Uma abordagem inteligente para suporte à detecção e classificação automática de *design smells* em sistemas de software orientados a objetos através de ontologias

Vinicius Jonathan Santos Silva

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2019

Vinicius Jonathan Santos Silva

Uma abordagem inteligente para suporte à detecção e classificação automática de *design smells* em sistemas de software orientados a objetos através de ontologias

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Fabiano Azevedo Dorça

Uberlândia
2019

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S586 Silva, Vinícius Jonathan Santos, 1992-
2019 Uma abordagem inteligente para suporte à detecção e classificação automática de design smells em sistemas de software orientados a objetos através de ontologias [recurso eletrônico] / Vinícius Jonathan Santos Silva. - 2019.

Orientador: Fabiano Azevedo Dorça.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2019.2566
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Dorça, Fabiano Azevedo,1979-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação de Mestrado Acadêmico, 15/2019, PPGCO				
Data:	23 de setembro de 2019	Hora de início:	**09hrs00min**	Hora de encerramento:	**11hrs30min**
Matrícula do Discente:	11722CCP012				
Nome do Discente:	Vinicius Jonathan Santos Silva				
Título do Trabalho:	Uma Abordagem Inteligente para Suporte à Detecção e Classificação Automática de Design Smells em Sistemas de Software Orientados a Objetos Através de Ontologias				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Inteligência Artificial				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Marcelo de Almeida Maia - FACOM/UFU, Bernardo Pereira Nunes - DINF/PUC e Fabiano Azevedo Dorça - FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. Bernardo Pereira Nunes participou da defesa por meio de videoconferência desde a cidade do Rio de Janeiro -RJ. Os outros membros da banca e o aluno participaram in loco.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Fabiano Azevedo Dorça, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Fabiano Azevedo Dorça**, **Professor(a) do Magistério Superior**, em 25/09/2019, às 16:49, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério**

Superior, em 30/09/2019, às 14:34, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Bernardo Pereira Nunes**, **Usuário Externo**, em 11/10/2019, às 16:14, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1569385** e o código CRC **0EAB55BD**.

Referência: Processo nº 23117.084737/2019-83 SEI nº 1569385

Este trabalho é dedicado aos meus pais que, com grande esforço e trabalho duro, me ajudaram a chegar até aqui.

Agradecimentos

Agradeço, primeiramente a Deus, por me abençoar todos os dias proporcionando saúde e força para realizar meus objetivos.

Ao meu orientador, Professor Doutor Fabiano Azevedo Dorça, pelos ensinamentos e por me ajudar nos momentos de dificuldade.

Aos meus pais, Roberto e Vanilda, que sempre batalharam para proporcionar uma vida melhor aos seus filhos. Com certeza eu não seria quem sou hoje se não fosse por eles.

À minha irmã Monique, por sempre estar comigo nos momentos bons e ruins.

À minha namorada Jordana, por toda a paciência e compreensão durante os anos da pós graduação e por sempre me apoiar e nunca me deixar desistir.

Aos Professores Doutores Bernardo Pereira Nunes e Marcelo de Almeida Maia, por aceitarem o convite para compor a banca de avaliação.

Aos Professores da PPGCO por todo o ensinamento passado.

Aos amigos e colegas que me ajudaram direta ou indiretamente.

"O sucesso é a soma de pequenos esforços repetidos dia após dia."
(Robert Collier)

Resumo

Programação Orientada a Objetos (POO) é uma disciplina bastante complexa que exige o conhecimento de vários conceitos como herança, encapsulamento e polimorfismo. Esses conceitos são muito importantes para o aprendizado de conceitos mais avançados, como padrões de projeto e refatoração de código. Alguns conceitos quando aplicados de maneira incorreta podem levar ao surgimento de falhas de *design*, também conhecidas como *design smells*. *Design smells* são estruturas de *software* que podem indicar problemas de código ou *design* que tornam complexo o processo de evolução e manutenção do *software*. Aprender como evitar essas falhas e como refatorá-las é uma das habilidades mais importantes para se tornar um bom desenvolvedor orientado a objetos. Dessa forma, esse trabalho tem como objetivo o desenvolvimento de um *plug-in* para o ambiente de desenvolvimento Eclipse, para tornar a identificação de *smells* em códigos OO mais simples e rápida. Nosso método foi testado contra 4 ferramentas de propósito similar(DECOR, JDeodorant, CheckStyle e PMD), executando todas elas em 3 projetos *open-source* (JUnit, Log4J e ArgoUML) com a finalidade de identificação de 4 *design smells* (Insufficient Modularization, Long Method, Long Parameter List e Deficient Encapsulation). Logo após, realizamos uma análise por amostragem com o objetivo de demonstrar o poder de expressividade da utilização de ontologias ao identificar todos os 14 *design smells* propostos nesse trabalho. Os resultados obtidos mostraram que nossa ferramenta apresentou acurácia de 100% em todos os testes, resultados estes que se igualaram a algumas ferramentas em alguns casos ou foi superior a elas em outros.

Palavras-chave: Aprendizado. Refatoração. Programação orientada a objetos. Engenharia de Software. Design smells.

Abstract

Object oriented programming is a fairly complex discipline that requires knowledge of various concepts such as inheritance, encapsulation, and polymorphism. These concepts are very important for learning more advanced concepts such as design patterns and code refactoring. Some concepts when applied incorrectly can lead to design flaws, also known as design smells. Design smells are software structures that may indicate code or design problems that make the process of software evolution and maintenance complex. Learning how to avoid these failures and how to refactor them is one of the most important skills for becoming a good object-oriented developer. Thus, this work aims to develop a plug-in for the Eclipse development environment to make the process of identifying smells in OO code simpler and faster. Our method has been tested against 4 similar purpose tools (DECOR, JDeodorant, CheckStyle and PMD), running them all in 3 open source projects (JUnit, Log4J and ArgoUML) for the purpose of identifying 4 design smells (Insufficient Modularization, Long Method, Long Parameter List and Deficient Encapsulation). Soon after, we performed a sample analysis in order to demonstrate the expressive power of using ontologies by identifying all 14 design smells proposed in this work. The results obtained showed that our tool presented 100 % accuracy in all tests, results that were equal to some tools or were superior to them in others.

Keywords: Learning. Refactoring. Object-oriented programming. Software Engineering. Design smells.

Lista de ilustrações

Figura 1 – Projeto Java representado em JDT 37
Figura 2 – Ontology 101 40
Figura 3 – Data Properties 42
Figura 4 – Object Properties 43
Figura 5 – Grafo da ontologia 44
Figura 6 – Exemplo de código usando JDT 50
Figura 7 – Carregando uma ontologia utilizando a OWL API 51
Figura 8 – Criando indivíduos utilizando a OWL API 51
Figura 9 – Tornando indivíduos diferentes utilizando a OWL API 52
Figura 10 – Resultado da execução do OWLSmell em um projeto OO 52
Figura 11 – Projeto Java contendo todos os smells 65
Figura 12 – Todos os smells validados pelo OWLSmell 66

Lista de tabelas

Tabela 1 – Matriz de Confusão 55
Tabela 2 – Quantidade de *smells Insufficient Modularization* encontrados por cada ferramenta 59
Tabela 3 – Quantidade de ocorrências existentes do *smell Insufficient Modularization* por sistema 59
Tabela 4 – Quantidade válida de *smells Insufficient Modularization* segundo (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) 59
Tabela 5 – *Precision and Recall* do *smell Insufficient Modularization* 60
Tabela 6 – Quantidade de *smells Long Method* encontrados por cada ferramenta . 60
Tabela 7 – Quantidade de ocorrências existentes do *smell Long Method* por sistema 61
Tabela 8 – Quantidade válida de *smells Long Method* segundo (FOWLER, 1999) . 61
Tabela 9 – *Precision and Recall* do *smell Long Method* 61
Tabela 10 – Quantidade de *smells Long Parameter List* encontrados por cada ferramenta 62
Tabela 11 – Quantidade de ocorrências existentes do *smell Long Parameter List* por sistema 62
Tabela 12 – Quantidade válida de *smells Long Parameter List* segundo (FOWLER, 1999) 63
Tabela 13 – *Precision and Recall* do *smell Long Parameter List* 63
Tabela 14 – Quantidade de *smells Deficient Encapsulation* encontrados por cada ferramenta 63
Tabela 15 – Quantidade de ocorrências existentes do *smell Deficient Encapsulation* por sistema 64
Tabela 16 – Quantidade válida de *smells Deficient Encapsulation* segundo (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) 64
Tabela 17 – *Precision and Recall* do *smell Deficient Encapsulation* 64
Tabela 18 – Identificação dos smells propostos 67
Tabela 19 – Comparação dos smells analisados nos trabalhos correlatos 73

Lista de siglas

AST Abstract Syntax Tree

DMP Declarative Meta Programming

GOF Gang of Four

JDT Java Development Tools

OO Orientação a Objetos

OWL Web Ontology Language

POO Programação Orientada a Objetos

SWRL Semantic Web Rule Language

UML Unified Modeling Language

W3C World Wide Web Consortium

Sumário

1 INTRODUÇÃO 23
1.1 Problema e Motivação 24
1.2 Objetivos e Contribuições 24
1.3 Estrutura da Dissertação 25

2 FUNDAMENTAÇÃO TEÓRICA 27
2.1 *Design Smells* 27
2.2 Sistemas Especialistas 32
2.3 Ontologias e Web Semântica 34
2.4 O projeto Eclipse JDT 37

3 METODOLOGIA 39
3.1 Perguntas de pesquisa 39
3.2 *Criação da ontologia* 40
3.3 Inferência de conhecimento 45

4 O *PLUG-IN* OWLSMELL 49

5 VALIDAÇÃO DO MÉTODO PROPOSTO 53
5.1 Comparação de ferramentas 53
5.2 Análise por amostragem 54
5.3 Métricas *Precision and Recall* 54

6 EXPERIMENTOS E ANÁLISE DOS RESULTADOS 57
6.1 Comparação de ferramentas 57
6.1.1 Smell Insufficient Modularization 58
6.1.2 Smell Long Method 60
6.1.3 Smell Long Parameter List 62
6.1.4 Smell Deficient Encapsulation 63

6.2 Análise por amostragem 64
6.3 Avaliação dos Resultados 67

7 TRABALHOS CORRELATOS 69

8 CONCLUSÃO 75
8.1 Principais Contribuições 75
8.2 Produção Bibliográfica 76
8.3 Trabalhos Futuros 76

REFERÊNCIAS 79

Capítulo 1

Introdução

A programação orientada a objetos surgiu com o objetivo de tornar o desenvolvimento de *softwares* mais parecido com o mundo real, de forma a representar um conjunto de dados em forma de objetos e permitir que um sistema funcione através da comunicação e relacionamento entre esses objetos.

Para isso o paradigma orientado a objetos apresenta alguns conceitos básicos como herança (capacidade de um objeto herdar características de outro objeto), encapsulamento (capacidade de um objeto se proteger contra ações exteriores) e polimorfismo (capacidade de um objeto possuir diversas formas).

A primeira vista, esses conceitos podem ser bastante simples. No entanto, são conceitos que podem ser considerados complexos de se aplicar em determinados casos. Um exemplo bastante comum é aplicar esses conceitos para refatorar um código já existente e que possui baixa qualidade.

A utilização desses conceitos deve ser bem planejada, pois a má utilização dos mesmos pode desencadear problemas no *design* de um *software* orientado a objetos. Tais problemas, também conhecidos como *design smells* ou *code smells*, tornam o *software* incapaz de evoluir e aumentam a complexidade e o custo de manutenção. Assim, é de grande importância para a qualidade de um *software* que esses problemas sejam identificados e refatorados.

No entanto, os *smells* podem ser removidos do código aplicando-se boas práticas de refatoração e fazendo bom uso dos conceitos básicos da Orientação a Objetos (OO). Todavia, esse processo tende a ser complexo e custoso, uma vez que o ciclo de vida de um *software* não acaba, e todos os dias surgem novos requisitos a serem implementados, tornando quase impossível para o desenvolvedor conhecer todo o código que foi implementado no projeto.

Apesar de serem problemas relacionados a utilização de más práticas de programação, *code smells* e *design smells* possuem características bastante distintas. Atualmente, é possível encontrar diversos estudos a respeito dos *code smells*: (FOWLER, 1999), (SOBRINHO; LUCIA; MAIA, 2018), (MOHA et al., 2010), (FONTANA; BRAIONE; ZANONI,

2012) etc.

Desde o surgimento do termo em 1999, quando Martin Fowler apresentou o termo à comunidade, os estudos sobre *code smells* têm sido bastante difundidos. Mas com o passar dos anos, vários pesquisadores começaram a encontrar problemas de código mais amplos, associados ao *design* de um projeto de *software* e que impactaria uma porção maior de código fonte. Então em 2004, surgiu o termo *design smell* apresentado em (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Dessa forma, é importante destacar que o nível de complexidade ao se trabalhar com *design smells* é maior do que *code smells*, pois enquanto o segundo trabalha em um nível de características de cada classe, como quantidade de variáveis e/ou métodos, o primeiro trabalha em um nível mais profundo, analisando uma estrutura de classes e como essa estrutura impactará no *design* implementado.

1.1 Problema e Motivação

Melhorar a qualidade de um *software* é uma tarefa extremamente importante que deve ser realizada em toda a fase de desenvolvimento. Mas na maioria das vezes, os desenvolvedores de *software* não realizam essa tarefa por uma série de motivos, entre elas a quantidade excessiva de requisitos e/ou a falta de tempo.

Essa melhoria ocorre quando desenvolvedores aplicam boas práticas de desenvolvimento para resolução de problemas e criação de regras de negócio. Práticas simples como *clean code* já tornam o código mais fácil de ser entendido e, por consequência, melhora a qualidade de manutenção.

No entanto, a implementação de más práticas de desenvolvimento podem levar ao surgimento de *design smells* e/ou *code smells* tornando o *software* difícil de manter e evoluir.

O processo de identificar um *smell* em um projeto é considerado custoso e complicado, pois o desenvolvedor precisa conhecer toda a estrutura de código implementada, e essa estrutura sofre alterações constantemente devido a implementação de novos requisitos.

Sabendo desses problemas, esse trabalho tem como principal motivação auxiliar o processo de identificação de *design smells* de maneira automática e inteligente para que esse processo se torne mais simples e mais prático para o desenvolvedor.

1.2 Objetivos e Contribuições

Apresentado os problemas e motivações, o objetivo deste trabalho é fornecer uma ferramenta que identifique *design smells* em projetos orientados a objetos e que também apresenta sugestões de como refatorá-los.

Tal ferramenta foi desenvolvida como um *plug-in* do ambiente de desenvolvimento Eclipse[1] onde foram utilizados recursos como o Eclipse Java Development Tools (JDT) para extração de informações do código fonte (como nome das classes, quantidade de métodos, nível de herança, entre outros) e também foi utilizada uma ontologia para representar os principais conceitos da OO e inferir novos conhecimentos através de regras lógicas no padrão estabelecido pela Web Semântica conhecidas como Semantic Web Rule Language (SWRL).

Também comparamos o nível de acurácia da nossa ferramenta com as ferramentas mais comuns encontradas na literatura: Decor[2] (MOHA et al., 2010), JDeodorant[3] (TSANTALIS; CHATZIGEORGIOU, 2009), PMD[4] (FONTANA; BRAIONE; ZANONI, 2012) e Checkstyle[5] (FONTANA; BRAIONE; ZANONI, 2012). Essa comparação foi realizada através da identificação de *smells* dessas ferramentas em projetos *open source* e logo após foram analisados os níveis de *Precision* e *Recall* de cada teste.

