IDENTIFICAÇÃO DE *BAD SMELLS* EM SOFTWARE A PARTIR DE MODELOS UML

HENRIQUE GOMES NUNES

IDENTIFICAÇÃO DE *BAD SMELLS* EM SOFTWARE A PARTIR DE MODELOS UML

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Co-orientadora: Kecia Aline Marques Ferreira

Belo Horizonte
Março de 2014

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Identificação de bad smells em software a partir de modelos UML

HENRIQUE GOMES NUNES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROFA. KÉCIA ALINE MARQUES FERREIRA - Coorientadora
Departamento de Computação - CEFET-MG

PROF. ANTONIO FRANCISCO DO PRADO
Departamento de Computação - UFSCar

PROF. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 28 de fevereiro de 2014.

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Nunes, Henrique Gomes.

N972i Identificação de badsmells em software a partir de modelos UML / Henrique Gomes Nunes — Belo Horizonte, 2014.
xvii, 80 f.: il.; 29 cm.

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Coorientadora: Kécia Aline Marques Ferreira.

1. Computação - Teses. 2. Engenharia de software – Teses. 3. UML (Linguagem de modelagem padrão) – Teses. 4. Software – Controle de qualidade - Teses. I. Orientadora. II. Coorientadora. III. Título.

519.6*32(043)

Agradecimentos

Quero agradecer primeiramente a Deus, a força, determinação e fé durante toda esta caminhada.

À Mariza e Kecia, que me acompanharam durante todo o meu mestrado, a paciência na orientação e incentivo que tornaram possível a conclusão desta dissertação.

À minha esposa, pai, mãe e irmã, que sempre me apoiaram nos momentos difíceis e fizeram-me acreditar que tudo era possível.

Aos amigos do meu trabalho pelo apoio e compreensão, aos amigos pessoais pelos bons momentos e a todos que contribuiram para meu crescimento neste período.

Muito obrigado!

Resumo

Métricas de software podem auxiliar na identificação de desvios de projeto, conhecidos na literatura como *bad smells*, e são úteis para avaliar a qualidade do código-fonte. Métricas também podem ser usadas para identificar problemas estruturais nas fases iniciais do ciclo de vida do software. Esta dissertação visa contribuir nesse aspecto, propondo um método e uma ferramenta para a identificação de *bad smells*, via métricas de software, em sistemas orientados por objetos a partir de modelos UML.

Neste trabalho, o método proposto foi avaliado em dois experimentos: um com o objetivo de analisar os resultados do método aplicado a versões antigas e versões refatoradas de um conjunto de seis softwares abertos; e outro com o objetivo de comparar os resultados do método com a análise manual. Os resultados desses experimentos indicam que o método proposto mostra-se útil para a identificação dos *bad smells* considerados nesta dissertação.

Palavras-chave: *Bad Smells*, Métricas, Qualidade de Software, Estratégias de Detecção, Valores Referência, Modelo UML.

Abstract

Software metrics may aid to identify design deviances, known in the literature as bad smells and are useful for evaluating the quality of source code. They also can be used for identifying design deviances in the early stages of the software lifecycle. This dissertation aims to contribute in this aspect, proposing a method and a tool for identifying bad smells, using software metrics, in UML models.

In this work, we carried out two experiments to evaluate the proposed method: the first one aimed to evaluate the results of our method when applied to old versions as well as to refactored versions of six open source projects; in the second experiment, we compare the results of our method with the results of manual inspections. The results of these experiments indicate that our method is able to identify the bad smells analyzed in this study.

Keywords: Bad Smells, Metrics, Software Quality, Detection Strategies, Thresholds, UML Model.

Lista de Figuras

3.1 Filtros de dados. Fonte: Marinescu [2004] 20
3.2 Estratégia de detecção do *God Class*. Fonte: Marinescu [2002] 21
3.3 Estrutura do iPlasma. Fonte: Marinescu et al. [2005] 25
3.4 Níveis do modelo de qualidade QMOOD. Fonte: Bertran [2009] 29
3.5 Métricas utilizadas no modelo QMOOD. Fonte: Bertran [2009] 30

4.1 Estratégia de detecção para o *God Class*. 37
4.2 Estratégia de detecção para o *Shotgun Surgery*. 38
4.3 Estratégia de detecção para o *Indecent Exposure*. 38
4.4 Casos de Uso da UMLsmell. 39
4.5 Estrutura da UMLsmell. 40
4.6 Diagrama de Classes da UMLsmell. 41
4.7 Tela com a lista de projetos criados por *UMLsmell*. 43
4.8 Tela para seleção de métricas de *UMLsmell*. 43
4.9 Telas para a definição dos valores das métricas de *UMLsmell*. 44
4.10 Tela para seleção de *métricas* para avaliação em *UMLsmell*. 44
4.11 Tela com as estatísticas de métricas de *UMLsmell*. 45
4.12 Tela para a seleção de *bad smells* para avaliação em *UMLsmell*. 45
4.13 Tela para as estatísticas de *bad smells* de *UMLsmell*. 46

5.1 Tabela com os resultados de *UMLsmell*. 51

A.1 Classe *Loader* do software *Picasso* 75
A.2 Classe *ServiceURL* do software *JSLP* 76

Lista de Tabelas

2.1 Valores referência definidos por Ferreira et al. [2012]. 15

3.1 Tabela de decisão usada para classificar os suspeitos das duas versões do sistema analisado. Fonte: Marinescu [2002]. 22
3.2 Resultado da avaliação automática de Marinescu [2002]. FP = Falsos Positivos, CE = Casos Especiais, FR = Falhas Reais. 23
3.3 Resultado de avaliação manual de Marinescu [2002]. DC = Detecção Correta, OB = Outro *Bad Smell*, PE = Precisão Estrita, PS = Precisão Solta. 24

5.1 *Bad smells* no *JHotdraw* Versão 5.2 52
5.2 *Bad smells* no *JHotdraw* Versão 7.0.6 53
5.3 Comparação dos *bad smells* no *JHotdraw* Versões 5.2 e 7.0.6 53
5.4 *Bad smells* no *Struts* Versão 1.1 54
5.5 *Bad smells* no *Struts* Versão 1.2.7 55
5.6 Comparação dos *bad smells* no *Struts* Versões 1.1 e 1.2.7 55
5.7 *Bad smells* no *HSqlDB* Versão 1.8.1 56
5.8 *Bad smells* no *HSqlDB* Versão 2.3.1 56
5.9 Comparação dos *bad smells* no *HSqlDB* Versões 1.8.1 e 2.3.1 56
5.10 *Bad smells* no *Jslp* Versão 0.7 57
5.11 *Bad smells* no *Jslp* Versão 1.0 57
5.12 Comparação dos *bad smells* no *Jslp* Versões 0.7 e 1.0 58
5.13 *Bad smells* no *Log4j* Versão 1.2 59
5.14 *Bad smells* no *Log4j* Versão 1.3 alpha 6 59
5.15 Comparação dos *bad smells* no *Log4j* Versões 1.2 e 1.3a6 59
5.16 *Bad smells* no *Sweethome 3D* Versão 2.5 60
5.17 *Bad smells* no *Sweethome 3D* Versão 3.5 60
5.18 Comparação dos *bad smells* no *Sweethome 3D* Versões 2.5 e 3.5 60
5.19 Avaliação manual do *Picasso*. 62

5.20 Avaliação do *Picasso* pela *UMLsmell*. 63
5.21 Avaliação manual do *JSLP* 64
5.22 Avaliação do *JSLP* pela *UMLsmell* 65

Sumário

Agradecimentos **v**

Resumo **vii**

Abstract **ix**

Lista de Figuras **xi**

Lista de Tabelas **xiii**

1 Introdução **1**

1.1 Definição do Problema 2
1.2 Objetivos 3
1.3 Contribuições 3
1.4 Organização do Texto 4

2 Referencial Teórico **5**

2.1 Métricas de Software para Diagramas de Classes 5
2.1.1 Métricas CK 6
2.1.2 Métricas de Li e Henry 7
2.1.3 Métricas MOOD 7
2.1.4 Métricas de Lorenz e Kidd 8
2.1.5 Métricas de Briand 9
2.1.6 Métricas de Marchesi 9
2.1.7 Métrica de Harrison 10
2.1.8 Métricas de Bansiya et al. 10
2.1.9 Métricas de Genero et al. 11
2.2 *Bad Smells* 12
2.3 Valores Referência 14

2.4 Conclusão 15

3 Trabalhos Relacionados **17**
3.1 Ferramentas de Medição Aplicadas a Modelos UML 17
3.2 Estratégias de Detecção de *Bad Smells* 19
3.2.1 Mecanismos de Filtragem de Métricas 19
3.2.2 Mecanismos de Composição 20
3.2.3 Avaliação das Estratégias de Detecção 21
3.3 Falhas de Software e *Bad Smells* 25
3.3.1 Análise de Correlação 26
3.3.2 Análise Delta 27
3.4 *Bad Smells* em Modelos UML 28
3.4.1 O Modelo de Qualidade QMOOD 28
3.4.2 Estudos Experimentais 31
3.5 Considerações Finais 32

4 Identificação de *Bad Smells* a partir de Diagramas de Classes **35**
4.1 Método Proposto 35
4.2 A Ferramenta *UMLsmell* 38
4.2.1 Requisitos de *UMLsmell* 38
4.2.2 Implementação de *UMLsmell* 40
4.2.3 Interface com o Usuário 43
4.3 Diferenciais do Método Proposto 46

5 Avaliação do Método Proposto **49**
5.1 Metodologia 49
5.1.1 Diagramas de Classes dos Experimentos 50
5.2 Análise de Softwares Reestruturados 51
5.2.1 Resultados da Avaliação Automática via *UMLsmell* 52
5.3 Avaliação Manual 61
5.3.1 Resultado da Avaliação Manual da Biblioteca Picasso 61
5.3.2 Resultado da Avaliação Manual do Protocolo JSLP 64
5.4 Análise dos Resultados das Avaliações Automática e Manual 65
5.5 Considerações Finais 67

6 Conclusão **69**
6.1 Trabalhos Futuros 70
6.2 Publicações 70

A Avaliação Manual dos *Bad Smells* **73**
A.1 Definição dos *Bad Smells* 73
A.2 Softwares Usados na Avaliação Manual 74
A.3 Questionário Usado para a Avaliação Manual 74
A.3.1 Picasso 75
A.4 Exemplo de Classes Avaliadas 75

Referências Bibliográficas **77**

Capítulo 1

Introdução

No contexto de Engenharia de Software, métrica corresponde a um método para determinar se um sistema, componente ou processo possui um determinado atributo (IEEE [1990]). Historicamente as métricas são utilizadas para estimar custo, esforço e tempo de desenvolvimento e manutenção.

As métricas mais comuns são usadas para avaliar o tamanho de um software e estimar o tempo que levará para desenvolvê-lo (Nuthakki et al. [2011]). A medição de softwares permite que uma organização melhore seu processo de desenvolvimento, auxiliando no planejamento, acompanhamento e controle, além de ser útil para a avaliação do software. Soliman et al. [2010] afirmam que muitas métricas de software têm sido propostas na literatura, para contabilizar: linhas de códigos, porcentagem de comentários, número de módulos, contagem de elementos, abstração de dados acoplados, etc. Eles ainda afirmam que a maioria das métricas são aplicadas nas fases posteriores ao projeto, entretanto sabe-se que é necessário compreender o projeto desde o seu início.

As métricas de software também podem auxiliar na identificação dos desvios de projeto, conhecidos na literatura como *bad smells* (Fowler [1999]). Alguns trabalhos têm sido desenvolvidos para identificar *bad smells* em software com o uso de métricas a partir de código fonte (Marinescu [2002], Lanza et al. [2006]). Todavia, poucos trabalhos propuseram avaliar *bad smells* nas fases iniciais de projetos de software (Bertran [2009]).

A *Unified Modeling Language* (UML) é uma família de notações gráficas, apoiada por um metamodelo único, que auxilia na descrição e no projeto de sistemas de softwares orientados por objetos (Fowler [2005]). A UML surgiu com a proposta de padronizar questões relacionadas ao projeto orientado por objetos, permitindo assim que um sistema seja representado independente da linguagem de programação utilizada. Ser capaz de obter métricas a partir de modelos da UML é importante, porque

a UML fornece uma visão da estrutura do software. Além disso, com a avaliação de modelos UML, a qualidade do software pode ser medida desde os estágios iniciais, o que pode tornar as intervenções no software menos dispendiosas. No entanto, poucas são as discussões de como aplicar as métricas propostas na literatura via UML (Soliman et al. [2010]).

Yi et al. [2004] citam em seu trabalho que diversos pesquisadores definiram métricas para softwares orientados por objetos, e que um grupo específico de estudiosos se concentrou na definição de métricas para modelos da UML. Os autores ainda afirmam que não há um consenso sobre quais as métricas a serem aplicadas nesses modelos. Alguns atribuem maior importância às métricas baseadas em métodos e atributos, enquanto outros acreditam que as classes e seus relacionamentos são os mais importantes.

Assim como no caso do código fonte, as métricas na UML podem permitir analisar aspectos como confiabilidade, manutenibilidade e complexidade de sistemas, porém, neste caso, logo nas fases iniciais do ciclo de vida do software. Dentre os diagramas definidos na UML destaca-se o diagrama de classes, que representa classes de objetos e suas relações. A análise desse diagrama pode fornecer diversas métricas relacionadas à manutenibilidade e complexidade do software logo no início do projeto. Porém, um projeto real necessita de uma ferramenta para automatizar este processo, pois é inviável calcular tais métricas manualmente (Girgis et al. [2009]).

O trabalho de dissertação proposto visa contribuir nesse aspecto, propondo um método e uma ferramenta para identificação de *bad smells* automaticamente em software orientado por objetos a partir de diagramas de classes da UML, aplicando métricas de software.

1.1 Definição do Problema

As pesquisas sobre *bad smells*, na maioria dos estudos, analisa desvios de projeto em código fonte (Fowler [1999], Marinescu [2002], Lanza et al. [2006]). Encontrar *bad smells* no código fonte é necessário para identificar demandas de refatoração. Porém, também é importante identificar *bad smells* nas fases iniciais do ciclo de vida do software, o que poderá contribuir para a diminuição do custo total do projeto, uma vez que realizar modificações quando o sistema já está em operação é mais dispendioso.

