que atributos que estejam mapeados com a estratégia de busca EAGER na entidade sejam carregados.

Analisando o exposto até então, poderia-se sugerir como recomendação de refatoração o uso de DTOs. Todavia, o uso do padrão DTO para este contexto não é um consenso. Apesar das vantagens dessa abordagem, para Bauer & King (2005) utilizar DTOs com objetivo de recuperar informação do banco de dados ao invés da própria entidade poderia causar outras situações indesejadas. Segundo o autor, o uso do padrão DTO, neste contexto, poderia adicionar dois *code smells* elencados por Fowler (2018):

- *Shotgun change smell*: Uma pequena alteração em alguma parte do código requer alterações em várias classes. No caso do DTO, se a classe da entidade for alterada, os DTOs que utilizam atributos daquela entidade também deverão ser alterados.
- *Parallel class hierarchies*: Duas hierarquias de classes distintas contêm classes semelhantes em uma correspondência de um para um. Nesse caso, fica evidente a hierarquia de classes paralelas. Utilizando DTO, teríamos por exemplo: `Usuario` e `UsuarioDTO`, `Discente` e `DiscenteDTO`.

Entretanto, ao analisar apenas o aspecto de desempenho, o DTO pode apresentar uma superioridade por não ser gerenciado pelo contexto do JPA. Segundo Janssen (2019a), o

fato das entidades serem gerenciadas pelo JPA faz com que sejam armazenadas no cache de primeiro nível. Isso evita instruções repetidas e otimiza gravações no banco de dados. Porém, o gerenciamento de cache de primeiro nível leva tempo e pode ser um problema se executadas grandes quantidades de consultas em entidades.

Perante o exposto, podemos catalogar como *code smells* consultas ORM que não utilizam projeção e são utilizadas somente para consulta, pois podem gerar problemas futuros de desempenho e manutenibilidade pelo excesso de colunas desnecessárias. Consideramos como *code smell* apenas consultas ORM declaradas, pois consultas usando o `find` do `EntityManager` ou geradas automaticamente a partir do nome do método são muito comuns e geralmente menos problemáticas por utilizar apenas uma entidade. No caso de um objeto de retorno ser atualizável, também não deve ser considerado como *code smell*. Como recomendação, podemos citar a utilização de projeções utilizando a própria classe da entidade ou DTOs, deixando o desenvolvedor escolher a melhor estratégia para cada contexto.

4.1.3 Atualização desnecessária de toda a entidade

4.1.3.1 Descrição do Smell

Atualização de todas as colunas de uma entidade, quando somente um atributo do objeto é atualizado, acarreta em atualização de dados em excesso e pode gerar problemas de desempenho, especialmente quando a entidade possui índices não clusterizados, dados grandes binários ou um grande número de colunas. O Exemplo 12 demonstra o *smell* no qual é desejado atualizar apenas o campo `observação` da entidade `Discente`, mas a instrução `Update` gerada pelo ORM atualiza todas as colunas da entidade, incluindo índices e tipo BLOB.

4.1.3.2 Sugestão de Refatoração

Sugere-se utilizar parametrizações nos *frameworks* ORM para que realizem atualização apenas das alterações realizadas. No *Hibernate* por exemplo é possível utilizar a anotação `@DynamicUpdate` ou realizar instruções ORM (HQL/JPQL ou Criteria) para atualizar somente as colunas desejadas. Alguns *frameworks* como o EclipseLink apresentam esta parametrização por padrão. O Exemplo 13 apresenta a utilização da anotação `@DynamicUpdate` fazendo com que seja atualizado apenas o atributo `observacao` na instrução `Update` gerada pelo ORM.

4.1.3.3 Detalhamento e Discussão

Com relação a atualização de todos os campos da entidade, existe uma distinção dependendo do *framework* ORM utilizado. Utilizando o Hibernate, é atualizado o objeto inteiro por padrão, mesmo que a atualização seja realizada apenas em um atributo. Isso ocasiona

Exemplo 12 - Update Smell

```
1  @Entity
2  class Discente{
3  ...
4   @Column
5   @Index
6   private Long matricula;
7
8   @Column
9   private UploadedFile arquivo;
10  ...
11 }
12
13 public static void main(String[] args) {
14  ...
15  Discente d = findDiscenteById(1);
16  d.setObservacao("Discente com pendencias");
17  entityManager.persist(d);
18 }
```

Instrução SQL gerada:

```
UPDATE Discente
SET    id = 1, matricula = 111, observacao = "Discente com pendencias",
       arquivo = "arquivo BLOB"
WHERE  id = 1;
```

Exemplo 13 - Ajuste Update Smell

```
1  @Entity
2  @DynamicUpdate
3  class Discente{
4  ...
5   @Column
6   @Index
7   private Long matricula;
8
9   @Column
10  private UploadedFile arquivo;
11  ...
12 }
13
14 public static void main(String[] args) {
15  ...
16  Discente d = findDiscenteById(1);
17  d.setObservacao("Discente com pendencias");
18  entityManager.persist(d);
19 }
```

Instrução SQL gerada:

```
UPDATE discente SET observacao = "Discente com pendências" WHERE id = 1;
```

processamento desnecessário ao banco de dados, principalmente em entidades fortemente indexadas e com grande número de colunas (Chen *et al.*, 2016a). O mesmo não se aplica ao padrão do EclipseLink, que realiza a atualização somente dos campos modificados. No EclipseLink a atualização de todos os campos somente será realizada de forma explicita pelo programador através de SQL ou JPQL (EclipseLink, 2017). Porém, para Mihalcea *et al.* (2018), o fato do Hibernate realizar a atualização com todos os atributos possui algumas vantagens:

- Permite que sejam utilizados recursos de cache;
- Permite a atualização em lotes, mesmo se várias entidades modificarem propriedades diferentes.

Mesmo com estes fatores, quando a entidade possui índices não clusterizados, os quais o banco de dados poderá atualizá-los de forma redundante, recomenda-se não atualizar todos os campos (Chen *et al.*, 2016b). Para isso, o Hibernate disponibiliza uma anotação a nível de classe chamada `@DynamicUpdate`. Esta anotação faz com que sempre seja atualizado apenas o campo alterado na entidade representada pela classe (Mihalcea *et al.*, 2018).

Com base no exposto, podemos considerar *code smell* quando possuir as seguintes premissas: o *framework* ORM em uso utiliza a atualização de toda entidade por padrão; a classe da entidade não possuir configuração para atualização apenas dos atributos alterados, como `@DynamicUpdate` do Hibernate; possuir índices não clusterizados relacionados a entidade. Nesse caso, conforme exposto na introdução deste capítulo referente a definição de um *code smell* ORM, indica uma necessidade de refatoração sendo uma prática não recomendada.

4.1.4 Falta de paginação quando não necessário todo o resultado

4.1.4.1 Descrição do Smell

Recuperar entidades completas do banco de dados sem especificar parâmetros de paginação quando os registros não são totalmente utilizados, implica em recuperação de dados em excesso causando uma sobrecarga desnecessária, além de utilizar um alto consumo de memória para armazenar a informação recuperada gerando problemas de desempenho. O Exemplo 14 apresenta a ocorrência do *Smell* recuperando todos os registros da entidade `Discente` que entraram no ano de 2020, mas utilizando efetivamente apenas 10 registros que foram solicitados pela camada de visão, supondo que o método `discentesPeriodo` foi chamado com este propósito.

4.1.4.2 Sugestão de Refatoração

A especificação JPA disponibiliza os seguintes métodos: `setFirstResult(int)` e `setMaxResults(int)` fornecidos pelas interfaces `Query` ou `TypedQuery` provendo

Exemplo 14 - Paginação Smell

```
1  public List<Discente> findByYear(int year){
2   String hql = "FROM Discente d WHERE ano = :ano";
3   Query q = entityManager.createQuery(hql.toString());
4   ...
5  }
6
7   public List<Discente> discentesPeriodo(int year,int page,int limit) {
8   int fromIndex = (page - 1) * limit;
9   List<Discente> discentes = findDiscenteByYear(year);
10  return discentes.subList(fromIndex, Math.min(fromIndex + limit,
        discentes.size()));
11 }
```

Instrução SQL gerada:

```
SELECT d.* FROM Discente d WHERE ano = 2020;
```

recursos de paginação que podem ser utilizados ao realizar uma consulta ORM. Desta forma é possível realizar uma limitação nos resultados que solicitam uma coleção de dados. O Exemplo 15 apresenta refatoração do Exemplo 14 utilizando uma consulta paginada para gerar a instrução SQL com limites para o resultado, recuperando apenas os 10 registros da entidade `Discente` solicitados pela camada de visão, evitando problemas de desempenho com o crescimento dos dados.

Exemplo 15 - Correção Paginação Smell

```
1  public List<Discente> findByYear(int year, int fromIndex, int limit){
2   String hql = "FROM Discente d WHERE ano = :ano";
3   Query q = entityManager.createQuery(hql.toString());
4   q.setFirstResult(fromIndex);
5   q.setMaxResults(limit);
6   ....
7  }
8
9  public List<Discente> discentePeriodo(int year,int page,int limit) {
10  int fromIndex = (page - 1) * limit;
11  return List<Discente> discentes = findDiscenteByYear(year,fromIndex,
        limit);
12 }
```

Instrução SQL gerada:

```
SELECT d.* FROM Discente d WHERE ano = 2020 LIMIT 10 OFFSET 10;
```

4.1.4.3 Detalhamento e Discussão

Segundo Mihalcea (2019b) para obter o melhor resultado nas instruções SQL geradas pelo ORM, é importante não apenas considerar o número de colunas, consultas e junções

realizados, mas também o número de registros que esta sendo recuperado do banco de dados. Para recuperar apenas um conjunto específico de dados, é possível configurar elementos como a paginação especificando um limite para quantidade de registros na execução de uma consulta (Mihalcea, 2017) através dos seguintes métodos disponibilizados pela especificação JPA (Gomes, 2016):

- `setFirstResult(int)`: Parâmetro utilizado para especificar o deslocamento;
- `setMaxResults(int)`: Parâmetro utilizado para especificar o limite.

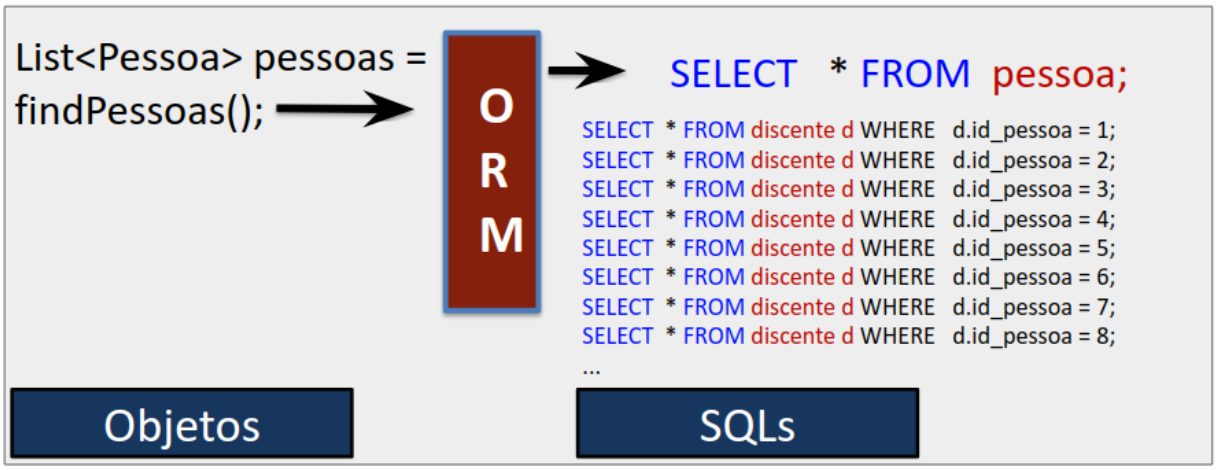
Na camada de visualização, as informações geralmente são apresentadas ao usuário em um formulário paginado, apresentando uma relação natural com a parametrização referente a paginação de dados ORM (Medeiros, 2015). Exemplo: Uma entidade possui 1.000.000 registros. Em uma pagina web é apresentado um subconjunto de informações exibindo 100 registros. Portanto os parâmetros `setFirstResult(1)` e `setMaxResults(100)` devem ser utilizados para recuperar apenas os registros utilizados. Outro agravante em não utilizar a paginação, é que os dados tendem a crescer com o tempo (Mihalcea, 2019b). Em um primeiro momento pode não apresentar problemas de desempenho mas com o aumento de dados pode se tornar um grande problema. Também dificulta a manutenção do código, pois o ajuste implica em alterações na estrutura dos métodos conforme Exemplo 15.

Com base no exposto, podemos considerar um *code smell* ORM a não utilização de paginação em coleções de dados quando os registros recuperados não serão integralmente utilizados, sendo uma má escolha de implementação, cabível de refatoração por indicar possíveis problemas futuros relativos a desempenho com o crescimento da quantidade de registros.

4.2 PROBLEMA DO N+1

Os *frameworks* ORM abstraem o SQL e o mapeamento manual dos objetos, simplificando o desenvolvimento especialmente em operações de criação, consulta, atualização e destruição (CRUD). Mas isso tem um preço, e conforme a estratégia de busca adotada em cada situação, o *framework* ORM pode precisar realizar mais consultas a partir de uma consulta para completar a informação de seus relacionamentos e montar o objeto desejado e seus relacionamentos. Essa ação pode causar um problema de desempenho dependendo da quantidade de consultas que são realizadas para completar a operação (Mihalcea, 2019d). Esse problema é conhecido como $N+1$, quando a partir de uma instrução ORM são geradas N outras, sendo apresentado no Capítulo 1 como exemplo motivacional desta pequisa. A Figura 9 apresenta um exemplo no qual ao solicitar uma lista do objeto `Pessoa`, a partir da consulta principal (`select * from pessoa`) são geradas N outras consultas para obter o relacionamento com a entidade `discente`.

Figura 9 – N+1



Fonte: Elaborada pelo autor (2020)

Através da análise do código, podemos detectar *code smells* que podem gerar o problema de $N+1$, os quais são descritos a seguir.

4.2.1 Falta de JOIN FETCH nas consultas de atributos do tipo EAGER

4.2.1.1 Descrição do Smell

Consultas ORM utilizando objetos que contenham atributos de relacionamento com a estratégia de busca do tipo EAGER e sem realizar o JOIN FETCH para o objeto relacionado, faz com que o *framework* ORM realize N consultas até que todos os objetos associados aos atributos do tipo EAGER fiquem com os dados completos gerando o problema do $N+1$. O Exemplo 16 apresenta um código no qual uma consulta para retornar uma lista da entidade `Discente` apresenta N consultas adicionais para entidade `Pessoa` realizadas pelo ORM para preencher as informações do objeto `Pessoa` que esta com relacionamento EAGER.

4.2.1.2 Sugestão de Refatoração

Para corrigir, uma das formas possíveis seria alterar o mapeamento para LAZY. Porém, se não for possível alterar o tipo de estratégia na classe devido a dependências, uma possibilidade de refatoração para evitar o $N+1$ seria realizar JOIN FETCH para todos os atributos EAGER na consulta ORM. Desta forma o *framework* ORM fará junções para trazer as informações dos relacionamentos em uma única consulta em vez de gerar N consultas extras. O Exemplo 17 apresenta a correção para o problema de N+1 gerado pelo Exemplo 16 adicionando JOIN FETCH para recuperar em forma de junção a entidade `Pessoa`.

Exemplo 16 - *Join Fetch Smell*

```
1  @Entity
2  class Discente{
3   ...
4   @ManyToOne(fetch = FetchType.EAGER)
5   private Pessoa pessoa;
6   ...
7  }
8  public List<Discente> findDiscentesByYear(int year){
9   String hql = "FROM Discente d WHERE d.ano = :ano";
10  Query q = entityManager.createQuery(hql.toString());
11  q.setInteger("ano", year);
12  return (List<Discente>) q.getResultList();
13 }
14
15 public static void main(String[] args) {
16  List<Discente> d = findDiscenteByYear(2020);
17  ...
18 }
```

Instrução SQL gerada:

```
SELECT * FROM Discente d WHERE   d.ano=2020;
-- Consultas adicionais geradas:
SELECT * FROM Pessoa p where p.id_pessoa = 1;
SELECT * FROM Pessoa p where p.id_pessoa = 2;
SELECT * FROM Pessoa p where p.id_pessoa = 3;
....
```

4.2.1.3 Detalhamento e Discussão

Utilizar EAGER como estratégia de busca para relacionamentos é um *code smell* como visto na Seção 4.1.1, portanto a melhor abordagem é evitar o uso de EAGER como estratégia de busca a nível de classe. Porém para casos onde se deseja realizar consultas ORM e seja complexo a refatoração no código ao alterar a estratégia de busca (de EAGER para LAZY) na classe devido a dependências, é necessário utilizar a cláusula JOIN FETCH para todos os atributos do EAGER para evitar o problema N + 1 (Mihalcea, 2019b). O problema de dependência mencionado ocorre quando o código existente espera que o objeto relacionado já esteja carregado. A alteração para LAZY pode causar exceções, como `LAZYInitializationException` no caso do Hibernate (Mihalcea *et al.*, 2018). Existem opções boas e ruins de tratar essa exceção. Utilizar o padrão *Open Session in View* ou ativar a opção `hibernate.enable_lazy_load_no_trans` são consideradas anti-padrões e portanto opções ruins para evitar a exceção (Mihalcea, 2020a). Esses são considerados anti-padrões pois tratam apenas os sintomas e não resolvem a causa real do problema (Mihalcea, 2019c). A melhor opção, nesse caso, é buscar todos os relacionamentos necessários antes de fechar o contexto de persistência por meio da cláusula JOIN FETCH (Mihalcea *et al.*, 2018). Por esse motivo, para evitar o $N+1$, a solução recomendada quando não possível realizar a refatoração

Exemplo 17 - Ajuste *Join Fetch Smell*

```
1  @Entity
2  class Discente{
3   ...
4   @ManyToOne(fetch = FetchType.EAGER)
5   private Pessoa pessoa;
6   ...
7  }
8  public List<Discente> findDiscentesByYear(int year){
9   ...
10  hql.append("FROM Discente d ");
11  hql.append("JOIN FETCH d.pessoa p WHERE d.ano = :ano");
12  Query q = entityManager.createQuery(hql.toString());
13  q.setInteger("ano", year);
14  return (List<Discente>)  q.getResultList();
15 }
16
17 public static void main(String[] args) {
18  List<Discente> d = findDiscenteByYear(2020);
19  ...
20 }
```

Instrução SQL gerada:

```
SELECT * FROM Discente d INNER JOIN Pessoa p
ON     p.id_pessoa = d.id_pessoa WHERE d.ano=2020;
```

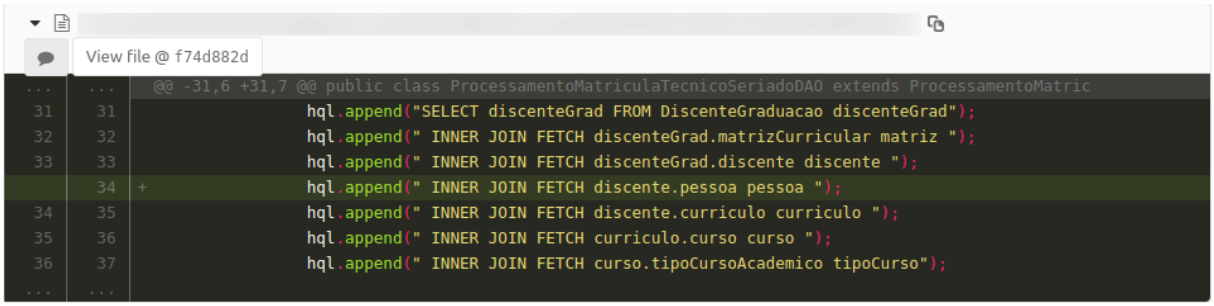
de EAGER para LAZY é utilizar JOIN FETCH nas consultas ORM conforme recomendado na Seção 4.2.1.2.