É importante ressaltar que nossa ferramenta funciona como um sistema especialista, onde contamos com uma base de conhecimento composta por uma ontologia, onde todo o conhecimento a respeito dos projetos OO e das características necessárias para identificação dos *design smells* estarão armazenadas. Também contamos com o motor de inferência representado aqui pela OWLApi, que é uma biblioteca que nos permite manipular os dados de nossa ontologia e apresentar ao usuário. Dessa forma, garantimos que, de forma inteligente, nossa ferramenta apresente os melhores resultados.

Assim, podemos dizer que nossa abordagem utiliza um modelo determinístico, onde o resultado será determinado pelos fatos contidos na base de conhecimento, diferente das outras ferramentas analisadas, que dependem de variáveis e métricas para realizarem suas execuções.

1.3 Estrutura da Dissertação

O restante deste trabalho está estruturado da seguinte forma: no capítulo 2 serão apresentados os principais conceitos teóricos utilizados no desenvolvimento deste trabalho; o capítulo 3 apresentará a metodologia aplicada para desenvolver este trabalho; o capítulo 4 apresentará como foi realizado o desenvolvimento do *plug-in*; no capítulo 5 serão apresentados os recursos utilizados para validar a abordagem proposta; o capítulo 6 apresentará como foram conduzidos os experimentos e os resultados encontrados; no capítulo 7 apresentaremos os trabalhos correlacionados à nossa pesquisa; e, por fim, no capítulo 8 apresentaremos as conclusões obtidas, principais contribuições e trabalhos futuros.

[1] https://www.eclipse.org/
[2] https://wiki.ptidej.net/
[3] https://github.com/tsantalis/JDeodorant
[4] https://pmd.github.io/
[5] http://checkstyle.sourceforge.net/index.html

Capítulo 2

Fundamentação Teórica

Nesse capítulo serão apresentados os conceitos utilizados para desenvolvimento deste trabalho.

2.1 *Design Smells*

Design smells são determinadas estruturas que indicam violação dos principios fundamentais de *design* e impactam negativamente a qualidade do código (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) tornando o *software* difícil de evoluir e manter podendo desencadear a necessidade de refatoração do código (FONTANA; BRAIONE; ZANONI, 2012).

O termo *smell* foi definido a primeira vez por Martin Fowler (FOWLER, 1999) que descreveu vinte e um *code smells* no livro *"Refactoring: Improving the Design of Existing Code"*. Desde então, muitas pesquisas têm sido realizadas no intuito de melhorar a qualidade de código fonte através da detecção e identificação de *smells* (SOBRINHO; LUCIA; MAIA, 2018).

Os *smells* citados por Fowler podem ser encontrados logo abaixo:

1. ***Duplicated code***: Esse *smell* ocorre quando a mesma estrutura de código existe em mais de um lugar. O exemplo mais comum de *duplicated code* é quando a mesma expressão aparece em dois métodos de uma mesma classe. Assim, tudo o que se tem a fazer é extrair essa expressão para um método e chamá-lo nos locais onde havia o código duplicado.

2. ***Long method***: Esse *smell* ocorre quando um método possui muitas linhas de código. Na maioria dos casos, métodos muito grandes precisam ser comentados para que possam ser entendidos, e se um método precisa de um comentário, então o mesmo deve ser refatorado para métodos menores.

3. ***Large class***: Esse *smell* ocorre quando uma classe possui muitas responsabilidades, e geralmente classes assim possuem muitas variáveis.

4. ***Long parameter list***: Esse *smell* ocorre quando um método possui excessivos parâmetros em sua assinatura.

5. ***Divergent change***: Esse *smell* ocorre quando uma classe é comumente alterada em diferentes maneiras por diferentes motivos. Por exemplo, se for necessário alterar muitos métodos de uma classe toda vez que um novo banco de dados surgir no sistema.

6. ***Shotgun surgery***: Esse *smell* é basicamente o oposto do *divergent change*. Nesse caso, toda vez que surge alguma alteração no sistema, o desenvolvedor precisa fazer pequenas alterações em diversos lugares.

7. ***Feature envy***: Esse *smell* ocorre quando uma determinada classe está mais interessada em métodos de outras classes do que os seus próprios métodos.

8. ***Data clumps***: Esse *smell* ocorre quando um conjunto de dados aparecem separados no código mas deveriam fazer parte de um objeto.

9. ***Primitive obsession***: Esse *smell* ocorre quando uma classe possui muitos atributos primitivos que poderiam ser transformados em uma classe.

10. ***Switch statements***: Esse *smell* ocorre quando há várias ocorrências de *switchs* no código que poderiam ser trocados por polimorfismo.

11. ***Parallel inheritance hierarchies***: Esse *smell* é um caso especial do *shotgun surgery*. Nesse caso, toda vez que é necessário criar uma subclasse de uma classe, também é necessário criar uma subclasse de outra classe.

12. ***Lazy class***: Esse *smell* ocorre quando uma classe não está fazendo o suficiente para permanecer no sistema. Geralmente é uma classe que foi adicionada pensando em alguma mudança que foi planejada mas não foi desenvolvida.

13. ***Speculative generality***: Esse *smell* ocorre quando uma classe é criada pensando em um requisito que pode ser necessário no futuro, mas tal requisito acaba não sendo implementado.

14. ***Temporary field***: Esse *smell* ocorre quando um atributo de uma classe é utilizado somente em certas circunstâncias.

15. ***Message chains***: Esse *smell* ocorre quando um objeto pede uma referência de outro objeto que por sua vez pede referência a outro objeto que também pede referência a outro objeto e assim por diante.

16. ***Middle man***: Esse *smell* ocorre quando uma classe é utilizada somente para delegar tarefas para outras classes.

17. ***Inappropriate intimacy***: Esse *smell* ocorre quando diversas classes dependem muito umas das outras para realizar suas tarefas.

18. ***Alternative classes with different interfaces***: Esse *smell* ocorre em classes que possuem métodos com nomes e implementações iguais mas assinaturas diferentes.

19. ***Incomplete library class***: Esse *smell* ocorre quando o sistema depende de uma biblioteca externa que precisa ser atualizada, pois não está fornecendo mais os requisitos esperados.

20. ***Data class***: Esse *smell* ocorre em classes que possuem somente atributos e métodos acessores e nada mais, sendo bastante provável que esse tipo de classe seja manipulada por várias outras classes no sistema.

21. ***Refused bequest***: Esse *smell* ocorre quando subclasses herdam métodos de suas superclasses mas negam a implementação desse método.

Além de Fowler, Suryanarayana (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) também descreve um total de vinte e cinco *design smells* divididos em quatro categorias: *smells* de hierarquia, *smells* de modularização, *smells* de abstração e *smells* de encapsulamento. Abaixo segue a lista dos vinte e cinco *design smells* apresentados em (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014):

1. ***Missing Abstraction***: Esse *smell* ocorre quando grupos de dados ou *strings* são utilizados em vez de criar uma classe ou *interface*.

2. ***Imperative Abstraction***: Esse *smell* ocorre quando um método é transformado em uma classe. Esse *smell* se manifesta em classes que possuem somente um método, e na maioria das vezes o nome do método e da classe são idênticos.

3. ***Incomplete Abstraction***: Esse *smell* ocorre quando uma abstração não suporta completamente métodos complementares ou inter-relacionados (simétricos). Por exemplo, uma *interface* provê um método *add()* mas não provê um método *delete()*.

4. ***Multifaceted Abstraction***: Esse *smell* ocorre quando uma abstração tem mais de uma responsibilidade atribuída a ela.

5. ***Unnecessary Abstraction***: Esse *smell* ocorre quando uma abstração que não é necessária (e poderia ser evitada) é introduzida no *design*.

6. ***Unutilized Abstraction***: Esse *smell* pode ocorrer de duas maneiras: (i) classes concretas que não são mais utilizadas; (ii) classes abstratas e/ou *interfaces* que não estão sendo implementadas por ninguém.

7. ***Duplicate Abstraction***: Esse *smell* ocorre quando duas ou mais classes têm nomes idênticos e/ou implementações idênticas.

8. ***Deficient Encapsulation***: Esse *smell* ocorre quando a acessibilidade de um ou mais membros da classe é mais permissível que o necessário. Por exemplo, uma classe que possui atributos públicos.

9. ***Leaky Encapsulation***: Esse *smell* ocorre quando uma classe expõe ou vaza detalhes de implementação por meio de uma interface pública.

10. ***Missing Encapsulation***: Esse *smell* ocorre quando variações de implementação não são encapsuladas em uma abstração ou hierarquia.

11. ***Unexploited Encapsulation***: Esse *smell* ocorre quando o código usa verificações de tipo explícitas (usando instruções encadeadas *if-else* ou *switch* que verificam o tipo do objeto) em vez de explorar a variação em tipos já encapsulados em uma hierarquia.

12. ***Broken Modularization***: Esse *smell* ocorre quando dados e/ou métodos que idealmente deveriam ter sido localizados em uma única abstração são separados e espalhados por várias abstrações.

13. ***Insufficient Modularization***: Esse *smell* ocorre quando existe uma abstração que não foi completamente decomposta, e uma decomposição adicional poderia reduzir seu tamanho, complexidade de implementação ou ambos.

14. ***Cyclically-dependent Modularization***: Esse *smell* ocorre quando duas ou mais abstrações dependem uma da outra direta ou indiretamente (criando um acoplamento rígido entre as abstrações).

15. ***Hub-like Modularization***: Esse *smell* ocorre quando uma abstração tem dependências (entrada e saída) com um grande número de outras abstrações.

16. ***Missing Hierarchy***: Esse *smell* ocorre quando um segmento de código usa lógica condicional (geralmente em conjunto com "tipos marcados") para gerenciar explicitamente a variação no comportamento em que uma hierarquia poderia ter sido criada e usada para encapsular essas variações.

17. ***Unnecessary Hierarchy***: Esse *smell* ocorre quando toda a hierarquia de herança é desnecessária, indicando que a herança foi aplicada desnecessariamente para o contexto de design específico.

18. ***Unfactored Hierarchy***: Esse *smell* ocorre quando há duplicação desnecessária entre tipos em uma hierarquia.

19. ***Wide Hierarchy***: Esse *smell* ocorre quando uma hierarquia de herança é muito ampla, indicando que tipos intermediários podem estar ausentes.

20. ***Speculative Hierarchy***: Esse *smell* ocorre quando um ou mais tipos em uma hierarquia são fornecidos especulativamente (ou seja, com base em necessidades imaginadas, em vez de requisitos reais)

21. ***Deep Hierarchy***: Esse *smell* ocorre quando uma hierarquia de herança é excessivamente profunda.

22. ***Rebellious Hierarchy***: Esse *smell* ocorre quando um subtipo rejeita os métodos fornecidos pelos seus supertipos.

23. ***Broken Hierarchy***: Esse *smell* ocorre quando um supertipo e seu subtipo conceitualmente não compartilham uma relação "é-um", resultando em substituibilidade quebrada.

24. ***Multipath Hierarchy***: Esse *smell* ocorre quando um subtipo herda direta e indiretamente de um supertipo, levando a caminhos de herança desnecessários na hierarquia.

25. ***Cyclic Hierarchy***: Esse *smell* ocorre quando um supertipo em uma hierarquia depende de qualquer um de seus subtipos.

Além de catalogar vinte e cinco *design smells*, (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) também cita os principais motivos que levam ao surgimento de *smells* em projetos orientados a objetos.

- ❑ Violação dos princípios de design

  Os princípios de design guiam os desenvolvedores a criar soluções de *software* efetivas e de alta qualidade. Quando esses princípios são violados, o resultado se manifesta como um *smell*.

  Considerando a classe *Calendar* do Java que foi criada para abstrair as funcionalidades de um calendário do mundo real, verifica-se que nela é violado o princípio da abstração, pois ela também suporta funcionalidades relacionadas a tempo. Portanto, é possível encontrar um *smell* na classe *Calendar* por possuir múltiplas responsabilidades.

- ❑ Uso inapropriado de padrões

Muitas vezes desenvolvedores se deparam com problemas que em um primeiro momento parece perfeito para ser resolvido com um *design pattern*, e sem entender tal problema e como a utilização do *pattern* implicará o projeto, o desenvolvedor realiza a refatoração acreditando que é a melhor solução.

O problema é que um *design pattern* mal aplicado cria uma estrutura que sofre de sintomas como muitas classes ou classes altamente acopladas com poucas responsabilidades.

Assim, cabe ao desenvolvedor entender a melhor utilização dos design patterns para ter certeza se será a melhor solução para acabar com determinado *smell*.

- Limitações de linguagens de programação

  Algumas linguagens de programação nos seus primórdios de existência não possuíam muitos recursos, assim desenvolvedores teriam que encontrar soluções alternativas a seus problemas. Por exemplo, as versões iniciais do Java que não suportava *enums*, obrigando desenvolvedores a inserir classes ou interfaces para conter constantes. Isso implicava na violação da abstração, surgindo assim um *smell*.

- Pensamento procedural em orientação a objetos

  Geralmente quando programadores com conhecimento de alguma linguagem procedural começam a desenvolver no paradigma orientado a objetos, a forma de pensar não acompanha tal transição, pois não basta utilizar uma linguagem orientada a objetos. Mais importante que isso é aplicar os conceitos da OO no código.

  Isso faz com que na maioria das vezes, técnicas procedurais sejam aplicadas juntas de técnicas orientadas a objetos, violando alguns princípios e criando *smells*.

- Não aderir a melhores práticas e processos

  Ao longo dos anos, *softwares* das mais diversas utilidades são criados para atender a um número cada vez maior de usuários. Essa demanda implicou a criação de processos e práticas para melhor gerenciar tempo e mão de obra para o desenvolvimento. Porém, algumas empresas não conseguem seguir tais práticas e acabam atingindo os programadores, que para atender a expectativa desejada no tempo estipulado precisam passar por cima dos padrões utilizados no projeto violando, assim, princípios fundamentais para um código de qualidade.

2.2 Sistemas Especialistas

Um sistema especialista é definido como um sistema computacional interativo e confiável que utiliza fatos e heurística para resolver problemas complexos de tomada de decisão (GURU99, 2019).

A ideia básica por trás de um sistema especialista é a transferência de uma vasta quantidade de conhecimento especialista de um humano para um computador. Esse conhecimento é armazenado e então o computador pode fazer inferências e chegar a conclusões específicas. Então, como um especialista humano, o sistema apresenta sugestões e explica, caso necessário, a lógica realizada até chegar à conclusão(LIAO, 2005).

O conhecimento especialista possui uma natureza dinâmica, ou seja, o conhecimento e a experiência estão continuamente sujeitos a mudanças. A percepção dessas propriedades levou a visão de que a separação explícita de algoritmos para aplicação de conhecimento altamente especializado do próprio conhecimento é altamente desejável, se não mandatório, para o desenvolvimento de sistemas especialistas (LUCAS; GAAG, 1991).

Essa compreensão para desenvolvimento de sistemas especialistas na atualidade é formulada pela equação 1, às vezes chamada de paradigma de design de sistemas especialistas (LUCAS; GAAG, 1991):

$$SistemaEspecialista = Conhecimento + Inferência \tag{1}$$

Consequentemente, um sistema especialista possui dois componentes essenciais, apresentados a seguir:

- Uma base de conhecimento que captura o conhecimento específico do domínio; e
- Um motor de inferência que consiste em algoritmos para manipulação de conhecimento representado pela base de conhecimento.

O processo de criação de sistemas especialistas pode ser resumido pelos items abaixo (GURU99, 2019):

- Determinar características do problema;
- Engenheiro de conhecimento e especialista no domínio trabalham em coerência para definir o problema;
- O engenheiro de conhecimento traduz o conhecimento em uma linguagem compreensível por computadores. Ele projeta um mecanismo de inferência, uma estrutura de raciocínio, que possa usar o conhecimento quando necessário.
- O especialista em conhecimento também determina como integrar o uso de conhecimento incerto no processo de raciocínio e que tipo de explicação seria útil.