Isto posto, na pesquisa realizada para esta dissertação foram identificados alguns problemas que ainda não possuem soluções disponíveis na literatura, e as poucas existentes ainda não são satisfatórias. São eles:

- carência de métodos que identifiquem desvios de projetos nas fases iniciais do

desenvolvimento de software.

- ausência de métodos e ferramentas que automatizem a identificação de *bad smells* em modelos UML;
- há poucos experimentos que validam estratégias de detecção em modelos UML.

Contornar esses problemas é importante, pois a identificação de desvios de projeto nas fases iniciais do projeto de software, como na fase de modelagem por exemplo, contribuem para que os problemas nas fases posteriores do desenvolvimento de software possam ser reduzidos. O presente trabalho contribui para a solução desses problemas.

1.2 Objetivos

Este trabalho tem por objetivos:

- definir um método de identificação de *bad smells* em softwares a partir de diagrama de classes, aplicando métricas de software;
- desenvolver uma ferramenta que permita automatizar a coleta de métricas e a aplicação delas na identificação dos *bad smells* a partir de diagramas de classes;
- avaliar o método e a ferramenta propostos por meio de experimentos com softwares abertos.

1.3 Contribuições

A seguir, as principais contribuições desta dissertação.

1. Identificação das métricas que melhor representam os aspectos relacionados a *bad smells* em diagramas de classe.
2. Uma ferramenta, denominada *UMLsmell*, para auxiliar os engenheiros de software na identificação de *bad smells* logo nas fases iniciais do projeto.
3. Avaliação do método proposto via experimentos com software aberto.
4. Disponibilização de *UMLsmell*, em formato de código aberto, para que engenheiros de software possam fazer uso da ferramenta.

1.4 Organização do Texto

O restante desta dissertação está organizada da seguinte forma: Capítulo 2 faz uma breve revisão sobre conceitos fundamentais de métricas, atendo-se em especial àquelas para diagramas de classes, valores referência e *bad smells*. Capítulo 3 apresenta os trabalhos relacionados com o tema da dissertação. Os Capítulos 4 e 5 constituem o cerne desta dissertação. Capítulo 4 apresenta o modelo proposto e a ferramenta desenvolvida, além dos pontos positivos deste trabalho em relação aos outros. O Capítulo 5 discute os resultados dos experimentos feitos automaticamente com a ferramenta desenvolvida, e, manualmente com o auxílio de avaliadores com formação em Engenharia de Software. Apresenta também algumas limitações identificadas no decorrer de seu desenvolvimento. Capítulo 6 apresenta as conclusões do trabalho desta dissertação, e aponta direções a serem seguidas, relacionando os trabalhos futuros. Apêndice A apresenta o questionário usado pelos avaliadores para a avaliação manual, além das informações pertinentes para esta avaliação. Parte dos diagramas de classes dos dois sistemas avaliados são mostrados para exemplificar o processo realizado.

Capítulo 2

Referencial Teórico

Este capítulo faz uma revisão sobre conceitos fundamentais relacionados a métricas, em especial àquelas para diagramas de classes, valores referência e *bad smells*.

Sommerville [2007] define medição como um valor numérico relacionado a algum atributo de software, permitindo a comparação entre atributos da mesma natureza. Ele argumenta que a medição permite realizar previsões gerais de um sistema e identificar componentes com anomalias. Sommerville [2007] argumenta que por meio das métricas é possível controlar um software. É possível também, por exemplo, via atributos internos de um software, definir atributos externos como facilidade de manutenção, confiabilidade, portabilidade e facilidade de uso.

2.1 Métricas de Software para Diagramas de Classes

Em seu trabalho, Genero et al. [2005] relatam o resultado de uma pesquisa, baseada em 12 outras obras, na qual apresentam conjuntos de métricas para diagramas de classes. De acordo com Genero et al. [2005], a escolha pelo modelo de diagrama de classes justifica-se pelo fato desse representar a *espinha dorsal* do desenvolvimento orientado por objetos, fornecendo informações sobre a concepção e implementação de um software logo no início de um projeto, tal que o impacto de falhas nas fases posteriores seja minimizado. Eles também afirmam que a coleta de métricas relacionadas a diagramas de classes ainda é escassa, e que por muitos anos o foco principal de medição tem sido o código fonte. Para eles, as métricas de diagramas de classes são:

- identificam os pontos fracos de um software nas fases iniciais do ciclo de vida do projeto, quando uma modificação nele custará menos;

- permitem escolher alternativas de projeto de forma mais objetiva;
- e possibilitam prever características externas de qualidade, como: manutenção e reutilização, além de possibilitar uma melhor alocação de recursos para solucionar tais questões.

Os autores definem os seguintes elementos como fundamentais para a extração de métricas em diagramas de classes: pacotes, classes, atributos, métodos, parâmetros e relacionamentos de associação, agregação, generalização e dependência. Identificados esses elementos, a escolha das métricas feita por Genero et al. [2005] seguiram os seguintes princípios básicos:

- a métrica deve ser clara ao definir seus objetivos;
- a métrica deve medir o atributo o qual ela se propõe avaliar;
- a métrica deve ser validada para saber se ela realmente é útil;
- o cálculo da métrica deve ser fácil e de possível automatização.

Para validar a escolha de suas métricas os autores utilizam cinco critérios: definição da métrica, objetivo da métrica, validação teórica, validação empírica e possibilidade de automatização.

A seguir são definidas as métricas levantadas por Genero et al. [2005]. Para cada um dos conjuntos de métricas apresentados nas próximas seções, são descritas aquelas identificadas como possíveis colaboradoras no reconhecimento dos *bad smells* em diagramas de classes.

2.1.1 Métricas CK

O objetivo das métricas CK (Chidamber & Kemerer [1994]) é medir a complexidade de um projeto em relação a atributos externos de qualidade como manutenção e reutilização de software. Ao total, existem seis métricas definidas no conjunto CK, porém apenas três, descritas a seguir, podem ser extraídas a partir de diagramas de classe:

- WMC (*Weighted Methods per Class*): diz respeito à complexidade dos métodos de uma classe. Usualmente, o WMC equivale à quantidade de métodos em uma classe no diagrama de classes.
- DIT (*Depth of Inheritance*): refere-se à distância entre uma determinada classe e a superclasse raiz na árvore de herança do software.

- NOC (*Number of Children*): representa o número de subclasses subordinadas diretamente à uma superclasse.

2.1.2 Métricas de Li e Henry

O objetivo das métricas de Li & Henry [1993] é medir atributos internos como acoplamento, complexidade e tamanho. Estas métricas são definidas no nível de classes:

- DAC (*Data Abstraction Coupling*): número de atributos que referenciam outra classe.
- DAC': número de diferentes classes que são usadas em uma determinada classe.
- NOM (*Number of Methods*): número de métodos locais.
- SIZE2 (*number of methods plus number of attributes*): número de atributos da classe somado ao seu número de métodos locais.

2.1.3 Métricas MOOD

O objetivo das métricas MOOD (Abreu & Carapuça [1994]) é medir a produtividade no desenvolvimento. Essas métricas são definidas para prover uma avaliação em nível de sistema, em vez de em nível de classe. Questões como herança, encapsulamento e polimorfismo são tratadas. Dentre as métricas MOOD, existem seis que são aplicáveis a diagrama de classes:

- MHF (*Method Hiding Factor*): representa o quociente entre o total de métodos privados no sistema e a quantidade total de métodos no sistema.
- AHF (*Attribute Hiding Factor*): diz respeito ao quociente entre os atributos privados e a quantidade total de atributos no sistema.
- MIF (*Method Inheritance Factor*): refere-se ao quociente entre os métodos herdados pela quantidade de métodos locais mais os métodos herdados.
- AIF (*Attribute Inheritance Factor*): representa o quociente entre os atributos herdados pela quantidade de atributos locais mais os atributos herdados.
- PF (*Polymorphism Factor*): representa o quociente entre o número atual de situações de polimorfismo e o numéro total de métodos reescritos, considerando que um método pode ser reescrito várias vezes.

2.1.4 Métricas de Lorenz e Kidd

As métricas de Lorenz e Kidd (Lorenz & Kidd [1994]) têm como objetivo avaliar questões estáticas de projeto de software como herança e responsabilidade das classes. Elas se dividem em três grupos: métricas de tamanho, de herança e de aspectos internos da classe.

Métricas de Tamanho de Classe

- PIM (*Public Instance Methods*): conta o número total de métodos públicos em uma classe.
- NIM (*Number of Instance Methods*): conta todos os métodos públicos, protegidos e privados definidos na classe.
- NIV (*Number of Instance Variables*): conta o número total de atributos em uma classe.
- NCM (*Number of Class Methods*): conta o número total de métodos em uma classe.
- NCV (*Number of Class Variables*): conta o número total de atributos que são do tipo classe.

Métricas de Herança de Classes

- NMO (*Number of Methods Overridden*): conta o número total de métodos redefinidos nas subclasses de uma superclasse.
- NMI (*Number of Methods Inherited*): é o número total de métodos herdados por uma classe.
- NMA (*Number of Methods Added*): conta o número total de métodos definidos em uma classe.
- SIX (*Specialization Index*): é o produto de métodos redefinidos na classe e o nível de profundidade hierárquica dividido pelo número total de métodos de uma classe.

Questões Internas da Classe

- APPM (*Average Parameter per Method*): representa a divisão do número total de parâmetros pelo número total de métodos.

2.1.5 Métricas de Briand

As métricas de Briand et al. [1997] têm o objetivo de medir o acoplamento entre as classes. O nome dessas métricas são definidas por letras da seguinte forma: ACAIC, OCAIC, DCAEC, OCAEC, ACMIC, OCMIC, DCMEC, OCMEC, tal que o significado de cada letra é:

- primeira letra indica o relacionamento: *A* representa o acoplamento com as classes ancestrais, *D*, os descendentes e *O*, outros tipos de relacionamentos;
- a segunda e terceira letras indicam o tipo de interação: *CA* é interação de uma classe com um atributo e *CM* é a interação de uma classe com um método;
- a quarta e quinta letras indicam o direcionamento: *IC* (*import coupling*) são conexões aferentes, ou seja, aquelas que chegam à classe. *EC* (*export coupling*) são conexões eferentes, ou seja, conexões que se originam da classe.

2.1.6 Métricas de Marchesi

O objetivo das métricas de Marchesi [1998] é medir a complexidade do software e equilibrar as responsabilidades entre pacotes e classes. As métricas com as inicias *CL* referem-se a métricas de classes, *PK*, métricas de pacotes e *OA*, métricas de complexidade global.

Marchesi [1998] define responsabilidade como informações retidas ou cálculos que devem ser realizados. Também define que dependência são todos os relacionamentos, exceto os de herança.

Métricas de Classes

- CL1: número de responsabilidades, herdadas ou não, de uma classe.
- CL2: número de relacionamentos de dependência de uma classe.

Métricas de Pacotes

Seja Ca uma classe do pacote Pa:

- PK1: quantidade de relacionamentos de Ca com classes não pertencentes ao pacote Pa.
- PK2: quantidade de relacionamentos de Ca com outras classes do pacote Pa.
- PK3: valor médio dos resultados da métrica PK1 em todas as classes.

Métricas de Sistema

- OA1: número total de classes.
- OA2: número total de relacionamentos de herança.
- OA3: é a média de todos os valores de CL1.
- OA4: desvio padrão para a métrica OA3.
- OA5: é a média de todos os valores de CL2.
- OA6: desvio padrão da métrica OA5.
- OA7: porcentagem das responsabilidades herdadas em relação ao seu número total.

2.1.7 Métrica de Harrison

Harrison et al. [1998] definem a métrica NAS (*Number of Associations*) que tem como objetivo medir o nível de acoplamento entre as classes. Essa métrica se refere à quantidade de relações de associação que se origina de uma classe.

2.1.8 Métricas de Bansiya et al.

As métricas de Bansiya & Davis [2002] foram definidas para avaliar propriedades de projeto, como encapsulamento (DAM), acoplamento (DCC), coesão (CACM), composição (MOA) e herança (MFA).

- DAM (*Data Access Metric*): é a proporção dos atributos privados em relação à quantidade total de atributos.
- DCC (*Direct Class Coupling*): representa a quantidade de classes às quais uma classe está diretamente relacionada.
- CAMC (*Cohesion Among Methods of Class*): é um indicador da afinidade entre métodos de uma classe com base na lista de parâmetros de métodos.
- MOA (*Measure of Aggregation*): representa a quantidade de atributos que fazem referência a outras classes.
- MFA (*Measure of Functional Abstraction*): define a relação entre o número de métodos herdados pela classe avaliada e o número total de métodos, de outras classes, acessados por um método da classe avaliada.

- DSC (*Design Size in Classes*): conta o número total de classes no projeto.
- NOH (*Number of Hierarchies*): conta o número total de hierarquias no projeto.
- ANA (*Average Number of Ancestors*): conta o número de classes ao longo de todos os caminhos da classe raiz para todas as classes em uma estrutura de herança.
- NOP (*Number of Polymorphism*): conta a quantidade total de métodos que foram reescritos em outras classes.

2.1.9 Métricas de Genero et al.

As métricas de Genero [2002] avaliam atributos de qualidade externos como manutenção via relacionamentos de associações, generalização, agregação e dependências. Essas métricas se dividem em dois grupos: métricas de escopo do diagrama de classe e métricas de escopo de classe.

Algumas métricas de Genero [2002] são ambíguas e de difícil interpretação, mas como elas também são identificadas como possíveis colaboradoras no reconhecimento dos *bad smells* em diagramas de classes, não poderiam ser ignoradas. A seguir, são apresentadas as descrições das métricas exatamente como o autor delas as define.