Com relação a refatoração apresentada utilizando JOIN FETCH é necessário um cuidado e testes após alteração, pois dependendo do contexto o número de junções também pode afetar o desempenho (Bauer *et al.*, 2016), cabendo uma análise do desenvolvedor para avaliar qual a melhor opção. Existem diferenças entre realizar um JOIN comum e um JOIN FETCH. Segundo Janssen (2020) a palavra FETCH da instrução JOIN FETCH é específica da JPA. Ela informa ao provedor de persistência para inicializar também a associação no objeto recuperado e não somente realizar a junção entre as duas entidades. Para exemplificar de forma prática o exposto até o momento, no capítulo 1 utilizamos como motivação desta pesquisa um problema ocasionado pelo HQL do Exemplo 1, e pelo mapeamento realizado no Exemplo 2, que gerava o problema $N+1$ no sistema SIG do IFSC. Foram realizadas duas refatorações que são detalhadas nesta seção utilizando as formas descritas no parágrafo anterior:

- Refatoração 1 - Ajuste das N consultas na entidade `Pessoa`: Uma das soluções possíveis seria alterar as estratégias de busca do tipo EAGER para LAZY em ambos os casos do Exemplo 2, de forma que o ORM buscasse as informações somente quando solicitadas. Seria a melhor estratégia na fase de concepção do sistema, pois atualmente existem centenas de casos de uso que utili-

zam a entidade `DiscenteGraduacao`, dos quais muitos deles esperam que a informação de *Discente* e *Pessoa* já estejam carregadas. Para modificar a estratégia de busca seria necessária uma grande refatoração em todos os pontos do código que as utilizam, tendo em vista que uma chamada ao atributo (`discenteGraduacao.getDiscente().getPessoa()`) em um momento que a sessão que carregou o objeto estiver fechada, iria gerar a exceção `LazyInitializationException`. Isso também evidencia o problema de manutenção causado pelo *smell*. Com base nesses fatores, optou-se por adicionar *JOIN FECTH* na consulta HQL, para continuar recuperando a informação de `Pessoa` mas evitando N consultas extras, utilizando *JOIN* para trazer as informações da entidade `Pessoa` já na consulta principal. Após esta refatoração, a quantidade de consultas adicionais diminuiu de 6767 para 116.

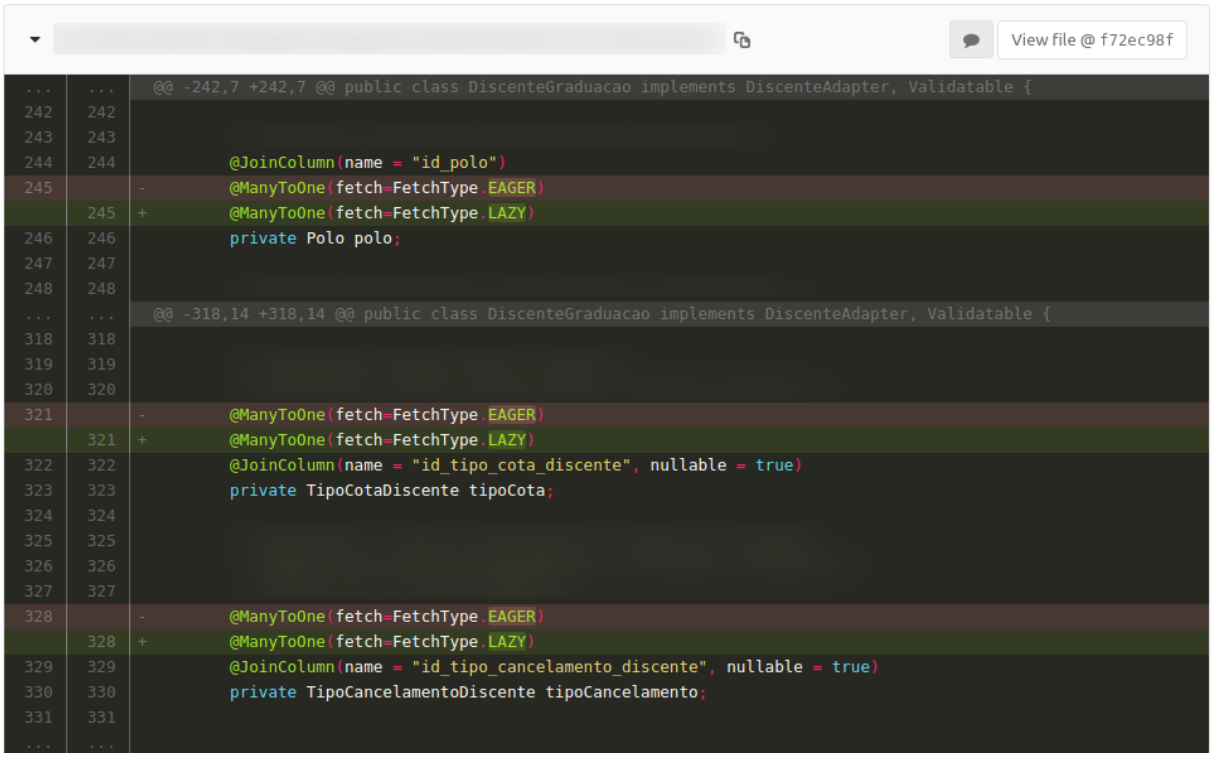
Figura 10 – Refatoração 1



Fonte: Elaborada pelo autor (2019)

- Refatoração 2 - Ajuste da estratégia de busca: Observou-se que as 116 consultas adicionais que restaram eram referentes a atributos EAGER de `DiscenteGraduacao` referentes as entidades relacionadas `Polo`, `TipoCotaDiscente` e `TipoCancelamento`. Observado no código que não possuíam muitos casos de usos que aguardavam essas informações de forma EAGER, sendo possível realizar a alteração para LAZY ajustando alguns poucos casos de usos. Após essa nova refatoração (Figura 11), o tempo de execução da consulta diminuiu de 14 segundos para 11 segundos em média e a quantidade de consultas adiciona diminuiu de 116 para 0.

Figura 11 – Refatoração 2



Fonte: Elaborada pelo autor (2019)

Conforme as situações expostas, consultas ORM utilizando entidades que contenham atributos com a estratégia de busca do tipo EAGER e sem realizar o JOIN FETCH para a entidade relacionada, poderão causar $N+1$, sendo uma prática não recomendada que indica necessidade de refatoração com potencial para gerar problemas de desempenho e manutenibilidade, podendo ser catalogado como um *code smell ORM*.

4.2.2 Acesso um por um: estrutura de repetição e estratégia de busca do tipo LAZY

4.2.2.1 Descrição do Smell

Quando necessário capturar determinado atributo em uma estrutura de coleção de objetos utilizando alguma estrutura de repetição, cujo atributo está com a estratégia de busca do tipo LAZY, em cada passagem pela estrutura de repetição serão geradas consultas adicionais. Essa questão é melhor visualizada a partir do Exemplo 18 que demonstra uma consulta para recuperação de uma lista do objeto `Pessoa` e ao acessar o atributo `discentes` em uma estrutura de repetição, são geradas consultas extras para carregar o relacionamento LAZY.

4.2.2.2 Sugestão de Refatoração

Abaixo listamos duas soluções possíveis para evitar $N+1$ neste caso:

Exemplo 18 - Acesso um por um Smell

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(fetch = FetchType.LAZY)
5   private List <Discente> discentes;
6   ...
7  }
8
9  public List<Pessoa> findAll(){
10  String hql = "FROM Pessoa pes";
11  ...
12 }
13
14 public static void main(String[] args) {
15  List<Pessoa> pessoas = findAll();
16  for (Pessoa pessoa : pessoas){
17   assertNotNull(pessoa.getDiscentes());
18  }
19 }
```

Instrução SQL gerada:

```
SELECT * FROM Pessoa p;
-- Consultas adicionais geradas:
SELECT * FROM Discente d where d.id_pessoa = 1
SELECT * FROM Discente d where d.id_pessoa = 2
SELECT * FROM Discente d where d.id_Pessoa = 3
....
```

- `@BatchSize(N)`: utilizar a anotação `@BatchSize` sobre o atributo do tipo LAZY. Dessa forma em vez de gerar N consultas, o *framework* ORM utilizará o operador `IN` do SQL. Poderá ser definido o número de elementos como parâmetro, definindo o tamanho para o carregamento em lote de coleções ou entidades. O Exemplo 19 apresenta a refatoração do Exemplo 18 utilizando a anotação `@BatchSize`

- JOIN FETCH ou projeções DTO: embora utilizar a anotação `@BatchSize` seja melhor do que carregar os registros um a um, utilizar JOIN FETCH para carregar antecipadamente os objetos ou utilizar DTOs pode ser uma estratégia melhor, sendo possível obter todos os dados em uma única consulta. O Exemplo 20 demonstra a utilização de JOIN FETCH como refatoração para o Exemplo 18

4.2.2.3 Detalhamento e Discussão

Por motivos de desempenho, conforme exposto na Seção 2.2.1, a especificação JPA utiliza as coleções `@OneToMany` e `@ManyToMany` com estratégia de busca LAZY por padrão. Isso significa, segundo Hut (2015), que o contexto de persistência não carregará a entidade de relacionamento até que uma chamada seja feita pelo código. Entretanto, quando o código

Exemplo 19 - Ajuste: Acesso um por um Smell (`@BatchSize`)

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(fetch = FetchType.LAZY)
5   @BatchSize(size = 5)
6   private List <Discente> discentes;
7   ...
8  }
9
10 public static void main(String[] args) {
11  List<Pessoa> pessoas = findAll();
12  for (Pessoa pessoa : pessoas){
13   assertNotNull(pessoa.getDiscentes());
14  }
15 }
```

Instrução SQL gerada:

```
SELECT * FROM Pessoa p;
-- Consultas adicionais geradas:
SELECT * FROM Discente d where d.id_pessoa in (1,2,3,4,5)
SELECT * FROM Discente d where d.id_pessoa in (6,7,8,9,10)
....
```

Exemplo 20 - Ajuste: Acesso um por um Smell (JOIN FETCH)

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(fetch = FetchType.LAZY)
5   private List <Discente> discentes;
6   ...
7  }
8
9  public List<Pessoa> findAll(){
10  String hql = "FROM Pessoa pes JOIN FETCH pes.discentes";
11  ...
12 }
13
14 public static void main(String[] args) {
15  List<Pessoa> pessoas = findAll();
16   for (Pessoa pessoa : pessoas){
17    assertNotNull(pessoa.getDiscentes());
18   }
19 }
```

Instrução SQL gerada:

```
SELECT * FROM Pessoa p INNER JOIN Discente d
ON     p.id_pessoa = d.id_pessoa;
```

solicita os dados, a transação já foi encerrada. Sem a possibilidade de junção, o *framework* ORM realiza consultas adicionais para recuperar a informação (Mihalcea *et al.*, 2018), o que

em uma estrutura de repetição gera um número de consultas em excesso prejudicando o desempenho do sistema. Conforme Marciniec (2019), uma das estratégias de correção muito utilizada e facilmente encontrada no StackOverFlow é a alteração para a estratégia de busca EAGER a nível de classe. Porém, esta resolução substitui um problema por outro, mostrando a dificuldade de manutenção causada pelo *smell*. Conforme comentado na Seção 4.1.1, utilizar EAGER a nível de classe poderá gerar o problema de dados em excesso, carregando mais dados do que o necessário, sendo uma melhor estratégia seguir as refatorações sugeridas na Seção 4.2.2.2.

Podemos concluir que o acesso um por um utilizando uma estrutura de repetição com a estratégia de busca do tipo LAZY é um *code smell* a ser catalogado pelos motivos expostos. As seguintes condições se aplicam para ser considerado um *code smell* ORM desta categoria: estar em uma estrutura de repetição com acesso a um atributo com estratégia LAZY; não ser carregado de forma antecipada na consulta (*JOIN FECTH*); não conter a anotação `@BatchSize` no atributo do relacionamento.

4.2.3 `@OneToMany` unilateral com uso inadequado de coleções

4.2.3.1 Descrição do Smell

Utilizar a anotação `@OneToMany` de forma unilateral (sem o correspondente `@ManyToOne` no outro lado da relação) com o uso inadequado de coleções, como `Collection` ou `List`, poderá gerar o problema de N+1 quando uma alteração em um elemento de uma coleção faz com que todos os registros relacionados sejam removidos e adicionados novamente, dependendo da semântica utilizada. O Exemplo 21 demonstra a ocorrência deste *smell*. Neste exemplo, ao adicionar um novo `Discente` na coleção `discentes` do objeto `Pessoa` (considera-se que existem registros pré-existentes de discentes na entidade representativa da relação `Pessoa_Discente` na base de dados), são geradas instruções adicionais, removendo todos os registros existentes da entidade `Pessoa_Discente` e adicionando-os novamente juntamente com o novo registro.

4.2.3.2 Sugestão de Refatoração

Existem duas formas possíveis de ajustar este *code smell*:

- Transformar em bidirecional: A melhor solução para este *smell* é transformar a relação em bidirecional, adicionando um atributo relacionado ao objeto no outro lado da relação com a anotação `@ManyToOne` informando o atributo da relação com o parâmetro `mappedBy` no `@OneToMany` (Mihalcea, 2016a). Assim, segundo Mihalcea *et al.* (2018) o *framework* ORM irá utilizar o lado com a anotação `@ManyToOne` da relação bidirecional sempre que um dos lados forem manipulados. O Exemplo 22 demonstra o ajuste realizado no Exemplo 21, adicionando a anotação

Exemplo 21 - OneToMany Smell

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(cascade = CascadeType.ALL)
5   private List<Discente> discentes = new ArrayList<Discente>();
6  }
7  @Entity
8  class Discente{
9   @Id
10  private Integer id;
11  private Integer matricula;
12 }
13
14 public static void main(String[] args) {
15  Pessoa pessoa = findById(1);
16  Discente discente = new Discente(5,12345);
17  pessoa1.add(discente);
18  entityManager.persist(pessoa);
19 }
```

Instrução SQL gerada:

```
INSERT INTO Discente (id, matricula) VALUES (5,12345);
DELETE FROM Pessoa_Discente WHERE pessoa_id=1;
INSERT INTO Pessoa_Discente (pessoa_id, discente_id) VALUES (1, 1);
INSERT INTO Pessoa_Discente (pessoa_id, discente_id) VALUES (1, 2);
...
INSERT INTO Pessoa_Discente (pessoa_id, discente_id) VALUES (1, 5);
```

`@ManyToOne`, o atributo responsável pelo relacionamento na classe `Discente` e informando através da anotação `mappedBy` na classe `Pessoa`. Além das instruções geradas pelo ORM serem otimizadas, a estrutura das entidades é modificada ficando a entidade `Discente` com a chave estrangeira relacionado a Pessoa, deixando de ter uma tabela adicional para representar a relação como no Exemplo 21.

- Utilizar a coleção `Set`: Caso não seja possível ou desejável ter uma relação bidirecional, sugere-se trocar o tipo da coleção para a coleção `SET`. Esta coleção tem como característica não armazenar valores duplicados, fazendo com que *framework* ORM altere no banco de dados apenas o registro modificado na coleção. O Exemplo 23 demonstra a refatoração realizada no Exemplo 21 trocando a coleção `List` por `Set`. Desta forma o relacionamento continua sendo representado pela entidade `Pessoa_Discente`, mas a instrução gerada pelo ORM é igualmente proporcional a alteração realizada nos objetos, inserindo apenas o novo registro.

Exemplo 22 - Ajuste OneToMany smell - transformando em bidirecional

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(cascade = CascadeType.ALL, mappedBy = "pessoa")
5   private List<Discente> discentes = new ArrayList<Discente>();
6  }
7  @Entity
8  class Discente{
9   @Id
10  private Integer id;
11  private Integer matricula;
12  @ManyToOne(fetch = FetchType.LAZY)
13  private Pessoa pessoa;
14 }
15
16 public static void main(String[] args) {
17  Pessoa pessoa = findById(1);
18  Discente discente = new Discente(5,12345,pessoa);
19  pessoa1.add(discente);
20  entityManager.persist(pessoa);
21 }
```

Instrução SQL gerada:

```
INSERT INTO Discente (id, matricula, pessoa) VALUES (5,12345,1);
```

Exemplo 23 - Ajuste oneToMany smell com `Set`

```
1  @Entity
2  class Pessoa{
3   ...
4   @OneToMany(cascade = CascadeType.ALL)
5   private Set<Discente> discentes = new HashSet<Discente>();
6  }
7  @Entity
8  class Discente{
9   @Id
10  private Integer id;
11  private Integer matricula;
12 }
13
14 public static void main(String[] args) {
15  Pessoa pessoa = findById(1);
16  Discente discente = new Discente(5,12345);
17  pessoa1.add(discente);
18  entityManager.persist(pessoa);
19 }
```

Instrução SQL gerada:

```
INSERT INTO Discente (id, matricula) VALUES (5,12345);
INSERT INTO Pessoa_Discente (pessoa_id, discente_id) VALUES (1, 5);
```

4.2.3.3 *Detalhamento e Discussão*

De acordo com Węgrzynowicz (2013), a anotação `@OneToMany` em JPA utilizada no lado proprietário (ou seja, unidirecional de forma a gerenciar a persistência) é uma das formas mais comuns de representar associações um-para-muitos entre classes que representam entidades persistentes. A Tabela 8 apresenta três semânticas de coleções utilizadas pelo ORM com a combinação do tipo de coleção Java e anotações do JPA.

Tabela 8 – Três possíveis semânticas com anotação `@OneToMany` em Java

Semânticas	**Tipo Java**	**Anotações**
Semântica `Bag`	`java.util.Collection` `java.util.List`	`@OneToMany`
Semântica `List`	`java.util.List`	`@OneToMany` & (`@IndexColumn` v `@OrderColumn`)
Semântica `Set`	`java.util.Set`	`@OneToMany`

Fonte: adaptado de Węgrzynowicz (2013)

O número e o tipo de instrução executada pelo ORM muda de acordo com a semântica utilizada. A Tabela 9 demonstra o número de instruções geradas ao persistir uma coleção após uma inserção ou remoção dada a semântica utilizada. (Węgrzynowicz, 2013)

Tabela 9 – Número de instruções realizadas a partir de cada semântica utilizada

Semânticas	**Adicionando um elemento**	**Removendo um elemento**
Semântica `Bag`	1 remoção, N inserções	1 remoção, N inserções
Semântica `List`	1 inserção, N atualizações	1 remoção, N atualizações
Semântica `Set`	1 inserção	1 remoção

Fonte: adaptado de Węgrzynowicz (2013)

Segundo Mihalcea (2016a), a utilização do tipo `Set` para representar uma coleção unilateral é a forma mais eficiente, seguido pelo tipo `List` com anotações de ordenação (`@IndexColumn` ou `@OrderColumn`), enquanto tipos da semântica `Bag` representadas por coleções não ordenadas são menos eficiente. Além disso, o fato de existir suporte para coleções em ORM não significa que todo relacionamento de um para muitos em banco de dados deve ser transformado para coleção, algumas vezes uma associação muitos para um (`@ManyToOne`) é suficiente e a coleção pode ser substituída por uma consulta simples na entidade. A utilização de `Set` de acordo com a tabela 9 demonstra ser mais eficiente, pois uma operação na coleção requer apenas uma única instrução no banco de dados evitando o N+1. Entretanto, para Węgrzynowicz (2013) ela não é uma exclusividade e existem alguns casos em que utilizar outros tipos pode ser mais eficaz. Como o uso de `Set` em Java requer uma verificação de exclusividade nos elementos, no caso de qualquer inserção de novos elementos,

todos os elementos da coleção devem ser carregados na memória principal. Sendo assim, casos em que a coleção na maioria das vezes é fortemente modificada com muitas alterações de uma vez, a utilização da semântica `Bag` ou `List` pode se tornar mais eficaz. Porém, para a maioria dos casos `Set` é mais eficiente.

A utilização de coleções em relacionamentos `@OneToMany` unilaterais que utilizam a semântica `Bag` ou `List` (conforme Tabela 8) é considerado *Code Smell* por indicar problemas de desempenho e manutenção de código quando necessária refatoração devido ao aumento de registros na entidade que é representada pela coleção. A troca para utilização de relacionamento bidirecional ou a utilização de Set são estratégias mais eficientes por executar apenas a instrução da modificação da coleção evitando N+1.

4.3 OUTROS *CODE SMELLS* ORM

4.3.1 Não uso de consultas somente leitura

4.3.1.1 Descrição do Smell

Por padrão todas as consultas a entidades em JPA são do tipo READ-WRITE (leitura-gravação) e são gerenciadas pelo contexto de persistência. Recuperar objetos da base de dados utilizando o próprio objeto da entidade para fins únicos de consulta e sem informar ao *framework* ORM que o objeto recuperado não necessita ser gerenciado pelo contexto de persistência, faz com que a busca seja menos eficiente, ocupando espaço desnecessário na memória ao guardar o estado do objeto e deixando a fase de liberação do contexto de persistência mais lenta. O Exemplo 24 apresenta a ocorrência do *smell* onde se deseja obter para fins únicos de consulta o objeto `Discente`, mas é recuperado no modo padrão READ-WRITE do JPA, mantendo o objeto de retorno `Discente` no contexto de persistência desnecessariamente.

Exemplo 24 - Somente Leitura Smell