Há ainda vários pré-requisitos para um formalismo de representação do conhecimento antes que ele possa ser considerado adequado para codificar o conhecimento do domínio. Um formalismo adequado de representação do conhecimento deve (LUCAS; GAAG, 1991):

- ❏ Possuir poder de expressividade suficiente para codificar o conhecimento do domínio específico;
- ❏ Possuir uma base semântica limpa, de modo que o significado do conhecimento presente na base de conhecimento seja fácil de entender, especialmente pelo usuário;
- ❏ Permitir interpretação algorítmica eficiente;
- ❏ Permitir explicações e justificativas das soluções obtidas, mostrando por que certas perguntas foram feitas ao usuário e como certas conclusões foram tiradas.

Parte dessas condições diz respeito à forma (sintaxe) de um formalismo de representação do conhecimento; outros dizem respeito ao seu significado (semântica) (LUCAS; GAAG, 1991).

Um formalismo amplamente utilizado para construção de sistemas baseados em conhecimento, ou sistemas especialistas, são as ontologias, devido à sua capacidade de representar um conhecimento rico e complexo sobre coisas, grupos de coisas e relações entre elas.

2.3 Ontologias e Web Semântica

Uma ontologia é uma descrição formal explícita de conceitos em um domínio de discurso (classes, às vezes chamadas de conceitos), propriedades de cada conceito descrevendo várias características e atributos dos conceitos (*slots*, as vezes chamados de papeis ou propriedades) e restrições nos *slots* (*facets*, às vezes chamados de restrições de papeis). Uma ontologia juntamente com um conjunto de instâncias individuais de classes constituem uma base de conhecimento. Na verdade, existe uma linha tênue onde a ontologia termina e a base de conhecimento começa (NOY; MCGUINNESS, 2001).

Segundo (NOY; MCGUINNESS, 2001) os principais benefícios em se construir uma ontologia são:

- ❏ Compartilhar o entendimento comum da estrutura de informações entre pessoas ou agentes de *software*;
- ❏ Permitir a reutilização de conhecimento de domínio;
- ❏ Tornar as suposições de domínio explícitas;
- ❏ Separar o conhecimento do domínio do conhecimento operacional;
- ❏ Analisar o conhecimento do domínio.

Uma ontologia é tipicamente composta de uma hierarquia de termos que descrevem objetos, relacionamentos entre os objetos e suas caracteristicas. Em geral, ontologias são compostas pelos seguintes conceitos (CHENG; LIAO, 2007)

- **Classe:** Uma classe representa um conceito ou um conjunto de conceitos com caracteristicas próprias no domínio. Uma classe ainda pode ter sub-classes que representam conceitos mais especificos.
- **Propriedade:** Propriedades são utilizadas para dar significado as classes, uma vez que uma classe por si só não quer dizer nada na ontologia. As propriedades podem ser definidas como propriedades descritivas e propriedades de relacionamento.
- **Relacionamento:** Relacionamentos descrevem como as classes interagem entre si no domínio.

Uma ontologia apenas define um vocabulário para compartilhar e padronizar o entendimento de determinadas informações sobre o domínio no qual se deseja trabalhar. Esse entendimento pode ser descrito em forma de tripla como "descrição-representação-interpretação"(ISOTANI; BITTENCOURT, 2015).

Uma das formas de descrever e representar um conhecimento de um modo que possa ser entendido e interpretado tanto por sistemas computacionais como pelos seres humanos é utilizar mecanismos e linguagens de representação/modelagem (visual ou lógica/formal) que explicitam as relações (restrições e hierarquias) entre conceitos(ISOTANI; BITTENCOURT, 2015).

Atualmente a linguagem mais utilizada para representação de ontologias é a Web Ontology Language (OWL), desenvolvida e aprovada pelo World Wide Web Consortium (W3C) para satisfazer ao formalismo exigido pela comunidade de Web Semântica e para que programas possam compreender e responder a consultas de agentes.

OWL é uma linguagem baseada em lógica computacional de modo que o conhecimento expresso em OWL possa ser explorado por programas de computador, por exemplo, para verificar a consistência desse conhecimento ou para tornar explícito o conhecimento implícito. Os documentos OWL conhecidos como ontologias podem ser publicados na *World Wide Web* e podem se referir a ou ser referidos por outras ontologias OWL (W3C, 2012).

(ISOTANI; BITTENCOURT, 2015) destacam que há uma ideia errada sobre o que é a OWL e como aplicá-la. Tal fato ocorre pela complexidade inerente ao termo "ontologia"e pela expressividade da linguagem OWL. Os autores também destacam três características não inerentes a OWL:

1. **Não é uma linguagem de programação:** OWL é uma linguagem declarativa que descreve um determinado universo do discurso de forma lógica. A partir do momento que descreve conhecimento, pode-se fazer uso de ferramentas conhecidas como *reasoners* para inferir novas informações sobre o universo de discurso.

2. **Não é uma linguagem de esquema para conformidade sintática:** Não faz parte do escopo da OWL prescrever como certo documento deve ser sintaticamente estruturado.

3. **Não é um banco de dados:** A principal diferença entre banco de dados e OWL é a semântica utilizada em cada um. Os bancos de dados são mundos fechados, o que quer dizer que, se determinado fato não está presente, ele é considerado falso. Enquanto isso ontologias são considerados mundos abertos, implicando que se determinado fato não está presente ele é considerado desconhecido, porque é possível que seja verdadeiro.

A especificação da W3C define três subconjuntos da linguagem OWL baseado na sua capacidade de representação e propriedades formais: OWL Lite, OWL-DL e OWL Full.

OWL Lite é o subconjunto com menor expressividade. Possui construtos para representação de taxonomias simples e algumas restrições sobre propriedades. O OWL-DL estende ao máximo a expressividade da OWL Lite, mantendo as suas propriedades computacionais de complexidade e decidibilidade. O conjunto mais completo, OWL Full, permite maior nível de representatividade de OWL, permitindo construções de meta modelagem. Porém, em OWL Full não há garantias computacionais para máquinas de inferência (ABEL; FIORINI, 2013).

Como dito anteriormente, a linguagem OWL permite a utilização de ferramentas chamadas *reasoners* para inferir novos conhecimentos a partir do conhecimento modelado. Para isso é necessário um conjunto de regras que descrevem logicamente um conhecimento novo que se espera ser obtido. Os *reasoners* mapeiam a base de conhecimento através dos conceitos, relações e fatos para que sejam inferidos e apresentados informações implícitas. Os principais motores de inferência disponíveis na literatura são: Pellet[6], Hermit[7] e Fact++[8](NONATO et al., 2017).

A *Semantic Web Rule Language* (SWRL) é um exemplo as regras lógicas utilizadas pelos motores de inferências. A SWRL foi criada pelo W3C para ser a linguagem padrão de inferência para ontologias e web semântica. Ela é composta por um antecedente conhecido como "corpo"e um consequente conhecido como "cabeça". Informalmente, isso quer dizer que se o antecendente é verdadeiro, o consequente também é verdadeiro. Como pode ser visto na equação 2, a regra SWRL verifica se a informação (Y é pai de X e Z é irmão de Y) é verdadeira, então a informação (Z é tio de X) também é verdadeira.

$$pai(?y, ?x) \wedge irmão(?z, ?y) \implies tio(?z, ?x) \tag{2}$$

Desta forma, a SWRL provê um formalismo para representação das regras de produção, no qual o conhecimento real sobre a solução de problemas é expresso.

[6] https://www.w3.org/2001/sw/wiki/Pellet
[7] http://www.hermit-reasoner.com/
[8] http://owl.man.ac.uk/factplusplus/

2.4 O projeto Eclipse JDT

O projeto Eclipse JDT *(Java Development Tools)* provê *APIs* para acessar e manipular código fonte Java através de modelos Java e Árvore de Sintaxe Abstrata (AST[9]). No modelo Java, cada projeto Java é representado internamente por meio de um modelo que é uma representação leve e tolerante a falhas do projeto Java. Ele não contém tantas informações quanto a AST, mas é mais rápido para criar (VOGEL; SCHOLZ; PFAFF, 2018).

O modelo Java é representado por um estrutura em árvore que pode ser descrita como a seguir:

- Projeto Java *(IJavaProject)*: o projeto Java em si que contém todos os outros arquivos e objetos;
- Pasta Src/bin ou bibliotecas externas *(IPackageFragmentRoot)*: mantém os códigos fonte ou arquivos binários, pode ser uma pasta ou uma biblioteca;
- Pacote *(IPackageFragment)*: cada pacote está abaixo do *IPackageFragmentRoot*, os subpacotes não são folhas do pacote, eles são listados diretamente sob *IPackageFragmentRoot*;
- Código fonte Java *(ICompilationUnit)*: o arquivo de origem é sempre o nó do pacote;
- Tipos/Atributos/Métodos *(IType/IField/IMethod)*: Tipos, atributos e métodos.

Na figura 1 é posssível visualizar como os componentes do JDT são representados em um projeto Java.

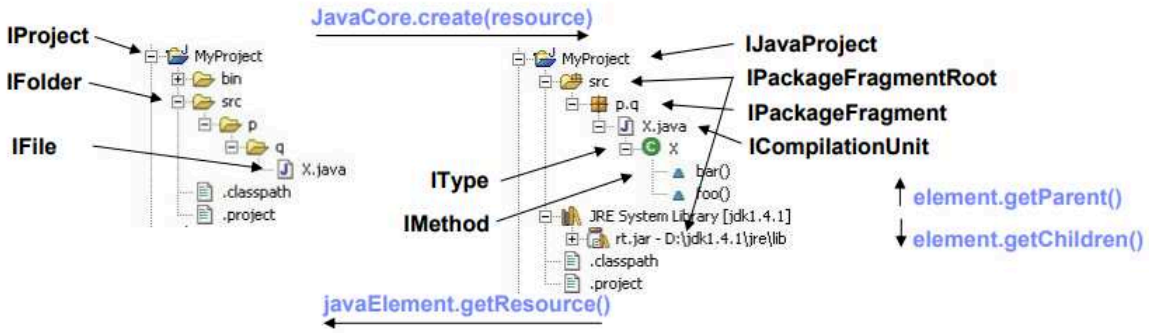

Figura 1 – Projeto Java representado em JDT

Fonte: (VOGEL; SCHOLZ; PFAFF, 2018)

A Árvore de Sintaxe Abstrata é uma representação em árvore detalhada do código-fonte Java. A AST define uma Interface de Programação de Aplicativos (API[10]) para

[9] Do inglês, Abstract Syntax Tree
[10] Do inglês, Application Programming Interface

modificar, criar, ler e excluir o código-fonte. Cada elemento de origem Java é representado como uma subclasse da classe ASTNode. Cada nó específico fornece informações específicas sobre o objeto que o representa (VOGEL; SCHOLZ; PFAFF, 2018).

Capítulo 3

Metodologia

Neste capítulo serão apresentados as perguntas de pesquisa e, logo após, a metodologia utilizada para responder tais perguntas. Serão apresentados os métodos abordados na construção da ferramenta, assim como as etapas de desenvolvimento e validação.

3.1 Perguntas de pesquisa

Como dito no capítulo 1, a proposta desse trabalho é analisar o uso de tecnologias da web semântica no processo de identificação e classificação de *design smells* em códigos orientados a objetos. Sendo mais específico, o uso de ontologias e regras SWRL para criação de uma base de conhecimento e inferência de conteúdo implícito. Assim, com os resultados obtidos através desse trabalho, propõe-se responder as seguintes perguntas ($RQs\#$):

- *RQ #1* É possível e viável detectar e classificar *design smells* automaticamente através de ontologias e tecnologias da web semântica?

  Essa pergunta de pesquisa busca avaliar até que ponto é possível detectar e classificar *design smells* utilizando tecnologias da web semântica e o quão viável é esse processo.

- *RQ #2* A identificação e classificação de *design smells* apoiados pelo uso de ontologias e tecnologias da web semântica apresentam resultados significativamente positivos quando comparados com as técnicas de detecção mais comuns?

  Neste caso será analisado o nível de acurácia entre as ferramentas propostas. Assim conseguiremos medir o nivel de melhoria que nosso método apresentou sobre as outras ferramentas.

3.2 *Criação da ontologia*

Diversas metodologias têm sido criadas para apoio ao processo de criação de ontologias. Algumas das mais utilizadas são: *Toronto Virtual Enterprise* [TOVE], *ENTERPRISE* [Uschold et al.], *METHONTOLOGY* [Fernandez et al.] e *Ontology Development 101* [Noy & McGuinness] (HOSS, 2006).

A metodologia escolhida para realização desse trabalho foi a *Ontology Development 101* (NOY; MCGUINNESS, 2001), pois ela fornece um guia prático de como começar a criação de sua ontologia, incluindo uma série de etapas práticas e dicas de erros comuns a serem evitados.

A figura 2 ilustra a sequência de etapas criadas por (NOY; MCGUINNESS, 2001) para apoio na criação de ontologias. Através desta figura podemos observar que o processo é contínuo, ou seja, cada etapa leva a próxima e todas as etapas sempre se repetirão no decorrer do desenvolvimento da ontologia.

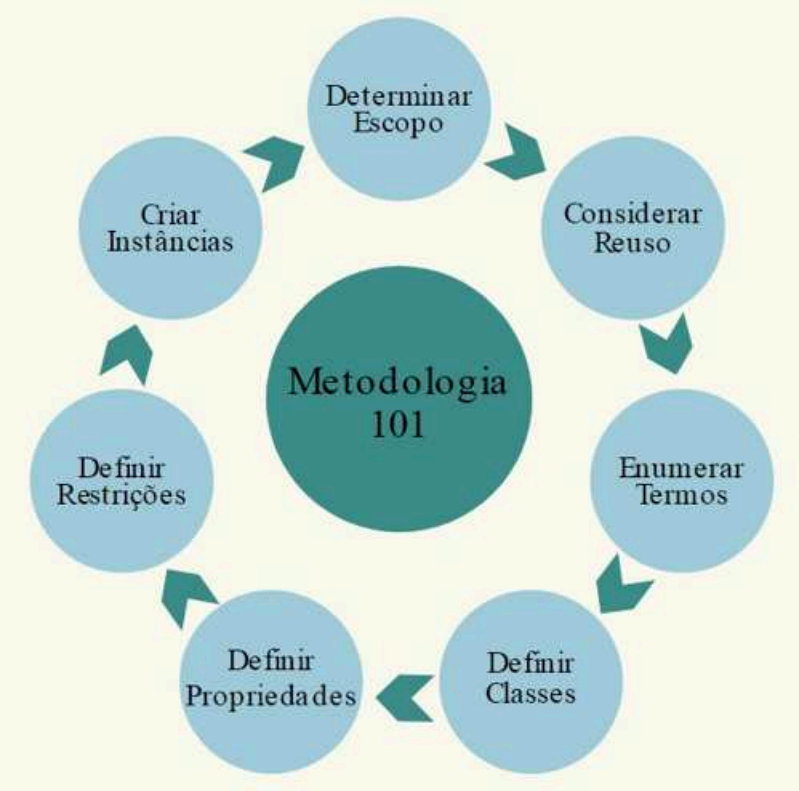

Figura 2 – Ontology 101

Fonte: Adaptado de (NOY; MCGUINNESS, 2001)

Seguindo o modelo acima, começamos o desenvolvimento da nossa ontologia determinando o escopo do nosso trabalho. Como iremos trabalhar com identificação e classificação de *design smells* no paradigma orientado a objetos, foi necessário pesquisar e estudar so-

bre tais conceitos. Dessa forma, fomos capazes de obter conhecimento adequado para a modelagem da nossa ontologia.

Após a aquisição de tal conhecimento, era importante considerarmos possíveis ontologias já existentes. Contudo, não fomos capazes de encontrar uma ontologia que modelasse o conhecimento da forma que precisávamos, então decidimos criar nossa ontologia do zero.