Métricas de Escopo de Classe

- NAssocC (*Number of Association per Class*): representa o número total de associações de uma classe.
- HAgg (*Height of a Class*): considerando o diagrama como um grafo, esta métrica indica o maior caminho definido por relacionamentos de agregação, a partir da classe avaliada.
- NODP (*Number of Direct Parts*): total de classes parte, diretas ou indiretas, que integram uma classe composta.
- NP (*Number of Parts*): total de classes parte, diretas ou indiretas, de uma classe todo.
- NW (*Number of Wholes*): total de classes todo, diretas ou indiretas, de uma classe parte.
- MAgg (*Multiple Aggregation*): total de classes todo diretas da qual uma classe é parte-de, dentro de uma hierarquia de agregação.

- NDepIn (*Number of Dependencies In*): total de classes que dependem de uma classe.
- NDepOut (*Number of Dependencies Out*): total de classes que tem classes dependentes.

Métricas de Escopo do Diagrama de Classes

- NAssoc (*Number of Association*): conta o número total de relacionamentos de associação em um diagrama de classes.
- NAgg (*Number of Aggregation*): conta o número total de relacionamentos de agregação em um diagrama de classes.
- NDep (*Number of Dependencies*): conta o número total de relacionamentos de dependência em um diagrama de classes.
- NGen (*Number of Generalization*): conta o número total de relacionamentos de generalização em um diagrama de classes.
- NGenH (*Number of Generalization Hierarchies*): conta o número total de hierarquias de generalização em um diagrama de classes.
- NAggH (*Number of Aggregation Hierarchies*): conta o número total de caminhos compostos por relacionamentos de agregação em um diagrama de classes.
- MaxDIT (*Maximum DIT*): é o número do maior nível de hierarquia no diagrama de classes. O nível de hierarquia de uma classe é dado conforme definido pela métrica DIT do conjunto CK.
- MaxHAgg (*Maximum HAgg*): é o maior caminho definido pela métrica HAgg.

2.2 *Bad Smells*

Para Fowler [1999], *bad smell* é um indicador de possível problema estrutural em código-fonte, que pode ser melhorado via refatoração. Fowler [1999] define o termo *bad smells* e apresenta uma lista com *bad smells* que aborda refatoração em código fonte, mas não usa as estratégias de detecção de *bad smells* para reconhecê-los. Fowler apenas cita características de como identificar as anomalias em código fonte, e deixa em aberto quais metodologias poderiam ser utilizadas para tal tarefa.

O trabalho de Marinescu [2002] é possivelmente um dos primeiros a definir estratégias de detecção de *bad smells* utilizando métricas. Lanza et al. [2006] também definem uma lista de *bad smells*, alguns novos e outros modificados a partir da obra de Fowler [1999], e usam estratégias de detecção para identificação deles. Em seu trabalho sobre refatoração, Hamza et al. [2008] descrevem os *bad smells* das obras de Fowler [1999] e Kerievsky [2005]. Eis as descrições de alguns *bad smells*:

- *God Class*: classes com alta responsabilidade no sistema.
- *Data Class*: classes que não definem funcionalidades próprias.
- *Long Parameter List*: métodos com uma longa lista de parâmetros.
- *Shotgun Surgery*: classes que ao serem alteradas causam impacto em outras classes.
- *Misplaced Class*: classes que se relacionam pouco com as classes de seu pacote, porém se relacionam muito com as classes de outros pacotes.
- *God Package*: pacotes grandes, que possuem muitos dependentes.
- *Divergent Change*: classes que podem mudar diversas vezes e de formas distintas.
- *Data Clumps*: um conjunto considerável de atributos ou parâmetros que andam sempre juntos.
- *Parallel Inheritance Hierarchies*: mostra que toda vez que for feita uma subclasse de uma superclasse, terá que ser feita uma subclasse de outra classe.
- *Indecent Exposure*: quando uma classe está mal encapsulada.
- *Brain Method*: um método que centraliza as funcionalidades de uma classe.
- *Feature Envy*: uma classe que usa os métodos de outra classe em excesso.
- *Intensive Coupling*: métodos de uma classe intensivamente acoplados a métodos de outra classe.
- *Dispersed Coupling*: métodos de uma classe intensivamente acoplados a métodos de várias outras classes.

2.3 Valores Referência

Crespo et al. [1996] afirmam que na literatura a maior parte dos estudos realizados sobre *bad smells* valem-se de métricas para sua identificação e que a relação entre as métricas e *bad smells* é determinada pelos valores referência. Em Marinescu [2002], os valores referência fazem parte dos mecanismos de filtragem. Para determinar os valores referência de métricas que identificam *bad smells*, Marinescu [2002] avaliou 45 softwares escritos em Java e 37 em C++. Para cada uma das métricas foram calculados as médias e os desvios padrões, sendo identificadas três faixas de valores referência para cada métrica:

- Acima de = média + desvio padrão. Corresponde a valores ligeiramente acima do aceitável.
- Abaixo de = média – desvio padrão. Identificam valores abaixo do esperado.
- Muito acima de = (média + desvio padrão) * 1,5. Identificam valores muito acima do esperado.

Crespo et al. [1996] relatam que em estudos anteriores utilizaram-se métricas para identificação de *bad smells* em códigos fonte. Os autores relatam a necessidade de se avaliar melhor a definição de valores referência para as métricas. Nos resultados do estudo de caso apresentado, Crespo et al. [1996] notam que os valores referência variam de acordo com o contexto dos softwares avaliados. Softwares de contextos similares obtiveram valores referência similares e em contextos diferentes os valores referência foram mais distantes.

As métricas e valores referência utilizados nesta dissertação foram definidos por Ferreira [2011] e Ferreira et al. [2012] que identifica valores referência para 6 métricas de software orientado por objetos: NCA (número de conexões aferentes), NAP (número de atributos públicos), NMP (número de métodos públicos), DIT (*depth in inheritance tree*), LCOM (*lack of cohesion in methods*) e COF (fator de acoplamento). O estudo de Ferreira et al. [2012] conclui que a essas métricas, exceto DIT, seguem uma distribuição de cauda pesada (*heavy-tailed distribution*), o que significa que não há um valor típico para essas métricas. Esse estudo foi realizado em um conjunto de 40 softwares abertos, de diferentes contextos e tamanhos. Os valores referência identificados estão representados na Tabela 2.1. O valor *bom* representa os valores mais frequentes, em softwares tidos como de boa qualidade, para as métricas avaliadas, *regular* para valores pouco frequentes e *ruim* para valores raramente frequentes.

Métrica	*Bom*	*Regular*	*Ruim*
COF	até 0.02	0.02 a 0.14	superior a 0.14
NCA	até 1	2 a 20	superior a 20
NAP	0	1 a 10	superior a 10
NMP	até 10	11 a 40	superior a 40
DIT	média igual a 2		
LCOM	0	1 a 20	superior a 20

Tabela 2.1. Valores referência definidos por Ferreira et al. [2012].

2.4 Conclusão

Neste capítulo foi apresentado o conceito de métricas e foi feito um levantamento das principais métricas de diagramas de classes encontradas na literatura. As métricas permitem quantificar a qualidade de um software, por isso é importante o conhecimento delas.

Também foi apresentado o conceito de *bad smells*, que indica possíveis desvios de projeto em softwares via métricas de softwares. Esse conceito é muito importante, e é o principal ponto deste trabalho de dissertação.

Para que as métricas possam identificar os *bad smells*, é necessário identificar algo que os relacione. Esse relacionamento é possível via valores referência, que definem valores numéricos que determinam quantitativamente se o valor de uma métrica é aceitável.

No Capítulo 3 são apresentados os trabalhos que influenciaram e que estão mais relacionados com o trabalho de pesquisa desenvolvido nesta dissertação.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são abordados trabalhos relacionados a *bad smells*, com foco principalmente em modelos UML e nesse sentido são apresentadas ferramentas desenvolvidas para coleta de métricas orientadas por objetos nestes modelos. Posteriormente descreve-se o trabalho de Marinescu [2002], um dos trabalhos pioneiros que utiliza métricas de software para identificação de *bad smells*, assim como a ferramenta iPlasma proposta por Marinescu et al. [2005]. Discute-se sobre os estudos realizados por Marinescu [2002] e Ferreira et al. [2012] sobre valores referência, e, na sequência, apresenta-se o trabalho de D'Ambros et al. [2010] que relaciona falhas de software com *bad smells*. Por fim, mas não menos importante, destaca-se o trabalho de Bertran [2009] que realiza experimentos de identificação de *bad smells* em diagramas de classes. Este capítulo termina com uma análise crítica dos trabalhos relacionados ao tema desta dissertação.

3.1 Ferramentas de Medição Aplicadas a Modelos UML

Na literatura há relato de ferramentas propostas para extraírem métricas de modelos da UML, na maioria dos casos, de diagrama de classes. Algumas delas são descritas a seguir.

Girgis et al. [2009] apresenta uma ferramenta capaz de calcular automaticamente diversas métricas relacionadas ao diagrama de classes. O objetivo dos autores é que a partir delas seja possível tomar decisões acerca de um projeto de software. Eles dividem as métricas em quatro grupos: métricas de complexidade, métricas de tamanho de software, métricas de acoplamento e métrica de herança, todas extraídas de diagramas de classes. A ferramenta recebe como entrada um diagrama de classe, porém no formato

de uma *Extensible Markup Language* (XML) chamado XMI, que é a sigla de XML *Metadata Interchange*[1]. A ferramenta coleta as seguintes informações para as classes do modelo UML: atributos (nome, tipo e visibilidade), métodos (nome, tipo, parâmetros e visibilidade) e relacionamentos (quantidades de agregações, composições, associações e heranças). As métricas são geradas a partir desses dados e os resultados são exibidos para o usuário. Para avaliar o trabalho, Girgis et al. [2009] realizaram um estudo de caso em que são coletadas métricas de três versões de um sistema em Java de código aberto. Os autores apresentam as alterações do software em dados quantitativos e concluem que o comportamento das métricas permite avaliar a evolução do software em diversos aspectos: avaliação da qualidade do projeto, melhor visualização do sistema, detecção de falhas dos padrões de projetos e detecção da necessidade de refatorações.

Soliman et al. [2010] apresentam uma ferramenta que coleta seis métricas CK. Essas métricas são extraídas de diagramas da UML, que são representados em arquivos do tipo XMI. A justificativa dos autores para escolherem as métricas CK é que além delas cobrirem todos os aspectos de modelos orientados por objetos, as mesmas são amplamente aceitas e são adotadas pela Object Constraint Language (OCL) (Fowler [2005]), linguagem que define parte do padrão da UML. Para avaliar a ferramenta desenvolvida, os autores realizaram dois estudos de caso. No primeiro, eles escolhem um modelo de um software fictício de matrícula em cursos *online*, fornecido no sítio da Microsoft [2013], que disponibiliza os diagramas UML com o objetivo de fazer uma demonstração da linguagem UML. Os três diagramas usados foram: diagrama de classes, diagrama de sequência e diagrama de atividades. Os autores coletaram as 6 métricas de cada classe e forneceram o valor médio de cada uma delas, comparando com o maior e menor valor obtido para cada métrica. O segundo estudo de caso foi realizado em um sistema de código aberto chamado Midas, que é apresentado no próprio trabalho de Soliman et al. [2010]. Nesse estudo, os diagramas não são fornecidos pelo criador do sistema. Os autores utilizam a ferramenta de modelagem Argo UML (ArgoUML [2013]) e realizam uma engenharia reversa no código-fonte para obter os diagramas.

O trabalho de coletar métricas feito por Nuthakki et al. [2011] se diferencia dos outros trabalhos uma vez que a ferramenta proposta por eles, UXSOM, é capaz de extrair métricas em diferentes formatos de arquivos gerados pelas ferramentas de modelagem. UXSOM suporta diagramas de classes exportados para os formatos XML, UXF (UML *exchange format*) e XMI, e, utiliza os arquivos de representação de diagramas gerados pelas seguintes ferramentas: ArgoUML [2013], UMLet [2013], ESS-Model

[1] A justificativa do autor para utilizar este formato é que este é o padrão adotado pela *Object Management Group*, também conhecida como OMG (OMG [2012]), a maior organização internacional que aprova padrões abertos para aplicações orientadas por objetos.

[2013], MagicDraw [2013] e Enterprise-Architect [2013]. Para cada arquivo são extraídas as seguintes informações: formato do arquivo do diagrama, quantidade de métodos públicos e privados , quantidade de métodos públicos, quantidade de atributos públicos e privados, quantidade de atributos públicos, quantidade de associações, quantidade de filhos diretos, quantidade de classificadores e pacotes, e quantidade de relações entre classificadores e pacotes. O mesmo diagrama de classes é gerado para cada ferramenta e o mesmo é exportado para algum dos formatos citados - XML, UXF ou XMI. Os autores, então, comparam os valores obtidos e observam que apesar de algumas diferenças, os valores são bem próximos.

3.2 Estratégias de Detecção de *Bad Smells*

Esta seção apresenta regras para estratégias de detecção definidas por Marinescu [2002] e reapresentadas detalhadamente em Marinescu [2004]. Estratégia de detecção nesse contexto é definida como *"uma expressão quantificável de uma regra, aonde podemos verificar se fragmentos do código fonte estão em conformidade com esta regra definida"* (Bertran [2009]). Os aspectos abordados por essas estratégias de detecção de *bad smells* são: mecanismos de filtragem de métricas e mecanismos de composição.

3.2.1 Mecanismos de Filtragem de Métricas

A filtragem possibilita a redução do conjunto inicial de dados, de modo que se possa verificar uma característica especial de fragmentos que têm suas métricas capturadas. Os filtros permitem definir limites de valores para determinados aspectos. A Figura 3.1 mostra os tipos de filtros e os divide em dois grupos:

- Filtro marginal: é dividido em dois sub-filtros denominados semânticos (variáveis) e estáticos (invariáveis).
- Filtro de intervalo: define um limite inferior e outro superior para o conjunto de dados.

Marinescu [2002] e Marinescu [2004] apresentam um conjunto de regras com o objetivo de orientar o engenheiro de software a decidir qual tipo de filtro de dados deve ser aplicado sobre os resultados de uma métrica em particular:

- Regra 1: escolher um filtro absoluto quando for quantificar regras de projeto que especificam limites explicitamente concretos dos valores.