```
1 public Discente findDiscenteById(Integer id){
2  String hql = "FROM Discente d WHERE id = :id";
3  Query q = entityManager.createQuery(hql.toString(), Discente.class);
4  q.setInteger("id", id);
5  return q.uniqueResult();
6 }
```

4.3.1.2 Sugestão de Refatoração

Existem diferentes configurações possíveis que podem ser utilizadas para informar que o objeto recuperado será utilizado somente para consulta, diferenciando para cada *framework* ORM. Estas configurações podem ser a nível de classe, sessão ou consulta. Outra opção é

a utilização de DTOs em vez de recuperar a entidade mapeada. O Exemplo 25 demonstra a utilização da configuração a nível de consulta para o *framework* ORM Hibernate como refatoração para o Exemplo 24, fazendo com que o objeto `Discente` não seja gerenciado pelo contexto de persistência. Na Seção 4.3.1.3 são detalhadas outras formas possíveis de configuração.

Exemplo 25 - Ajuste Somente Leitura Smell

```
1 public Discente findDiscenteById(Integer id){
2  String hql = "FROM Discente d WHERE id = :id";
3  Query q = entityManager.createQuery(hql.toString(), Discente.class);
4  q.setInteger("id", id);
5  q.setHint("org.hibernate.readOnly", true)
6  return q.uniqueResult();
7 }
```

4.3.1.3 Detalhamento e Discussão

Recuperar entidades no modo somente leitura é mais eficiente do que buscar entidades em modo de leitura e gravação (Mihalcea *et al.*, 2018). É possível informar ao *framework* ORM que determinada consulta é do tipo somente leitura para consultas que retornam objetos representativos de entidades no banco de dados e que não serão modificados posteriormente. Como todas as consultas a entidades em JPA são do tipo READ-WRITE por padrão, modificações no estado da entidade são traduzidas em comandos de atualização (SQL UPDATE). Ao definir que uma consulta é somente leitura, o estado do objeto não será gerenciado pelo *framework* ORM e, dessa forma, não necessita detectar modificações na entidade representada pelo objeto. Além disso, as entidades somente leituras são ignoradas durante o `flush` (Mihalcea, 2019b). As configurações podem ser realizadas a nível de entidade, nível de sessão ou a nível de consulta:

- A nível de entidade: permite que a entidade ou a coleção da entidade seja do tipo somente leitura. Qualquer tentativa de salvar nova informação na base será lançada uma exceção (Janssen, 2017a). Pode ser através da anotação `@Immutable` para o Hibernate (Mihalcea *et al.*, 2018), conforme Exemplo 26, ou anotação `@ReadyOnly` para o EclipseLink (EclipseLink, 2017).

- A nível de sessão: É possível transformar a sessão do `entityManager` em uma sessão que fará consultas somente leitura (Bauer *et al.*, 2016). O Exemplo 27 demonstra a utilização no código fonte.

- A nível de consulta: é possível definir na própria consulta ORM, também com distinções entre os *frameworks* ORM. Para o Hibernate, é possível realizar a con-

Exemplo 26 - Utilização `@Imutable`

```
1 @Entity(name = "parametros") @Immutable
2 public static class Parametros {
3  @Id
4  private Long id;
5  private String nome;}
```

Exemplo 27 - Definição somente-leitura na sessão

```
1 Session session = entityManager.unwrap(Session.class);
2 session.setDefaultReadOnly(true);
```

sulta utilizando `setHint("org.hibernate.readOnly",true)`, conforme já demonstrado no Exemplo 24.

Portanto, se um objeto representativo de uma entidade for recuperado da base para fins únicos de consulta a dados e não esteja definido como somente leitura a nível de entidade, sessão ou consulta será considerado um *code smell* ORM. Manter o objeto gerenciado pelo *framework* ORM desnecessariamente pode ser considerado um *code smell* ORM por indicar uma possibilidade de refatoração por questões de desperdício de recurso computacional.

5 AVALIAÇÃO DO CATÁLOGO DE CODE SMELLS ORM

Para avaliar o catálogo de *code smells* em relação aos *smells* e às soluções propostas, apresentamos o catálogo para desenvolvedores com alguma familiaridade com *frameworks* Java ORM. A pesquisa foi realizada por meio de *survey* e teve a participação de 86 profissionais brasileiros e internacionais. A seguir, detalhamos a metodologia e os resultados da pesquisa.

5.1 METODOLOGIA DA APLICAÇÃO DO *SURVEY*

Utilizamos uma estratégia mista para garantir uma boa generalização da validação e incentivo a participação. Uma das abordagens foi através de *snowballing* (Goodman, 1961), compartilhando o link da pesquisa com profissionais que atendessem os requisitos e incentivando a indicação de outros profissionais adequados para participação. Também compartilhamos em redes sociais como *Twitter* e *Reddit* informando os requisitos para participação. Outra abordagem utilizada foi através de mineração na base de dados do Stack Overflow com a finalidade de obter profissionais que interagiram através de perguntas e respostas referente aos assuntos relacionados com frameworks ORM em Java. Utilizamos a ferramenta Stack Exchange Data Explore[1] e formulamos uma consulta (Exemplo 28) para extração de dados. Após realizado o tratamento para extrair os e-mails do campo *AboutMe*, obtivemos uma lista de 368 e-mails de candidatos a avaliarem a pesquisa. Para fomentar uma maior participação

Exemplo 28 - SQL utilizado para mineração no Stackoverflow