Precisávamos, então, definir quais os conceitos mais importantes do nosso domínio. Definimos que os conceitos mais gerais seriam: classe, atributo, método, parâmetro e interface. E então, a partir desses conceitos surgiriam conceitos mais específicos.

Dessa forma fomos capazes de criar nossa hierarquia de classes na ontologia. Conceitos como classe e atributos seriam ramificados para representação de conceitos mais específicos. Assim, definimos as classes *ClassOO* (para representação das classes OO), *AbstractClassOO* (para representação das classes OO abstratas), *NormalAttribute* (para representação de atributos gerais), *ConstantAttribute* (para representação de atributos constantes), *Method* (para representação de métodos), *Paremeter* (para representação de parâmetros contidos na assinatura de métodos) e *Interface* (para representação de interfaces).

Definidas as classes, precisaríamos definir as propriedades responsáveis por gerar significado as classes e as relações entre elas. A figura 3 apresenta as propriedades que darão significado as classes definidas em nossa ontologia. Como é possível notar, todas elas começam com um prefixo para definição de qual conceito tal propriedade fará parte. Os prefixos foram definidos como *at* (para classe *NormalAttribute* e *ConstantAttribute*), *cl* (para classe *ClassOO* e *AbstractClassOO*), *enum* (para classe *Enum*), *in* (para classe *Interface*), *mt* (para classe *Method*) e *pm* (para classe *Parameter*).

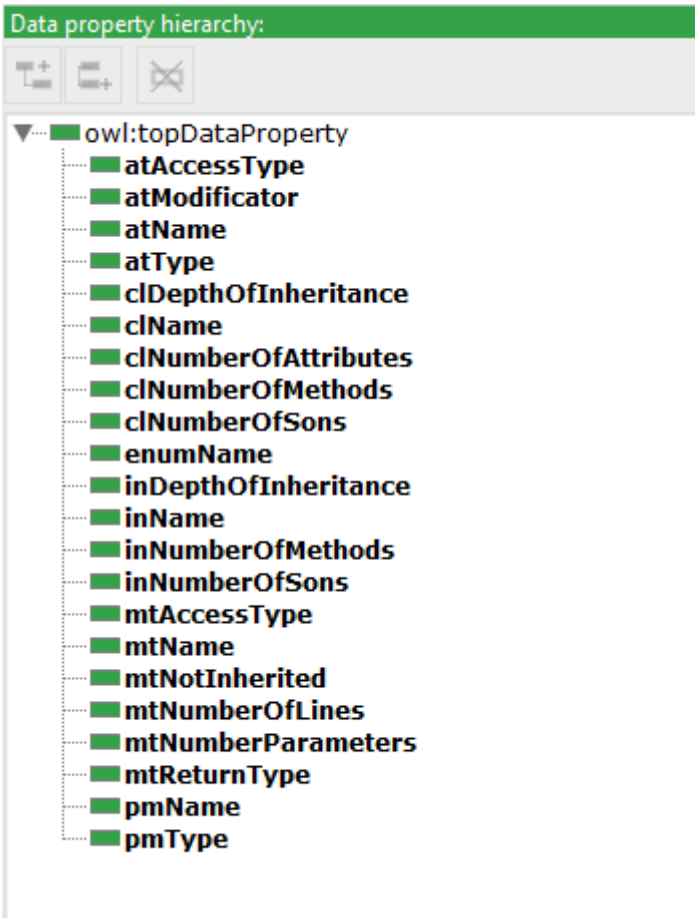

Figura 3 – Data Properties

Fonte: Dados do trabalho

Já a figura 4 apresenta as propriedades utilizadas para representação das relações entre as classes na ontologia. A propriedade *hasAttribute* define uma relação de uma classe que contém um atributo. A propriedade *hasMethod* define uma relação de uma classe que contém um método. A propriedade *hasParameter* define uma relação de um método que contém um parâmetro. A propriedade *hasSub* define uma relação de uma classe que possui outra classe filha. A propriedade *hasSuper* define uma relação de uma classe que possui outra classe pai. A propriedade *implementss* define uma relação de uma classe que implementa uma interface. E a propriedade *isTypeOf* define uma relação de um atributo que é do tipo de alguma classe.

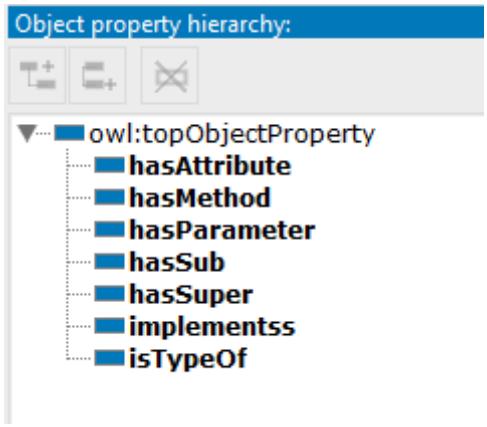

Figura 4 – Object Properties

Fonte: Dados do trabalho

Com nossas classes e propriedades criadas, o último passo foi a instanciação dos indivíduos na nossa ontologia através dos dados extraídos com o JDT. Uma vez que os indivíduos foram criados, bastava o *reasoner* realizar a inferência através das regras SWRL modeladas.

Como estamos trabalhando com identificação e classificação de *design smells* em projetos orientados a objetos, é importante também modelarmos as características dos *smells* que espera-se serem identificados. Nossa abordagem é capaz de identificar um total de 14 *design smells*, nos quais tiveram suas classes criadas na ontologia e suas características modeladas através de regras SWRL. Utilizamos essa abordagem pois os *smells* não são conceitos concretos em um projeto OO como uma classe por exemplo, mas sim um conjunto de características de uma ou mais classes que levam ao seu surgimento no projeto.

A figura 5 apresenta o grafo completo da ontologia modelada. Nela é possível observar como são realizados os relacionamentos das principais classes. Na próxima seção, serão apresentadas as regras SWRL criadas para a realização de inferência do conhecimento no qual se espera a ser obtido.

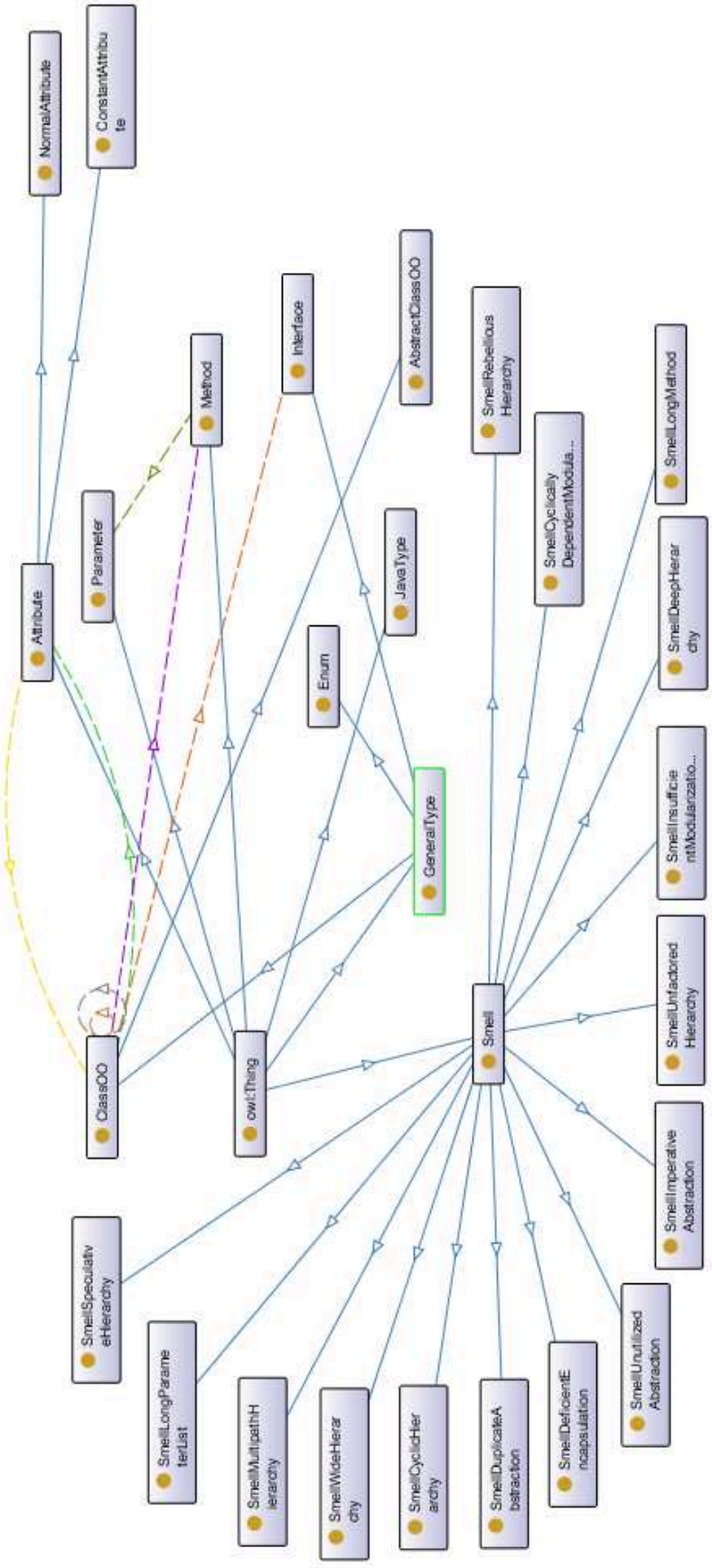

Figura 5 – Grafo da ontologia

Fonte: Dados do trabalho

3.3 Inferência de conhecimento

Para criação da ontologia foi utilizada a linguagem OWL *(Web Ontology Language)* para modelagem dos dados. Contudo, a OWL não consegue inferir conhecimento por si só e para auxilar nessa tarefa, o W3C criou a *Semantic Web Rule Language* SWRL que é uma linguagem baseada em regras de primeira ordem no formato "se-então"que permite aos seus usuários realizarem inferências de novos conhecimentos através dos dados modelados pela OWL.

Nesse trabalho foram criadas 14 regras para inferência dos *design smells* identificados nesse trabalho, sendo 2 citados por (FOWLER, 1999) e 12 por (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

- **Regra 1: Smell Cyclically Dependent Modularization**

  $ClassOO(?a) \wedge ClassOO(?b) \wedge NormalAttribute(?a1) \wedge NormalAttribute(?b1) \wedge hasAttribute(?a, ?b1) \wedge hasAttribute(?b, ?a1) \wedge isTypeOf(?a1, ?a) \wedge isTypeOf(?b1, ?b) \rightarrow SmellCyclicallyDependentModularization(?a) \wedge SmellCyclicallyDependentModularization(?b)$

  Na regra 1 é verificado se uma classe A possui um atributo com o tipo de uma classe B e vice-versa através das propriedades *hasAttribute(?a, ?b1)* e *hasAttribute(?b, ?a1)*. Caso essa afirmação seja verdadeira, ambas as classes A e B são consideradas ocorrências do *design smell Cyclically Dependent Modularization*

- **Regra 2: Smell Cyclic Hierarchy**

  $ClassOO(?a) \wedge ClassOO(?b) \wedge hasSuper(?a, ?b) \wedge NormalAttribute(?a1) \wedge NormalAttribute(?b1) \wedge isTypeOf(?a1, ?a) \wedge isTypeOf(?b1, ?b) \wedge hasAttribute(?b, ?a1) \rightarrow SmellCyclicHierarchy(?b)$

  Na regra 2 é verificado se uma Classe B que é pai de uma classe A possui um atributo com o tipo da própria classe filha. Tal verificação é realizada através das propriedades *hasSuper(?a, ?b)* e *hasAttribute(?b, ?a1)*. Caso essa afirmação seja verdadeira, a classe B é considerada uma ocorrência do *design smell Cyclic Hierarchy*.

- **Regra 3: Smell Deep Hierarchy**

  $ClassOO(?c) \wedge clDepthOfInheritance(?c, ?a) \wedge greaterThan(?a, "6" \wedge \wedge xsd : int) \rightarrow SmellDeepHierarchy(?c)$

  Na regra 3 é verificado através das propriedades *clDepthOfInheritance(?c, ?a)* e $greaterThan(?a, "6" \wedge \wedge xsd : int)$ se uma classe possui hierarquia de herança com

nível maior ou igual a 6. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Deep Hierarchy*.

❑ **Regra 4: Smell Deficient Encapsulation**

$ClassOO(?c) \wedge NormalAttribute(?a) \wedge hasAttribute(?c, ?a) \wedge atAccessType(?a, "public" \wedge \wedge xsd : string) \rightarrow SmellDeficientEncapsulation(?c)$

Na regra 4 é verificado através da propriedade $atAccessType(?a, "public" \wedge \wedge xsd : string)$ se uma classe possui algum atributo com o encapsulamento do tipo público. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Deficient Encapsulation* .

❑ **Regra 5: Smell Duplicate Abstraction**

$ClassOO(?a) \wedge ClassOO(?b) \wedge clName(?a, ?a1) \wedge clName(?b, ?b1) \wedge equal(?a1, ?b1) \wedge DifferentFrom(?a, ?b) \rightarrow SmellDuplicateAbstraction(?b) \wedge SmellDuplicateAbstraction(?a)$

Na regra 5 é verificado se diferentes classes (propriedade *DifferentFrom (?a, ?b)*) possuem nomes iguais (propriedade *equal(?a1, ?b1)*) em um mesmo projeto. Caso a afirmação seja verdadeira, essas classes são consideradadas ocorrências do *design smell Duplicate Abstraction*.

❑ **Regra 6: Smell Imperative Abstraction**

$ClassOO(?c) \wedge equal(?a, "1" \wedge \wedge xsd : int) \wedge clNumberOfMethods(?c, ?a) \rightarrow SmellImperativeAbstraction(?c)$

Na regra 6 é verificado através da propriedade $equal(?a, "1" \wedge \wedge xsd : int)$ se uma classe possui somente um método. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Imperative Abstraction*.

❑ **Regra 7: Smell Insufficient Modularization**

$ClassOO(?c) \wedge clNumberOfAttributes(?c, ?a) \wedge greaterThan(?a, "50" \wedge \wedge xsd : int) \rightarrow SmellInsufficientModularization(?c)$

Na regra 7 é verificado através das propriedades *clNumberOfAttributes(?c, ?a)* e $greaterThan(?a, "50" \wedge \wedge xsd : int)$ se uma classe possui 50 ou mais atributos. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Insufficient Modularization*.

❑ **Regra 8: Smell Multipath Hierarchy**

$ClassOO(?c) \wedge ClassOO(?b) \wedge Interface(?i) \wedge implementss(?b, ?i) \wedge implementss(?c, ?i) \wedge hasSuper(?c, ?b) \rightarrow SmellMultipathHierarchy(?c)$

Na regra 8 é verificado através da propriedade se duas classes implementam a mesma *interface* (propriedade *implementss*) e se uma é pai da outra (propriedade *hasSuper(?c, ?b)*). Caso a afirmação seja verdadeira, as classes são consideradas ocorrências do *design smell Multipath Hierarchy*.

❑ **Regra 9: Smell Speculative Hierarchy**

$ClassOO(?c) \wedge clNumberOfSons(?c, ?a) \wedge equal(?a, "1" \wedge \wedge xsd : int) \rightarrow SmellSpeculativeHierarchy(?c)$

Na regra 9 é verificado se uma classe possui somente um filho através da propriedade *clNumberOfSons(?c, ?a)*. Caso a afirmação seja verdadeira, a classe é considerada uma ocorrência do *design smell Speculative Hierarchy*.