Type of Data Filter	Limit Specifiers		Filter Example
Marginal	Semantical	*Relative*	▪ `TopValues(10)`, ▪ `BottomValues(5%)`
		Absolute	▪ `HigherThan(20)` ▪ `LowerThan(6)`
	Statistical		▪ `Box-Plot`
Type of Data Filter	**Specification**		**Filter Example**
Interval	Composition of two marginal filters, with semantical limit specifiers of opposite polarities		`Between(20,30) := HigherThan(20) ∧ LowerThan(30)`

Figura 3.1. Filtros de dados. Fonte: Marinescu [2004]

- Regra 2: escolher um filtro semântico quando a regra do projeto for definida em termos de valores difusos marginais, como *acima* ou *abaixo* de valores ou em valores extremos.
- Regra 3: para grandes sistemas, utilizar os filtros do tipo marginal, mais especificamente, os semânticos. Para sistemas menores, utilizar intervalos.
- Regra 4: escolher um filtro estático para os casos em que as regras de projeto façam referência a valores extremamente alto / baixo, sem especificar qualquer limite preciso.

Em relação aos mecanismos de filtragem para métricas, Lanza et al. [2006] os definem como: *muito* (valores altos), *pouco* (valores baixos), *metade* de algum valor, não utilizando valores numéricos, pois acreditam que eles dependem do contexto. As estratégias de detecção são definidas pelos autores no formato de circuito lógico.

3.2.2 Mecanismos de Composição

Os mecanismos de composição permitem utilizar as métricas em operações matemáticas, possibilitando assim a mescla de métricas para obter informações relevantes. Marinescu [2002] e Marinescu [2004] definem dois tipos de mecanismos de composição:

- Ponto de vista lógico: permite aplicar operadores lógicos às métricas.
- Ponto de vista de conjuntos: permite agrupar as métricas em conjuntos.

Marinescu [2004] utiliza-se do *bad smell* denominado *God Class*, para exemplificar o processo de formulação das estratégias de detecção de *bad smells*. Na Figura 3.2 é possível ver o exemplo da definição da estratégia de detecção do *God Class*.

Primeiramente, as regras definidas para *God Class* tratam de encontrar uma classe com alta complexidade, baixa coesão e um alto nível de acoplamento. No trabalho de

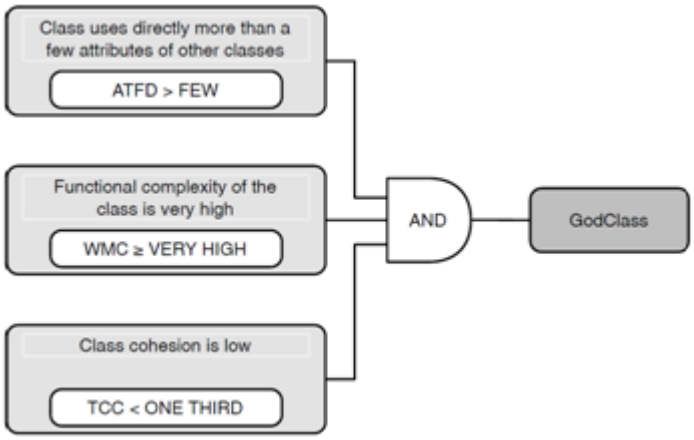

Figura 3.2. Estratégia de detecção do *God Class*. Fonte: Marinescu [2002]

Marinescu [2002] e Marinescu [2004], foram usadas as seguintes métricas, também usadas por Lanza et al. [2006], para identificar *God Class*:

- *Weighted Method Count* (WMC): soma de complexidade dos métodos.
- *Tight Class Cohesion* (TCC): representa o número de métodos conectados diretamente, ou seja, conta o número relativo de métodos de uma classe que acessam pelo menos um atributo em comum.
- *Access to Foreign Data* (ATFD): número de acessos a atributos de classes externas.

O *God Class* foi usado apenas como um exemplo para que Marinescu apresentasse passo-a-passo a contrução de uma estratégia de detecção de *bad smells*. A partir daí foram apresentadas outras estratégias de detecção para outros *bad smells*.

3.2.3 Avaliação das Estratégias de Detecção

Para avaliar os resultados de sua proposta, Marinescu [2002] destaca os seguintes critérios:

Dentre as estratégias de detecção para identificar *bad smells* em código fonte destacam-se os seguintes critérios usados por Marinescu [2002] para avaliar os resultados de sua proposta:

- Escalabilidade: verifica se problemas em sistemas de grande escala foram identificados.
- Precisão: verifica se os resultados estão de acordo ou dentro do esperado.

Suspeito em V1 ?	Suspeito em V2 ?	Conclusão
SIM	SIM	Falso positivo
SIM	NÃO	Falha real
NÃO	NÃO	Caso especial

Tabela 3.1. Tabela de decisão usada para classificar os suspeitos das duas versões do sistema analisado. Fonte: Marinescu [2002].

A validação das estratégias de detecção definidas por Marinescu [2002] foi baseada nos seguintes itens:

- estratégias de detecção devem ser um mecanismo de mapeamento entre os resultados de medição e os problemas de projeto;
- estratégias de detecção devem identificar corretamente os problemas;
- estratégias de detecção devem ser capazes de localizar o problema.

Apesar de sutil, existe uma diferença entre o primeiro e o último item. O primeiro utiliza a estratégia de detecção como mecanismo de interpretação em alto nível para resultados de métricas que conduz para fragmentos de falhas de projeto, enquanto o último requer a estratégia de detecção para identificar especificamente a falha que pretende detectar.

Foram usados dois métodos para avaliar escalabilidade e precisão: um automático e outro manual. Marinescu [2002] realizou esse experimento em um software com aproximadamente 152 classes e outro com 387 classes para avaliar se é possível identificar automaticamente *bad smells* em códigos fonte. Os dois softwares representam o mesmo sistema, porém em versões distintas, Versão V1 e Versão V2, tal que a Versão V2 representa a refatoração da Versão V1. Isso significa que se a metodologia de Marinescu [2002] apontar a presença de mais *bad smells* em V1 que em V2, está comprovada sua eficácia, pois como o V2 é uma versão arquiteturalmente melhorada de V1, obviamente V1 apresentará mais falhas de projeto.

Na avaliação automática, foram avaliados os resultados da primeira e segunda versões do sistema em questão, respectivamente V1 e V2. Na Tabela 3.1 é possível ver o modelo adotado por Marinescu [2002] para avaliar automaticamente as estratégias de detecção. A avaliação ocorreu da seguinte forma:

- caso o problema exista na primeira versão e na segunda, isso significa que ou a estratégia não detectou a falha ou a reengenharia não foi feita para tal aspecto. Nesse caso Marinescu [2002] considerou o item como falso positivo;

Bad Smell	V1	V2	FP	CE	FR	Precisão
Feature Envy	40	15	11	4	25	63%
God Method	4	4	1	3	3	75%
Shotgun Surgery	15	7	6	1	9	60%
Refused Bequest	22	6	4	2	18	81%
God Class	5	2	2	0	3	60%
Data Class	3	2	1	1	2	66%
God Package	2	1	1	0	1	50%
Misplaced Class	4	2	1	1	3	75%
Wide Subsystem Interface	5	1	1	0	4	80%
Lack of Bridge	0	0	0	0	0	-
Lack of Strategy	4	2	2	0	2	50%
Lack of Singleton	4	5	2	3	2	50%

Tabela 3.2. Resultado da avaliação automática de Marinescu [2002]. FP = Falsos Positivos, CE = Casos Especiais, FR = Falhas Reais.

- caso o problema exista na primeira versão, mas não na segunda, significa que a refatoração ocorreu com sucesso, ou que houve alguma mudança na estrutura de pacotes que mascarou a falha. Nesse caso Marinescu [2002] considerou o item como *falha real*;
- caso o problema exista na segunda versão, mas não na primeira, significa que o caso deve ser avaliado separadamente.

Os resultados dessa análise são apresentados na Tabela 3.2. Na coluna *Precisão* é possível ver o critério de avaliação precisão. Para obter esse valor, o número de *falhas reais* é dividido pela quantidade de suspeitos na Versão 1 do sistema.

Por considerar a avaliação automática como cega, o autor optou por realizar inspeções manuais nos resultados da avaliação automática. Foram utilizados os critérios de escalabilidade e precisão em fragmentos suspeitos (*falhas reais*) e feita a avaliação manual em cada um desses trechos. Após a inspeção manual, os suspeitos foram classificados em três categorias:

- detecção correta: quando a avaliação manual julga que o elemento realmente possui o problema do *bad smell* avaliado;
- outro *bad smell*: ocorre quando o *bad smell* avaliado não é detectado, porém outro *bad smell* é detectado;
- falso positivo: ocorre quando a avaliação manual julga que o suspeito não possui problema algum.

Bad Smell	Suspeitos	DC	OB	FP	PE	PS
God Method	4	2	1	1	50%	75%
Shotgun Surgery	15	10	2	3	66%	80%
God Class	5	3	1	0	80%	100%
Data Class	3	3	0	0	100%	100%
Wide Subsystem Interface	5	3	1	1	60%	80%
Lack of Strategy	4	3	1	0	75%	100%
Lack of Singleton	4	1	1	2	25%	50%

Tabela 3.3. Resultado de avaliação manual de Marinescu [2002]. DC = Detecção Correta, OB = Outro *Bad Smell*, PE = Precisão Estrita, PS = Precisão Solta.

Na avaliação manual, Marinescu [2002] utiliza dois tipos de precisão para quantificar a eficácia da estratégia de detecção.

- Estrita (*strict*): considera apenas o *bad smell* avaliado, ou seja, apenas a detecção correta.
- Solta (*loose*): considera o *bad smell* avaliado mais outros presentes no item avaliado, ou seja, detecção correta mais outro *bad smell*.

Os resultados dessa análise realizada por Marinescu [2002] podem ser vistos na Tabela 3.3. Seus resultados mostram que é possível identificar automaticamente *bad smells* em código fonte. Percebe-se que a quantidade de falso positivo é inferior a quantidade de *falhas reais* encontradas, indicando a eficácia do método. A metodologia definida por Marinescu é de grande importância para avaliar estratégias de detecção.

Posteriormente, Marinescu et al. [2005] definem uma ferramenta, iPlasma [2013], para automatizar algumas de suas estratégias de detecção para *bad smells*. A ferramenta foi feita para detectar *bad smells* em código fonte. A Figura 3.3 mostra a estrutura de iPlasma.

A ferramenta iPlasma é definida em quatro camadas. A primeira camada, denominada *Model Extractors*, é a camada responsável por realizar a análise sintática do código fonte, seja o código em C++ ou Java. As informações extraídas do código fonte pelo analisador sintático são importantes para construir uma nova camada denominada *Models*, responsável por organizar e armazenar todas as informações extraídas do código fonte. Essa camada é utilizada pela terceira camada que se chama *Analyses*. A camada *Analyses* é responsável por gerar as métricas, definir as estratégias de detecção para identificação dos *bad smells* e definir os modelos de qualidade. A última camada refere-se ao *Front-end* que faz a interface entre o sistema e o usuário; é ela que fornece as informações para o engenheiro de software.

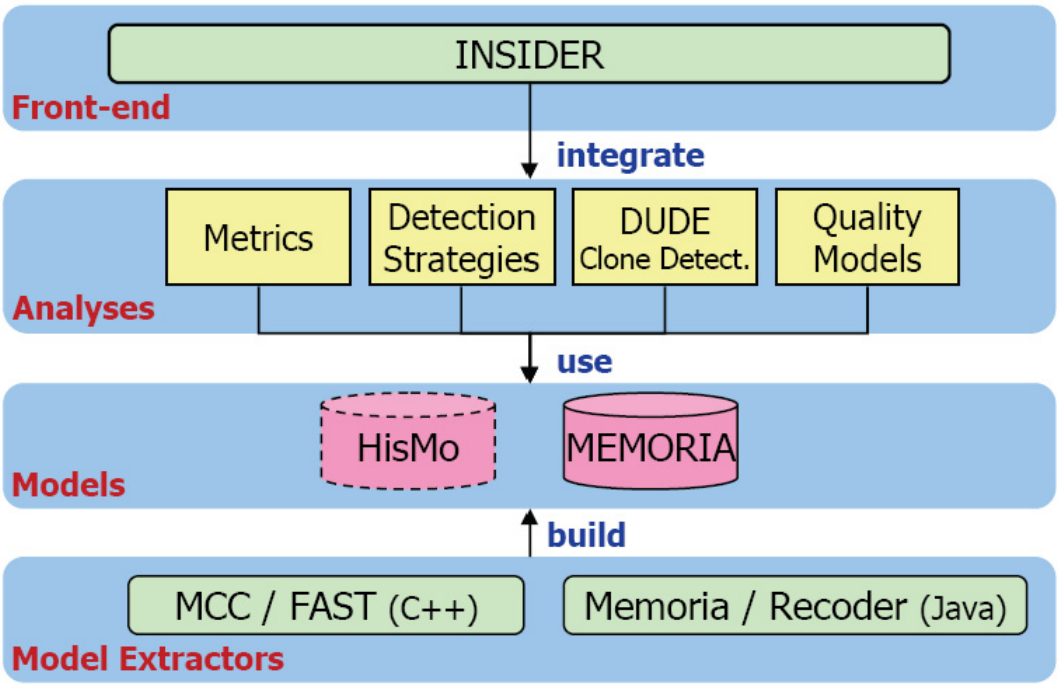

Figura 3.3. Estrutura do iPlasma. Fonte: Marinescu et al. [2005]

3.3 Falhas de Software e *Bad Smells*

Sommerville [2007] define falha de software como "incapacidade do software de realizar a função requisitada (aspecto externo)". D'Ambros et al. [2010] propõe relacionar *bad smells* com falhas de software. Os autores afirmam que 90% dos problemas de software estão relacionados a manutenção e que métricas somente serão efetivas se forem combinadas formando estratégias de detecção. D'Ambros et al. [2010] analisam métricas e dados nas ocorrências de falhas em seis softwares: Lucene, Maven, Mina, Eclipse CDT, Eclipse PDE UI e Equinox, que são das comunidades do Apache e do Eclipse. Essa escolha se deu devido ao fato dos softwares pertencerem a diferentes contextos e por terem muitas pessoas envolvidas no seu desenvolvimento.