```
SELECT u.id, u.AboutMe, count(*)
FROM Tags t
INNER JOIN PostTags pt ON pt.TagId = t.id
INNER JOIN Posts     p ON p.ParentId = pt.PostId
INNER JOIN Users     u ON u.Id = p.OwnerUserId
WHERE (Upper(TagName) = 'ORM'
OR     Upper(TagName) = 'HIBERNATE'
OR     Upper(TagName) = 'ECLIPSELINK'
OR     Upper(TagName) = 'OPENJPA'
OR     Upper(TagName) = 'JPA')
AND    ((AboutMe LIKE '%@%.%'))
GROUP BY u.id, u.AboutMe
ORDER BY count(*) DESC
```

bem como uma maior abrangência da pesquisa foram desenvolvidos dois questionários, um em português e outro em inglês para ser respondido por pessoas de outras nacionalidades (como os participantes localizados através de mineração no Stack Overflow). O questionário online foi composto por:

- Uma introdução explicando os requisitos para responder à pesquisa.

[1] https://data.stackexchange.com/

- Uma seção demográfica solicitando aos participantes: grau de estudo, anos de experiência em Java, número de aplicações desenvolvidas utilizando ORM.
- Uma introdução explicando o significado de *code smell* ORM.
- Um conjunto de 8 *code smells* ORM e suas soluções com respectivos exemplos em código.
- Um campo para comentários em geral sobre a pesquisa

5.2 FORMA DE AVALIAÇÃO

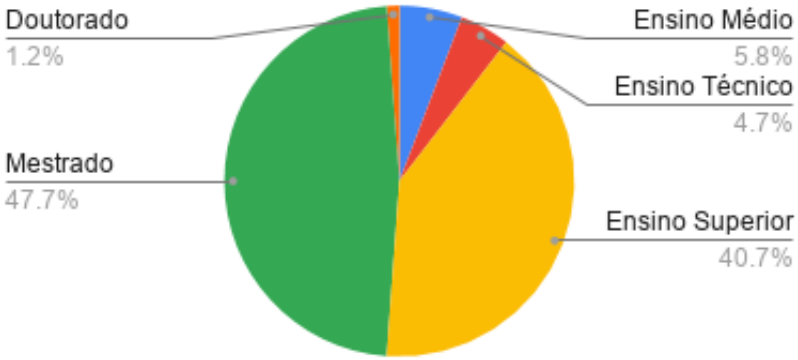
Pedimos que fosse avaliado o nível de concordância para cada *code smell* e o nível de concordância para cada solução apresentada utilizando a escala *Likert* (Oppenheim, 2000) com cinco níveis representados de forma numérica (5 - concordo totalmente, 4 - concordo, 3 - não concordo nem discordo, 2 - discordo, 1 - discordo totalmente) e um campo opcional de comentários permitindo ao avaliador tecer comentários adicionais sobre o *code smell* ou sobre a solução proposta. Utilizamos a ferramenta Google Forms[2] para administrar o questionário. Além das postagens em redes sociais, foi encaminhado um *link* por e-mail como convite à participação. Não foram relatados problemas específicos para responder o questionário a partir da ferramenta utilizada.

5.3 RESULTADOS

Obtivemos 86 respostas de desenvolvedores de softwares, sendo 48 respostas no questionário administrado em inglês e 38 respostas do questionário administrado em português. Desses, a maioria dos participantes possui o título de mestrado (41) e bacharelado (35), como mostra o gráfico da Figura 12. Com relação ao nível de experiência no desenvolvimento Java, a maioria dos participantes possui de 5-10 anos (34) e 3-5 anos (22), conforme Figura 13. Referente ao número de aplicativos Java desenvolvidos utilizando ORM, houve uma maior diversificação entre os participantes, com 3-5 (24) mais que 10 (23) e de 0-2 aplicativos desenvolvidos (22), conforme gráfico da Figura 14 .

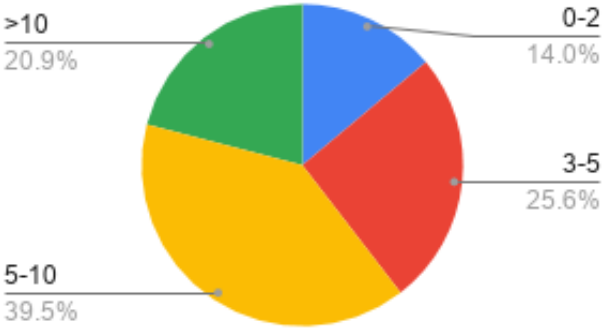
[2] `https://google.com/forms`

Figura 12 – Qualificação



Fonte: Elaborada pelo autor (2020)

Figura 13 – Experiência de desenvolvimento Java



Fonte: Elaborada pelo autor (2020)

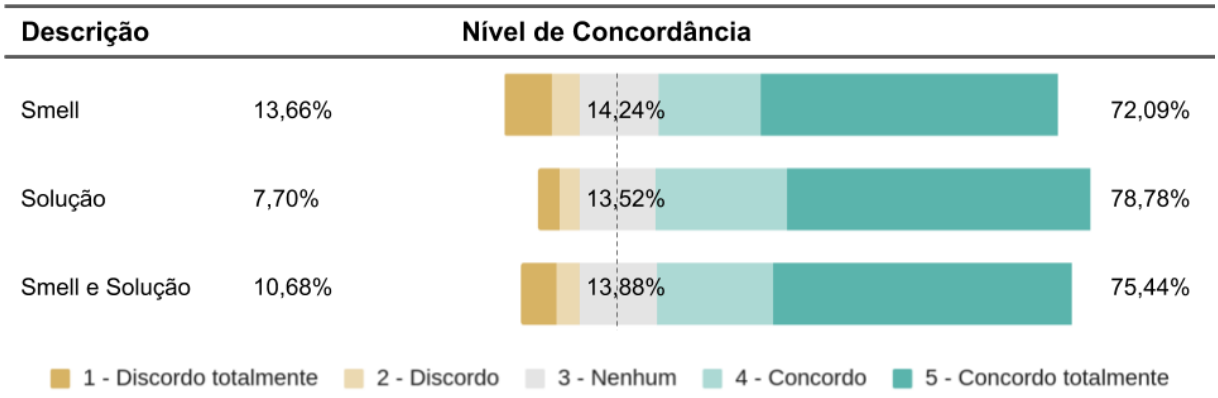
Figura 14 – Número de aplicações utilizando ORM

Fonte: Elaborada pelo autor (2020)

Os resultados referentes ao nível de concordância geral do catálogo de *code smell* ORM considerando a média dos *smells*, soluções e uma média geral, considerando *smells* e soluções, podem ser verificados na Figura 15 através do gráfico de barras empilhadas

divergentes (*diverging stacked bar chart*. As três porcentagens mostradas no gráfico indicam a proporção de discordâncias, respostas neutras e concordâncias, respectivamente. Além disso, as cores verde escuro e verde claro indicam a proporção de respostas "Concordo totalmente"e "Concordo", marrom escuro e marrom claro indicam a proporção de respostas "Discordo totalmente"e "Discordo". A maioria dos participantes avaliou positivamente o catálogo (75,44%), considerando *smell* e solução. Analisando separadamente, foram 72,09% avaliações positivas para os *smells* apresentados e 78,27% para suas respectivas soluções.

Figura 15 – Nível de concordância geral (média)



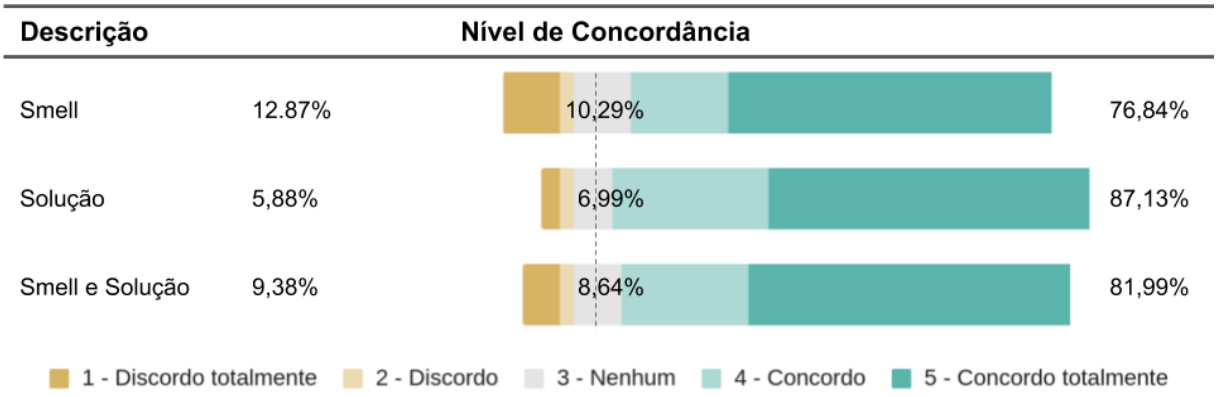
Fonte: Elaborada pelo autor (2020)

Verificamos que o nível de concordância difere quando verificado o resultado da avaliação de acordo com o nível de experiência em relação a ORM em Java. Assim, também analisamos os dados separadamente entre as respostas de acordo com o nível de experiência. Separamos as respostas em dois grupos considerando o nível de experiência de acordo com as respostas na seção demográfica:

- Desenvolvedores experientes: possuem mais de cinco anos de experiência com desenvolvimento Java e uma vasta experiência com a utilização de ORM (cinco ou mais aplicativos desenvolvidos).
- Desenvolvedores iniciantes ou intermediários: demais participantes que não se enquadram no grupo de desenvolvedores experientes.

A Figura 13 apresenta o gráfico considerando o nível de concordância utilizando apenas as respostas dos participantes considerados desenvolvedores experientes (total de 34). Houve um aumento de 8,7% no nível de concordância geral para *smells* e soluções combinados quando comparado à média de todos os participantes, totalizando 81,99%. Em relação aos *smells* o nível de concordância é de 76,84% e de 87,13% para as soluções.

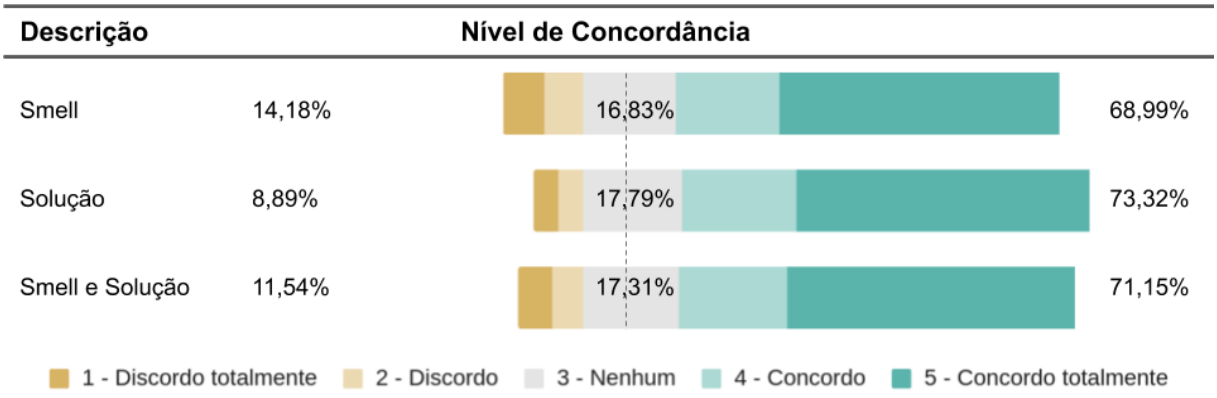
Figura 16 – Nível de concordância considerando desenvolvedores experientes (média)



Fonte: Elaborada pelo autor (2020)

Para o grupo referentes a desenvolvedores (Figura 17) iniciantes ou intermediários (total de 52), houve um decréscimo 5,7% no nível de concordância comparado ao gráfico de todos os participantes, totalizando 71,15%. Sendo 68,99% para os *smells* e 73,32% para a solução.

Figura 17 – Nível de concordância considerando desenvolvedores iniciantes ou intermediários (média)

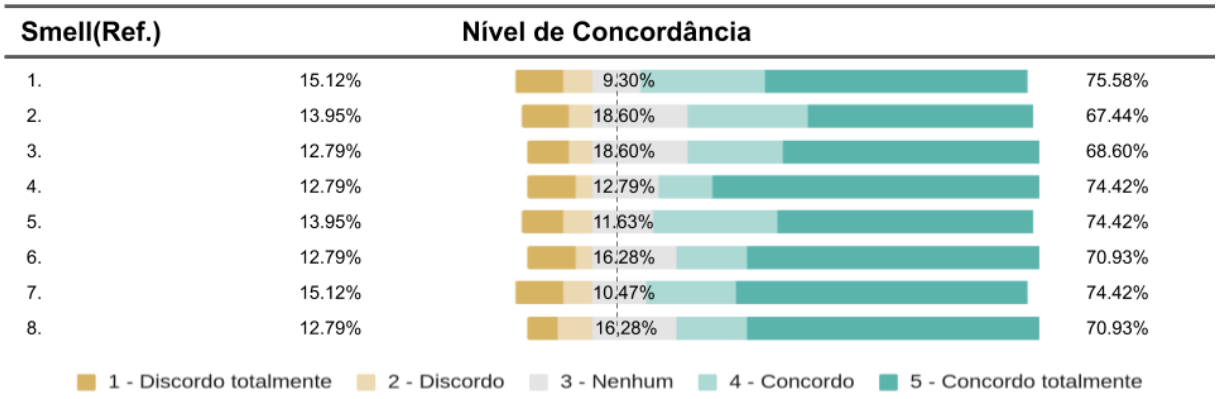


Fonte: Elaborada pelo autor (2020)

A Figura 18 apresenta o resultado da concordância para cada *code smell* ORM representados por números, relacionados a Tabela 7 . O *smell* que teve o maior nível de concordância foi 1. EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE (ESTÁTICO) com 75,58%, seguido por 4. FALTA DE PAGINAÇÃO QUANDO NÃO NECESSÁRIO TODO O RESULTADO, 5. FALTA DE JOIN FETCH NAS CONSULTAS DE ATRIBUTOS DO TIPO EAGER e 7. @ONETOMANY UNILATERAL COM USO INADEQUADO DE COLEÇÕES todos com 74,42%. No entanto, o resultado da concordância em relação a solução teve uma

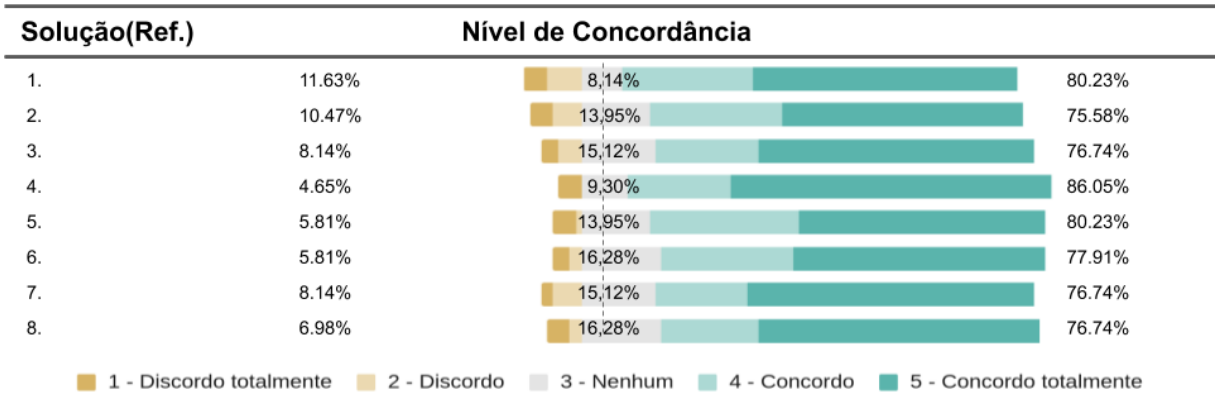
ordem diferente, conforme Figura 19. A solução apresentada com o nível de concordância mais alto foi 4. FALTA DE PAGINAÇÃO QUANDO NÃO NECESSÁRIO TODO O RESULTADO com 86,05% seguido de 1. EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE (ESTÁTICO) e 5. FALTA DE JOIN FETCH NAS CONSULTAS DE ATRIBUTOS DO TIPO EAGER ambos com 80,23%.

Figura 18 – Nível de concordância por *code smell*



Fonte: Elaborada pelo autor (2020)

Figura 19 – Nível de concordância por solução



Fonte: Elaborada pelo autor (2020)

Realizando um comparativo entre a média das respostas quanto ao nível de concordância dos desenvolvedores experientes (81,99%) com os desenvolvedores iniciantes ou intermediários (71,15%), observamos que houve uma variação percentual positiva de 15,23% para o grupo de desenvolvedores mais experientes. Usualmente, desenvolvedores experientes tendem a trabalhar com a parte mais complexa do código, assim como problemas relacionados a diversas causas (Tsunoda *et al.*, 2017). Acreditamos que o maior nível de concordância se deu pelo fato dos desenvolvedores experientes presenciarem mais vezes *code smells* ORM

e os problemas gerados para as aplicações na prática do que desenvolvedores iniciantes ou intermediários.

Recebemos alguns comentários gerais sobre o catálogo apresentado na pesquisa. Alguns participantes mostraram apreço pelo catálogo, por exemplo: "*Ótima pesquisa e pontos interessantes relançados*";"*... Foi esclarecedor e útil aprender esses bad smells. Meu código será melhor a partir de agora*.", entre outros. Outros participantes expressaram desprezo pelo ORM, por exemplo: "*A suposição de que o ORM é válido (e que pode ser corrigido) é falso. O ORM precisa ser eliminado ...*";"*SQL é melhor que o ORM*". Algumas sugestões foram apresentadas pelos participantes e ajustadas no catálogo, como por exemplo uma sugestão quanto a solução proposta do *smell* 4. FALTA DE PAGINAÇÃO QUANDO NÃO NECESSÁRIO TODO O RESULTADO: "*Logo na sugestão eu utilizaria uma StringBuilder (para concatenar String), pois da forma que o exemplo foi escrito, gera processamento desnecessário em memória por causa do operador +.*". Outras sugestões foram apresentadas de forma geral com relação ao catálogo, as que não puderam ser tratadas nesse trabalho, foram utilizadas como recomendação de trabalhos futuros, como em: "*Talvez, para code smell / solução, pudesse ser apresentado um benchmark, algo assim, destacando a diferença em termos de desempenho e recursos computacionais envolvidos.*"

Concluímos que o catálogo foi positivamente avaliado em sua maioria, tanto em relação aos *code smells* quanto as soluções apresentadas. Os comentários também foram em sua maioria positivos, com exceção de alguns desenvolvedores que não acham vantajoso o uso da tecnologia, possivelmente pelos problemas já citados que o mau uso pode ocasionar.

6 FERRAMENTA PARA DETECÇÃO DE CODE SMELLS ORM

Além do desenvolvimento do catálogo de *code smells*, outra contribuição desta pesquisa é o desenvolvimento de uma ferramenta para detecção automática destes smells através de análise estática do código, apresentando ao desenvolvedor o local do *code smell*, exibindo a descrição e sugestão para ajuste, conforme indicação do catálogo descrito no Capítulo 4. A ferramenta para detecção de *code smells* ORM foi desenvolvida utilizando a arquitetura de desenvolvimento de *plugins* para SpotBugs. Neste capítulo será apresentada a arquitetura e detectores desenvolvidos para quatro *code smells* ORM presentes no catálogo. O código-fonte da ferramenta desenvolvida está disponível online.[1]

6.1 ARQUITETURA

A arquitetura do projeto teve como base o padrão recomendado para desenvolvimento de um novo *plugin* para o SpotBugs (SpotBugs, 2018a). Java foi a linguagem de desenvolvimento utilizada com o gerenciador de bibliotecas Maven, contendo os pacotes descritos a seguir.

6.1.1 Artefatos Principais

O pacote principal do projeto é composto por subpacotes para cada *smell*, contendo uma classe principal com a lógica necessária para detecção do *code smell* relacionado no código fonte. Essa classe deve estender uma das seguintes classes, conforme (SpotBugs, 2018a):

- `AnnotationDetector`: classe base responsável por analisar anotações em classes, métodos, campos e parâmetros de métodos;
- `BytecodeScanningDetector`: classe base responsável por analisar *bytecode* Java em arquivos de classe;
- `OpcodeStackDetector`: é uma subclasse de `BytecodeScanningDetector`. Permite analisar o *bytecode* de um método através de uma pilha de operações.

6.1.2 Artefatos de Testes

Contém casos de testes para validação a partir de testes unitários separados em pacotes para cada *code smell* ORM, também seguindo a arquitetura do *plugin*, conforme (SpotBugs, 2018a). Cada pacote possui três classes:

[1] https://bit.ly/3aOnMJi

- Caso de teste: uma classe contendo um caso de teste para o detector de *code smell* relacionado ao pacote;
- *BadCase*: uma classe contendo um código com o *code smell* para validação;
- *GoodCase*: uma classe contendo um código que o detector não deve encontrar *code smells* ORM para evitar falsos-positivos.

6.1.3 Artefatos de Recursos

O pacote de recursos contém dois arquivos XML referentes ao SpotBugs (*findbugs.xml* e *messages.xml*) que devem ser editados para adicionar cada detector de *code smell* ORM desenvolvido, relacionando a seus pacotes e classes, adicionando descrições curtas e longas a serem exibidas na detecção para cada *code smell* ORM.

6.2 DETECTORES

Para cada *code smell* ORM foi desenvolvido um detector seguindo a arquitetura descrita. Devido a restrições de tempo com relação ao término da pesquisa, realizamos a seleção de quatro *code smells* dos oito possíveis para desenvolvimento dos detectores para a ferramenta. Nesta seção, além de explicar os critérios de seleção, iremos detalhar os detectores desenvolvidos apresentando: os artefatos gerados, regras para detecção, funcionamento e limitações conhecidas. Os casos de testes unitários apresentados nos detectores auxiliaram durante o desenvolvimento, fazendo com que a cada rodada de testes o código fosse aprimorado até que todos os testes fossem executados com sucesso.

6.2.1 Metodologia de seleção

O critério de escolha dos *code smells* a serem desenvolvidos foi baseado com relação ao nível de risco de falsos-positivos e falsos-negativos. Para isto, realizamos um levantamento de possíveis falsos-positivos e falsos-negativos para cada *code smell* tendo em vista o cenário de detecção através de análise estática. A partir dessa análise, os *code smells* foram classificados em três níveis de risco: baixo, médio e alto. O procedimento realizado, contendo os riscos identificados e o nível de classificação destes riscos está disponível online.[2] Dois *code smells*, EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE e @ONETOMANY UNILATERAL COM USO INADEQUADO DE COLEÇÕES foram classificados com risco baixo e foram selecionados para desenvolvimento. Selecionamos, também, dois *code smells*, RECUPERAÇÃO DE DADOS SEM PROJEÇÃO PARA SOMENTE LEITURA e FALTA DE JOIN-FETCH: ESTRATÉGIA DE BUSCA EAGER, classificados com risco médio (detalhes dos riscos são apresentados nas Seções 6.2.5.3 e 6.2.3.3) para completar quatro detectores, chegando

[2] https://bit.ly/3gmD49c

a 50% dos *smells* relacionados no catálogo. A Tabela 10 apresenta os quatro *code smells* selecionados relacionados ao detector desenvolvido na ferramenta. Para fins de facilitar a visualização, esses *smells* são tratados com o nome de seus respectivos detectores conforme relacionado na tabela.

Tabela 10 – *Code smells* ORM selecionados

Code Smell	**Detector**
EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE	EAGERSMELL
FALTA DE JOIN FETCH: ESTRATÉGIA DE BUSCA EAGER	N1JOINFECHSMELL
@ONETOMANY UNILATERAL COM USO INADEQUADO DE COLEÇÕES	ONETOMANYCOLLECTIONSSMELL
RECUPERAÇÃO DE DADOS SEM PROJEÇÃO PARA SOMENTE LEITURA	PROJECTIONSMELL

Fonte: Elaborada pelo autor (2020)

6.2.2 EAGERSMELL

O detector EAGERSMELL é o responsável pela detecção do *code smell*: EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE, relatado na Seção 4.1.1. Possui a lógica de detecção através da classe principal `EagerDetector`, anotações XML e as classes responsáveis pelos testes unitários no pacote `EagerSmell` dentro do pacote de testes da aplicação.

6.2.2.1 Regras para Detecção

Deve ser reportado como *code smell* ORM dessa categoria qualquer classe que representa uma entidade (com anotação `@Entity`) e que contenha anotações de relacionamento (`@ManyToMany`, `@OneToMany`, `@ManytoOne`, `@OneToOne`) com a estratégia de busca `FetchType.EAGER` explicitamente defina, ou, em específico para anotações de relacionamentos `@ManyToOne` e `@OneToOne` que não estejam explicitamente com anotação `FetchType.LAZY`. Para os relacionamentos `@ManyToMany` e `@OneToMany` com a estratégia de busca `FetchType.EAGER` o detector deve considerar um nível de prioridade mais alto. É necessário apontar o local da ocorrência, apresentar uma descrição e as soluções propostas no catálogo.

6.2.2.2 Funcionamento do Detector

A classe principal (Exemplo 29) que contém a lógica para detecção é chamada de `EagerDetector`, foi desenvolvida de forma a estender a classe `AnnotationDetector`, responsável por verificar as anotações no código. O método `visitAnnotation` tem como

objetivo principal realizar um filtro para que anotações não alusivas aos relacionamentos de entidades fiquem de fora da análise. Caso contrário, a estratégia de busca é capturada e encaminhada para o método `detectEager` em conjunto com a anotação utilizada. É

Exemplo 29 - Classe *EagerDetector*

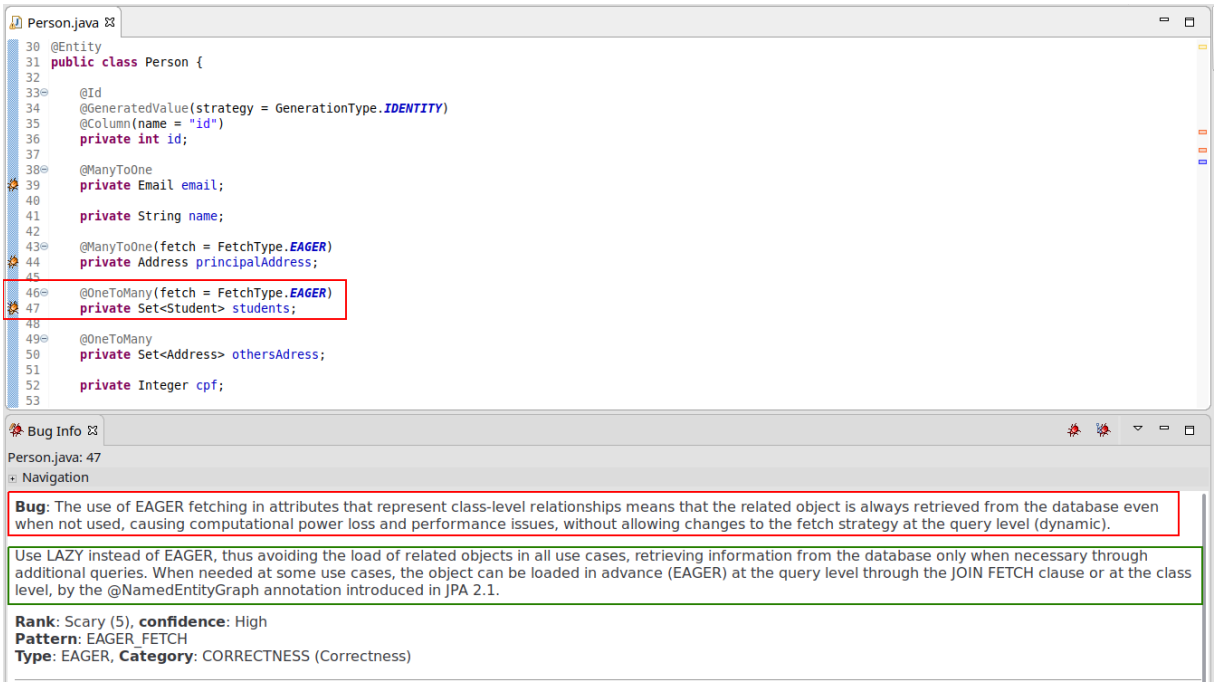
```
1  public class EagerDetector extends AnnotationDetector {
2
3   private final BugReporter bugReporter;
4
5   public EagerDetector(BugReporter bugReporter) {
6    this.bugReporter = bugReporter; }
7
8   @Override
9   public void visitAnnotation(@DottedClassName String annotationClass,
        Map<String, ElementValue> map,
10     boolean runtimeVisible) {
11    if (!Objects.equals(annotationClass, "javax.persistence.ManyToOne")
12      && !Objects.equals(annotationClass, "javax.persistence.OneToMany")
13      && !Objects.equals(annotationClass, "javax.persistence.OneToOne")
14      && !Objects.equals(annotationClass, "javax.persistence.ManyToMany")
            ) {
15     return; }
16    String estrategia = null;
17    if (map.get("fetch") != null) {
18     estrategia = map.get("fetch").stringifyValue(); }
19    detectEager(estrategia, annotationClass); }
20
21   private void detectEager(String estrategia, String annotationClass) {
22    if (((Objects.equals(annotationClass, "javax.persistence.ManyToOne"))
23      || (Objects.equals(annotationClass, "javax.persistence.OneToOne")))
24      && (!Objects.equals(estrategia, "LAZY"))) {
25     badSmellReport("EAGER_SMELL", NORMAL_PRIORITY);
26    } else if ("EAGER".equals(estrategia)) {
27     badSmellReport("EAGER_SMELL", HIGH_PRIORITY); }}
28
29   private void badSmellReport(String badSmell, Integer prior) {
30    bugReporter.reportBug(new BugInstance(this, badSmell, prior).addClass
          (this).addField(this));
31   }
32  }
```

possível identificar se o mapeamento realizado é considerado um *code smell*, através do método `detectEager`, seguindo a lógica descrita na regra de detecção. Primeiramente é verificado se a anotação é relacionada a `@ManyToOne` ou `@OneToOne` com a estratégia de busca não sendo explicitamente definida como LAZY. Se estiver nessa condição é reportado como *code smell* do tipo `EAGER_SMELL` com a classificação `NORMAL_PRIORITY`. Para as anotações restantes (`@OneToMany` e `@ManyToMany`) com a estratégia EAGER definida será reportado como *code smell* de mesmo tipo, mas com a classificação `HIGH_PRIORITY`. Ambos são reportados através do método `badSmellReport`, alterando os parâmetros passados. Os mapeamentos entre a classe *EagerDetector* com as configurações do *SpotBug*

e textos relacionados são realizados via XML conforme Seção 6.1.3.

Para execução dos testes unitários, foram criados para validação inicial dois arquivos com mapeamentos ORM. O primeiro, chamado `BadCase`, contendo duas anotações (uma com EAGER implícito, através da anotação `@ManyToOne` e outra com EAGER de forma explícita) que são detectadas como um *code smell* ORM com níveis de prioridades diferentes. O segundo arquivo, chamado `GoodCase`, contendo duas anotações com a estratégia de busca LAZY que não devem ser detectadas como *code smells* ORM. A Figura 20 apresenta um exemplo da utilização do detector (após compilar, adicionar como biblioteca do *plugin* e executar no Eclipse), mostrando o local do *code smell* no código através da imagem de um inseto (*bug*), conforme padrão do SpotBugs (SpotBugs, 2018b), exibindo também a descrição do *smell* e sugestão de solução.

Figura 20 – Teste de detecção - *EagerSmell*



Fonte: Elaborada pelo autor (2020)

6.2.2.3 Limitações conhecidas

Em uma análise inicial não foram encontradas limitações no funcionamento deste detector. As classes representativas de entidades são mapeadas de forma única com anotação `@Entity` e os mapeamentos para a estratégia de busca também são feitos de forma semelhante, deixando pouca margem para erro na detecção.

6.2.3 N1JOINFETCHSMELL

Para o *code smell* FALTA DE JOIN FETCH: ESTRATÉGIA DE BUSCA EAGER, relatado na Seção 4.2.1, foi desenvolvido o detector N1JOINFETCHSMELL. O detector é composto pela classe principal `N1MissedJoinFetchDetector`, que possui a lógica de detecção, juntamente com uma classe auxiliar `CollectEagerAtributes`, dois DTOs para utilização na classe principal (`ClassEagerAttDTO` e `QlMethodDTO`), anotações XML e as classes responsáveis pelos testes unitários no pacote `N1JoinFetchSmell` dentro do pacote de testes da aplicação.

6.2.3.1 Regras para Detecção

Para ser reportado como *code smell* ORM nesta categoria, o detector deve verificar consultas ORM que retornem entidades com atributos EAGER (mesmas regras do detector 6.2.2) e sem possuir JOIN FETCH na consulta para estes atributos. A consulta também deve retornar a entidade completa, ou seja, sem projeção. Além de apontar o local do CODE SMELL, apresentar a descrição e solução descrita no catálogo, deve também informar quais são os atributos EAGER sem JOIN FETCH da classe principal da consulta.

6.2.3.2 Funcionamento do Detector

Para este detector, necessitamos criar uma classe auxiliar chamada COLLECTEAGERATRIBUTES. O objetivo dessa classe é coletar no código as classes que possuem atributos de relacionamentos do tipo EAGER e salvar em uma lista do tipo `ClassEagerAttDTO`. Este DTO contem o nome da classe coletada e uma lista de atributos EAGER referentes a classe coletada. A classe principal do detector, `N1MissedJoinFetchDetector`, assim como no caso do EAGERSEMELL também estende o comportamento da classe `AnnotationDetector`. A classe faz a verificação no contexto das classes percorridas a procura de métodos contendo execução de consultas ORM (podendo ser diferentes conforme o *framework* ORM), como por exemplo o método `createQuery` do pacote `javax.persistence.Query`. Após localizar, faz uma análise para montar a consulta ORM em ordem (*SELECT ... FROM ... WHERE*) a partir das *Strings* disponíveis no método. Esta ordenação é necessária, pois a instrução ORM pode estar desordenada logicamente e espalhada em diversas variáveis *Strings*, sendo concatenadas ao final. Também é verificado se a consulta retorna a entidade completa pois a utilização de projeção não causa o *smell*. O Exemplo 30 apresenta uma parte do código, demonstrando o método responsável por extrair a instrução ORM, capturar a linha da instrução e armazenar em um lista para posterior verificação. Ao fim é verificado na consulta ORM qual a classe principal de retorno da consulta, verificado pelo método `methodGen.getReturnType().toString()`. Com a informação da classe, é percorrida a lista a procura de classes coletadas com atributos `EAGER` pelo coletor COLLECTEAGERATRIBUTES. Caso encontrada, verifica se todos os atributos coletados como EAGER estão sendo utilizados

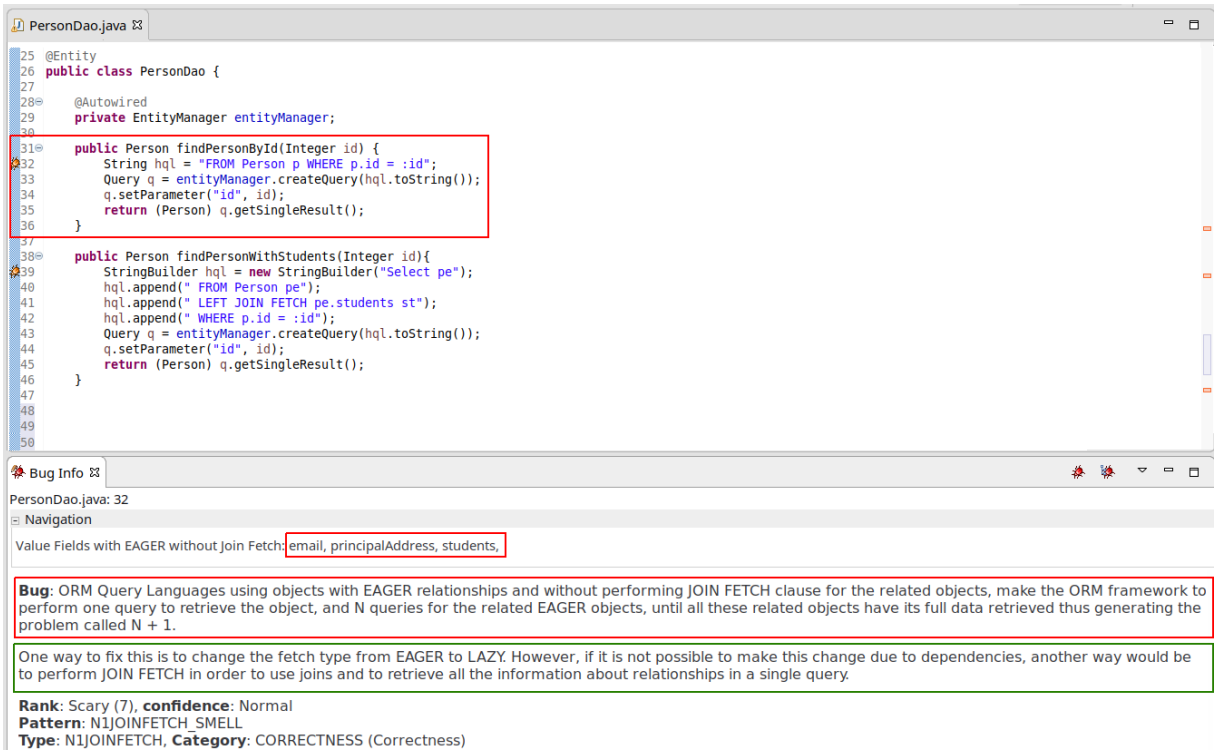
na consulta a partir do *JOIN FETCH*. Se algum atributo estiver faltando, armazena os atributos que não estão presentes na consulta e reporta o *Code Smell* ORM, adicionando na mensagem quais os campos que estão faltando na consulta.

Exemplo 30 - Classe *N1MissedJoinFetchDetector*