❑ **Regra 10: Smell Unfactored Hierarchy**

$ClassOO(?a) \wedge ClassOO(?b) \wedge ClassOO(?c) \wedge Method(?m) \wedge Method(?n) \wedge Parameter(?p) \wedge Parameter(?q) \wedge hasSuper(?a, ?c) \wedge hasSuper(?b, ?c) \wedge hasMethod(?a, ?m) \wedge hasMethod(?b, ?n) \wedge mtName(?m, ?m1) \wedge mtName(?n, ?n1) \wedge pmName(?p, ?p1) \wedge pmName(?q, ?q1) \wedge hasParameter(?m, ?p) \wedge hasParameter(?n, ?q) \wedge pmType(?p, ?p2) \wedge pmType(?q, ?q2) \wedge equal(?p1, ?q1) \wedge equal(?m1, ?n1) \wedge equal(?p2, ?q2) \wedge DifferentFrom(?a, ?b) \wedge DifferentFrom(?p, ?q) \wedge DifferentFrom(?m, ?n) \rightarrow SmellUnfactoredHierarchy(?b) \wedge SmellUnfactoredHierarchy(?a)$

Na regra 10 é verificado se duas classes que possuem o mesmo pai (proprieade *hasSuper*) possuem métodos com nomes e paramêtros iguais (propriedades *hasMethod* e equal). Caso a afirmação seja verdadeira, as classes são consideradas ocorrências do *design smell Unfactored Hierarchy*

❑ **Regra 11: Smell Unutilized Abstraction**

$AbstractClassOO(?c) \wedge clNumberOfSons(?c, ?a) \wedge equal(?a, "0" \wedge \wedge xsd : int) \rightarrow SmellUnutilizedAbstraction(?c)$

Na regra 11 é verificado se uma classe abstrata não possui nenhum filho (propriedade *clNumberOfSons(?c, ?a)*), ou seja, não está sendo implementada por ninguém. Caso

a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Unutilized Abstraction*.

❑ **Regra 12: Smell Wide Hierarchy**

$ClassOO(?c) \wedge clNumberOfSons(?c, ?a) \wedge greaterThan(?a, "9" \wedge \wedge xsd : int) \rightarrow SmellWideHierarchy(?c)$

Na regra 12 é verificado através da propriedade *clNumberOfSons(?c, ?a)* e $greaterThan(?a, "9" \wedge \wedge xsd : int)$ se uma classe possui 9 ou mais filhos. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Wide Hierarchy*.

❑ **Regra 13: Smell Long Method**

$ClassOO(?c) \wedge Method(?m) \wedge hasMethod(?c, ?m) \wedge greaterThan(?a, "150" \wedge \wedge xsd : int) \wedge mtNumberOfLines(?m, ?a) \rightarrow SmellLongMethod(?c)$

Na regra 13 é verificado através das propriedades *mtNumberOfLines(?m, ?a)* e $greaterThan(?a, "150" \wedge \wedge xsd : int)$ se um método possui mais de 150 linhas de código. Caso a afirmação seja verdadeira, essa classe é considerada uma ocorrência do *design smell Long Method*.

❑ **Regra 14: Smell Long Parameter List**

$ClassOO(?c) \wedge Method(?m) \wedge hasMethod(?c, ?m) \wedge greaterThan(?a, "7" \wedge \wedge xsd : int) \wedge mtNumberParameters(?m, ?a) \rightarrow SmellLongParameterList(?c)$

Na regra 14 é verificado através das propriedades *mtNumberParameters(?m, ?a)* e $greaterThan(?a, "7" \wedge \wedge xsd : int)$ se um método possui mais de 7 parâmetros em sua assinatura. Caso a afirmação seja verdadeira, a classe que contém tal método é considerada uma ocorrência do *design smell Long Parameter List*.

Capítulo 4

O *plug-in* OWLSmell

A escolha por desenvolver um *plug-in* para o Eclipse[11] é devido ao fato de que tal ambiente é bastante utilizado por desenvolvedores no mundo todo. E também, procuramos desenvolver um artefato que seria de fácil utilização uma vez que o *plug-in* já estará integrado na IDE.

O código fonte do *plug-in* OWLSmell pode ser encontrado no repositório do *github* <https://github.com/viniciusjns/owlsmellplugin>.

Para alcançar este objetivo, nós usamos uma série de ferramentas providas pelo Eclipse que permite extrair e manipular código fonte na linguagem de programação Java. Essas ferramentas são conhecidas como *Java Development Tools* (JDT).

A figura 6 apresenta um exemplo de uso do Eclipse JDT para extrair informações de um projeto Java. Na linha 60 é obtido o nome de todos os pacotes do projeto, uma vez que a função *println* está dentro de uma estrutura de repetição. Na linha 85 é obtido o nome das classes do projeto, e na linha 87 é informado quantas linhas de código contém cada classe. No método *printIMethodDetails* encontrado na linha 91, encontram-se três chamadas a função *println*: a primeira para apresentar o nome do método (linha 95), a segunda para apresentar os tipos de parâmetros encontrados na assinatura do método (linha 96) e o terceiro para informar o tipo de retorno do método (linha 97).

[11] https://www.eclipse.org/

SampleHandler.java

```java
package sampleplugin.jdt;

import org.eclipse.core.commands.ExecutionEvent;

public class SampleHandler  {

    public Object execute(ExecutionEvent event) throws ExecutionException {
        // Get the root of the workspace
        IWorkspace workspace = ResourcesPlugin.getWorkspace();
        IWorkspaceRoot root = workspace.getRoot();
        // Get all projects in the workspace
        IProject[] projects = root.getProjects();
        // Loop over all projects
        for (IProject project : projects) {
            try {
                printProjectInfo(project);
            } catch (CoreException e) {
                e.printStackTrace();
            }
        }
        return null;
    }

    private void printProjectInfo(IProject project) throws CoreException,
            JavaModelException {
        System.out.println("Working in project " + project.getName());
        // check if we have a Java project
        if (project.isNatureEnabled("org.eclipse.jdt.core.javanature")) {
            IJavaProject javaProject = JavaCore.create(project);
            printPackageInfos(javaProject);
        }
    }

    private void printPackageInfos(IJavaProject javaProject)
            throws JavaModelException {
        IPackageFragment[] packages = javaProject.getPackageFragments();
        for (IPackageFragment mypackage : packages) {
            // Package fragments include all packages in the
            // classpath
            // We will only look at the package from the source
            // folder
            // K_BINARY would include also included JARS, e.g.
            // rt.jar
            if (mypackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
                System.out.println("Package " + mypackage.getElementName());
                printICompilationUnitInfo(mypackage);

            }

        }
    }

    private void printICompilationUnitInfo(IPackageFragment mypackage)
            throws JavaModelException {
        for (ICompilationUnit unit : mypackage.getCompilationUnits()) {
            printCompilationUnitDetails(unit);

        }
    }

    private void printIMethods(ICompilationUnit unit) throws JavaModelException {
        IType[] allTypes = unit.getAllTypes();
        for (IType type : allTypes) {
            printIMethodDetails(type);
        }
    }

    private void printCompilationUnitDetails(ICompilationUnit unit)
            throws JavaModelException {
        System.out.println("Source file " + unit.getElementName());
        Document doc = new Document(unit.getSource());
        System.out.println("Has number of lines: " + doc.getNumberOfLines());
        printIMethods(unit);
    }

    private void printIMethodDetails(IType type) throws JavaModelException {
        IMethod[] methods = type.getMethods();
        for (IMethod method : methods) {

            System.out.println("Method name " + method.getElementName());
            System.out.println("Signature " + method.getSignature());
            System.out.println("Return Type " + method.getReturnType());

        }
    }
}
```

Figura 6 – Exemplo de código usando JDT

Fonte: Adaptado de (VOGEL; SCHOLZ; PFAFF, 2018)

Utilizando esses conceitos, fomos capazes de adquirir informações de um projeto Java como classes, métodos, variáveis, herança, encapsulamento, entre outros. Após as informações serem coletadas, elas foram salvas em uma base de conhecimento.

Para representar a base de conhecimento, foi utilizada uma ontologia. Essa escolha é justificada por dois motivos(NOY; MCGUINNESS, 2001):

1. A ontologia permite representação de conceitos e propriedades a fim de ser facilmente reutilizada e, se necessário, ser extendida em diferentes contextos e/ou aplicações;

2. Ontologias permitem o raciocínio de informações que estão sendo representadas;

Para instanciar os indivíduos na ontologia com os dados obtidos do Eclipse JDT, utilizamos a OWL API[12] versão 4.2.8 que nos fornece operações para criação e manipulação de ontologias utilizando a linguagem OWL. A figura 7 apresenta um exemplo de código Java utilizando a OWL API.

```
72      private OWLOntologyManager manager;
73      private File file;
74      private OWLOntology ontology;
75      private OWLDataFactory factory;
76
77      private void loadOntology() {
78          try {
79              manager = OWLManager.createOWLOntologyManager();
80              file = new File("C:\\ontology.owl");
81              ontology = manager.loadOntologyFromOntologyDocument(file);
82              factory = ontology.getOWLOntologyManager().getOWLDataFactory();
83              System.out.println("Ontology loaded!");
84          } catch (OWLOntologyCreationException e) {
85              e.printStackTrace();
86          }
87      }
88
```

Figura 7 – Carregando uma ontologia utilizando a OWL API

Fonte: Dados do trabalho

Na figura 7 é apresentado um método para auxiliar no carregamento da ontologia. Carregar a ontologia é essencial para realizar a manipulação da mesma, seja para criação de indivíduos, associação de propriedades, tornar indivíduos diferentes e etc.

```
128     public void createIndividualByClass(String individualName, OWLSmellClass owlSmellClass) {
129         try {
130             OWLClass owlClass = factory.getOWLClass(IRI.create(URL + owlSmellClass.name()));
131             OWLIndividual individual = factory.getOWLNamedIndividual(IRI.create(URL + individualName));
132             OWLClassAssertionAxiom classAssertionAxiom = factory.getOWLClassAssertionAxiom(owlClass, individual);
133             AddAxiom addAxiom = new AddAxiom(ontology, classAssertionAxiom);
134             manager.applyChange(addAxiom);
135             manager.saveOntology(ontology);
136
137         } catch (OWLOntologyStorageException e) {
138             e.printStackTrace();
139         }
140     }
```

Figura 8 – Criando indivíduos utilizando a OWL API

Fonte: Dados do trabalho

Na figura 8 é apresentado um método para criação de indivíduos utilizando a OWL API. O código é bem simples e intuitivo, o único detalhe a ser notado é o *enum* OWL-

[12] http://owlapi.sourceforge.net/

ClassSmell passado como segundo paramêtro na assinatura do método. Esse *enum* é um dado interno da nossa aplicação criado para representar as classes criadas na ontologia.

```
143    public void makeAllIndividualsDifferent() {
144        try {
145            OWLDifferentIndividualsAxiom diff = factory.getOWLDifferentIndividualsAxiom(
146                    ontology.getIndividualsInSignature(Imports.EXCLUDED));
147            AddAxiom addAxiom = new AddAxiom(ontology, diff);
148            manager.applyChange(addAxiom);
149            manager.saveOntology(ontology);
150        } catch (OWLOntologyStorageException e) {
151            e.printStackTrace();
152        }
153    }
```

Figura 9 – Tornando indivíduos diferentes utilizando a OWL API

Fonte: Dados do trabalho

A figura 9 apresenta o método criado para tornar todos os indivíduos diferentes um dos outros. Como citado anteriormente, a OWL faz parte de um paradigma de mundo aberto, o que quer dizer que a criação de indivíduos com nomes diferentes não significa que ambos são distintos. Isso deve ser configurado na ontologia a partir de uma propriedade específica para tal ação.

E, por fim, a figura 10 apresenta o resultado final do *plug-in OwlSmell* aplicado a um projeto OO. Como demonstrado na figura, a ferramenta apresenta no console do próprio Eclipse quais as classes apresentaram *smells* e como refatorar os *smells* apresentados.

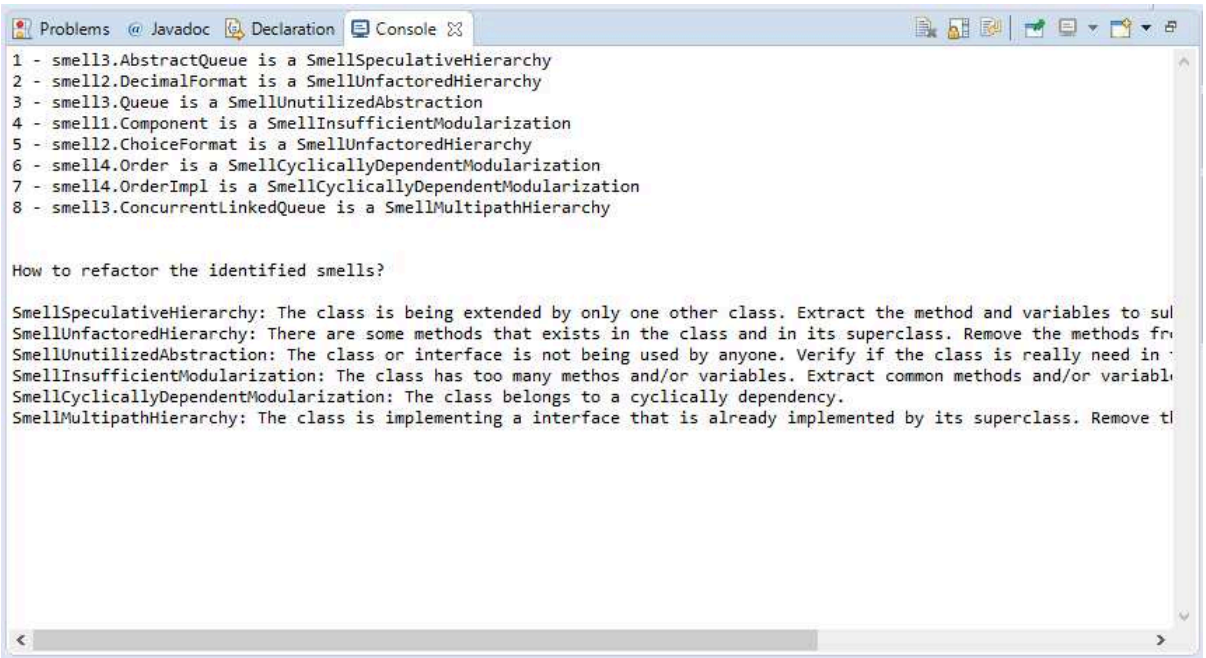

Figura 10 – Resultado da execução do OWLSmell em um projeto OO

Fonte: Dados do trabalho

Capítulo 5

Validação do método proposto

Nesse capítulo serão abordadas as validações que determinaram as respostas para as perguntas de pesquisa apresentadas no capítulo 3.

Primeiramente, comparamos nosso *plug-in* com outras ferramentas para medir a acurácia de cada uma delas ao identificar um mesmo conjunto de *smells*. Nessa validação, conseguimos demonstrar apenas uma pequena parte do poder de expressividade da nossa abordagem, pois as outras ferramentas não identificam a maioria dos *smells* que a nossa identifica.

Assim, realizamos um experimento demonstrativo, quando criamos um projeto Java contendo instâncias de todos os *design smells* propostos nesse trabalho. Com isso, conseguimos demonstrar as vantagens da utilização de ontologias para identificação de *smells*.

Por fim, com os dados obtidos nessas análises, utilizamos as métricas *Precision and Recall* para determinar a acurácia de cada experimento.

5.1 Comparação de ferramentas

Na primeira validação, nosso método foi comparado com algumas ferramentas de identificação e classificação de *smells* já existentes. A avaliação será realizada através da execução de ferramentas bastante conhecidas e conceituadas pela comunidade de engenharia de *software* em projetos *open source*.

As ferramentas escolhidas foram DECOR[13], JDeodorant[14], PMD[15] e CheckStyle[16]. A escolha das ferramentas foi baseada no estudo de (SOBRINHO; LUCIA; MAIA, 2018) onde os autores apontam as ferramentas de identificação e classificação de *smells* mais estudadas no período de 1990 até 2017.