Para a realização do trabalho, foi avaliada a frequência de *bad smells* nos softwares. Também foi avaliada a correlação entre os *bad smells* e falhas de softwares em várias versões deles. O objetivo dessa análise foi avaliar se os *bad smells* induzem às falhas de software.

As estratégias de detecção utilizadas para identificar os *bad smells* usados por D'Ambros et al. [2010] foram propostas por Lanza et al. [2006]. Os *bad smells* avaliados foram, *Shotgun Surgery*, *Brain Method*, *Feature Envy*, *Intensive Coupling* e *Dispersed Coupling* definidos na Seção 2.2.

Os códigos fonte dos softwares avaliados foram coletados em repositórios CVS

disponibilizados pelos desenvolvedores. Para analisar o código fonte, foi utilizado o software Infusion [2012], que realiza uma análise sintática e posteriormente gera um arquivo em uma linguagem independente de meta dados de sistemas orientados por objetos denominado FAMIX. O arquivo FAMIX foi dado como entrada para o *framework* Moose [2013], que informou as estatísticas dos *bad smells* nos softwares avaliados.

D'Ambros et al. [2010] extraíram e relacionaram as falhas com as classes dos sistemas avaliados utilizando o repositório CVS. Foi realizada novamente uma análise sintática de cada sistema gerando um arquivo FAMIX e, a partir dele, foi extraído o nome de cada classe dos sistemas. Foram analisados os *logs* de erros de cada *commit* do repositório e os dados de erros foram relacionados com suas respectivas classes. Para obter os relatórios de *bugs* foram usadas as ferramentas Bugzilla e Jira.

Para avaliar a relação entre *bad smells* e falhas de softwares foram realizados dois experimentos: análise de correlação e análise delta.

- Análise de correlação: analisa a correlação entre *bad smells* e defeitos.
- Análise delta: investiga, em um período de tempo, se o crescimento de *bad smells* acompanha o crescimento de defeitos.

Para os seis softwares foram coletadas múltiplas versões, uma a cada duas semanas. Em seguida foram extraídos os dados relativos a *bad smells* e falhas do software. O número de classes, falhas e *bad smells* varia entre versões. Diferente do *bad smells*, uma mesma falha pode afetar várias classes.

3.3.1 Análise de Correlação

A partir dos dados coletados, D'Ambros et al. [2010] os analisaram para identificar quais *bad smells* são mais recorrentes em um determinado sistema e quais são mais recorrentes em todos os sistemas. Na análise de correlação avaliaram-se duas questões.:

- Os *bad smells* estão relacionados com as falhas de software ?
- Algum *bad smell* está mais presente do que outros em todos os sistemas ?

Os resultados de D'Ambros et al. [2010] concluiram que não existe um *bad smell* predominante em todos os sistemas, e que valores de correlação acima de 0.4 configuram forte correlação de acordo com estudos da literatura. Os autores avaliaram que existem dois tipos de sistemas:

- um em que os *bad smells* não possuem tanta relação com as falhas;

- outro em que um ou poucos *bad smells* possuem forte relação com as falhas.

3.3.2 Análise Delta

O objetivo da análise delta é verificar se o crescimento de *bad smells* acompanha o crescimento das falhas de software. Esse experimento teve como objetivo investigar se:

- o crescimento dos *bad smells* está relacionado as falhas de software;
- há um *bad smell* que influencia falhas mais que os outros;
- a relação do crescimento de falhas acompanha o crescimento dos *bad smells*.

D'Ambros et al. [2010] concluíram que:

- o crescimento da quantidade de *bad smells* está relacionado às falhas de software. Porém, isso não é regra geral para todos os *bad smells* em todos os sistemas;
- não há um *bad smell* que influencie mais no crescimento do número de falhas do que os outros;
- existem softwares em que a correlação entre falhas e *bad smells* é baixa, mas o crescimento de um acompanha o outro ao longo das versões;
- existem softwares que falhas e *bad smells* não crescem juntos ao longo das versões, mas a correlação entre eles é alta.

Eles também concluíram que em alguns sistemas, no primeiro experimento observa-se que um *bad smell* F é predominantemente mais correlacionado com as falhas ao longo das versões. Porém no segundo experimento seu crescimento não acompanha o número de falhas no sistema. Uma possível explicação para isso é que um sistema possui por natureza o *bad smell F*, porém os desenvolvedores sabem lidar com esse problema. Também existem aqueles *bad smells* que possuem baixa correlação com as falhas de um sistema, porém um acréscimo desse *bad smell* causa falhas no sistema. Isso corrobora com a explicação anterior, pois como o *bad smell* não está na natureza do projeto, uma explicação possível é que qualquer acréscimo nele causa impacto nas falhas.

D'Ambros et al. [2010] afirmam que uma das limitações de seu trabalho está relacionado com o fato dos valores referência utilizados, extraídos dos estudos de Marinescu [2002], serem estatisticamente avaliados de um conjunto de sistemas.

Com os resultados dos experimentos, D'Ambros et al. [2010] concluiram afirmando que os *bad smells* possuem correlação com as falhas de software, que existem *bad smells* mais frequentes, mas que não existe um *bad smell* que seja mais correlacionado com falhas do que os outros.

3.4 *Bad Smells* em Modelos UML

A maioria dos trabalhos desenvolvidos até hoje estão relacionados à detecção de *bad smells* extraídos exclusivamente do código fonte. Embora detectar desvios de projeto em código fonte seja útil para evitar a degradação do software, é necessário também que existam meios para se identificar problemas desde o início do ciclo de vida do sistema.

Nesse sentido, o trabalho de Bertran [2009] define estratégias de detecção que podem ser aplicadas a diagramas UML. O propósito do trabalho é definir técnicas que encontrem os mesmos elementos problemáticos do projeto, quando extraídos do código fonte, mas que sejam aplicados a diagrama de classes. No trabalho de Bertran [2009] foram considerados seis *bad smells*: *God Class*, *Data Class*, *Long List Parameter*, *Shotgun Surgery*, *Misplaced Class* e *Shotgun Surgery*.

Bertran [2009] desenvolveu uma ferramenta, denominada QCDTool, com o objetivo de automatizar o controle de qualidade em modelos UML, ou seja, a aplicação automatizada das estratégias propostas e dos modelos de qualidade. Tal ferramenta foi utilizada em dois estudos experimentais. O primeiro estudo teve por objetivo verificar se os valores das métricas obtidos em diagramas de classes correspondiam àqueles obtidos em código fonte. Para isso, foram extraídos dados do código fonte e do diagrama de classes de 9 sistemas, e os valores obtidos foram comparados. O segundo estudo propôs a aplicação do modelo de qualidade QMOOD em uma sequência de versões do *framework* JHotdraw com o objetivo de avaliar, via métricas, a degradação do software. Dada a grande relação do trabalho de Bertran [2009] com o tema de pesquisa proposto nesta dissertação, esta seção apresenta o seu trabalho em mais detalhes.

3.4.1 O Modelo de Qualidade QMOOD

A qualidade de um software deve ser definida em termos de atributos de produtos de software que são de interesse do usuário. Utilizar os atributos internos de um software a fim de predizer os atributos externos que são definidos pelo usuário é uma tarefa fundamental. Por causa disso, engenheiros de software definiram na literatura modelos de qualidade para a estimativa dos atributos externos em termos de atributos internos.

Bertran [2009] utilizou o modelo de qualidade QMOOD (*Quality Model for Object-Oriented Design*) (Bansiya & Davis [2002]), que propõe estimar os atributos de qualidade que são de interesse para os usuários. Esse modelo tem um formato hierárquico, no qual os níveis superiores fazem referências aos atributos externos de qualidade de um software, como flexibilidade e facilidade de uso, e os atributos externos são compostos por um ou mais atributos internos de um software. QMOOD é decomposto em quatro níveis como mostra a Figura 3.4.

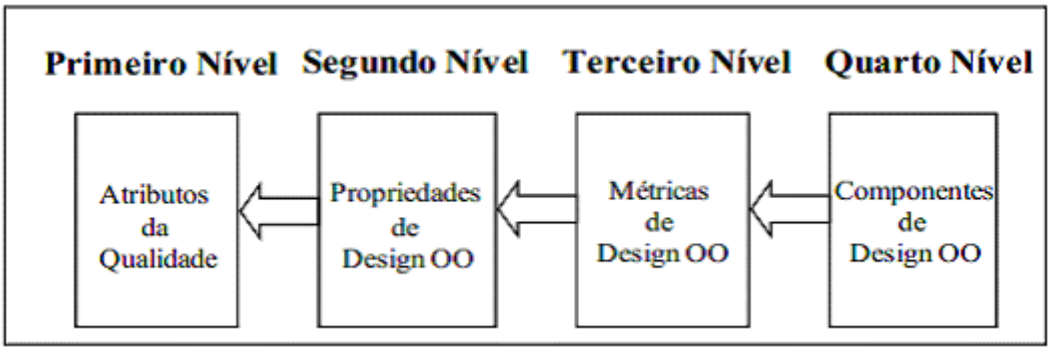

Figura 3.4. Níveis do modelo de qualidade QMOOD. Fonte: Bertran [2009]

Os atributos de qualidade estão relacionados a atributos externos de qualidade, que se baseiam em um conjunto definido pela norma ISO9126. Os atributos utilizados no QMOOD por Bertran [2009] foram: funcionalidade, eficácia, facilidade de compreensão, facilidade de extensão, usabilidade e flexibilidade.

As propriedades do projeto orientado por objetos são avaliadas examinando as classes, métodos e atributos de um modelo, ou seja, os atributos internos. Abordam tamanho, hierarquia, coesão, abstração, acoplamento, herança, polimorfismo, composição e troca de mensagens entre classes. As métricas de projeto orientado por objetos utilizadas no trabalho de Bertran [2009] são mostradas na Figura 3.5.

Os componentes de projeto orientados por objetos são os elementos do diagrama: classes, atributos, métodos, relações como composição e herança. Os componentes de projeto são os responsáveis por fornecer informações para gerar as métricas do modelo. Por sua vez, o conjunto de métricas forma as propriedades de projeto, como por exemplo, os valores de acoplamento e coesão de um modelo. As propriedades, quando aplicadas em fórmulas, geram os valores dos atributos de qualidade, ou seja, os atributos externos de um modelo. Um exemplo de uma fórmula do QMOOD para um dos atributos de qualidade externa, facilidade de extensão, utilizada no trabalho é:

Facilidade de extensão $= 0,25 * coesao - 0,25 * acoplamento + 0,25 * trocademensagens + 0,5 * tamanho$

Métrica	Nome	Descrição
DSC	Número de classes no *design*	Conta o número total de classes no projeto
NOH	Numero de Hierarquias	Conta a quantidade de hierarquias de classes no projeto
ANA	Média de Ancestrais	Representa a média de classes das quais a classe avaliada herda informação. É calculado utilizando o número total de classes na estrutura de herança.
DAM	Métrica de acesso a dados	Calcula a proporção que representam os atributos privados (protegidos) do total de atributos da classe (faixa 0-1)
DCC	Acoplamento direto entre classes	Conta a quantidade de classes com as quais a classe avaliada está direitamente relacionada. Na contagem, são incluídas as declarações de atributos e parâmetros.
CAM	Coesão entre os métodos da classe	Representa o grau de relação entre os métodos de uma classe baseada nos parâmetros dos métodos da classe. Primeiramente é calculado o conjunto máximo independente dos tipos dos parâmetros de todos os métodos. Depois é computada a soma da interseção dos tipos dos parâmetros de cada um dos métodos com o conjunto independente. Finalmente, a soma é divida pelo número total de parâmetros multiplicado pela quantidade de métodos.
MOA	Medida de agregação	Representa a composição de uma classe utilizando os atributos declarados. É computada contando o número de atributos cujo tipo é definido pelo usuário.
MFA	Abstração funcional	Calcula a proporção que representam os métodos herdados do total de métodos da classe (faixa 0-1)
NOP	Numero de métodos polimórficos	Conta a quantidade de métodos que podem ter comportamento polimórfico (e.x. em Java, os métodos não declarados como final).
CIS	Tamanho da interface de uma classe	Conta a quantidade de métodos públicos que a classe possui. Não são considerados nesta métrica os atributos públicos.
NOM	Quantidade de métodos	Conta a quantidade de métodos declarados na classe.

Figura 3.5. Métricas utilizadas no modelo QMOOD. Fonte: Bertran [2009]

No trabalho de Bertran [2009] o modelo QMOOD não foi integrado ao mecanismo de detecção de *bad smells*, foi desenvolvido separadamente dentro do QCDTool, mas a intenção de trabalho futuro da autora é de que utilizando os dois mecanismos de

forma integrada, se terá a estimativa dos atributos externos de qualidade e a detecção/localização de possíveis entidades que causam anomalias de projetos.

3.4.2 Estudos Experimentais

A ferramenta QCDTool, utilizada para realização dos estudos experimentais, foi desenvolvida pela própria Bertran [2009]. QCDTool é responsável por realizar a coleta de dados para identificar os 6 *bad smells* considerados no trabalho e por aplicar o modelo QMOOD, ambos para diagramas de classe.

No primeiro estudo de caso a ferramenta buscou avaliar se os problemas de projeto encontrados em códigos fonte são os mesmos identificados em seu diagrama de classes correspondente. O objetivo foi avaliar a acurácia[2], precisão[3] e o *recall*[4] entre os valores obtidos para código fonte e o diagrama para os 6 *bad smells*. Para isso, foram utilizados nove sistemas, sendo alguns softwares de código aberto e outros sistemas desenvolvidos em trabalhos de cursos acadêmicos. Todos os sistemas foram desenvolvidos na linguagem Java.

Os resultados do estudo de caso mostraram que as estratégias de detecção dos *bad smells Data Class*, *God Class* e *God Package* estavam correlacionadas significativamente com suas correspondentes versões para o código. As estratégias *Long Parameter List* e *Data Class* apresentaram 100% nos graus de *recall*. A *God Package* apresentou 100% de precisão, o que indica que a *Shotgun Surgery* apresentou alto grau de similaridade dos valores entre código fonte e o modelo. Isso significa que todas as entidades que tiveram esses seis problemas de projeto no código fonte foram detectadas pela estratégia proposta para o modelo. Porém, algumas das entidades identificadas como problemáticas no modelo não foram identificadas como tal no código.