```
1  ...
2  private void analyzeMethod(ClassContext classContext, Method method)
       throws DataflowAnalysisException, CFGBuilderException {
3    ...
4    String classReturnName = methodGen.getReturnType().toString();
5    classReturnName = CodeSmellsUtils.transformListReturnToEntity(
         classReturnName, methodGen);
6    if(sql.toString().toUpperCase().contains("FROM")) {
7     if (CodeSmellsUtils.existProjection(sql.toString())) {
8      return;
9     }
10   }
11   sourceLine=getSourceLine(classContext, method, javaClass, methodGen);
12   QlMethodDTO qlMethodDTO = new QlMethodDTO(classReturnName, sql.
         toString(), methodGen, javaClass.getSourceFileName(), sourceLine,
         interfaceClass, javaClass, method);
13   listQlMethodDTO.add(qlMethodDTO);
14  }
15 ...
```

Foram desenvolvidos também testes unitários para validação inicial do detector. O primeiro caso de teste chamado `BadCase`, contendo instruções ORM e atributos EAGER sem a utilização de JOIN FETCH. O segundo caso de testes, chamado `GoodCase`, contendo instruções ORM sem atributos EAGER e com atributos EAGER utilizando JOIN FETCH que não devem ser detectadas para este *code smells* ORM. A Figura 21 apresenta um exemplo da detecção exibindo a marcação no código referente ao método da consulta ORM correspondente ao *smell* e apresentando a mensagem do *smell*. Como adicional, apresenta na mensagem os atributos EAGER que estão sem JOIN FETCH na consulta de forma a facilitar a refatoração, além da sugestão de solução.

Figura 21 – Teste de detecção - *N1JoinFetchSmell*



Fonte: Elaborada pelo autor (2020)

6.2.3.3 Limitações conhecidas

Este detector possui como limitação a não verificação de atributos EAGER em classes que não a principal da consulta, podendo gerar falsos negativos. As entidades relacionadas através de JOIN FETCH podem também ter seus atributos EAGER e gerar o problema do $N + 1$. Mas por estarem descritos em forma de atributos na *String* de consulta ORM sem informação do caminho completo da classe, considerando que podem ter classes com o mesmo nome em diferentes pacotes, fica complexa essa verificação utilizando somente a análise estática fornecida por SpotBugs. O *framework* ORM consegue verificar utilizando o seu mapeamento convertendo a consulta em tempo de execução, diferente do nosso detector que utiliza análise estática. Desta forma, atualmente o detector se limita a verificação da classe principal na consulta. Outras limitações existentes neste detector são referentes à verificação da existência de projeção, que são descritas em detalhes na Seção 6.2.5.3, itens 1 e 2.

6.2.4 ONETOMANYCOLLECTIONSSMELL

Desenvolvemos o detector ONETOMANYCOLLECTIONSSMELL para o *code smell* ORM: @ONETOMANY UNILATERAL COM USO INADEQUADO DE COLEÇÕES, abordado na Seção 4.2.3. Este detector é composto por: uma classe principal nominada `OneToManyCollections-Detector`, anotações XML com a descrição e solução para o detector, além das classes

responsáveis pelos testes unitários no pacote `OneToManyCollectionsSmell` dentro do pacote de testes da aplicação.

6.2.4.1 Regras para Detecção

Classes que representam entidades (com anotação `@Entity`) e possuam atributos com anotação `@OneToMany` de forma unilateral (sem parametrização de `mappedBy`, ou anotações `JoinColumn` e `JoinTable`) e que o tipo do atributo é diferente de `Set` deve ser reportado como *code smell* ORM. O detector deve apontar o local do CODE SMELL, apresentar a descrição e solução descrita no catálogo.

6.2.4.2 Funcionamento do Detector

A detecção do *code smell* é realizada a partir da classe principal `OneToManyCollectionsDetector` (Exemplo 31). Esta classe, a exemplo dos detectores apresentados anteriormente, estende o comportamento da classe `AnnotationDetector`. A detecção inicia sobrescrevendo o método `visitAnnotation`, descartando os atributos com anotações que não o tipo `@OneToMany`. Se possuir essa anotação, também verifica se apresenta o parâmetro `mappedBy` para descartar relação bilateral fora do contexto do *code smell*. Após este filtro inicial, o método `collectionVerify` verifica se o atributo é o do tipo `Set`, caso não for, verifica se possui parâmetros do tipo `JoinColumn` ou `JoinTable` que também caracterizaria como bilateral. Por fim é relatado o *code smell* ORM.

Nos casos de testes, foram adicionados dois atributos com `@OneToMany` unilateral (um do tipo `List` e outro do tipo `Collection`) na classe de `BadCase`. Na classe `GoodCase` foram adicionados dois atributos, um do tipo `Set` unilateral e outro do tipo `List` mas com o parâmetro `mappedBy` para ser considerado como bilateral. A Figura 22 apresenta um exemplo da detecção do *code smell* com marcação na linha dos atributos anotados como `OneToMany` unilateral com uso inadequado de coleções. Além da marcação, exibe um texto contendo explicação e sugestão de solução.

Exemplo 31 - Classe *OneToManyCollectionsDetector*

```
1  public class OneToManyCollectionsDetector extends AnnotationDetector {
2   private final BugReporter bugReporter;
3   private static final Type typeSet = new ObjectType("java.util.Set");
4
5   public OneToManyCollectionsDetector(BugReporter bugReporter) {
6    this.bugReporter = bugReporter; }
7
8   @Override
9   public void visitAnnotation(@DottedClassName String annotationClass,
        Map<String, ElementValue> map,
10    boolean runtimeVisible) {
11   if (!Objects.equals(annotationClass, "javax.persistence.OneToMany")
        || Objects.nonNull(map.get("mappedBy"))) {return;}
12   collectionVerify( annotationClass);
13  }
14  private void collectionVerify(String annotationClass) {
15   if (Objects.equals(typeSet, getField().getType())) {return;}
16   Collection<ClassDescriptor> annotationDescriptors = ((FieldInfo)
        getFieldDescriptor()).getAnnotationDescriptors();
17   if (annotationDescriptors.stream()
18     .filter(annotation -> ("JoinColumn".equals(annotation.getSimpleName
          ()) || "JoinTable".equals(annotation.getSimpleName())))
19     .findAny().isPresent()) {
20    return; }
21   badSmellReport("ONETOMANYCOLLECTIONS_SMELL", HIGH_PRIORITY); }
22  private void badSmellReport(String badSmell, Integer prior) {
23   bugReporter.reportBug(new BugInstance(this, badSmell, prior).addClass
        (this).addField(this)); }
24 }
```

Figura 22 – Teste de detecção - *OneToManyCollectionSmell*

Fonte: Elaborada pelo autor (2020)

6.2.4.3 Limitações conhecidas

Não foram verificadas limitações para este detector em uma análise inicial. O contexto é semelhante ao verificado na Seção 6.2.2, com anotações padronizadas e com pouca margem para erro no processo de detecção.

6.2.5 PROJECTIONSMELL

Para o *code smell* ORM: RECUPERAÇÃO DE DADOS SEM PROJEÇÃO PARA SOMENTE LEITURA foi desenvolvido o detector PROJECTIONSMELL. Esse detector possui como classe principal `ProjectionDetector`, o detalhamento do *smell* e solução para apresentação na detecção em arquivos XML, além dos casos de testes unitários.

6.2.5.1 Regras para Detecção

Para ser reportado como *code smell* ORM, o detector deve verificar consultas ORM que não utilizam projeção, ou seja, consultas que comecem diretamente com `FROM`, ou que não possuam campos informados entre o `SELECT` e o `FROM`. A finalidade do objeto de retorno deve ser de somente leitura. É necessário ainda apontar o método da consulta, apresentar uma descrição e as soluções propostas no catálogo.

6.2.5.2 Funcionamento do Detector

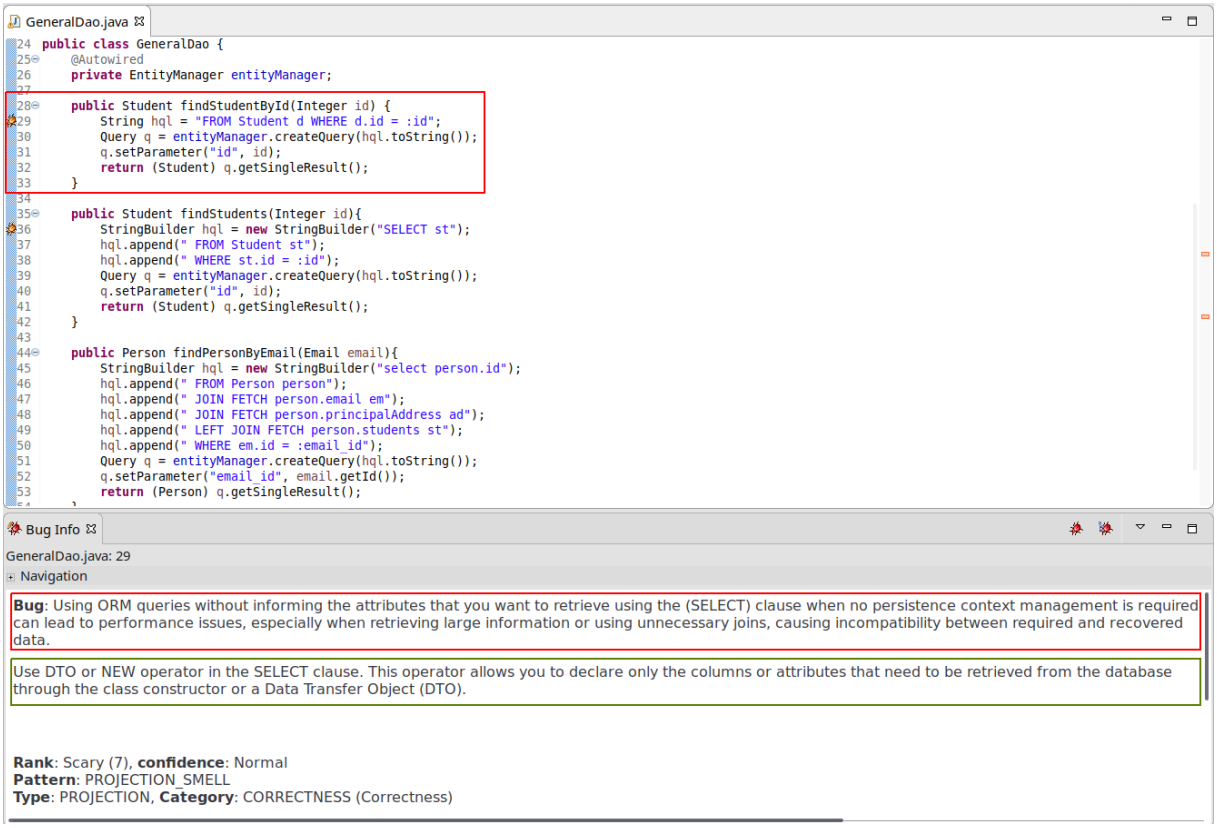
A classe `ProjectionDetector`, estende também o comportamento de `AnnotationDetector`. O Exemplo 33 ilustra como a classe realiza a verificação sobrescrevendo o método `visitClassContext`, verificando as classes e métodos que possuem consultas ORM, como por exemplo, métodos que utilizam o `createQuery` do pacote `javax.persistence. Query`. Após, utiliza métodos adicionais para extrair e ordenar a consulta ORM em uma variável `String` para análise. A partir da consulta extraída, verifica se possui a cláusula `SELECT`. Se existir, verifica também se entre essa cláusula e a cláusula `FROM` possui mais de três caracteres de distância. Esse número de caracteres foi escolhido, pois verificamos a partir de buscas aleatórias em código, que *alias* utilizados em consultas sem projeção costumam ter um ou dois caracteres. Também desconsidera a palavra `distinct` na contagem de caracteres. Se não possuir `SELECT` ou for menor que três caracteres a distância para a clausula `FROM`, relata como *code smell* ORM.

Nos casos de testes, foram adicionados métodos sem projeção para o `BadCase` escrito de diferentes formas e métodos com projeção para o `GoodCase`. É possível verificar um exemplo de funcionamento na Figura 23 , apresentando o local no código do *code smell*, bem como uma descrição e possível solução.

Exemplo 32 - Classe *ProjectionDetector*