[13] http://www.ptidej.net/tools/designsmells
[14] https://github.com/tsantalis/JDeodorant
[15] https://pmd.github.io/
[16] http://checkstyle.sourceforge.net/

Os projetos escolhidos para serem validados por todas as ferramentas são: JUnit v4.12[17], Log4J v1.2.1[18] e ArgoUML v0.19.8[19]. Esses projetos foram escolhidos por possuírem propósitos distintos (um *framework* para testes unitários, um *framework* para *debug* de código fonte e um sistema para modelagem de diagramas UML, respectivamente) além de já terem sido estudados nos trabalhos (PAIVA et al., 2017) e (MOHA et al., 2010).

Os *smells* inclusos na validação do método foram escolhidos com base em dois critérios: **1)** os *smells* mais estudados segundo o estudo de (SOBRINHO; LUCIA; MAIA, 2018); **2)** os *smells* que faziam parte dos métodos de detecção de duas ou mais ferramentas. Dentre os *smells* citados no capítulo 2, quatro foram selecionados seguindo os critérios acima, sendo dois abordados por (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) e dois abordados por (FOWLER, 1999).

- *Insufficient Modularization*: Esse *smell* ocorre em classes que possuem muitas responsabilidades e/ou muitos métodos e atributos.
- *Long Method*: Esse *smell* ocorre em classes que contém métodos com muitas linhas de código.
- *Long Parameter List*: Esse *smell* ocorre em classes que contém métodos que possuem muitos paramêtros em sua assinatura.
- *Deficient Encapsulation*: Esse *smell* ocorre em classes que os atributos da mesma tem sua visibilidade mais exposta que o necessário.

5.2 Análise por amostragem

Como dito anteriormente, criamos um projeto Java para realizar essa validação. O projeto pode ser encontrado no repositório do *github* <https://github.com/viniciusjns/ProjetoValidacaoMestrado>.

Com esse projeto, fomos capazes de demonstrar todos os *design smells* identificados pelo OWLSmell e também demonstrar que a utilização da ontologia para identificação de *smells* apresentou um avanço no estado da arte devido sua facilidade de configuração, utilização e capacidade de compartilhamento de conhecimento.

5.3 Métricas *Precision and Recall*

Para avaliar a acurácia dos dados analisados na seção anterior serão utilizadas métricas conhecidas como *Precision and Recall*, que são comumente utilizadas para avaliar com mais precisão os resultados, a fim de separar resultados positivos dos negativos.

[17] https://junit.org/junit4/
[18] https://logging.apache.org/log4j/1.2/source-repository.html
[19] http://argouml.tigris.org

Precision determina a fração de registros que realmente se mostra ser positivo no grupo que o classificador declarou como positivo. Quanto maior o *precision*, menor será o número de erros falsos positivos cometidos pelo classificador (TAN; STEINBACH; KUMAR, 2005).

Recall mede a fração de exemplos positivos corretamente previstos pelo classificador. Classificadores com grande *recall* têm muito poucos exemplos positivos mal classificados como negativos, sendo o valor de recall equivalente a verdadeira taxa positiva (TAN; STEINBACH; KUMAR, 2005).

Pode-se definir *precision and recall* com as funções abaixo:

$$Precision, p = \frac{TP}{TP + FP}$$

$$Recall, r = \frac{TP}{TP + FN}$$

A função acima tem suas variáveis estabelecidas a partir de uma matriz de confusão, que é mais comumente conhecida como uma tabela em que se organiza os dados levando em consideração a taxa de erro e acerto de cada classificação. Ela é dividida em quatro seções utilizadas para representar os valores verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos. Na tabela 1 é apresentado como é dividida a matriz de confusão e, logo abaixo, explicado cada valor.

Tabela 1 – Matriz de Confusão

		Predicted Class	
		+	-
Actual Class	+	*True Positive*	*True Negative*
	-	*False Positive*	*False Negative*

Fonte: (TAN; STEINBACH; KUMAR, 2005)

- Verdadeiro Positivo *(True Positive)*: corresponde ao número de exemplos positivos corretamente classificados pelo modelo de classificação;
- Verdadeiro Negativo *(True Negative)*: corresponde ao número de exemplos negativos corretamente classificados pelo modelo de classificação;
- Falso Positivo *(False Positive)*: corresponde ao número de exemplos negativos erroneamente classificados como positivos pelo modelo de classificação;
- Falso Negativo *(False Negative)*: corresponde ao número de exemplos positivos erroneamente classificados como negativos pelo modelo de classificação.

A contagem em uma matriz de confusão também pode ser expressada em termos de porcentagem. A taxa de verdadeiros positivos é definida como a fração de exemplos positivos classificados corretamente pelo modelo (TAN; STEINBACH; KUMAR, 2005).

$$TPR = \frac{TP}{TP + FN}$$

A taxa de verdadeiros negativos pode ser definida como a fração de exemplos negativos classificados corretamente pelo modelo.

$$TNR = \frac{TN}{TN + FP}$$

A taxa de falsos positivos é definida como exemplos negativos classificados como positivos pelo modelo.

$$FPR = \frac{FP}{TN + FP}$$

E, finalmente, a taxa de falsos negativos é a fração de exemplos negativos classificados como negativos pelo modelo.

$$FNR = \frac{FN}{TP + FN}$$

Capítulo 6

Experimentos e Análise dos Resultados

Neste capítulo serão apresentados os experimentos e resultados obtidos com as validações apresentadas no capítulo anterior.

6.1 Comparação de ferramentas

Como dito anteriormente, o primeiro experimento foi realizado com base em comparação de ferramentas já existentes e bastante conceituadas na comunidade de engenharia de *software*. Todas as ferramentas foram executadas em três sistemas *open source* com a finalidade de detectar quatro *smells* distintos. Após isso, foram aplicadas as métricas *precision and recall* para mensurar a acurácia da detecção de cada ferramenta.

É importante ressaltar que nossa ferramenta utiliza um método de detecção distinto das outras ferramentas. Nossa abordagem é baseada em um modelo determinístico, quando extraímos somente os dados necessários para a identificação dos *smells* e utilizamos regras de primeira ordem que estão explicitamente configuradas na ontologia para identificação dos *smells*.

As outras ferramentas por sua vez, utilizam modelos não determinísticos e não necessitam de regras ou arquivos explicitamente configurados, possuindo cada uma sua própria técnica de detecção, o que resulta em números distintos de ocorrências de *smells* identificadas por cada uma. Assim, todas as ocorrências foram avaliadas segundo a literatura de (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014) e (FOWLER, 1999) de forma que fosse possível haver paramêtros de comparação iguais para todas as detecções.

É importante dizer também que os sistemas analisados são *open source* e os autores deste trabalho não possuem conhecimento total do código fonte nem das regras de negócio de cada um. Assim, para avaliar quais classes realmente continham o *smell* sendo estudado, todas as ferramentas foram executadas nos sistemas e as ocorrências encontradas contabilizadas. Logo após, as ocorrências foram manualmente validadas segundo os paramêtros de comparação para descobrir a quantidade real de *smells* por sistema. Pois, só assim, teríamos um valor de referência para realizar os cálculos de *Precision and Recall*.

A ferramenta DECOR foi a mais complexa para uso e configuração. O código fonte é disponibilizado pela desenvolvedora para fins acadêmicos, no entanto, a mesma não proveu a documentação necessária para configuração e utilização. Por fim, depois de muitas pesquisas e estudos em cima do código fornecido, conseguimos executar um arquivo com extensão .java que analisava um projeto (através de uma variável que determinava o caminho do projeto) e identificava os *smells* previamente configurados.

A ferramenta JDeodorant apesar de ser de fácil utilização, não fornece opções para configuração. Em alguns experimentos, ela apresentou resultados abaixo do esperado, fato que pode ser justificado pela falta de configuração dos parâmetros utilizados.

As ferramentas PMD e Checkstyle foram as mais fáceis de configurar e utilizar. Utilizamos as ferramentas em forma de *plug-in* e o mesmo nos forneceu opções para realização da configuração da quantidade de métodos e atributos utilizados na validação dos *smells* a seguir.

Assim, verificamos que a ferramenta OWLSmell demonstrou o melhor grau de configuração e utilização. Como ela foi desenvolvida em forma de *plug-in*, a utilização é bastante fácil, uma vez que o *plug-in* é integrado a IDE. E, utilizando ontologias, nossa abordagem se torna altamente configurável e extensível, ou seja, a criação de regras para identificação de novos *smells* é simples e não modifica o código da aplicação nem o núcleo da ontologia.

6.1.1 Smell Insufficient Modularization

O primeiro *smell* analisado foi o *smell Insufficient Modularization* apresentado por (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014). Para identificação desse *smell* foram consideradas classes que continham uma quantidade maior ou igual a 50 métodos e/ou atributos.

Na tabela 2 é apresentado a quantidade total de instâncias do *smell Insufficient Modularization* identificado por cada ferramenta nos três sistemas avaliados. Como pode ser visto, a ferramenta DECOR identificou 0 ocorrências no JUnit, 1 ocorrência no Log4J e 4 no ArgoUML. O JDeodorant foi a ferramenta que mais apresentou ocorrências, sendo 24 no JUnit, 16 no Log4J e 72 no ArgoUML. Por outro lado, a ferramenta CheckStyle não apresentou nenhuma ocorrência do *smell* em questão, enquanto o *plug-in* OWLSmell apresentou um total de vinte e sete ocorrências, sendo duas para o projeto JUnit, três para o Log4J e vinte e duas para o ArgoUML.

Tabela 2 – Quantidade de *smells Insufficient Modularization* encontrados por cada ferramenta

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit 4.12	0	24	3	0	2
Log4J 1.2.1	1	16	16	0	3
ArgoUML 0.9.18	4	72	72	0	22
Total	5	112	91	0	27

Após a execução das ferramentas nos sistemas, foi realizada uma análise manual em cada classe classificada como uma ocorrência de *smell* a fim de verificar se a classe realmente apresentava um erro de *design* segundo os paramêtros usados. Essa análise foi realizada para identificar possíveis ocorrências falsas-positivas em alguma ferramenta.

A tabela 3 apresenta o resultado dessa análise, contendo a quantidade real de *smells* do tipo *Insufficient Modularization* em cada sistema.

Tabela 3 – Quantidade de ocorrências existentes do *smell Insufficient Modularization* por sistema

Sistema	Qtde de ocorrências
JUnit 4.12	2
Log4J 1.2.1	3
Argo UML 0.19.8	22
Total	27

O próximo passo foi comparar os dados obtidos na tabela 2 com os dados da tabela 3 para descobrir o número de ocorrências que poderiam ser consideradas válidas. O resultado é apresentado na tabela 4.

Tabela 4 – Quantidade válida de *smells Insufficient Modularization* segundo (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit 4.12	0	0	0	0	2
Log4J 1.2.1	0	0	3	0	3
ArgoUML 0.19.8	0	6	20	0	22
Total	0	6	23	0	27

Sabendo-se os valores encontrados por cada ferramenta, os valores reais de cada sistema e a quantidade total de *smells Insufficient Modularization* válidos, foi realizado o cálculo de *Precision and Recall* para avaliar a acurácia de cada ferramenta.

Tabela 5 – *Precision and Recall* do *smell Insufficient Modularization*

Sistema	DECOR		JDeodorant		PMD		CheckStyle		OWLSmell	
	P	R	P	R	P	R	P	R	P	R
JUnit 4.12	0%	0%	8%	0%	66%	0%	0%	0%	100%	100%
Log4J 1.2.1	100%	0%	18%	0%	18%	100%	0%	0%	100%	100%
ArgoUML 0.19.8	100%	0%	30%	27%	30%	90%	0%	0%	100%	100%

O DECOR é atualmente a ferramenta mais famosa e utilizada por pesquisadores no mundo inteiro (SOBRINHO; LUCIA; MAIA, 2018). No entanto, foi a ferramenta que apresentou o pior resultado para o *smell Insufficient Modularization*, seguido pelo *CheckStyle* que apesar de ser uma ferramenta bastante conhecida, não faz parte do top 5 das mais estudadas (SOBRINHO; LUCIA; MAIA, 2018). O JDeodorant mostrou péssimos resultados para os sistemas JUnit e Log4J, mas demonstrou uma pequena melhora no ArgoUML. Por outro lado, o PMD apresentou resultados baixos de *Precision* no Log4J e ArgoUML mas apresentou resultados muito bons de *Recall* nos mesmos sistemas com 100% e 90%, respectivamente. E, por último, o *plug-in* OWLSmell foi a ferramenta que apresentou os melhores resultados na avaliação, com 100% de *Precision and Recall* para todos os três sistemas.

6.1.2 Smell Long Method

O segundo *smell* analisado foi o *Long Method* (FOWLER, 1999). Para identificação desse *smell* foram consideradas classes que continham métodos com uma quantidade de linhas de código maior ou igual a 150.

Como apresentado anteriormente na tabela 2 para o *smell Insufficient Modularization*, a tabela 6 apresenta a quantidade de ocorrências do *smell Long Method* encontradas por ferramenta em cada sistema analisado.

Tabela 6 – Quantidade de *smells Long Method* encontrados por cada ferramenta

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit	46	0	0	0	0
Log4J	49	0	1	1	1
ArgoUML	315	0	10	10	10
Total	410	0	11	11	11

A partir dos dados encontrados na tabela 6, foi realizado uma análise manual para descobrir as ocorrências reais em cada sistema. O resultado dessa análise pode ser encontrado na tabela 7.

Tabela 7 – Quantidade de ocorrências existentes do *smell Long Method* por sistema

Sistema	Qtde ocorrências
JUnit 4.12	0
Log4J 1.2.1	1
Argo UML 0.19.8	10
Total	11

Como pode ser visto na tabela 6, o DECOR encontrou uma quantidade muito superior a existente na tabela 7, enquanto as outras ferramentas encontraram a mesma quantidade. No entanto, vale lembrar que mesmo encontrando a mesma quantidade apresentada na tabela 7, não significa que as classes identificadas são válidas. As quantidade de classes que realmente contém *smell* são apresentadas na tabela 8.

Tabela 8 – Quantidade válida de *smells Long Method* segundo (FOWLER, 1999)

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit	0	0	0	0	0
Log4J	0	0	1	1	1
ArgoUML	0	0	10	10	10
Total	0	0	11	11	11

A tabela 9 mostra os resultados da aplicação das métricas *Precision and Recall* nos dados apresentados na tabela tabela 6, 7 e 8.

Tabela 9 – *Precision and Recall* do *smell Long Method*

Sistema	DECOR		JDeodorant		PMD		CheckStyle		OWLSmell	
	P	R	P	R	P	R	P	R	P	R
JUnit 4.12	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Log4J 1.2.1	2%	0%	0%	0%	100%	100%	100%	100%	100%	100%
ArgoUML 0.19.8	3%	0%	0%	0%	100%	100%	100%	100%	100%	100%

Os resultados apresentados pela tabela 9 mostram que para o *smell Long Method*, a ferramenta DECOR e JDeodorant apresentaram os piores resultados. PMD, CheckStyle e OWLSmell tiveram resultados idênticos tanto para *Precision* quanto para *Recall*. Por outro lado, a ferramenta JDeodorant teve bons resultados somente para o sistema JUnit. Esses resultados apresentam a eficiência das ferramentas PMD, CheckStyle e OWLSmell para detectar o *smell Long Method*.

6.1.3 Smell Long Parameter List

O próximo *smell* estudado nos experimentos foi o *Long Parameter List*. Para identificação desse *smell* foram consideradas classes que continham métodos com uma quantidade de parâmetros maior ou igual a sete.

A tabela 10 mostra as ocorrências encontradas pelas ferramentas em cada sistema. A coluna da ferramenta JDeodorant se encontra sem resultados pois a mesma não identifica esse *smell*.