O segundo estudo de caso teve como objetivo avaliar a usabilidade do modelo de qualidade QMOOD em diagramas de classe. Este modelo foi aplicado em uma sequência de versões do *framework* JHotDraw [2012]. Foram avaliadas ao total 4 versões do aplicativo. Os atributos de qualidade externos avaliados nos modelos foram: facilidade de compreensão, flexibilidade e reusabilidade. A avaliação desse estudo de caso consistiu em avaliar valores manualmente de uma versão para outra e verificar se a alteração desses valores correspondiam a diferenças significativas na estrutura do software. As métricas consideradas avaliaram os seguintes aspectos: abstração, acoplamento, coesão, complexidade, composição, encapsulamento, polimorfismo, tamanho e troca de mensagens.

[2]Representa o grau de veracidade dos resultados.
[3]Corresponde à relevância dos resultados obtidos.
[4]Representa a habilidade para recuperação dos resultados relevantes de acordo à consulta realizada.

Os resultados da aplicação do modelo QMOOD nos diagramas de classe mostraram que os valores das métricas que avaliam os atributos de qualidade reusabilidade e flexibilidade aumentaram ao longo das quatro versões. Isso aconteceu pelo aumento das funcionalidades ao longo das versões do software. Além disso, o atributo relacionado a qualidade facilidade de compreensão degradou. Isto foi motivado pelo fato que novas funcionalidades e características foram inseridas para aumentar a capacidade do sistema por meio da implementação de novas classes e métodos das classes já existentes.

3.5 Considerações Finais

Os trabalhos de Girgis et al. [2009], Soliman et al. [2010] e Nuthakki et al. [2011] discutem acerca de extração de métricas a partir de modelos UML. Esses trabalhos indicam que o diagrama de classes é o modelo predominante quando o foco é medição de software a partir de modelos UML. Esses trabalhos também utilizam o formato de arquivo XMI para representar as métricas coletadas do diagrama de classes. Porém, nenhum desses três trabalhos relaciona as métricas com atributos externos de um sofware, ou seja, não fornece informações relevantes acerca do projeto a ser desenvolvido.

Os trabalhos de Marinescu [2002] e Marinescu [2004] apresentam uma solução para transformar métricas em informações relevantes para detectar *bad smells*. Porém, esses trabalhos lidam apenas com informações coletadas em códigos fonte, que em geral passam a existir apenas em fases já avançadas do ciclo de vida dos softwares.

O trabalho de Bertran [2009] é o mais próximo deste trabalho de mestrado. Nele, a autora associa métricas extraídas de diagramas de classes a *bad smells*. Para automatizar tal tarefa, a autora desenvolveu uma ferramenta para coleta desses dados. Muito embora o trabalho de Bertran [2009] tenha definido estratégias de detecção que podem ser aplicadas a diagramas UML, algumas questões foram deixadas em aberto: (1) Bertran [2009] relata, por exemplo, a importância de redefinir os valores referência utilizados em seus estudos experimentais, definidos por Marinescu [2002], pois considera que a técnica utilizada por Marinescu [2002] não seja a melhor forma de definí-los. (2) Bertran [2009] relata também que os experimentos foram realizados apenas por uma pessoa e que o contexto e tamanho dos softwares utilizados não diferem tanto. (3) A autora ainda sugere que novos experimentos sejam realizados para identificar novos *bad smells* em outros diagramas da UML.

O trabalho de dissertação apresentado neste documento visa contribuir para a solução do problema de identificar desvios de projeto em fases iniciais do ciclo de vida do sistemas. Especificamente, este trabalho se concentra em definir um método para

identificação de *bad smells* a partir de diagramas de classes, bem como desenvolver uma ferramenta para viabilizar a aplicação e a avaliação do método proposto. Para isso, este trabalho se baseia nos seguintes aspectos:

- definir um método para detecção de *bad smells* em diagramas UML que utilize métricas de software e seus respectivos valores referência propostos na literatura;
- criar e disponibilizar uma ferramenta, *UMLsmell*, que identifique *bad smells* em diagrama de classes;
- definir uma técnica para avaliar o método proposto nessa dissertação que não dependa de código fonte e avalie quantitativa e qualitativamente os resultados;
- avaliar uma quantidade maior de softwares e com tamanhos também maiores do que os utilizados por Marinescu [2002] e Bertran [2009], para avaliar a escalabilidade do método proposto nesta dissertação;
- realizar experimentos manuais com critérios bem definidos e com uma quantidade considerável de avaliadores.

Capítulo 4

Identificação de *Bad Smells* a partir de Diagramas de Classes

Neste capítulo é apresentado o método para identificação de *bad smells* proposto nesta dissertação. Além disso, são descritas as principais funcionalidades, arquitetura e interfaces da ferramenta, *UMLsmell*, que automatiza o método proposto.

4.1 Método Proposto

Diferente de outros métodos propostos na literatura que avaliam a qualidade de software em modelos UML, o método definido nesta dissertação considera valores referência que, associados às métricas de software, permitem identificar as partes problemáticas de um software, ou seja, o método vale-se das métricas de software para definir estratégias de detecção que identifiquem *bad smells*.

Inicialmente, foram identificados, dentre os *bad smells* descritos na literatura, aqueles que podem ser aplicados a modelos de classe da UML. Esses *bad smells* e as métricas que podem ser aplicadas para identificá-los são descritos a seguir.

- *Divergent Change*: este *bad smell* está associado às métricas de relacionamentos do tipo eferentes que permitem avaliar se uma classe pode sofrer alterações decorrente de modificações em outras classes.

- *God Class*: este *bad smell* está associado às métricas que calculam a quantidade de relacionamentos e métricas que calculam o número de métodos da classe avaliada, que permitem avaliar se uma classe possui muitos métodos e se muitas classes dependem da classe avaliada.

- *God Package*: este *bad smell* está associado às métricas de relacionamentos entre pacotes e classes de diferentes pacotes que permitem avaliar se as classes que possuem o maior número de outras classes dependentes (conexões aferentes) concentram-se em poucos pacotes.

- *Indecent Exposure*: este *bad smell* está associado às métricas de atributos públicos que permitem avaliar o encapsulamento das classes.

- *Large Class*: este *bad smell* está associado às métricas de quantidade de métodos que permitem avaliar o tamanho de uma classe.

- *Long List Parameter*: este *bad smell* está associado às métricas de parâmetros dos métodos que permitem avaliar a quantidade de parâmetros de um método.

- *Shotgun Surgery*: este *bad smell* está associado às métricas de relacionamentos do tipo aferentes que permitem avaliar se a alteração em uma classe implicará em alterações em outras classes.

A escolha dos *bad smells* avaliados nesta dissertação foi limitada pela carência de valores referência propostos na literatura. Os valores referência usados foram definidos por Ferreira et al. [2012]. A escolha desses valores referência se deu por duas razões principais: primeiro por que os mesmos consideram a frequência de vários softwares e não simplesmente a média, e segundo por que os valores referência foram avaliados empiricamente, e há poucos valores referência propostos na literatura. Dentre as métricas para as quais a autora propõe valores referência, três delas podem ser obtidas via diagramas de classes:

- Número de Conexões Aferentes (NCA);

- Número de Métodos Públicos (NMP);

- Número de Atributos Públicos (NAP).

Essas três métricas permitem identificar os *bad smells God Class*, *Shotgun Surgery* e *Indecent Exposure*. Os valores referência das métricas de Ferreira et al. [2012] são classificados como:

- *Bom*: *são os valores mais frequentes para métricas em softwares de boa qualidade*;

- *Regular*: *corresponde a valores pouco frequentes para métricas em softwares de boa qualidade*;

- *Ruim*: *corresponde a valores raros para métricas em softwares de boa qualidade.*

A idéia básica por trás das faixas *bom*, *regular* ou *ruim* é a seguinte: uma vez que os valores são frequentes em softwares, isso indica que eles correspondem à prática comum no desenvolvimento de software de alta qualidade, o que serve como um parâmetro de comparação de um software com os demais. Da mesma forma, os valores pouco frequentes indicam situações não usuais na prática, portanto, pontos a serem considerados como críticos. Por esta razão, neste trabalho, optou por considerar as faixas *regular* e *ruim* na identificação de *bad smells*. A seguir, são descritas as estratégias de detecção propostas para esses três *bad smells*.

As métricas que definem a estratégia de detecção do *bad smell God Class* consideram seus relacionamentos com outras classes e o tamanho da classe avaliada. O *God Class* é identificado via métricas NCA e NMP. A métrica NCA verifica a influência da classe avaliada sobre as outras classes do software, ou seja, a quantidade de classes que usam os serviços da classe avaliada. A métrica NMP permite avaliar o tamanho de uma classe e a quantidade de serviços a oferecer pela quantidade de métodos. A estratégia de detecção adotada para identificação do *God Class* pelo método proposto é apresentada na Figura 4.1. Caso as duas métricas avaliadas estejam dentro da faixa *regular*, a avaliação do *God Class* é considerada como *regular* e caso uma das métricas esteja dentro da faixa *ruim* de acordo com os valores referência , o *God Class* é considerado como crítico (*ruim*), caso contrário a classe não é considerada como problemática.

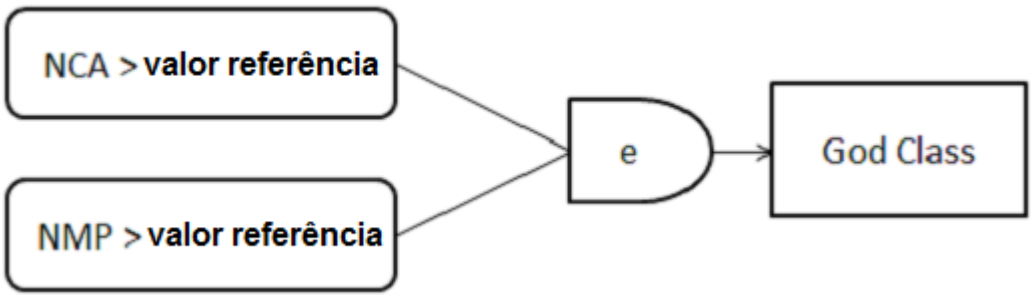

Figura 4.1. Estratégia de detecção para o *God Class*.

O *Shotgun Surgery* é identificado via métrica NCA, que define quantas classes dependem da classe avaliada, uma vez que quando uma classe é alterada, todas as outras classes que dependem dela estão sujeitas a mudanças. A estratégia de detecção para identificação do *Shotgun Surgery* é apresentada na Figura 4.2 e sua identificação em uma classe e em qual faixa se encontra, *regular* ou *ruim*, é equivalente à métrica avaliada.

Figura 4.2. Estratégia de detecção para o *Shotgun Surgery*.

Um bom encapsulamento acontece quando os dados, atributos, de uma classe são ocultos e seus serviços, métodos, úteis para as demais classes, são públicos (Sommerville [2007]). Classes seguras são bem encapsuladas. O *Indecent Exposure* é um *bad smell* referente à falta de encapsulamento de classes. Ele pode ser identificado via métrica NAP, que define quantos atributos de uma classe são públicos. A estratégia de detecção para identificação do *Indecent Exposure* é apresentada na Figura 4.3. A identificação do *Indecent Exposure* em uma classe e em qual faixa se encontra, *regular* ou *ruim*, é equivalente à métrica avaliada.

Figura 4.3. Estratégia de detecção para o *Indecent Exposure*.

4.2 A Ferramenta *UMLsmell*

O objetivo deste trabalho de dissertação é a automatização na identificação de *bad smells* em diagramas de classe da UML. Para atingir este objetivo foi proposto um método baseado em métricas e seus valores referências, bem como foi projetada e implementada uma ferramenta denominada *UMLsmell* que permite a aplicação do método proposto. Esta seção apresenta a sua arquitetura, a metodologia usada para seu desenvolvimento, incluindo as decisões de projeto, as funcionalidades oferecidas e sua implementação.

4.2.1 Requisitos de *UMLsmell*

A primeira tarefa para o desenvolvimento da *UMLsmell* foi a definição de seus casos de uso, ilustrados no diagrama da Figura 4.4 e descritos a seguir.

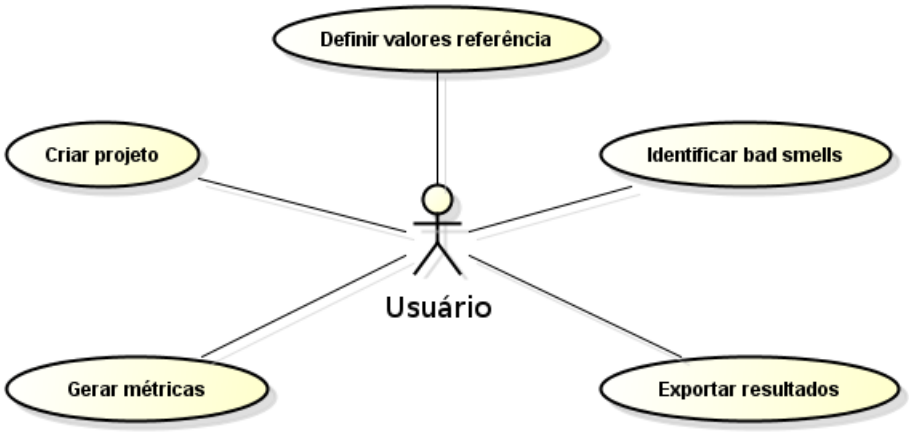

Figura 4.4. Casos de Uso da UMLsmell.

- Definir valores referência: permite que o usuário personalize os valores referência em *regular* e *ruim* para cada métrica coletada pela ferramenta. Esta funcionalidade foi implementada para que o usuário tenha liberdade de escolher os valores referência que preferir utilizar.

- Criar projeto: cria um arquivo que irá armazenar todas alterações feitas nos valores referência. Esta funcionalidade é útil caso o usuário altere os valores referência conforme sua necessidade e pretenda salvá-los para utilizações posteriores da *UMLsmell*.

- Gerar métricas: permite que o usuário possa selecionar, dentre as métricas existentes no sistema, aquelas para as quais ele deseja gerar relatórios para análises posteriores. Esta funcionalidade permite ao usuário verificar os valores das métricas selecionadas para as classes do software.