```
...
@Override
 public void visitClassContext(ClassContext classContext) {
  JavaClass javaClass = classContext.getJavaClass();
  Method[] methodList = javaClass.getMethods();
  for (Method method : methodList) {
   if (!CodeSmellsUtils.hasInterestingMethod(method.getConstantPool(),
       allMethods)) {
    return;
   }
   try {
    analyzeMethod(classContext, method);
   } catch (DataflowAnalysisException | CFGBuilderException e) {
    bugReporter.logError("ProjectionSmell caught exception while
        analyzing "
      + classContext.getFullyQualifiedMethodName(method), e);
   }
  }
 }
...
```

Figura 23 – Teste de detecção - *ProjectionSmell*



Fonte: Elaborada pelo autor (2020)

6.2.5.3 Limitações conhecidas

Este detector contém limitações quanto a verificação de projeção (também válida para o detector 6.2.3) e a verificação da utilização para somente leitura, conforme os seguintes riscos mapeados:

1. Falsos negativos em situações que o *alias* da consulta for maior que 3 caracteres. Exemplo: `SELECT pess FROM Pessoa pess`.

2. Falsos positivos em situações que o nome do atributo único a ser consultado for menor ou igual a 2 caracteres. Exemplo: `SELECT id FROM Pessoa`.

3. Falsos positivos pela impossibilidade de verificação do retorno ser utilizado para somente leitura. Uma das premissas do *code smell* é que o retorno da consulta ORM seja utilizada como somente leitura pelos motivos explanados na seção 4.1.2. Porém, não foi implementado devido à dificuldade de verificar esta premissa utilizando somente análise estática, tendo que identificar cada chamada para o método e analisar se o objeto será persistido em algum momento, podendo ser de diversas formas diferentes (inclusive utilizando métodos próprios ex: `dao.salvaObjeto(objeto)`). Nesse caso, caberá análise manual do desenvolvedor.

Após término da codificação e testes iniciais dos detectores para os *code smells* ORM selecionados do catálogo, realizamos procedimentos para avaliação da ferramenta, em especial quanto a critérios de precisão e revocação. O procedimento completo será detalhado no próximo capítulo.

7 AVALIAÇÃO DA FERRAMENTA DESENVOLVIDA

Ao fim do desenvolvimento e testes da ferramenta para detecção de *code smells* ORM, precisamos avaliar a precisão e revocação quanto aos *code smells* detectados pela ferramenta. Embora seja aceitável algum nível de imprecisão no processo de detecção com base nas limitações mapeadas no Capítulo 6 e também pelo fato do conceito de *code smells* ser relacionado a sintomas de más escolhas e não necessariamente um problema (Fowler, 1999), muitos falsos-positivos ou falsos-negativos poderiam inviabilizar a utilidade da ferramenta. Com esse objetivo, desenvolvemos uma pequena aplicação para servir de referência com exemplos de *code smells ORM* para uma primeira etapa de avaliação quanto à precisão e revocação. A aplicação desenvolvida também auxilia na busca por limitações desconhecidas. Em seguida, realizamos a avaliação com projetos reais extraídos do GitHub, além de projetos extraídos do GitLab do IFSC. Em adicional, também analisamos a difusão dos *code smells* com objetivo de verificar a quantidade de ocorrências detectadas em projetos reais. Este capítulo esta divido da seguinte forma: Seção 7.1 descreve a metologia do procedimento de avaliação da ferramenta. Seção 7.2 apresenta o resultado da execução da ferramenta na aplicação de referência e detalhes da aplicação desenvolvida. Os resultados da execução da ferramenta em aplicações reais são apresentados na Seção 7.3. Por fim, na Seção 7.4 apresentamos o resultado da aplicação para verificação da difusão dos *smells*.

7.1 METODOLOGIA

A ferramenta desenvolvida foi avaliada quanto à precisão e revocação. Inicialmente utilizando uma pequena aplicação de referência e posteriormente utilizando projetos reais. Também foi analisado a difusão dos *code smells* ORM detectados. Ambos os procedimentos metodológicos realizados são detalhados nos próximos parágrafos.

Utilizamos o cálculo de precisão e revocação para validação da ferramenta. A precisão é calculada com base no número de *smells* verdadeiros detectados pela ferramenta, através de uma verificação manual dentre os detectados para identificar o número de falsos-positivos. Já a revocação leva em consideração o número de *smells* verdadeiros não detectados, também com verificação manual necessária para identificar os *smells* e calcular o número de falsos-negativos gerados pela ferramenta. Após os cálculos, é possível obter uma média harmônica entre precisão e revocação sendo representadas pelo *F1-Score*. A Tabela 11 apresenta a equação baseada no estudo (Frakes & Baeza-Yates, 1992). Na Tabela em questão, TP significa verdadeiro-positivo, FP falso-positivo, FN falso-negativo, PRE precisão e REV revocação.

O processo para avaliação da precisão e revocação foi dividido em duas etapas. Para a realização da primeira etapa, desenvolvemos uma pequena aplicação de referência com três exemplos de casos positivos e negativos para cada *code smell* ORM para avaliação do número de falsos-positivos e falsos-negativos em um ambiente controlado. Os exemplos desenvolvidos

Tabela 11 – Cálculo Precisão e Revocação

Precisão	(TP)/(TP+FP)
Revocação	(TP)/(TP+FN)
F1-Score	2*(PRE*REV)/(PRE+REV)

Fonte: Elaborada pelo autor (2020)

nessa aplicação foram adaptações de exemplos encontrados em dois projetos reais, o SIGAA (sistema acadêmico do IFSC) e o PetClinic (Aplicação utilizada como base em outros estudos, como em Chen *et al.* (2014) e Dubois (2013)). Realizamos uma busca manual nesses projetos à procura de exemplos que dificultassem o trabalho de detecção da ferramenta para adaptação. Em adicional, a aplicação foi desenvolvida para ser uma padronização de referência para testes de novos detectores de *code smells* ORM que forem desenvolvidos, além de auxiliar na verificação de limitações desconhecidas dos detectores.

Na segunda etapa, aplicamos a ferramenta em quatro projetos reais. O número de projetos foi delimitado levando em consideração o esforço necessário de análise manual juntamente com o prazo para o término da pesquisa. Selecionamos dois projetos do GitLab do IFSC e dois projetos do GitHub. O objetivo foi buscar projetos de tamanhos distintos para que dois projetos menores fossem analisados integralmente no processo de análise manual e dois maiores utilizando uma parte do total do código para verificação da eficácia e revocação. Projetos do IFSC foram selecionados para avaliar a ferramenta em projetos de uma grande instituição, além de demonstrar a aplicabilidade da pesquisa na instituição interessada nos resultados. Selecionamos o Sistema Integrado de Gestão Acadêmica (SIGAA) e o *Programa de Assistência Estudantil* (PAEVS), dois projetos de tamanhos distintos que estão ativos atualmente na instituição e preenchem os requisitos por utilizarem a linguagem Java com utilização de *framework* ORM para acesso aos dados. Com relação aos projetos do GitHub, realizamos duas buscas distintas utilizando a busca de repositórios do GitHub (GitHub, 2020). Procuramos por repositórios com no mínimo 10 estrelas (classificação pelos usuários do GitHub) na linguagem Java, não arquivados e que tivessem no conteúdo do *Readme* a palavra Hibernate (*Framework* ORM mais utilizado conforme Figura 14). Utilizamos como padrão de busca as palavras chaves apresentadas na Tabela 12, adicionando de forma alternada apenas a variável referente ao tamanho do projeto (`size:>10000` e `size:<1000`). Foram analisados os projetos, ordenados por recentes atualizações para verificar projetos ativos. Para checar se continham código ORM, efetuamos a busca no repositório de cada projeto por `javax.persistence` referentes a métodos do JPA. Os projetos selecionados nessa etapa foram: BroadleafCommerce[1] e Shopping[2].

Utilizamos a IDE de desenvolvimento Eclipse para realização do procedimento de avaliação quanto à eficácia e revocação. Primeiramente, adicionamos o *Java Archive* (JAR)

[1] `https://github.com/BroadleafCommerce/BroadleafCommerce`
[2] `https://github.com/IcedSoul/Shopping`

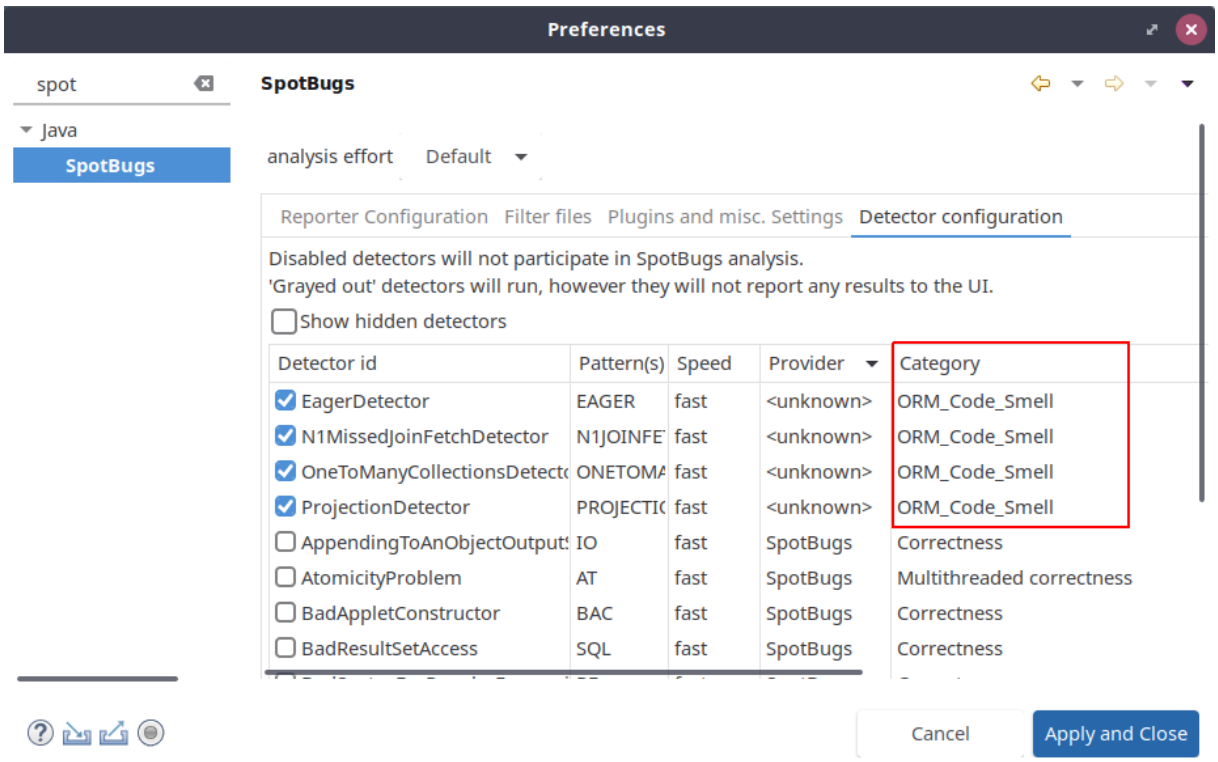
Tabela 12 – Padrão de busca - repositórios para avaliação manual

stars:>=10	language:java	Hibernate in:Readme	archived:false

Fonte: Elaborada pelo autor (2020)

da ferramenta como *plugin* para o SpotBugs no Eclipse, conforme documentação SpotBugs (2018a). Posteriormente, filtramos somente os *smells* referentes a ferramenta (categoria `ORM_Code_Smells`), conforme Figura 24. Os projetos avaliados foram importados no eclipse e compilados. Aplicamos a ferramenta utilizando a opção `spotbugs->find bugs` no menu de contexto de cada projeto.

Figura 24 – Filtro para mostrar somente os *Code Smells ORM*



Fonte: Elaborada pelo autor (2020)

Para avaliar a precisão, navegamos na árvore de *smells* detectados utilizando a perspectiva do SpotBugs no Eclipse e verificamos se estão de acordo com as regras de detecção descritas para cada detector na Seção 6.2. Em caso de desacordo com as regras, classificamos como falso-positivo.

Com relação à revocação, precisamos realizar o procedimento manual a partir do código, criando um oráculo com os *smells* para verificar os falsos-positivos, diferentemente da avaliação de precisão que partimos dos *smells* detectados. Avaliamos os projetos manualmente realizando os seguintes procedimentos para cada *smell*:

- EAGERSMELL: utilizamos a opção `Search->File Search` do Eclipse, realizando a busca utilizando a expressão regular `(@ManyToOne | @OneToMany | @OneToOne | @ManyToMany)` para procurar por anotações de relacionamentos. Verificamos se essas anotações estão definidas como EAGER conforme regra de detecção descrita na Seção 6.2.2.1. Caso positivo e a ferramenta não tenha relatado como *smell* classificamos como falso-negativo.

- N1JOINFETCHSMELL: também utilizamos a busca do Eclipse com a expressão `(createQuery|@Query)`. Esta busca procura por consultas ORM (HQL,JPQL) ou com a anotação `@Query` do Spring. No caso de `@Query` desconsideramos resultados com a opção `nativeQuery=true` por não se tratar de ORM. Verificamos se as consultas localizadas seguem as regras definidas na Seção 6.2.3.1. Nos casos positivos e que não foram identificados pela ferramenta, classificamos como falso-negativo.

- ONETOMANYCOLLECTIONSSMELL: utilizando a busca do Eclipse, procuramos pelo termo `@OneToMany`. Verificamos entre os resultados se estão de acordo com as regras de detecção descritas na Seção 6.2.4.1. Se estiver, classificamos como falso-negativo se a ferramenta não detectou.

- PROJECTIONSMELL: Realizamos a mesma busca do detector N1JOINFETCHSMELL, utilizando a expressão `(createQuery|@Query)` para verificar as consultas. Também desconsideramos resultados com a opção `nativeQuery=true` (`@Query`). Os demais resultados foram analisados seguindo a regra descrita na Seção 6.2.5.1. Caso a ferramenta não tenha identificado casos que seguem a regra descrita, classificamos como falso-negativo

Além da validação manual, utilizamos a ferramenta para verificar a difusão dos *code smells* ORM aplicando em projetos do GitHub. Para execução desta etapa, desenvolvemos um *script* utilizando Python para automatizar o processo de mineração de repositórios e execução da ferramenta com objetivo de aplicá-la no máximo de projetos possíveis. Realizamos alguns testes com combinações de busca no GitHub e modificamos alguns parâmetros das palavras chaves utilizadas na avaliação da precisão e revocação apresentadas na Tabela 12. Essa alteração se deve ao fato de verificar os melhores resultados visando a automação do processo.

Tabela 13 – Padrão de busca - repositórios para análise de difusão

stars:>=5	language:Java	archived:false	size:<100000	Hibernate in:Readme

Fonte: Elaborada pelo autor (2020)

O *script* realiza primeiro a busca por repositórios utilizando as palavras chaves descritas na Tabela 13. Após, são verificados os repositórios de forma individual para analisar o

conteúdo de forma a refinar os resultados. Em cada repositório é realizada uma nova busca pela palavra chave: `persistence.javax.ManyToOne`, procurando pelo relacionamento ORM mais utilizado (Mihalcea, 2020c). Esse processo é realizado para verificar a existência de código ORM no projeto, adaptado para descartar código de bibliotecas, exemplos, código incompleto, importante no contexto de um ambiente automatizado. Além disso, o *script* procura também pela existência de arquivos referentes à gerenciadores de dependências (Maven ou Gladle), pois facilitam a compilação, visto que os projetos precisam ser compilados para análise pela ferramenta, conforme verificado na Seção 2.5. Após essa etapa, os repositórios de projetos candidatos são listados em um arquivo `txt` de forma a guardar os projetos encontrados em caso de interrupções na execução. Na próxima etapa, o *script* segue os comandos em sequência para cada repositório encontrado. Primeiro, realiza o clone do projeto utilizando o comando `git clone –depth 1`. Após o clone, compila o projeto utilizando o gerenciador de dependência associado (ex: `mvn compile`) e o descarta em caso de falha na compilação. Em sequência, a ferramenta é executada através do SpotBugs por linha de comando (`spotbugs.jar`), utilizando as opções disponíveis como argumento (SpotBugs, 2018b): o JAR da ferramenta como *plugin*, um arquivo XML de configuração para utilizar apenas os detectores da ferramenta, o diretório do projeto e o arquivo de saída em XML com o resultado da detecção. Para leitura dos arquivos XML e compilação dos resultados, desenvolvemos uma aplicação em Java para esta finalidade. O resultado é apresentado na sequência das avaliações referentes a precisão e revocação.

7.2 AVALIAÇÃO UTILIZANDO A APLICAÇÃO DE REFERÊNCIA

A aplicação de referência foi desenvolvida com o objetivo de estabelecer um *benchmark* para realizar testes em menor escala. Isto nos possibilitou obter resultados preliminares da execução da ferramenta para detecção de *code smells* ORM. Em adicional, procuramos descobrir novas limitações antes da execução em projetos reais. O código fonte da aplicação está disponível *online*.[3]

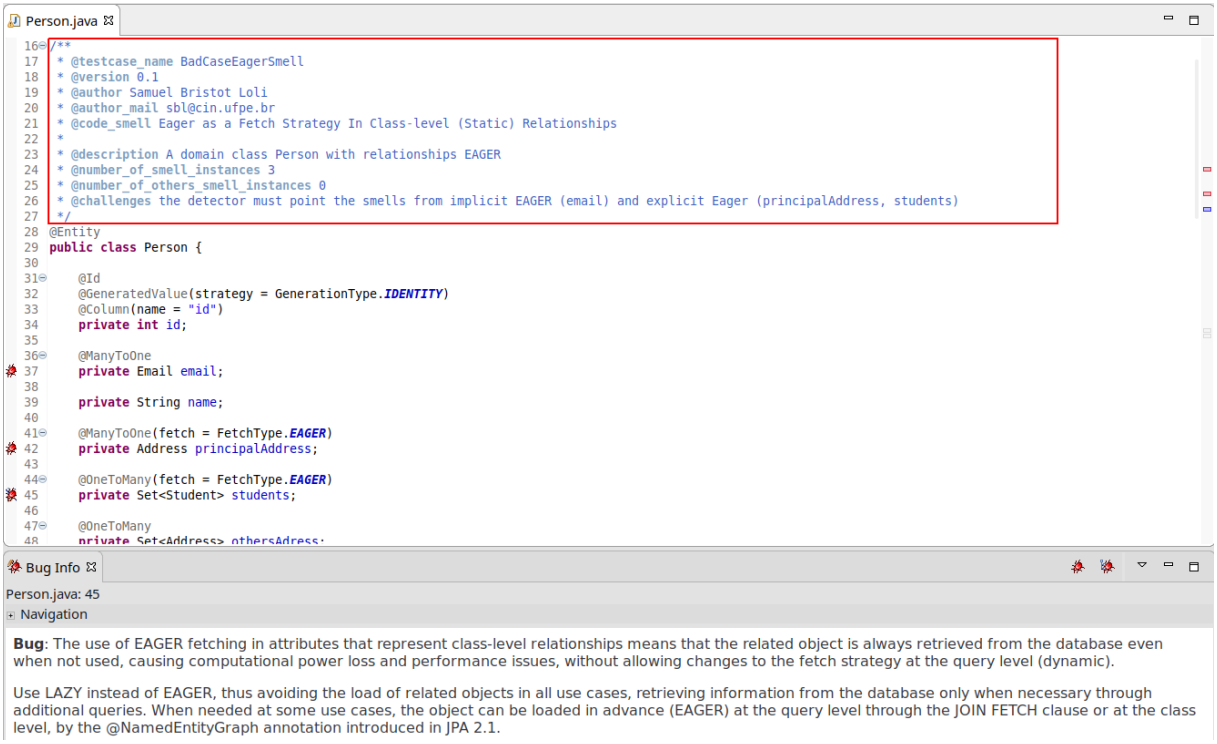
7.2.1 Aplicação de Referência

A aplicação foi estruturada em pacotes, sendo cada pacote responsável por um code smell. Dentro de cada pacote existem dois subpacotes, um chamado `BadCase` contendo exemplos que o detector deve identificar como *smell*, outro chamado `GoodCase` com exemplos que o detector deve ignorar. Cada classe com os exemplos está documentada de forma a facilitar a validação da eficácia/revocação do detector, contendo o nome do *smell*, descrição, quantas instâncias existem do *smell* na classe, quantas instâncias de outros *code smells* (que não o do caso de teste) e quais os desafios para o detector. A Figura 25 apresenta um

[3] `https://bit.ly/32kWd6u`

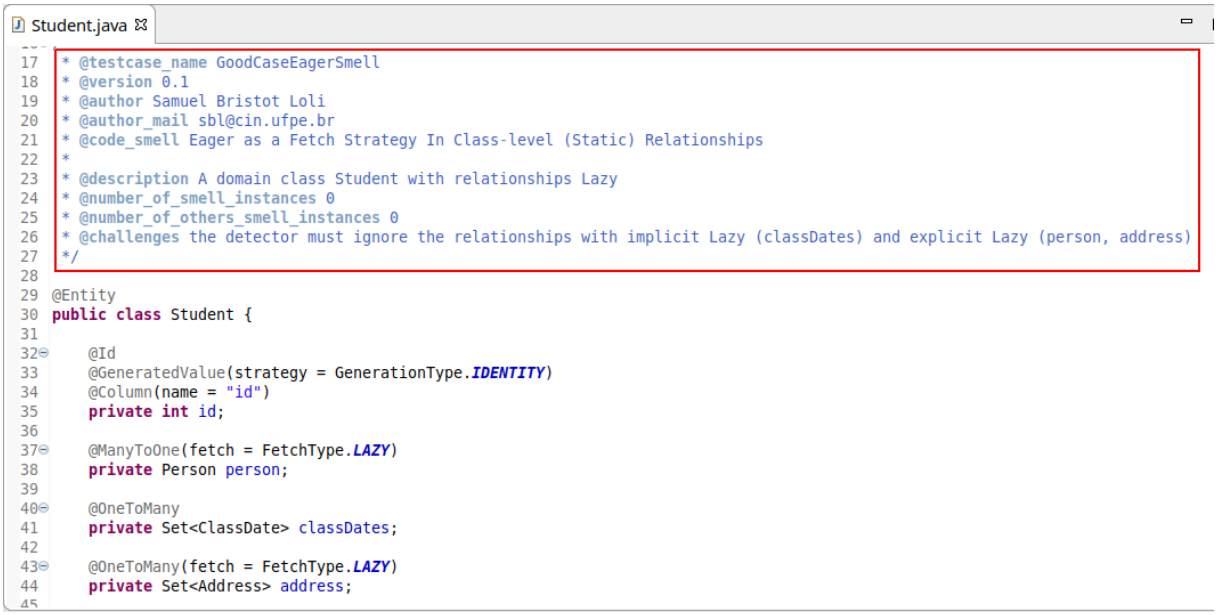
exemplo de `BadCase` e a Figura 26 um exemplo de *GoodCase*, ambos referentes ao *code smell* EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE.

Figura 25 – Caso de teste - BadCaseEagerSmell



Fonte: Elaborada pelo autor (2020)

Figura 26 – Caso de teste - GoodCaseEagerSmell



Fonte: Elaborada pelo autor (2020)

7.2.2 Resultado da avaliação

A avaliação foi realizada de acordo com os procedimentos metodológicos descritos na Seção 7.1 para análise da eficácia e revocação. Foram desenvolvidos três exemplos de testes positivos e três negativos para cada *smell*. O *smell* PROJECTIONSMELL possui seis exemplos devido a ser detectado também nos casos de testes para o *smell* N1JOINFETCHSMELL. A Tabela 14 apresenta a avaliação quanto à precisão da ferramenta utilizando a aplicação de referência. A coluna `detectados` apresenta a quantidade de *smells* identificados pela ferramenta, `TP` são os verdadeiros-positivos e `FP` os falsos-positivos verificados em análise manual. Dois detectores tiveram 100% de precisão, enquanto o detector PROJECTIONSMELL apresentou 80,00% e o detector N1JOINFETCH 66,67%, ambos com um falso-positivo do mesmo tipo, referente à projeção menor que três caracteres, limitação que foi mapeada na Seção 6.2.5.3. Esta limitação poderia ser evitada com a alteração da forma de análise das projeções, utilizando a verificação de atributos das entidades relacionadas na consulta ao invés do número de caracteres. Porém, não foi possível utilizar esta estratégia por limitações da arquitetura do SpotBugs em verificações de atributos e entidades distribuídos na *string* de consulta.

Tabela 14 – Analise de precisão - Aplicação de Referência

Smells	**Detectados**	**TP**	**FP**	**Precisão**
EagerSmell	3	3	0	100,00%
N1JoinFetchSmell	3	2	1	66,67%
OneToManyCollectionsSmell	3	3	0	100,00%
ProjectionSmell	5	4	1	80,00%

Fonte: Elaborada pelo autor (2020)

O resultado da avaliação quanto à revocação é apresentado na Tabela 11. A coluna `Analisados` apresenta a quantidade de instâncias de *code smells* ORM identificadas manualmente seguindo as etapas descritas na metodologia para cada *smell*. A coluna `TP` apresenta a quantidade de verdadeiros-positivos e `FN` refere-se aos falsos-negativos, que são *code smells* identificados mas não detectados pela ferramenta. Dois *smells* tiveram 100% de revocação enquanto outros dois obtiveram 66,67%. Para o smell PROJECTIONSMELL foram encontrados dois falsos-negativos referente ao *alias* maior que três caracteres, limitação também mapeada na Seção 6.2.5.3. Com relação ao N1JOINFETCHSMELL, foi encontrado um falso-negativo que não estava inicialmente mapeado, referente a objetos EAGER em classes de relacionamento que não a classe principal de retorno da consulta. Tentamos realizar o ajuste no detector sem sucesso, portanto adicionamos a limitação na seção 6.2.3.3, a qual também descreve a complexidade de implementação dessa verificação no detector.

A Tabela 16 apresenta o *f1-score* mostrando a média harmônica utilizando os dados de precisão e revocação para avaliação utilizando a aplicação de referência.

Tabela 15 – Análise de revocação - Aplicação de Referência

Smells	Analisados	TP	FN	Revocação
EagerSmell	3	3	0	100,00%
N1JoinFetchSmell	3	2	1	66,67%
OneToManyCollectionsSmell	3	3	0	100,00%
ProjectionSmell	6	4	2	66,67%

Fonte: Elaborada pelo autor (2020)

Tabela 16 – F1-Score - Aplicação de Referência

Smells	Precisão	Revocação	F1-Score
EagerSmell	100,00%	100,00%	100,00%
N1JoinFetchSmell	66,67%	66,67%	66,67%
OneToManyCollectionsSmell	100,00%	100,00%	100,00%
ProjectionSmell	66,67%	80,00%	72,73%

Fonte: Elaborada pelo autor (2020)

A aplicação de referência também pode ser útil como um conjunto de dados para testar a aplicação da ferramenta de detecção de *code smells* ORM. Além de verificar a precisão e revocação em um ambiente controlado, é possível utilizá-la para descobrir novas limitações, como relatado no caso do N1JOINFETCHSMELL. A aplicação se mostrou uma boa forma de testes iniciais para potenciais novos detectores ou ferramentas que avaliem *code smells* ORM. Em geral, a ferramenta apresentou uma precisão e revocação pior para os *smells* que necessitam analisar a *string* de consulta (PROJECTIONSMELL e N1JOINFETCHSMELL) e uma precisão e revocação melhor para detectores que analisam anotações (EAGERSMELL e ONTETOMANYCOLLECTIONSSMELL). Na próxima Seção aplicaremos a ferramenta em projetos reais.