Tabela 10 – Quantidade de *smells Long Parameter List* encontrados por cada ferramenta

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit	1	-	0	0	0
Log4J	7	-	0	5	5
ArgoUML	235	-	0	1	1
Total	243	-	0	6	6

A lista contendo a quantidade real de ocorrências em cada sistema é apresentada pela tabela 11.

Tabela 11 – Quantidade de ocorrências existentes do *smell Long Parameter List* por sistema

Sistema	Qtde ocorrências
JUnit 4.12	0
Log4J 1.2.1	5
Argo UML 0.19.8	1
Total	6

Como pode ser visto na tabela 10, as ferramentas CheckStyle e OWLSmell tiveram a mesma quantidade de ocorrências apresentadas pela tabela 11. Por outro lado, a ferramenta PMD não identificou nenhuma ocorrência desse *smell*. Enquanto isso, a ferramenta DECOR encontrou 235 ocorrências contra uma das outras ferramentas no sistema ArgoUML.

Na tabela 12 são apresentados os resultados da validação de todas as ocorrências encontradas pelas ferramentas. Apesar do número de ocorrências encontradas pelo DECOR no sistema ArgoUML, somente uma classe demonstrou ser válida contendo o *smell Long Parameter List*. Todos os resultados encontrados pelas ferramentas CheckStyle e OWLSmell são válidos.

Tabela 12 – Quantidade válida de *smells Long Parameter List* segundo (FOWLER, 1999)

Sistema	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
JUnit	0	-	0	0	0
Log4J	5	-	0	5	5
ArgoUML	1	-	0	1	1
Total	6	-	0	6	6

Com esses resultados os cálculos de *Precision and Recall* para esse *smell* são apresentados na tabela 13.

Tabela 13 – *Precision and Recall* do *smell Long Parameter List*

Sistema	DECOR		JDeodorant		PMD		CheckStyle		OWLSmell	
	P	R	P	R	P	R	P	R	P	R
JUnit 4.12	0%	100%	-	-	100%	100%	100%	100%	100%	100%
Log4J 1.2.1	71%	100%	-	-	0%	0%	100%	100%	100%	100%
ArgoUML 0.19.8	0%	100%	-	-	0%	0%	100%	100%	100%	100%

Como esperado, as ferramentas CheckStyle e OWLSmell tiveram os melhores resultados. O PMD apresentou bons resultados para o sistema JUnit porque nesse sistema não existe nenhuma ocorrência de *smell Long Parameter List* e a ferramenta identificou 0 ocorrências. Já a ferramenta DECOR apresentou bons resultados de *Recall* em todos os sistemas, mas os resultados de *Precision* não foram tão bons.

6.1.4 Smell Deficient Encapsulation

O último *smell* estudado nesse trabalho foi o *Deficient Encapsulation* apresentado por (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014). Para identificação desse *smell* foram consideradas classes que continham pelo menos um atributo com tipo de acesso público.

A tabela 14 apresenta o número de ocorrências encontradas por cada ferramenta. As únicas ferramentas que identificam esse *smell* são o CheckStyle e o OWLSmell.

Tabela 14 – Quantidade de *smells Deficient Encapsulation* encontrados por cada ferramenta

Sistema	CheckStyle	OWLSmell
JUnit	59	86
Log4J	101	116
ArgoUML	18	115
Total	178	317

A lista com a quantidade real de ocorrências para esse *smell* é apresentada na tabela 15.

Tabela 15 – Quantidade de ocorrências existentes do *smell Deficient Encapsulation* por sistema

Sistema	Qtde ocorrências
JUnit 4.12	86
Log4J 1.2.1	116
Argo UML 0.19.8	115
Total	317

A tabela 16 apresenta o número de ocorrências válidas em cada sistema. Como pode ser visto, esse *smell* parece ser bem comum nos três sistemas analisados.

Tabela 16 – Quantidade válida de *smells Deficient Encapsulation* segundo (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014)

Sistema	CheckStyle	OWLSmell
JUnit	59	86
Log4J	101	116
ArgoUML	18	115
Total	178	317

A tabela 17 apresenta os resultados da aplicação das métricas *Precision and Recall*. A ferramenta CheckStyle apresentou bons resultados para *Precision* mas não tão bons para *Recall*. A ferramenta OWLSmell apresentou ótimos resultados para ambas as métricas.

Tabela 17 – *Precision and Recall* do *smell Deficient Encapsulation*

Sistema	CheckStyle		OWLSmell	
	P	R	P	R
JUnit	100%	68%	100%	100%
Log4J	100%	87%	100%	100%
ArgoUML	100%	15%	100%	100%

6.2 Análise por amostragem

No segundo experimento, criamos um projeto Java contendo ocorrências de todos os *design smells* abordados nesse trabalho. Cada *smell* foi separado por pacote, facilitando a visualização das ocorrências ao executar a ferramenta OWLSmell, como pode ser visto na figura 11.

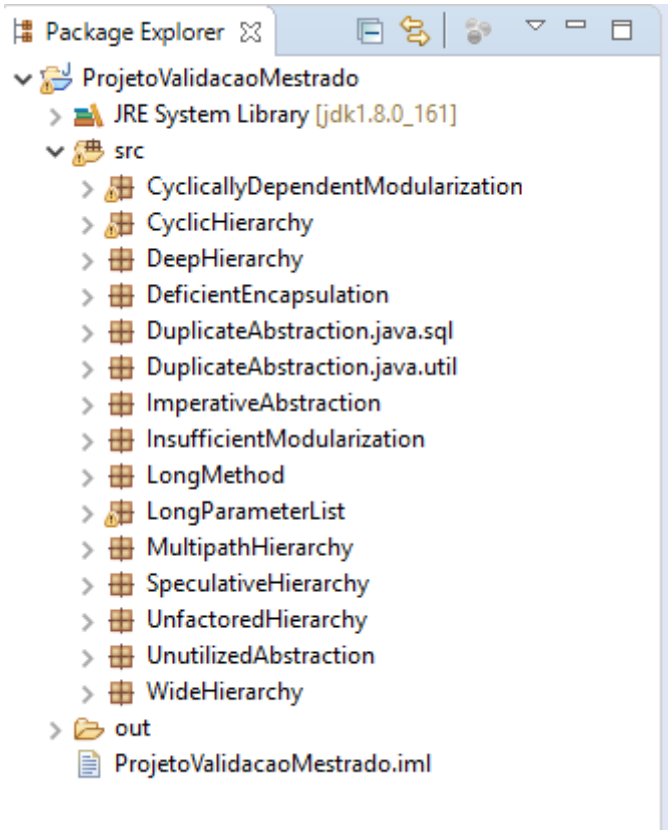

Figura 11 – Projeto Java contendo todos os smells

Fonte: Dados do trabalho

Nesse projeto, foi criado apenas uma ocorrência de cada *smell*, com exceção de alguns *smells* como o *CyclicHierarchy* que apresenta mais de uma ocorrência, pois é necessário mais de uma classe para o *smell* surgir no código. Com essa validação, conseguimos provar que a acurácia de nossa ferramenta chega aos 100%.

Como já dito anteriormente, a utilização de regras lógicas nos permite atingir altos níveis de acurácia, uma vez que as regras trabalham apenas com hipóteses binárias, ou seja, sim e não. Então, uma vez que os dados forem extraídos de forma correta pelo *plug-in* e esses mesmos dados forem compatíveis com as regras modeladas, a acurácia será sempre alta.

A figura 12 apresenta o console do Eclipse com o resultado da identificação no projeto mencionado anteriormente.

Figura 12 – Todos os smells validados pelo OWLSmell

Fonte: Dados do trabalho

A tabela 18 apresenta de forma mais clara a quantidade real e a quantidade encontrada de cada *smell* pela ferramenta OWLSmell. Nela, conseguimos observar claramente que todas as ocorrências existentes foram encontradas, e também podemos observar que como proposto nesse trabalho, a ferramenta é capaz de identificar 14 *smells*; mais que as outras ferramentas analisadas anteriormente.

Tabela 18 – Identificação dos smells propostos

Smell	Ocorrências reais	Ocorrências encontradas
SmellImperativeAbstraction	7	7
SmellUnfactoredHierarchy	2	2
SmellSpeculativeHierarchy	11	11
SmellDeepHierarchy	2	2
SmellCyclicallyDependentModularization	2	2
SmellWideHierarchy	1	1
SmellCyclicHierarchy	1	1
SmellDuplicateAbstraction	2	2
SmellUnutilizedAbstraction	3	3
SmellMultipathHierarchy	1	1
SmellDeficientEncapsulation	1	1
SmellInsufficientModularization	1	1
SmellLongParameterList	1	1
SmellLongMethod	1	1

Com essa validação conseguimos provar o poder de identificação de *smells* que a utilização de ontologias proporciona. O domínio da OO pode ser considerado bastante complexo, com muitas regras e muitas características. Contudo, a ontologia consegue expressar os conceitos de forma clara e simples, e quando associada a regras lógicas, o poder de expressividade aumenta ainda mais.

Dessa forma, conseguimos demonstrar que mesmo *smells*, com características complexas, podem ser identificados com a utilização de ontologias.

6.3 Avaliação dos Resultados

Nessa seção serão apresentados os resultados para as perguntas de pesquisa apresentadas no capítulo anterior.

- ❑ ***RQ #1* É possível e viável detectar e classificar *design smells* automaticamente através de ontologias e tecnologias da web semântica?**

Esse trabalho teve como objetivo a identificação e classificação automática de *design smells* em códigos orientados a objetos através do uso de ontologias e tecnologias da web semântica. Durante todo seu desenvolvimento, foi criada uma ontologia de forma a

modelar o conhecimento do domínio da OO e da engenharia de *software*, quando tivemos que lidar com muitos desafios, mudar nossa forma de pensar e de desenvolver a ontologia. Por fim, como visto nas seções anteriores, conseguimos criar uma ontologia capaz de identificar e classificar automaticamente quatorze *design smells* citados por (FOWLER, 1999) e (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Também conseguimos avaliar a viabilidade da identificação de *smells* automática com o uso de ontologias através do desenvolvimento de um *plug-in* para o ambiente Eclipse e a validação do mesmo em comparação com outras ferramentas existentes que possuem o propósito similar. Como visto na seção de experimentos, nosso *plug-in* obteve resultados tão bons quanto as melhores ferramentas criadas para tal, e em alguns casos, nossa ferramenta ainda apresentou resultados superiores.

Logo, conseguimos validar a hipótese de que é totalmente possível e viável a modelagem, identificação e classificação de *design smells* através do uso de ontologias e web semântica.

- ❑ ***RQ #2* A identificação e classificação de *design smells* apoiados pelo uso de ontologias e tecnologias da web semântica apresentam resultados significativamente positivos quando comparados com as técnicas de detecção mais comuns?**

Como visto anteriormente nos experimentos, a ferramenta OWLSmell, que tem todo seu núcleo de identificação e classificação formado por uma ontologia e regras SWRL, apresentou melhores resultados que as outras ferramentas que utilizam técnicas diferentes para identificação de *smells*.

Em alguns casos, a ferramenta OWLSmell demonstrou resultados similares a de outras ferramentas, como por exemplo na identificação do *smell Long Parameter List* na ferramenta CheckStyle, o que demonstra que o OWLSmell consegue acompanhar de perto os resultados de outras ferramentas já existentes há muito tempo.

Isso é proporcionado principalmente pelo uso da ontologia e sua capacidade de realizar inferências a partir das regras SWRL. Como a ontologia faz parte de um paradigma de mundo aberto, qualquer domínio de conhecimento é aceito e pode ser modelado, basta saber exatamente o que se quer representar no modelo e o que pretende-se inferir.

A ontologia da ferramenta OWLSmell foi muito bem planejada para abordar conceitos da POO de modo que as regras SWRL fossem as mais eficazes possíveis no momento da identificação e classificação dos *smells*. Os resultados podem ser vistos nas tabelas 5, 9, 13 e 17 apresentadas na seção anterior.

Sendo assim, é válido afirmar que o uso de ontologias e regras SWRL são altamente eficazes na busca por erros em códigos orientados a objetos e bastante recomendado a sua utilização em qualquer tipo de projeto que demande a inferência de conhecimento.

Capítulo 7

Trabalhos Correlatos

Neste capítulo serão apresentados os trabalhos encontrados na literatura que possuem alguma relação com este trabalho e que, de alguma forma, foram essenciais para a criação do mesmo.

No trabalho de (STOIANOV; ŞORA, 2010) foi proposto um método para detecção de padrões de projeto e *antipatterns* utilizando uma abordagem lógica baseada em predicados da linguagem Prolog.

Também foram avaliados os seguintes padrões de projeto definidos pelo Gang of Four (GOF) em (GAMMA et al., 1994): *Observer*, *Singleton*, *Strategy*, *Adapter* e *Decorator*. E o conjunto de *antipatterns* avaliados são descritos em (FOWLER, 1999) e (BROWN et al., 1998): *Data class*, *Call super*, *Constant interface*, *The blob*, *Refused interface*, *Yoyo problem* e *Poltergeist*.

Para avaliar a precisão de identificação do mecanismo desenvolvido, os autores comparam sua abordagem com a ferramenta Pinot[20], executando ambas as ferramentas em 6 projetos desenvolvidos na linguagem Java: *JHotDraw6.0b1*, *Java AWT 1.3*, *Java Swing 1.4*, *Java.io 1.4.2*, *Java.net 1.4.2* e *Apache Ant 1.6.2*.

Como resultados a respeito dos padrões de projeto, os autores verificaram que sua abordagem apresentou resultados relativamente similares aos da outra ferramenta, com exceção aos padrões *Observer* e *Adapter*. Com relação ao *Adapter*, apesar dos resultados apresentarem maiores ocorrências do que a ferramenta Pinot, foi realizada uma verificação manual para validação das ocorrências. Com essa verificação, concluiu-se que as ocorrências encontradas eram válidas segundo a definição encontrada no (GAMMA et al., 1994).

Por outro lado, para o padrão *Observer*, os resultados apresentaram menores ocorrências, devido ao fato de que a abordagem desenvolvida pelos autores ser mais rigorosa quanto as regras de detecção. Contudo, os autores consideraram sua abordagem mais precisa que a ferramenta Pinot, segundo as definições encontradas em (GAMMA et al., 1994).

[20] http://www.cs.ucdavis.edu/ shini/research/pinot/index.html

Por fim, os resultados encontrados na detecção de *antipatterns* identificaram que não havia a existência de falsos positivos entre as ocorrências. No entanto, estudos futuros foram propostos para identificação de falsos negativos (*antipatterns* que não foram identificados pela abordagem).

No trabalho de (TOURWE; MENS, 2003) é proposto um método para, automáticamente, identificar oportunidades de refatoração de uma aplicação e propor técnicas de refatoração adequadas para cada oportunidade. Tal abordagem é realizada através do uso de programação meta lógica.

Para avaliar a abordagem, os autores buscaram a identificação de dois *bad smells* (*Obsolete parameter* e *Inappropriate interface*) executando o método em uma ferramenta desenvolvida por eles próprios, devido a necessidade de conhecimento do código fonte e regras de negócio.

Apesar de propor técnicas para refatoração, não foi o objetivo desse trabalho realizar refatoração automática, uma vez que pode haver diversas maneiras de refatorar um certo trecho de código e nem sempre fica claro qual a melhor técnica. Diante disso, os autores proporam listar uma série de técnicas de forma a permitir o desenvolvedor escolher a mais adequada.

Com o resultado da execução, chegou-se a seguinte conclusão: a maioria das refatorações propostas foram efetivamente aplicada para obter um *design* mais limpo e melhor. Se uma refatoração específica não foi aplicada, por qualquer motivo, uma análise mais detalhada do *bad smell* identificado revelou que havia realmente um problema que deveria ser resolvido. Assim, foi suposto que esses problemas só agravam os aplicativos em larga escala trabalhados por muito mais desenvolvedores.