- Identificar *bad smells*: é o ponto central da ferramenta. Sua funcionalidade consiste em receber como entrada um arquivo XMI, que corresponde ao diagrama de classes a ser analisado, e identificar via métricas, em quais classes foram identificados os *bad smells* e em qual nível – *regular* ou *ruim*, conforme o método para identificação de *bad smells* proposto neste trabalho.

- Exportar resultados: permite que a análise das métricas e dos *bad smells* seja exportada para o formato de planilha, aceito pela maioria das ferramentas de planilhas disponíveis atualmente.

4.2.2 Implementação de *UMLsmell*

Definidas as funcionalidades de UMLsmell, o próximo passo foi definir a sua estrutura. A estrutura de UMLsmell é ilustrada na Figura 4.5.

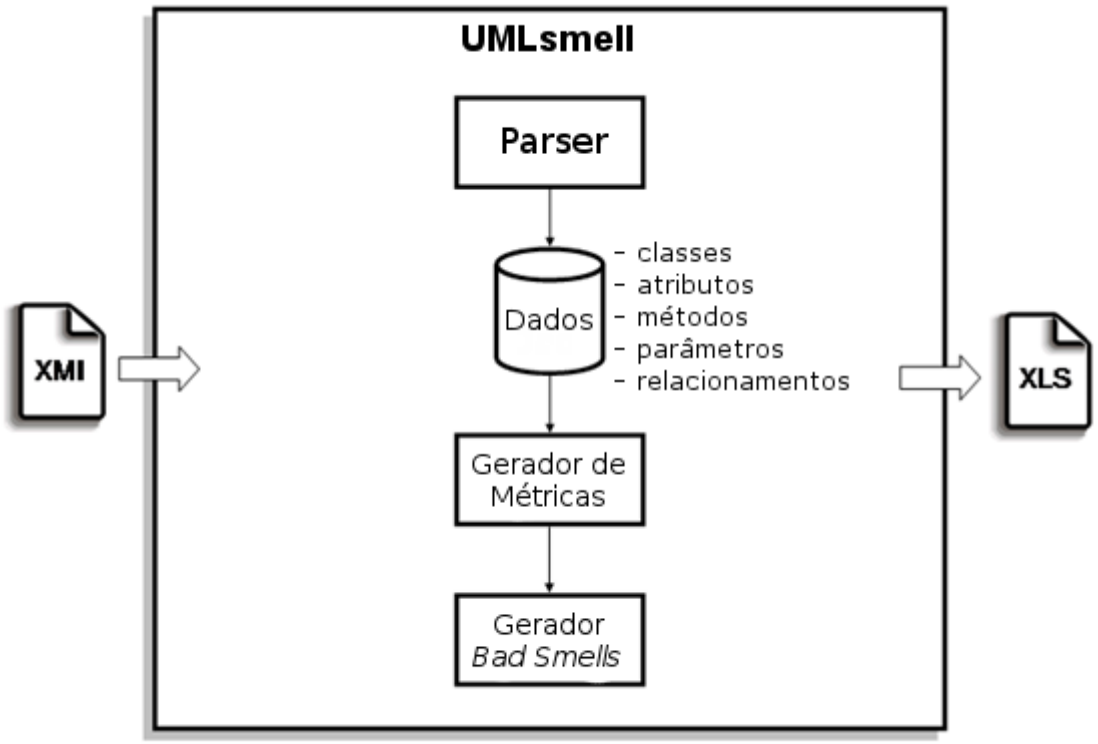

Figura 4.5. Estrutura da UMLsmell.

UMLsmell recebe como entrada um arquivo XMI, no qual é realizada a análise sintática com o objetivo de extrair as seguintes informações: classes, atributos, métodos, parâmetros e relacionamentos. Esses dados são armazenados internamente nas estruturas de dados definidas pelo programa para consultas posteriores, tal que, a partir desses dados a funcionalidade *Gerar métricas* possa fornecer as informações de métricas das classes do software ao usuário. Essas métricas permitem que a funcionalidade *Identificar bad smells* detecte os *bad smells* do software analisado.

A ferramenta foi desenvolvida na linguagem Java e contém 3205 linhas de código. *UMLsmell* foi projetada e desenvolvida por dois programadores, sendo um aluno de mestrado, autor deste trabalho, e outro de iniciação científica. A Figura 4.6 exibe o diagrama de classes, das camadas de controle e modelo, usado para desenvolver a ferramenta. Na camada de modelo foram criadas quatro classes: *ClassData*, *Attribute*,

Method e *Relationship*. Essas classes são responsáveis por armazenar na memória as informações coletadas durante a análise sintática.

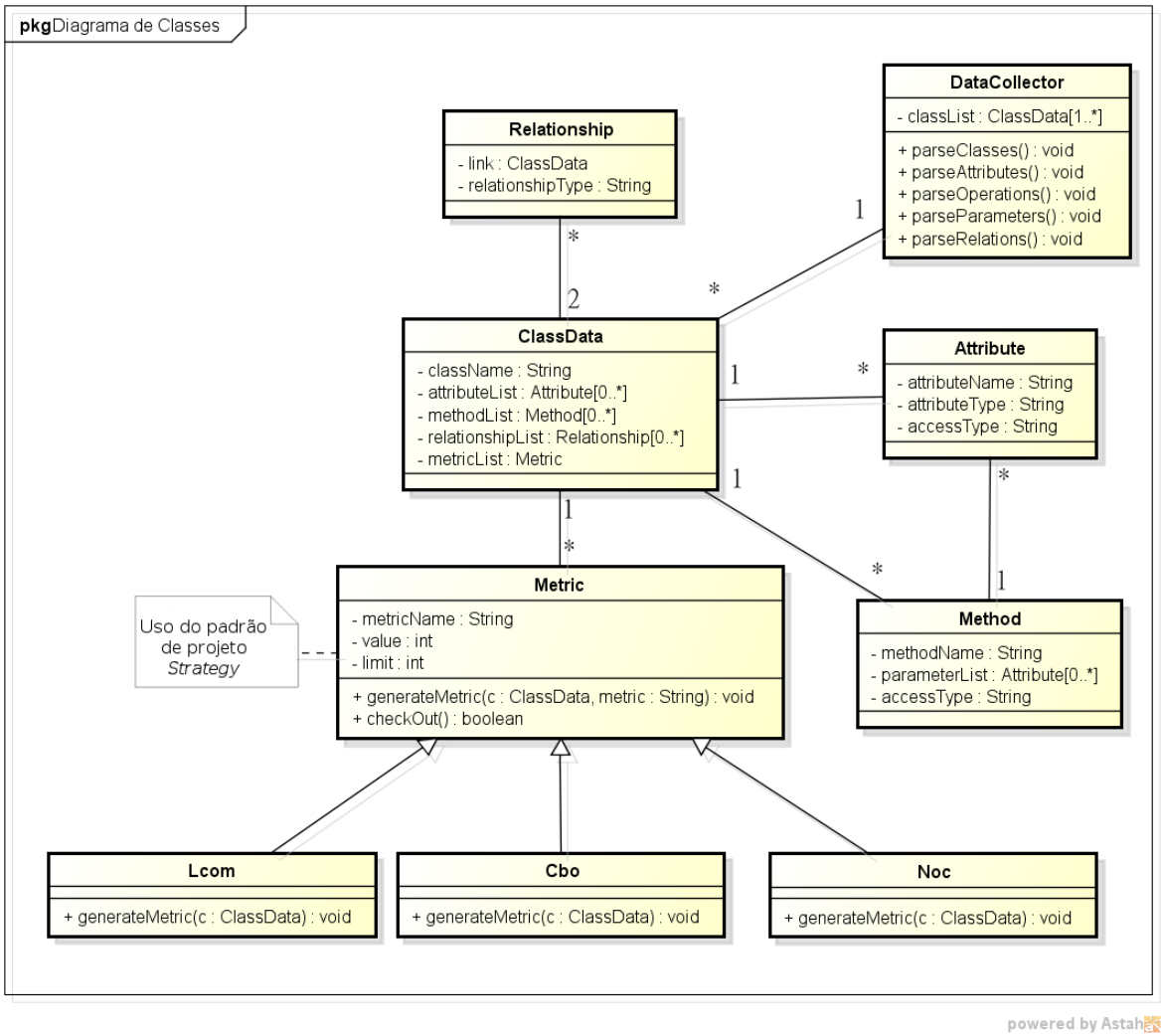

Figura 4.6. Diagrama de Classes da UMLsmell.

- *ClassData*: classe responsável por informações relativas às classes e interfaces do software avaliado. Também mantém vetores com dados sobre atributos, métodos e relacionamentos das classes.
- *Attribute*: classe responsável pelas informações relativas aos atributos do software. Informa o nome de um atributo, tipo e tipo de acesso.
- *Method*: classe responsável pelas informações relacionadas aos métodos do software. Informa o nome do método, seu tipo de acesso e possui uma lista do tipo *Attribute* com os parâmetros de entrada do método.

- *Relationship*: classe responsável pelas informações relativas aos relacionamentos entre as classes do software. Informa as classes nas duas pontas do relacionamento e qual o tipo de relacionamento.

Existe também um conjunto de classes que possui como superclasse a classe Metric, tal que todas as suas subclasses são as métricas existentes no sistema. O padrão de projeto *Strategy*, apresentado por Gamma et al. [2006], foi usado nesta parte da implementação de tal forma que as subclasses herdam as informações gerais das métricas, como nome, valor e valor referência, e, as subclasses, via polimorfismo, implementam o método *generateMetric*, tal que cada subclasse possui sua forma de calcular a respectiva métrica. Esse método permite facilmente criar outras métricas no sistema.

Na camada de controle foi criada uma classe, a *DataCollector*. Essa classe é responsável por ler o arquivo XMI, coletar todas as informações e criar as instâncias da camada de entidade. Essa classe funciona como o analisador sintático do arquivo XMI.

Para coletar os dados necessários para gerar as métricas que permitirão a identificação de *bad smells*, é necessário definir quais são as informações relevantes em um diagrama de classes. Neste trabalho foram usadas as seis informações sugeridas por Genero et al. [2005], apresentadas na Seção 2.1. São elas: pacotes, classes, atributos, métodos, parâmetros e relacionamentos.

As informações extraídas do diagrama de classes são armazenadas em blocos muito bem definidos no XMI. Os blocos do tipo *<element>* representam tanto os pacotes, quanto as classes e interfaces. O que os diferencia é o atributo *"xmi:type"* definido nas *tags* desses blocos.

Fora do bloco *<element>* também existem os blocos *<packagedElement>* que definem todas informações como classes, interface, atributos, métodos, parâmetros e relacionamentos presentes em um pacote.

Dentro do bloco *<element>* existem os blocos *<attribute>*, *<operation>* e *<links>* que fornecem todas informações referentes aos atributos, métodos e relacionamentos do elemento em questão, classe, interface ou pacote. Dentro do bloco *<operation>* existem os blocos *<parameter>* que são os parâmetros do método em questão.

Informações referentes a nomes, níveis de acesso (privado, público e protegido), número de identificação dos elementos e todas outras informações referentes a cada item, encontram-se dentro dos blocos ou em forma de atributo em cada bloco.

4.2.3 Interface com o Usuário

Desenvolvida a estrutura de *UMLsmell*, o próximo passo foi a criação das interfaces gráficas do sistema. São elas:

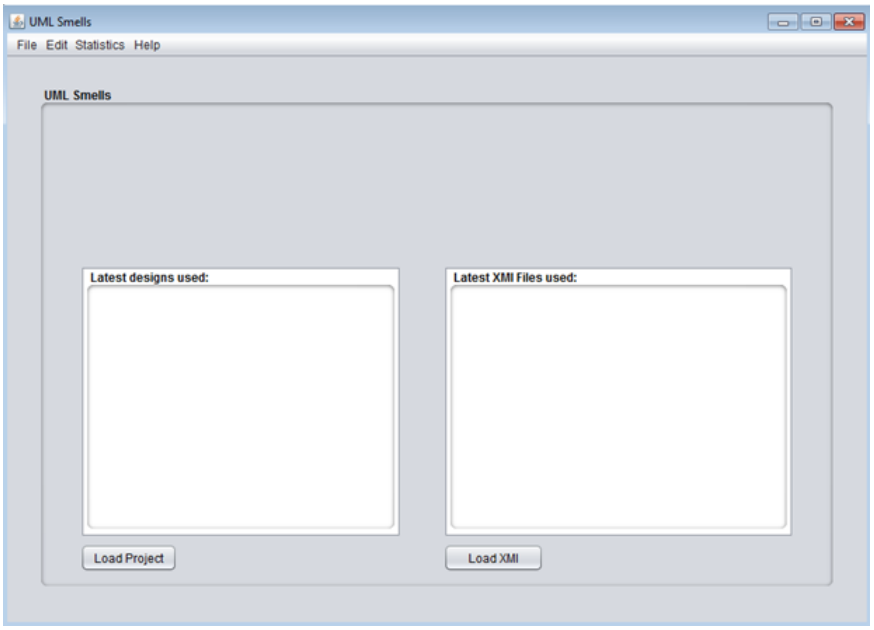

Figura 4.7. Tela com a lista de projetos criados por *UMLsmell*.

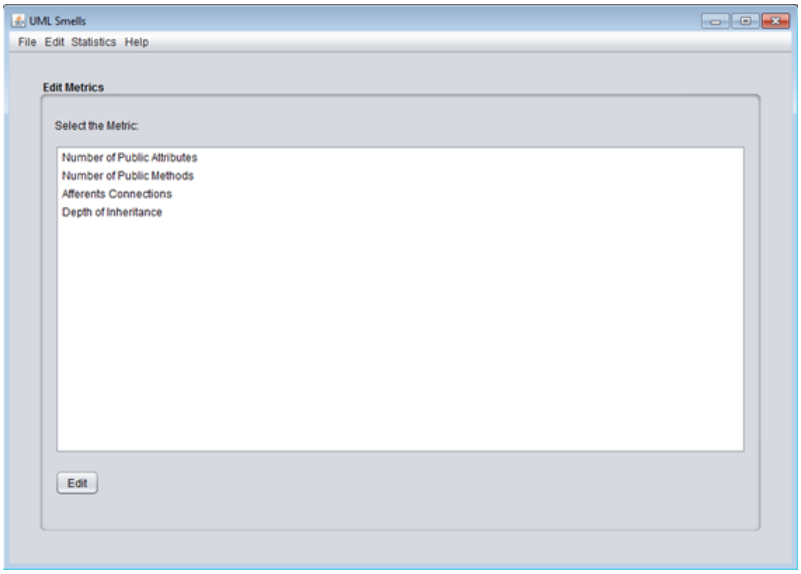

Figura 4.8. Tela para seleção de métricas de *UMLsmell*.