7.3 AVALIAÇÃO UTILIZANDO PROJETOS REAIS

Selecionamos alguns projetos do GitHub e GitLab do IFSC para executar a ferramenta e confrontar com a análise manual dos *smells* para verificação em termos de precisão e revocação. Os projetos foram selecionados em procedimento detalhado na Seção 6.2.1.

7.3.1 Aplicações Selecionadas

Selecionamos quatro aplicações de tamanhos distintos. Dois projetos maiores (SIGAA e BroadleafCommerce) e dois menores (PAEVS e Shopping). Abaixo descrevemos uma visão rápida sobre cada projeto utilizado:

- SIGAA: Sistema de gestão acadêmica utilizado pelo IFSC. O sistema foi adquirido

junto a Universidade Federal do Rio Grande do Norte (UFRN) e modificado para se adequar as regras do IFSC. Atualmente é mantido por uma equipe de 12 desenvolvedores. O sistema possui atualmente 1.397.997 Lines of code (LOC) e utiliza o *Hibernate* como *framework* ORM.

- BroadleafCommerce: Fornece um *framework* de comércio eletrônico em Java e Spring para utilização por empresas do ramo. A aplicação utiliza JPA e Hibernate como *framework* ORM. O projeto possui mais de 1400 estrelas e 1100 *forks* no GitHub, possuindo um total de 337.161 LOC.

- PAEVS: Sistema utilizado para gerenciamento de bolsas referentes a assistência estudantil. Possui 24.640 LOC e também é mantido pela mesma equipe do SIGAA.

- Shopping: Sistema de shopping online, utilizando Java e Spring MVC. O projeto utiliza Hibernate como *framework* ORM possui um total de 2.305 LOC, 141 estrelas e 78 *forks* no GitHub.

7.3.2 Resultado

Realizamos a avaliação manual dos *code smells* detectados seguindo os procedimentos metodológicos descrito na Seção 6.2.1. Devido à dificuldade de analisar integralmente os dois projetos maiores, extraímos de forma aleatória três pacotes para detecção e análise de cada projeto. No SIGAA utilizamos os pacotes: ENSINO com 110.517 LOC, SISU com 9.973 LOC e ESTAGIO com 29.751 LOC. Para o BroadleafCommerce, utilizamos: BROADLEAF-CONTENTMANAGEMENT-MODULE com 18.636 LOC, BROADLEAF-FRAMEWORK com 107.606 LOC e BROADLEAF-PROFILE com 9808 LOC. Os dois projetos menores foram analisados integralmente. A Tabela 17 apresenta o resultado da avaliação de precisão da ferramenta, com a quantidade de instâncias por *smells* aplicada aos quatro projetos supracitados. O nível de precisão variou de 100% (EAGERSMELL e ONETOMANYCOLLECTIONSSMELL) e à 92,86% (N1JOINFETCHSMELL). Dois falsos-positivos detectados para PROJECTIONSMELL são referentes à consultas ORM com objetos que foram posteriormente atualizados, portanto, não sendo consideradas consultas com objetivo somente leitura. A dificuldade para o detector identificar o tipo de objetivo do retorno é explicada na Seção 6.2.5.3. Os demais falsos-positivos detectados, tanto para PROJECTIONSMELL, quanto para N1JOINFETCHSMELL estão relacionados a uma limitação conhecida relacionada as consultas ORM com projeção contendo menos de três caracteres fazendo o detector reportar como *smell*, também explicado com mais detalhes na Seção 6.2.5.3.

O resultado da análise para avaliar a revocação nos quatro projetos selecionados é apresentado na Tabela 18. Os *smells* EAGERSMELL e N1JOINFETCHSMELL obtiveram 100% de revocação, repetindo o resultado apresentado na avaliação da precisão. Para os demais, o nível de revocação ficou inferior ao nível de precisão, gerando um número maior de falsos-

Tabela 17 – Análise de precisão - Aplicação em projetos reais

Smells	Detectados	TP	FP	Precisão
EagerSmell	294	294	0	100,00%
N1JoinFetchSmell	14	13	1	92,86%
OneToManyCollectionsSmell	1	1	0	100,00%
ProjectionSmell	106	101	5	95,33%

Fonte: Elaborada pelo autor (2020)

negativos que falsos-positivos. A maior parte dos falsos-negativos de *ProjectionSmell* estão relacionados à consultas ORM contendo o *alias* da entidade maior que três caracteres, fazendo o detector identificar como projeção de forma incorreta e não acusando o *smell* conforme limitação abordada na Seção 6.2.5.3. Quatro falsos-negativos estão relacionados a consultas ORM que estão divididas entre métodos diferentes, estando a projeção com uma série de condicionais para formulação e em um método distinto do corpo da consulta. A dificuldade pro detector neste caso é por não conseguir analisar estaticamente condicionais para saber se o método irá retornar uma *String* vazia ou uma projeção. Os dois falsos-negativos de N1JOINFETCHSMELL estão relacionados a atributos EAGER sem JOIN FETCH de entidades diferentes da principal da consulta, limitação explicada em mais detalhes na Seção 6.2.3.3.

Tabela 18 – Analise de revocação - Aplicação em projetos reais

Smells	Analisados	TP	FN	Revocação
EagerSmell	294	294	0	100,00%
N1JoinFetchSmell	15	13	2	86,67%
OneToManyCollectionsSmell	1	1	0	100,00%
ProjectionSmell	117	101	16	87,18%

Fonte: Elaborada pelo autor (2020)

A Tabela 19 apresenta o *F1-Score* para verificação da média harmônica utilizando os dados de precisão e revocação calculados através da aplicação da ferramenta em projetos reais.

Tabela 19 – F1-Score - Aplicação em projetos reais

Smells	Precisão	Revocação	F1-Score
EagerSmell	100,00%	100,00%	100,00%
N1JoinFetchSmell	92,86%	86,67%	89,66%
OneToManyCollectionsSmell	100,00%	100,00%	100,00%
ProjectionSmell	95,33%	87,18%	91,07%

Fonte: Elaborada pelo autor (2020)

Os distintos níveis de precisão e revocação entre os diferentes tipos de *smells* podem

ser explicados pela forma como a ferramenta realiza a análise para identificação. Os *smells* que apresentaram 100% são relativos à verificação de anotações que contêm mapeamentos padronizados pelo JPA, facilitando o desenvolvimento das regras de detecção e tornando a identificação dos *smells* mais precisa. Diferentemente, os outros dois *smells* analisam *Strings* de consultas ORM. Essa é uma análise mais complexa para o detector pelo fato de trabalhar com *Strings*, o que dificulta a análise por utilização do grafo de chamadas. Além disso as consultas podem ser criadas de diferentes formas, a depender do método utilizado, podem estar desordenadas e espalhadas por variáveis e métodos condicionais no código. Com relação aos *smells* que tiveram 100%, o *smell* ONETOMANYCOLLECTIONSSMELL teve apenas um resultado, o que não seria uma margem segura para afirmar que possui 100% de revocação/precisão. Acreditamos que mesmo com um número maior de resultados sua precisão e revocação manteriam-se no mesmo nível por também se basear em anotações, assim como o EAGERSMELL. Com os resultados iniciais obtidos, a ferramenta aparenta possuir um nível de precisão e revocação aceitável e pode ajudar a localizar os *code smells* ORM no código de forma a apresentar para o desenvolvedor analisar a necessidade ou não de refatoração. A partir desta análise, também foi possível identificar e validar limitações da ferramenta para futuras evoluções.

7.4 AVALIAÇÃO DA DIFUSÃO DOS *CODE SMELLS* ORM

Considerando o limite de precisão e revocação da ferramenta, podemos avaliar se os *code smells* ORM implementados realmente acontecem nos projetos reais e com que frequência. O resultado pode mostrar informações úteis, como averiguar se um *code smell* é extremamente raro para reavaliar esforços em melhorias no detector, ou se um *smell* é extremamente comum chegando a gerar um excessivo número de avisos. Para aplicar a ferramenta no maior número possível de projetos, utilizamos a abordagem através de *script* para automação das etapas de mineração de repositório, *download*, compilação dos projetos, execução da ferramenta e compilação dos dados. Foram analisados 379 projetos, contendo 164307 classes, 14288 pacotes. Encontramos um total de 2829 *code smells* ORM em 284 projetos, correspondendo a 75% dos projetos analisados. Detalhes sobre os projetos, bem como os arquivos resultantes da análise em XML, contendo tamanho e os *smells* encontrados por linha, estão disponíveis *online*.[4] A Tabela 20 apresenta a difusão por *code smell* ORM, apresentando o total de instâncias, a quantidade de projetos afetados por aquele *smell*, a quantidade mínima de instâncias e a quantidade máxima encontradas considerando o projeto com menos e mais instâncias de *smells* respectivamente.

Conforme observado na tabela, o maior número de instâncias de *smells* ocorre para EAGERSMELL com 1671 instâncias. Esse *smell* também tem o maior número de projetos afetados, 66% do total de projetos analisados possuem ao menos uma instância. Acreditamos

[4] `https://github.com/samuelbl/projects-orm-smells-report`

Tabela 20 – Difusão dos *smells* em projetos reais

Smells	Total Inst.	Proj. Afetados	Min p/ Proj.	Max p/ Proj.
EagerSmell	1671	250 (66%)	0	200
N1JoinFetchSmell	184	51 (13%)	0	16
OneToManyCollectionsSmell	72	18 (5%)	0	42
ProjectionSmell	902	142 (37%)	0	78

Fonte: Elaborada pelo autor (2020)

que um dos possíveis motivos de sua grande utilização é por ser o valor padrão para anotações `@OneToOne` e `@ManyToOne` por regras impostas pelo JPA, apesar de não recomendado pelos *frameworks* ORM (Oracle, 2015; Mihalcea *et al.*, 2018) conforme discutido na Seção 4.1.1 do catálogo. Outro possível motivo, é a utilização como uma forma de corrigir o problema do $N + 1$ discutido na Seção 4.2.2. Apesar de gerar um grande número de avisos aos desenvolvedores devido ao alto número de instâncias, acreditamos que é extremamente importante alertar sobre a existência desse *smell*, pois se ajustado precocemente, pode evitar problemas relacionados à manutenção do código ao alterar para LAZY quando ainda há pouca ou nenhuma dependência, conforme também explicado na Seção 4.1.1. O segundo *smell* com mais casos detectados foi o PROJECTIONSMELL com 902 instâncias, afetando 37% dos projetos analisados, um número menor comparado ao EAGERSMELL. Esse *smell* apresentou um bom número de instâncias e está relacionado ao problema de excesso de dados, gerando incompatibilidade entre dados requeridos e dados recuperados conforme descrito na Seção 4.1.2. A quantidade desse *smell* deve ser analisada sob a ótica do percentual de precisão e revocação obtidos na análise da Seção 7.3, podendo ser mais ou menos dentro da margem analisada. Isso também deve ser considerado para o *smell* N1JOINFETCHSMELL com 184 instâncias detectadas e com 13% dos projetos afetados. Mesmo apresentando um menor número de instâncias se comparado aos dois primeiros, sua detecção continua importante já que a falta de JOIN FETCH está relacionada ao problema do $N + 1$, conforme Seção 4.2.1. Por último, temos o ONETOMANYCOLLECTIONSSMELL com 72 instâncias, abrangendo 5% dos projetos analisados. Apesar do baixo percentual de projetos afetados, acreditamos que investir esforços em sua detecção continua importante, pois esse *smell*, além de problemas de desempenho, pode impactar a manutenibilidade do código na correção, pois alterar o `@OneToMany` para bidirecional requer alteração da estrutura do banco de dados.

O resultado da difusão dos *smells* foi próximo ao da análise realizada na Seção 7.3, apresentando a mesma ordenação entre os *smells* mais e menos detectados. Acreditamos que todos os *smells* detectados, mesmo que alguns em poucos projetos, são importantes para alertar o desenvolvedor de forma precoce para efetuar, se for o caso, uma correção antes de gerar maiores problemas com a expansão do código do sistema e da base de dados.

8 CONCLUSÕES

O incentivo ao desenvolvimento ágil, diante da necessidade de gerar produtos melhores com um menor tempo de execução, transforma a tecnologia ORM em uma grande aliada ao acesso a dados de maneira simplificada, considerando o contexto de orientação de objetos. *Frameworks* ORM fornecem uma abstração conceitual, mapeando os registros do banco de dados para os objetos, possibilitando aos desenvolvedores se concentrarem mais no desenvolvimento das regras de negócio e menos em questões de acesso a dados. No entanto, essa abstração pode levar a vários problemas devido a opções de implementação abaixo do ideal no código ORM. Esses sintomas que indicam código de baixa qualidade são chamados de *code smells*. Nesta pesquisa, propomos um catálogo de *code smells* ORM em Java (Loli *et al.*, 2020) de maneira sistemática através de evidências encontradas no contexto da literatura e da prática.

O catálogo foi avaliado em relação ao nível de concordância aos code *smells* e soluções apresentadas aos desenvolvedores Java por meio de um questionário, obtendo 86 respostas com uma concordância média de 75,44 %. Houve um aumento de 15,23 % no nível de concordância geral com o catálogo quando comparadas respostas de desenvolvedores experientes com desenvolvedores iniciantes/intermediários, podendo estar relacionado ao fato de desenvolvedores experiente estarem mais propensos a corrigir problemas complexos (Tsunoda *et al.*, 2017) e presenciarem com mais frequência os erros causados pelos *code smells* ORM apresentados no catálogo. Acreditamos que este catálogo pode ser útil não apenas para pesquisadores, mas também para profissionais. Como por exemplo, apresentar o catálogo de *code smells* ORM a desenvolvedores Java com o objetivo de evitar más práticas de codificação ORM já no início do desenvolvimento, evitando problemas futuros de manutenção e desempenho.

Como proposta adicional, desenvolvemos uma ferramenta de detecção automatizada que utiliza análise estática do código, possuindo detectores para quatro *code smells* ORM apresentados no catálogo. Com a ferramenta é possível facilitar a localização destes *code smells* no código auxiliando o desenvolvedor de software na identificação durante o desenvolvimento, facilitando a correção na fase de menor custo de mudanças no código. Realizamos duas avaliações na ferramenta para verificar a precisão e revocação. Para primeira avaliação desenvolvemos uma aplicação de referência contendo um conjunto de dados referentes a exemplos de *code smells* obtendo uma avaliação inicial positiva, descobrindo também novas limitações dos detectores antes da aplicação da ferramenta em outros projetos. Na segunda avaliação, aplicamos a ferramenta em quatro projetos reais com resultados próximos ao executado anteriormente, com dois *smells* obtendo 100% de precisão e revocação e dois *smells* obtendo uma precisão e revocação menor. Adicionalmente aplicamos a ferramenta em 379 projetos, detectando 2829 *code smells* ORM em 75% dos projetos analisados. O *smell* EAGER COMO ESTRATÉGIA DE BUSCA NOS RELACIONAMENTOS A NÍVEL DE CLASSE (ESTÁTICO) foi

o mais citado pelas fontes nas revisões realizadas (Capítulo 3), o que teve o maior nível de concordância entre os participantes da avaliação (Capítulo 5) e o que foi mais detectado pela ferramenta nos projetos através de seu detector EAGERSMELL, com 1671 instâncias e um percentual de 66% de projetos afetados (Capítulo 7). Isso mostra a importância do catálogo, visto que o *smell* com mais fontes, maior consenso entre os desenvolvedores é também o mais detectado nos projetos atuais. No restante do capítulo, apresentamos os trabalhos relacionados a pesquisa, as limitações e ameças a validade das etapas executadas, bem como a sugestão de trabalhos futuros.

8.1 TRABALHOS RELACIONADOS

Nesta seção, discutimos os trabalhos relacionados a presente pesquisa. Iremos abordar estudos referentes à duas áreas associadas: *code smells* e problemas relacionados a ORM.

Estudos anteriores propõem várias abordagens para catalogar ou detectar *code smell* em diferentes contextos. Monteiro & Fernandes (2006) apresentam um catálogo de refatorações e *code smells* para o AspectJ, concentrando-se em um grupo específico de *code smell* para código-fonte orientado a aspectos. Garcia *et al.* (2009) discute *code smells* em aspectos arquiteturais de um projeto, descrevendo quatro *code smells* com relação a arquitetura encontrados através de engenharia reversa de código. Em estudo complementar, Garcia *et al.* (2009) expande sua pesquisa, descrevendo em maiores detalhes os *code smells* arquiteturais, utilizando também como fonte uma revisão da literatura em conjunto com análise de engenharia reversa. Aniche *et al.* (2018) provê um catálogo de seis *code smells* relacionados à arquitetura Model-View-Controller (MVC), os *code smells* foram definidos a partir de uma pesquisa com 53 desenvolvedores que utilizam a arquitetura MVC. Fard & Mesbah (2013) propõem um conjunto de 13 *code smell* JavaScript coletados através de diversos recursos utilizadas pelos desenvolvedores com abordagem baseada em métricas para detectar *code smells* no código. Sharma *et al.* (2018) apresentam um catálogo contendo 13 *smells* relacionados à esquemas de banco de dados. Os autores realizaram uma pesquisa por meio de *survey* com desenvolvedores para avaliar as perspectivas sobre os *smells* elicitados. Também foram extraídas, através de expressões regulares, instruções SQL incorporadas no código de 2925 projetos reais. Essas instruções foram analisadas através de uma ferramenta desenvolvida para análise da difusão desses *smells*. De forma diferente do nosso trabalho, esses *smells* estão relacionados ao esquema de banco de dados, como por exemplo: *God table* relacionado a tabelas com excesso de colunas e *Index abuse* referente ao uso incorreto de índices em tabelas. Borrelli *et al.* (2020) verificam *smells* no domínio de desenvolvimento de jogos usando o *framework* Unity, apresentando sete *bad smells*, encontrados através de análise da literatura existente e fóruns de discussão. Alguns dos *smells* propostos afetam o desempenho dos jogos, enquanto outros afetam a manutenção. Os *smells* foram avaliados quanto a sua relevância em uma pesquisa com 68 participantes. Os autores implementaram uma ferramenta chamada

UnityLinter para verificação através de análise estática destes *smells*. A ferramenta foi aplicada em 100 projetos para analisar a ocorrência dos *smells* detectando-os em de 39% à 97% dos projetos analisados. Também avaliaram a precisão e revocação, obtendo entre 86% à 100% de precisão e 50% à 100% de revocação. Da mesma forma que esses estudos, nossa pesquisa tem uma dimensão específica de domínio, ou seja, o *code smells* relacionados a utilização de ORM em Java.

Com relação a problemas relacionados a ORM, estudos anteriores propõem várias abordagens para identificar e verificar esses problemas. Nazário *et al.* (2019) desenvolveram um *framework* que detecta e relata 12 problemas referentes ao ORM identificados por meio de um estudo observacional, avaliado positivamente em um experimento com 13 desenvolvedores. Os problemas abordados nesse estudo são relacionados à incompatibilidade entre o esquema do banco de dados e o mapeamento manual feito nas classes de entidades. Isso ocorre pelo fato dos desenvolvedores não possuírem acesso à base de produção. A maior parte desses problemas geram avisos ou erros em tempo de compilação ou tempo de execução ao integrar o código ORM ao banco de dados. Em nosso estudo, os *code smells* ORM não geram avisos ou impedem a compilação/execução do sistema, são práticas com sintomas silenciosos que podem levar a problemas futuros. Chen *et al.* (2014) propuseram um *framework* para detectar anti-padrões de desempenho ORM com base na análise de código estático, com dois anti-padrões de desempenho apresentados. Avaliaram o *framework* aplicando em três sistemas, com uma melhora de desempenho em 98%. Posteriormente, eles forneceram uma abordagem automatizada para localizar problemas de dados redundantes no código ORM, aplicando também em três sistemas e localizando dados redundantes em 87% das transações realizadas (Chen *et al.*, 2016a). Węgrzynowicz (2013) também realiza um estudo a respeito de anti-padrões de desempenho, apresentando cinco anti-padrões relacionados ao uso de associações um-para-muitos no Hibernate. A maioria dos estudos anteriores sobre ORM visa detectar anti-padrões focados no desempenho. No entanto, os *code smells* não se limitam a esses problemas, apresentam sintomas que podem gerar ou não problemas de manutenção e desempenho. Não encontramos estudos que apresentassem *code smells* no código ORM.