Uma desvantagem potencial da abordagem é que muitas refatorações podem ser propostas. Assim, o desenvolvedor pode receber uma grande lista de refatorações e pode não ver mais resultados. Isso é inevitável, no entanto, uma vez que um *bad smell* em particular pode ser sanado por uma infinidade de refatorações.

No trabalho de (FONTANA; BRAIONE; ZANONI, 2012) os autores fazem uma revisão sobre o estado atual de ferramentas que detectam *smells* automaticamente. São analisados os resultados da execução de quatro ferramentas em seis versões diferentes do GanttProject[21], um sistema *open source* escrito na linguagem Java.

Para escolher as quatro ferramentas que seriam utilizadas no estudo, os autores primeiro avaliaram as ferramentas mais citadas na literatura, sendo elas: CheckStyle, DECOR, inFusion, iPlasma, JDeodorant, PMD e Stench Blossom. Após isso, foi realizado um estudo sobre quais *smells* abordados por (FOWLER, 1999) são detectados por cada ferramenta. Após esses estudos, foram escolhidas as ferramentas JDeodorant, inFusion, PMD e CheckStyle, devido ao fato dessas ferramentas estarem disponíveis para *download* e até o momento do estudo elas eram ativamente mantidas.

[21] https://www.ganttproject.biz/

Sobre os *smells* analisados, o experimento considerou um total de seis *smells*, sendo eles compartilhados por pelo menos duas ferramentas: *Duplicated Code, Feature Envy, God Class, Large Class, Long Method, Long Parameter List*.

Com esse estudo, os autores buscaram responder as seguintes perguntas: **1)** Diferentes ferramentas de detecção apresentam resultados similares quando aplicadas ao mesmo sistema? **2)** Quão relevante é a detecção automática de *smells* para a evolução de um *software*? **3)** A presença de *smells* está relacionada a alguma característica observável do código-fonte ou do processo? E as respostas encontradas são listadas abaixo:

1. Os experimentos demonstraram que diferentes ferramentas para o mesmo *smell* produziram diferentes resultados mesmo quando elas possuíam técnicas de identificação similares. A única exceção foi a respeito das ferramentas que identificam o *smell God Class*.

2. Os experimentos mostraram um número bastante positivo na porcentagem de *smells* refatorados de uma versão para a próxima. Isso sugere que as ferramentas são aptas a detectar regiões de código sensíveis a refatoração. Esse resultado demonstra que a detecção automática de *smells* é bastante relevante para a evolução de um *software*.

3. Os experimentos mostraram que a presença de *smells* está aparentemente relacionada com características observáveis dos sistemas analisados. Não foi possível verificar uma significante correlação estatística devido ao tamanho do conjunto de dados analisado serem pequenos e porque características observáveis não são facilmente mensuráveis.

No trabalho (ITO et al., 2014), os autores desenvolveram um método para detectar *bad smells* em código utilizando Declarative Meta Programming (DMP) e árvore de sintaxe abstrata (Abstract Syntax Tree (AST)), associados com o Prolog.

O método DMP é essencialmente o uso de uma linguagem de programação declarativa para raciocionar e manipular programas em uma linguagem base subjacente. Esse método permitiu aos autores descrever vários *bad smells* como uma notação unificada que consiste em um programa declarativo.

A árvore de sintaxe abstrata foi utilizada para melhorar o processo de detecção, uma vez que uma AST representa o código fonte de maneira que cada nó da árvore contém uma informação detalhada sobre o código, como por exemplo o nome de uma classe, ou informações a respeito de variáveis e métodos.

Utilizando esses métodos juntamento com o Prolog, os autores foram capazes de analisar a estrutura do código fonte em detalhes, e criar regras de inferência utilizando lógica de primeira ordem para detectar os *smells*.

Então, a partir do método proposto foi desenvolvida uma ferramenta em forma de *plug-in* para o ambiente de desenvolvimento Eclipse. O Eclipse foi escolhido, pois, segundo os autores, é um ambiente bastante conhecido e muito utilizado por desenvolvedores.

Dessa forma, os autores buscaram aplicar tal ferramenta na educação para que alunos pudessem aprender conceitos da engenharia de *software* com mais facilidade. Eles chegaram a conclusão que a melhor forma de aprender sobre qualidade de *software* seria através da habilidade prática. Assim, aprender o conceito de *bad smells* e como refatorá-los seria de grande importância para os alunos.

Baseado nessa ideia, eles determinaram que o *plug-in* seria integrado também a uma plataforma de controle de versão, como o Git[22], por exemplo. Assim, os estudantes iriam desenvolver seus códigos, commitar no repositório, e o *plug-in* analisaria os *commits* e enviaria a um servidor *logs* contendo informações a respeito de possíveis *smells* encontrados no código.

Logo, os estudantes teriam um histórico das modificações realizadas, quais modificações apresentaram problemas e quais foram as soluções que removeram os problemas do código.

Com a análise dos trabalhos mencionados acima, foi possível realizar um comparativo entre os *smells* analisados pelas principais ferramentas de detecção abordadas. A partir da tabela 19 é possível ter um panorama geral sobre os *smells* estudados.

Na tabela 19 pode-se observar que a ferramenta desenvolvida nesse trabalho é capaz de identificar uma maior quantidade de *smells* quando comparadas as outras ferramentas. Tal característica foi possível devido a utilização de um sistema especialista que utiliza uma ontologia como motor de inferência e regras lógicas em sua base de conhecimento.

Ao utilizar uma ontologia e regras lógicas, nossa abordagem se torna capaz de ser extensível e compartilhável, uma vez que a ontologia não fica associada ao código fonte do sistema, possuindo ainda a capacidade de ser reaproveitada em outros contextos e poder ser reutilizada para o desenvolvimento de diferentes tipos de sistemas, seja *plug-ins*, sistemas *web*, sistemas *mobile* e etc.

A abordagem proposta é implementada em OWL e SWRL, linguagens estas que se baseiam em um formalismo lógico (lógica de predicados de primeira ordem) para representação de conhecimento, caracterizadas pela decidibilidade (existe um algoritmo de prova para um conjunto de sentenças), consistência (um algoritmo de inferência gera apenas sentenças dedutíveis) e completude (é possível achar a prova de todo predicado dedutível). Dessa forma, como resultado obteve-se uma acurácia de 100% em todos os testes realizados (como poder ser visto no capítulo 4).

Portanto, quando comparados os benefícios da nossa abordagem com as outras existentes, pode-se visualizar um avanço no estado da arte a respeito de técnicas de identificação de *smells*, ressaltando novamente os benefícios na utilização de sistemas especialistas

[22] https://git-scm.com/

associados a tecnologias da web semântica para modelagem e compartilhamento de conhecimento.

Tabela 19 – Comparação dos smells analisados nos trabalhos correlatos

Smell	DECOR	JDeodorant	PMD	CheckStyle	OWLSmell
Cyclic Hierarchy					•
Deep Hierarchy					•
Multipath Hierarchy					•
Speculative Hierarchy					•
Unfactored Hierarchy					•
Wide Hierarchy					•
Cyclically Dependent Modular.					•
Insufficient Modularization	•	•	•	•	•
Duplicate Abstraction					•
Imperative Abstraction					•
Deficient Encapsulation				•	•
Unutilized Abstraction					•
Long Method	•	•	•	•	•
Long Parameter List	•		•	•	•
Refused Bequest	•				
Functional Decomposition	•				
Spaghetti Code	•				
Swiss Army Knife	•				
Feature Envy		•			
Type Checking		•			
Duplicate Code		•	•		
Illegal Type				•	

Capítulo 8

Conclusão

Neste trabalho, foram avaliados os resultados da execução de quatro ferramentas identificadores de *smells* em três projetos *open source*, sendo eles, JUnit, Log4J e ArgoUML. As perguntas de pesquisa tiveram suas respostas baseadas nessa ánalise a partir da aplicação das métricas *Precision and Recall*. A seguir, serão apresentadas as principais contribuições deste trabalho.

8.1 Principais Contribuições

Este trabalho apresentou uma ferramenta para apoiar o processo de melhroria de qualidade de código fonte OO. A validação dessa ferramenta foi realizada através da comparação com outras ferramentas já abordadas na literatura, e os resultados foram extremamente positivos quando usado um sistema especialista para representar o domínio.

O principal objetivo desse trabalho foi verificar a importância da utilização de ontologias e tecnologias da *web* semântica para modelagem de conhecimento do domínio da engenharia de *software*. Para atingir esse objetivo foi necessário criar uma ontologia seguindo os conceitos de um diagrama de classe da Unified Modeling Language (UML) para representação de projetos orientados a objetos e identificação de *design smells* seguindo os principios catalogados por (FOWLER, 1999) e (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014).

Esse trabalho provou que a abordagem utilizada possui vantagens quando comparadas com as outras analisadas. Em primeiro lugar, apresentamos os benefícios em se utilizar um sistema especilista para identificação e classificação de *smells*. Em segundo, esse trabalho demonstrou que é possível modelar um domínio bastante complexo e realizar inferência a partir desse conhecimento, provando o enorme poder de expressividade da *web* semântica aliada a utilização de ontologias.

E a ferramenta desenvolvida ainda foi integrada a um ambiente de desenvolvimento bastante utilizado para desenvolvimento de *softwares* OO. Isso garante que desenvolve-

dores possam validar seus códigos utilizando somente uma aplicação, tornando assim o processo mais rápido.

Nossa ontologia também foi colocada em prova quando a ferramenta OWLSmell precisou ser validada com as ferramentas DECOR, JDeodorant, CheckStyle e PMD. Logo, os resultados apresentados pela OWLSmell foram superiores as outras abordagens, o que reforça o fato de que sistemas especialistas apoiados por *web* semântica são extremamente eficazes.

Como visto nos experimentos, nossa abordagem apresentou melhores resultados para todos os testes. Isso ocorreu pois nossa ferramenta utiliza um modelo deterministico em sua execução. Uma vez que temos os dados corretamente recuperados de um projeto de *software* orientado a objetos, e tais dados combinem com as regras que estão explicitamente configuradas na ontologia, nosso resultado será sempre ótimo, pois o núcleo de inferência de nossa abordagem é composto por regras "se-então", logo, sempre que a primeira parte da regra for verdadeira, consequentemente a segunda parte também será verdadeira, implicando em um resultado acertivo em 100% dos testes.

Assim esse trabalho buscou contribuir para a disseminação de informação sobre o domínio de engenharia de *software*, avaliar as principais abordagens existentes e melhorar qualidade da identificação de *smells* em ferramentas propondo a utilização de ontologias e regras SWRL bem como utilizar o conhecimento gerado para lecionar conceitos complexos.

E, por fim, a ontologia utilizada nesse trabalho possui a capacidade de ser compartilhada, uma vez que essa é uma das principais características em se utilizar ontologias. Com isso, diversos pesquisadores podem evoluí-la e também contribuir com esse projeto.

8.2 Produção Bibliográfica

- **Silva, V. J. S.**; Dorça, F. A. (2019). An automatic and intelligent approach for supporting teaching and learning of software engineering considering design smells in object-oriented programming. In (ICALT 2019) 19th IEEE International Conference on Advanced Learning Technologies - **Qualis B1**.

8.3 Trabalhos Futuros

Esse trabalho buscou introduzir a utilização de ontologias e conceitos da *web* semântica no domínio da engenharia de *software*. Nele foi desenvolvida a primeira versão da ferramenta OWLSmell, que utiliza uma ontologia para realizar a identificação de *smells* em projetos OO. Como projeto futuro, pretendemos evoluir o *design* visual da ferramenta, para que a apresentação das informações fique mais amigável para os usuários.

Também pretendemos evoluir mais a ontologia, criar mais regras, aumentar a quantidade de *smells* e também melhorar o código utilizado para recuperar os dados dos projetos

e popular a ontologia.

E, por último, também pretendemos realizar testes com estudantes da disciplina de programação orientada a objetos para melhorarmos nossa ferramenta de forma a adequá-la para ambos os contextos, mercado e acadêmico.

Referências

ABEL, M.; FIORINI, S. Uma revisão da engenharia do conhecimento: Evolução, paradigmas e aplicações. **International Journal of Knowledge Engineering and Management (IJKEM)**, v. 2, p. 1–35, 01 2013.

BROWN, W. et al. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. [S.l.: s.n.], 1998.

CHENG, Y.-P.; LIAO, J.-R. An ontology-based taxonomy of bad code smells. ACTA Press, Anaheim, CA, USA, p. 437–442, 2007.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. **Journal of Object Technology**, v. 11, 01 2012. Disponível em: <http://dx.doi.org/10.5381/jot.2012.11.2.a5>.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley, 1999. ISBN 0-201-48567-2.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.: s.n.], 1994.

GURU99. **Expert System in Artificial Intelligence: What is, Applications, Example**. 2019. <https://www.guru99.com/expert-systems-with-applications.html>. Acessado em: 26/09/2019.

HOSS, A. M. Ontology-based methodology for error detection in software design. 07 2006.

ISOTANI, S.; BITTENCOURT, I. **Dados Abertos Conectados: em Busca da Web do Conhecimento**. [s.n.], 2015. ISBN 978-85-7522-449-6. Disponível em: <http://dx.doi.org/10.13140/RG.2.1.4355.6329>.

ITO, Y. et al. A method for detecting bad smells and its application to software engineering education. p. 670–675, 2014. Disponível em: <http://dx.doi.org/10.1109/IIAI-AAI.2014.139>.

LIAO, S.-H. Expert system methodologies and applications - a decade review from 1995 to 2004. **Expert Systems with Applications**, v. 28, p. 93–103, 01 2005. Disponível em: <http://dx.doi.org/10.1016/j.eswa.2004.08.003>.

LUCAS, P. J.; GAAG, L. C. Principles of expert systems. In: _____. [S.l.: s.n.], 1991.

MOHA, N. et al. Decor: A method for the specification and detection of code and design smells. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 36, n. 1, p. 20–36, jan. 2010. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2009.50>.

NONATO, H. et al. Uma abordagem baseada em ontologias para modelagem e avaliação do estudante em sistemas adaptativos e inteligentes para educação. 10 2017. Disponível em: <http://dx.doi.org/10.5753/cbie.sbie.2017.1197>.

NOY, N. F.; MCGUINNESS, D. Ontology development 101: A guide to creating your first ontology. **Knowledge Systems Laboratory**, v. 32, 01 2001.

PAIVA, T. et al. On the evaluation of code smells and detection tools. **Journal of Software Engineering Research and Development**, v. 5, p. 7, 12 2017. Disponível em: <http://dx.doi.org/10.1186/s40411-017-0041-1>.

SOBRINHO, E. Vicente de P.; LUCIA, A. D.; MAIA, M. A systematic literature review on bad smells — 5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, PP, p. 1–1, 11 2018. Disponível em: <http://dx.doi.org/10.1109/TSE.2018.2880977>.

STOIANOV, A.; ŞORA, I. Detecting patterns and antipatterns in software using prolog rules. In: . [s.n.], 2010. p. 253 – 258. Disponível em: <http://dx.doi.org/10.1109/ICCCYB.2010.5491288>.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for Software Design Smells: Managing Technical Debt**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN 0128013974, 9780128013977.

TAN, P.-N.; STEINBACH, M.; KUMAR, V. **Introduction to Data Mining, (First Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321321367.

TOURWE, T.; MENS, T. Identifying refactoring opportunities using logic meta programming. In: . [s.n.], 2003. p. 91– 100. ISBN 0-7695-1902-4. Disponível em: <http://dx.doi.org/10.1109/CSMR.2003.1192416>.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of move method refactoring opportunities. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 35, n. 3, p. 347–367, maio 2009. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2009.1>.

VOGEL, L.; SCHOLZ, S.; PFAFF, F. **Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model**. 2018. <https://www.vogella.com/tutorials/EclipseJDT/article.html>. Acessado em: 18/03/2019.

W3C. **Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model**. 2012. <https://www.w3.org/OWL/>. Acessado em: 18/09/2019.