8.2 LIMITAÇÕES E AMEAÇAS À VALIDADE

Nesta seção, apresentamos as ameaças à validade para: metodologia de pesquisa, o catálogo de *code smells* ORM, a avaliação realizada para validação do catálogo, a ferramenta proposta e a avaliação realizada na ferramenta.

Alguns procedimentos realizados utilizando RR apresentam ameaças à validade devido à sua metodologia mais leve se comparado a uma revisão sistemática completa da literatura (Cartaxo *et al.*, 2018). O GLR também possui suas ameaças especialmente pelo conteúdo não científico. As ameaças e limitações aplicáveis às duas revisões (RR e GLR) consistem no procedimento de seleção, conduzido por um único pesquisador, embora permitido por ambas

revisões aplicadas para celeridade no processo, pode ocasionar viés de seleção. Além disso, não foram realizadas avaliações de qualidade nas fontes, o que pode reduzir a confiança nas evidências encontradas. Com relação ao GLR, pelo fato de serem fontes não científicas, para atenuar essa situação realizamos em uma das etapas do procedimento de seleção uma análise qualitativa de forma geral nos estudos. Especificamente para RR, apenas um mecanismo de pesquisa foi usado durante o procedimento de busca, o que pode reduzir o número de evidências encontradas. Atenuamos isso usando um mecanismo que agrega vários bancos de dados. Para a GLR, limitar os resultados ao algoritmo de classificação do Google impõe um limite restrito ao número de recursos analisados, apesar de ser uma técnica recomendada por (Garousi *et al.*, 2019) como condição de parada, pode ter descartado evidências úteis. Devido à natureza prática da pesquisa, foram utilizados os 50 primeiros resultados da pesquisa, pois geralmente os usuários do Google tendem a se concentrar principalmente nas primeiras páginas. Por fim, o uso da literatura cinza devido a poucos estudos em literatura formal, embora importante nesse contexto, é considerado uma ameaça à validade por não possuir rigor científico.

Com relação ao catálogo, embora a linguagem de desenvolvimento Java possua uma padronização ORM com a implementação JPA, os frameworks podem ter algumas particularidades com problemas e smells específicos que podem não ser capturados pelo catálogo. Isso ocorre porque a escolha é ter um catálogo geral de *code smells* em JAVA e não específico a algum *framework*. Outra questão, é que nem todo *code smell* necessariamente precisa ser modificado. Por exemplo, o uso de EAGER em nível de classe pode nunca levar a um problema de desempenho ou manutenibilidade em casos específicos nos quais as informações são sempre necessárias ao relacionamento e com poucos registros na base de dados. No entanto, na maioria dos casos de uso, esse *smell* pode causar problemas, portanto, é recomendável uma verificação manual do desenvolvedor antes de uma modificação.

Em relação ao *survey* realizado, apesar de detalharmos os requisitos, como familiaridade com algum *framework* Java ORM para participação na avaliação da pesquisa, não podemos confirmar que os participantes realmente têm requisitos. Também não incluímos o detalhamento e discussão de cada *code smell* por razões de escopo, dificultando ao participante entender melhor as exceções ao *smell*.

A ferramenta desenvolvida possui limitações discutidas no Capítulo 6, como a não verificação de somente-leitura para consultas sem projeção, que podem trazer a presença de falsos-positivos e falsos-negativos no processo de detecção. As limitações foram abordadas nas Seção 6.2 referentes aos detectores.

A avaliação manual da ferramenta foi realizada pelo desenvolvedor, o que pode causar viés. Realizamos a descrição das etapas na metodologia de forma a atenuar e o processo poder ser repetido para validação. Para analisar os projetos do GitHub quanto a difusão dos *smells* tivemos que restringir a projetos com gerenciadores de dependência (Maven ou Gradle), pois facilitam a compilação do código a partir de *script*.

8.3 TRABALHOS FUTUROS

Com a análise dos resultados obtidos nas etapas desta pesquisa, foi possível identificar trabalhos futuros relacionados ao tema. Dentre eles podemos citar:

- Os *frameworks* ORM de diferentes linguagens de programação tem suas especificidades relacionadas a tecnologia ORM, o que dificulta a realização de um estudo geral sem um nível adequado de detalhes. Trabalhos futuros podem estudar *code smells* ORM em outras linguagens de desenvolvimento, como *frameworks* ORM de Python e PHP.
- Trabalhos futuros podem usar o catálogo para medir problemas gerados por *code smells* ORM em termos de desempenho e manutenção.
- Outro possível trabalho futuro é medir a qualidade do código ORM em uma empresa antes e depois do conhecimento adquirido com o catálogo.
- Um estudo utilizando engenharia reversa em projetos diversos pode ser realizado para descobrir novos *code smells* ORM de forma a expandir o catálogo.
- Trabalhos futuros podem expandir a ferramenta de forma a desenvolver novos detectores para os quatro *code smells* do catálogo não abordados na atual versão da ferramenta.

REFERÊNCIAS

Ahmed, S. (2018). Jpa tips: Avoiding the n + 1 select problem. Disponível em: https://www.javacodegeeks.com/2018/04/jpa-tips-avoiding-the-n-1-select-problem.html. Acesso em: 16 fev. 2020.

Aniche, M., Bavota, G., Treude, C., Gerosa, M. A., & van Deursen, A. (2018). Code smells for model-view-controller architectures. *Empirical Software Engineering*.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. *IEEE Software*, 25(5):22–29.

Barry, D. & Stanienda, T. (1998). Solving the java object storage problem. *Computer*, 31(11):33–40.

Bauer, C. & King, G. (2005). *Hibernate in Action*. In Action Series. Manning.

Bauer, C., King, G., & Gregory, G. (2016). *Java Persistence with Hibernate*. Manning Publications Co.

Borrelli, A., Nardone, V., Lucca, G. A. D., Canfora, G., & Penta, M. D. (2020). Detecting video game-specific bad smells in unity projects. In *17th International Conference on Mining Software Repositories (MSR '20)*.

Cartaxo, B., Pinto, G., & Soares, S. (2018). The role of rapid reviews in supporting decision-making in software engineering practice. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 24–34.

Cartaxo, B., Pinto, G., & Soares, S. (2020). *Contemporary Empirical Methods in Software Engineering*, chapter Rapid Reviews in Software Engineering. Springer, first edition.

Cartaxo, B., Pinto, G., Vieira, E., & Soares, S. (2016). Evidence briefings: Towards a medium to transfer knowledge from systematic reviews to practitioners. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 57.

Chen, T. (2015). Improving the quality of large-scale database-centric software systems by analyzing database access code. In *2015 31st IEEE International Conference on Data Engineering Workshops*, 245–249.

Chen, T., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016a). Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 42(12):1148–1161.

Chen, T., Shang, W., Yang, J., Hassan, A. E., Godfrey, M. W., Nasser, M., & Flora, P. (2016b). An empirical study on the practice of maintaining object-relational mapping code in java systems. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 165–176.

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, 1001–1012.

Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., & Saraiva, J. (2012). Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 243–244.

Dhingra, N., Abdelmoghith, E., & Mouftah, H. (2017). Review on jpa based orm data persistence framework. *International Journal of Computer Theory and Engineering*, 9:318–328.

Dubois, J. (2013). Improving the performance of the spring-petclinic sample application (part 1 of 5). Disponível em: `encurtador.com.br/bxIUV`. Acesso em: 19 fev. 2019.

Durix, H. (2020). Boost the performance of your spring data jpa application. Disponível em: `https://blog.ippon.tech/boost-the-performance-of-your-spring-data-jpa-application/`. Acesso em: 16 fev. 2020.

EclipseLink (2017). Faq - migrate hibernate to eclipselink. Disponível em: `https://www.eclipse.org/eclipselink/documentation/2.5/solutions/migrhib002.htm`. Acesso em: 09 set. 2019.

EclipseLink (2019). What is object-relational mapping. Disponível em: `https://wiki.eclipse.org/EclipseLink/FAQ/JPA#What_is_Object-Relational_Mapping`. Acesso em: 25 abr. 2019.

Elmari, R. & Navathe, S. (2010). *Sistemas de Banco de Dados. 6ª Edição*. Editora Pearson.

Elssamadisy, A. & Schalliol, G. (2002). Recognizing and responding to "bad smells"in extreme programming. In *Proceedings of the 24th International Conference on Software Engineering*, 617–622.

Engines, D. (2019). Db engines ranking. Disponível em: `https://db-engines.com/en/ranking_categories/`. Acesso em: 30 set. 2019.

Fabrício, U. (2015). I discovered an undocumented way to improve jpa performance. Disponível em: `https://bewire.be/blog/i-discovered-an-undocumented-way-to-improve-jpa-performance/`. Acesso em: 16 fev. 2020.

Fard, A. M. & Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 116–125.

Fontana, F. A., Ferme, V., Zanoni, M., & Yamashita, A. (2015). Automatic metric thresholds derivation for code smell detection. In *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*, 44–53.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series (Fowler). Pearson Education.

Frakes, W. B. & Baeza-Yates, R. (1992). *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc.

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Identifying architectural bad smells. In *CSMR'09*.

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In Mirandola, R., Gorton, I., & Hofmeister, C., editors, *Architectures for Adaptive Software Systems*.

Garousi, V., Felderer, M., & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101 – 121.

GitHub (2020). Pesquisar repositórios. Disponível em: `https://docs.github.com/pt/github/searching-for-information-on-github/searching-for-repositories`. Acesso em: 16 jun. 2020.

Gomes, G. (2016). Performance improvement in java applications: Orm / jpa. Disponível em: `https://dzone.com/articles/performance-improvement-in-java-applications-orm-j`. Acesso em: 16 fev. 2020.

Goodman, L. (1961). Snowball sampling. *Ann Math Stat*, 32.

Grigera, J., Garrido, A., & Rivero, J. M. (2014). A tool for detecting bad usability smells in an automatic way. In Casteleyn, S., Rossi, G., & Winckler, M., editors, *Web Engineering*.

Hut, C. (2015). Avoiding jpa performance pitfalls. Disponível em: `https://www.veracode.com/blog/secure-development/avoiding-jpa-performance-pitfalls`. Acesso em: 16 fev. 2020.

Janssen, T. (2015). How to improve jpa performance | rebel. Disponível em: `https://www.jrebel.com/blog/how-to-improve-jpa-performance/`. Acesso em: 16 fev. 2020.

Janssen, T. (2017a). *Hibernate Tips*. Nermina Miller.

Janssen, T. (2017b). Solve hibernate performance issues in development. Disponível em: `https://stackify.com/find-hibernate-performance-issues/`. Acesso em: 16 fev. 2020.

Janssen, T. (2018). 3 common hibernate performance issues in your log. Disponível em: `https://www.baeldung.com/hibernate-common-performance-problems-in-logs`. Acesso em: 16 fev. 2020.

Janssen, T. (2019a). Entities or dtos – when should you use which projection. Disponível em: `https://thoughts-on-java.org/entities-dtos-use-projection`. Acesso em: 09 set. 2019.

Janssen, T. (2019b). Mapping definitions in jpa and hibernate – annotations, xml or both. Disponível em: `https://thoughts-on-java.org/mapping-definitions-jpa-hibernate-annotations-xml/`. Acesso em: 30 set. 2019.

Janssen, T. (2020). 7 tips to boost your hibernate performance - thoughts on java. Disponível em: https://thoughts-on-java.org/tips-to-boost-your-hibernate-performance/. Acesso em: 16 fev. 2020.

Keith, M., Schincariol, M., & Nardone, M. (2018). *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs*. Apress.

Khomh, F., Di Penta, M., & Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, 75–84.

Loli, S., Teixeira, L., & Cartaxo, B. (2020). A catalog of object-relational mapping code smells for java. In *SBES 2020 Research*.

Maple, S. & Binstoc, A. (2018). Jvm ecosystem report 2018. In *JVM Ecosystem Report 2018*.

Marciniec, M. (2019). Jpa: N+1 select problem [spring & jpa pitfalls series. Disponível em: https://codete.com/blog/jpa-n-plus-1-select-problem/. Acesso em: 16 fev. 2020.

Mat, B. (2011). Hibernate performance - stack overflow. Disponível em: https://stackoverflow.com/questions/5155718/hibernate-performance. Acesso em: 16 fev. 2020.

Medeiros, I. (2015). Java persistence api: Otimizando a performance das aplicações. Disponível em: https://www.devmedia.com.br/java-persistence-api-otimizando-a-performance-das-aplicacoes/32091. Acesso em: 16 fev. 2020.

Mihalcea, V. (2016a). Hibernate performance tuning and best practices. Disponível em: https://in.relation.to/2016/09/28/performance-tuning-and-best-practices/. Acesso em: 16 fev. 2020.

Mihalcea, V. (2016b). High-performance hibernate devoxx france. Disponível em: https://www.slideshare.net/VladMihalcea/high-performance-hibernate-devoxx-france. Acesso em: 16 fev. 2020.

Mihalcea, V. (2016c). *High-Performance Java Persistence*. Vlad Mihalcea.

Mihalcea, V. (2017). Performance features. Disponível em: https://www.eclipse.org/eclipselink/documentation/2.7/solutions/performance001.htm/. Acesso em: 02 fev. 2020.

Mihalcea, V. (2019a). Eager fetching is a code smell when using jpa and hibernate. Disponível em: https://vladmihalcea.com/eager-fetching-is-a-code-smell/. Acesso em: 16 fev. 2020.

Mihalcea, V. (2019b). Hibernate performance tuning tips. Disponível em: https://vladmihalcea.com/hibernate-performance-tuning-tips/. Acesso em: 30 ago. 2019.

Mihalcea, V. (2019c). The hibernate.enable_lazy_load_no_trans anti-pattern. Disponível em: https://vladmihalcea.com/the-hibernate-enable_lazy_load_no_trans-anti-pattern. Acesso em: 16 jan. 2019.

Mihalcea, V. (2019d). How to detect the hibernate n+1 query problem during testing. Disponível em: https://vladmihalcea.com/how-to-detect-the-n-plus-one-query-problem-during-testing/. Acesso em: 28 jul. 2019.

Mihalcea, V. (2020a). The best way to handle the lazyinitializationexception. https://vladmihalcea.com/the-best-way-to-handle-the-lazyinitializationexception/. Acesso em: 29 mar. 2019.

Mihalcea, V. (2020b). The best way to prevent jpa and hibernate performance issues. Disponível em: https://vladmihalcea.com/jpa-hibernate-performance-issues/. Acesso em: 16 fev. 2020.

Mihalcea, V. (2020c). Manytoone jpa and hibernate association best practices. Disponível em: https://vladmihalcea.com/manytoone-jpa-hibernate/. Acesso em: 20 jul. 2020.

Mihalcea, V., Ebersole, S., Andrea Boriero, G. M., Badner, G., Cranford, C., Bernard, E., Grinovero, S., Meyer, B., Ferentschik, H., King, G., Bauer, C., Andersen, M. R., Maesen, K., Vansa, R., & Jacomet, L. (2018). Hibernate orm 5.4.3.final user guide. Disponível em: https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html. Acesso em: 16 fev. 2020.

Mihalcea, V., Ebersole, S., Andrea Boriero, G. M., Badner, G., Cranford, C., Bernard, E., Grinovero, S., Meyer, B., Ferentschik, H., King, G., Bauer, C., Andersen, M. R., Maesen, K., Vansa, R., & Jacomet, L. (2019). Hql and jpql. Disponível em: https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#hql. Acesso em: 30 jul. 2019.

Monteiro, M. P. & Fernandes, J. M. (2006). Towards a catalogue of refactorings and code smells for aspectj. In Rashid, A. & Aksit, M., editors, *Transactions on Aspect-Oriented Software Development I*.

Munhoz, J. (2019). Hibernate and the n+1 selections problem - quintoandar tech. Disponível em: https://medium.com/quintoandar-tech-blog/hibernate-and-the-n-1-selections-problem-c497710fa3fe. Acesso em: 16 fev. 2020.

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

Nazário, M. F. C., Guerra, E., Bonifácio, R., & Pinto, G. (2019). Detecting and reporting object-relational mapping problems: An industrial report. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–6.

OpenJpa, A. (2019). Apache openjpa 3.1 user's guide. Disponível em: http://openjpa.apache.org/builds/3.1.0/apache-openjpa/docs/manual.pdf. Acesso em: 30 abr. 2019.

Oppenheim, A. (2000). *Questionnaire Design, Interviewing and Attitude Measurement*. Bloomsbury Academic.

Oracle (2015). 9 oracle toplink (eclipselink) jpa performance tuning. Disponível em: `https://docs.oracle.com/middleware/1212/core/ASPER/toplink.htm`. Acesso em: 16 fev. 2020.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 268–278.

Sharma, T., Fragkoulis, M., Rizou, S., Bruntink, M., & Spinellis, D. (2018). Smelly relations: Measuring and understanding database schema quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 55–64.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2006). *Sistema de Banco de Dados*. Elsevier Editora Ltda.

Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics based refactoring. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 30–38.

SpotBugs (2018a). Implement spotbugs plugin. Disponível em: `https://spotbugs.readthedocs.io/en/stable/implement-plugin.html`. Acesso em: 10 jun. 2019.

SpotBugs (2018b). Spotbugs manual. Disponível em: `https://spotbugs.readthedocs.io/en/stable/`. Acesso em: 11 set. 2019.

Terra, R. & Bigonha, R. S. (2008). Ferramentas para análise estática de códigos java. *Monografia, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte*.

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498 – 1516.

Tomassi, D. A. (2018). Bugs in the wild: Examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 980–982.

Tsunoda, T., Washizaki, H., Fukazawa, Y., Inoue, S., Hanai, Y., & Kanazawa, M. (2017). Evaluating the work of experienced and inexperienced developers considering work difficulty in sotware development. In *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 161–166.

Vial, G. (2018). Lessons in persisting object data using object-relational mapping. *IEEE Software*.

White, O. (2014). Java tools and technologies landscape for 2014. *Retrieved Octubre*, 30:2014.

Węgrzynowicz, P. (2013). Performance antipatterns of one to many association in hibernate. In *2013 Federated Conference on Computer Science and Information Systems*, 1475–1481.

Yamashita, A. & Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering*, 682–691.