- Tela com a Lista de Projetos Criados: é a tela inicial, que apresenta uma lista

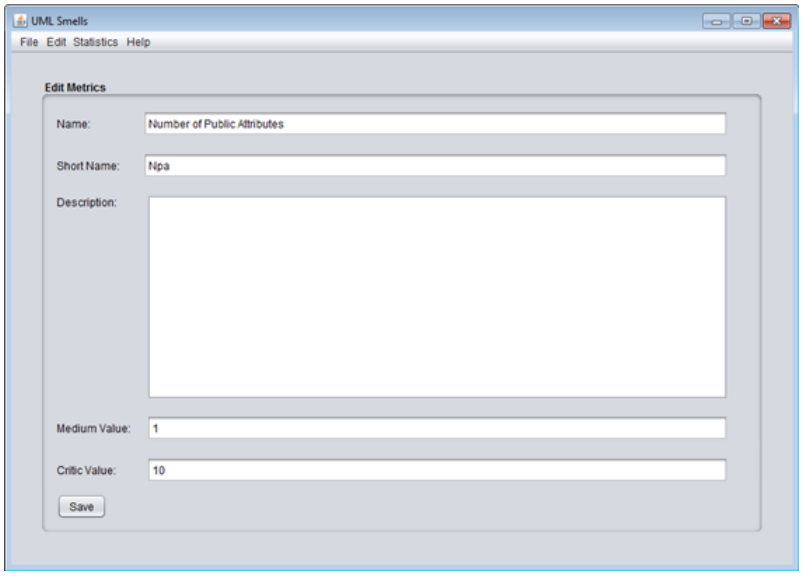

Figura 4.9. Telas para a definição dos valores das métricas de *UMLsmell*.

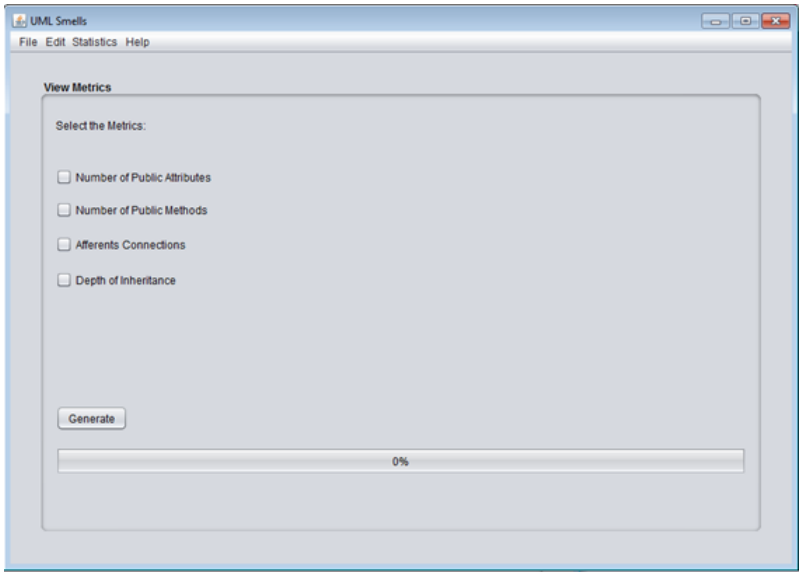

Figura 4.10. Tela para seleção de *métricas* para avaliação em *UMLsmell*.

de todos os projetos já criados para serem carregados. Esta tela é mostrada na Figura 4.7.

- Telas para a Definição dos Valores das Métricas: são duas telas, uma para selecionar uma métrica e outra para definir os valores referência da métrica selecionada, ilustradas nas Figuras 4.8 e 4.9, respectivamente.

- Telas com as Estatísticas das Métricas: são duas telas, uma para selecionar as métricas e outra para exibir os valores das métricas do arquivo XMI carregado,

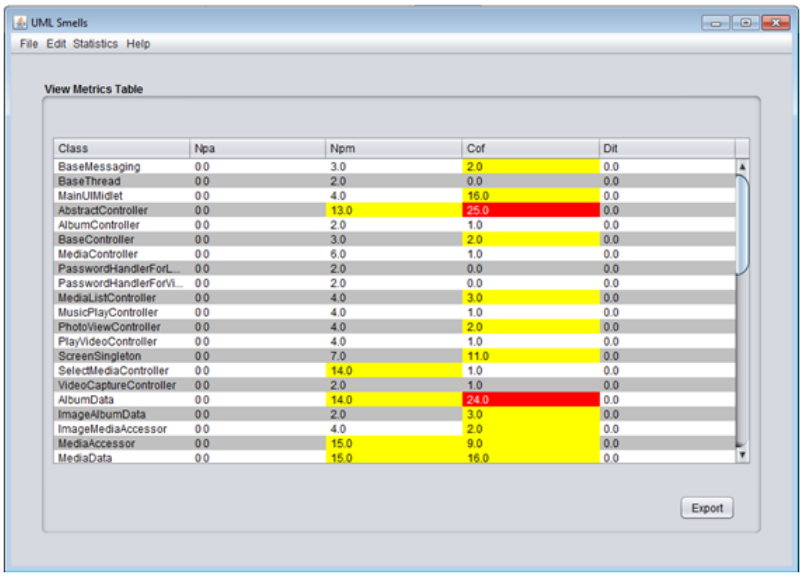

Figura 4.11. Tela com as estatísticas de métricas de *UMLsmell*.

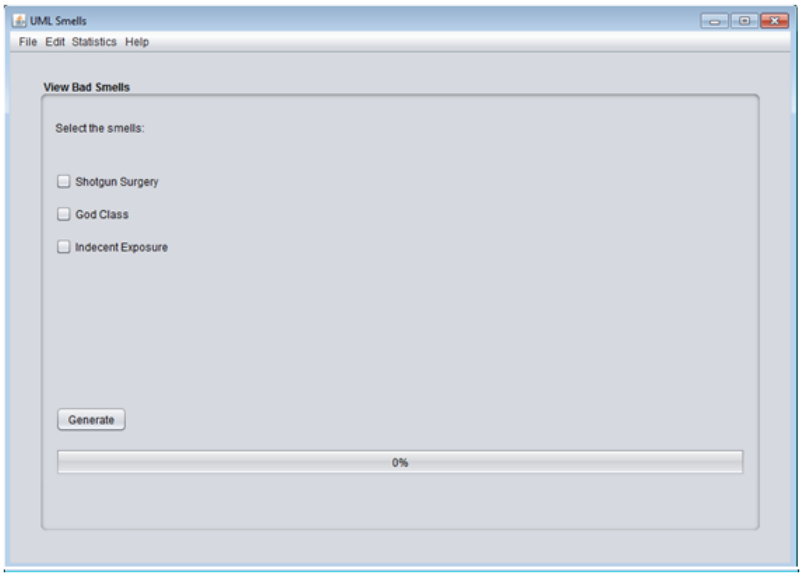

Figura 4.12. Tela para a seleção de *bad smells* para avaliação em *UMLsmell*.

mostradas nas Figuras 4.10 e 4.11, respectivamente.

- Telas com as Estatísticas dos *Bad Smells*: são duas telas, uma para selecionar os *bad smells* a serem avaliados, mostrada na Figura 4.12, e outra com os valores dos *bad smells* para cada classe avaliada, mostrada na Figura 4.13.

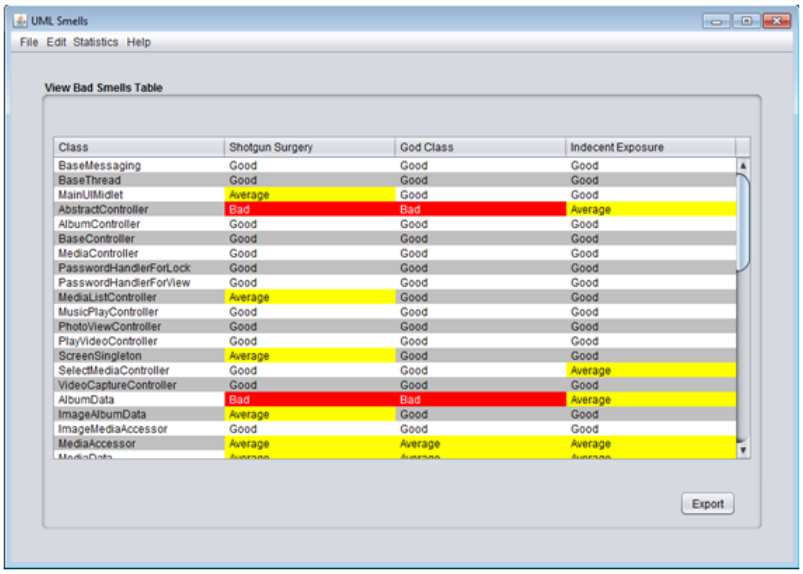

Figura 4.13. Tela para as estatísticas de *bad smells* de *UMLsmell*.

4.3 Diferenciais do Método Proposto

Os principais trabalhos relacionados a esta dissertação são os de Marinescu [2002] e Bertran [2009]. Estes trabalhos possuem algumas limitações que foram exploradas pelo método proposto nesta dissertação.

O primeiro ponto é que tanto Marinescu [2002] quanto Bertran [2009] utilizam valores referências obtidos a partir da média de vários softwares. Essa técnica pode prejudicar os resultados pois parte-se do princípio que os valores das métricas seguem uma distribuição normal. Todavia, a literatura (Ferreira et al. [2012]) tem convergido para os achados de que boa parte das métricas têm valores que seguem uma distribuição de cauda pesada e, consequentemente, a média não é representativa. Neste trabalho de dissertação foram escolhidos valores referência que consideram frequência em vez da média dos valores referência dos softwares.

Diferente dos softwares avaliados por Marinescu [2002] e Bertran [2009], neste trabalho também foram avaliados softwares de diversos tamanhos e contextos.

A avaliação do método proposto é descrita no Capítulo 5. A avaliação realizada também se diferencia daquelas realizadas em trabalho anteriores.

Ao contrário dos trabalhos prévios, o método proposto se baseia na utilização de valores referência de métricas associados à interpretação apropriada para eles, o que pode facilitar o uso do método nas tarefas de refatoração de software.

Outro aspecto comum nos dois trabalhos é que a avaliação manual para identificar erros na avaliação automática envolveu apenas os autores das obras, de tal forma que

os resultados da avaliação manual podem favorecer involuntariamente os resultados da avaliação automática. O método proposto neste capítulo foi avaliado por 15 avaliadores de forma que os resultados automáticos possam ser comparados com o consenso de um grupo consideravelmente grande.

Outro ponto do trabalho de Bertran [2009] é que para validar seus resultados, esses foram comparados aos resultados de código fonte. Porém, existem aspectos no código fonte que não podem ser avaliados em modelos de diagramas de classes. Neste trabalho de dissertação, a avaliação automática dos resultados considerou apenas os diagramas de classes.

Nos experimentos realizados por Bertran [2009], os relacionamentos que não foram gerados automaticamente via engenharia reversa foram criados manualmente. Todavia, a quantidade de relacionamentos não criados pelo Enterprise-Architect [2013] é grande. É portanto possível que por fatores humanos, alguns relacionamentos sejam ignorados. Para evitar esse problema, nesse trabalho de dissertação foi projetada e implementada uma ferramenta para identificar e criar automaticamente os relacionamentos não criados pelo Enterprise-Architect [2013].

Capítulo 5

Avaliação do Método Proposto

Este capítulo relata os experimentos realizados para avaliar o método proposto e os seus respectivos resultados. O objetivo desses experimentos foi avaliar o método para identificação de *bad smells* a partir de diagramas de classes.

5.1 Metodologia

Os experimentos investigam as seguintes questões de pesquisa:

RQ1: as métricas de software e seus respectivos valores referência auxiliam a identificar *bad smells* em um software?

RQ2: os *bad smells* identificados pelo método em um software são correspondentes àqueles identificados por avaliadores com formação em Engenharia de Software?

Para responder essas questões foram realizados os dois conjuntos de experimentos descritos a seguir.

- Análise de softwares reestruturados: este experimento avalia a identificação automática de *bad smells* em duas versões de cada software, tal que uma versão é a refatoração da outra. Tendo em vista que uma refatoração é uma modificação realizada no software para melhorar sua estrutura, espera-se encontrar mais *bad smells* na versão não refatorada, indicando que o método proposto é eficaz no reconhecimento de *bad smells* automaticamente. Esse tipo de análise foi empregada por Marinescu [2002] para avaliar sua proposta, porém ele a aplicou em somente um software. Neste trabalho, são analisados os projetos de sete softwares abertos.

- Avaliação Manual: este experimento avalia manualmente a identificação de *bad smells* e os compara com os resultados da *UMLsmell* a fim de avaliar a eficácia da estratégia de detecção proposta.

Seção 5.2 apresenta a análise de softwares reestruturados e Seção 5.3 a avaliação manual.

5.1.1 Diagramas de Classes dos Experimentos

Foram realizados experimentos a partir dos diagramas de classes dos seguintes softwares abertos desenvolvidos em Java: *JHotdraw*, *Struts*, *HSqlDB*, *JSLP*, *Log4J* e *Sweethome 3D*. A seleção desses softwares deve-se ao fato dos mesmos serem considerados como refatorados, conforme relatado pelos autores Dig & Johnson [2005] e Soares et al. [2011]. Isso significa que os desenvolvedores do software, a cada versão lançada, se preocuparam em melhorar a arquitetura desses softwares. Para a escolha dos softwares da avaliação manual, um conjunto de software pequenos foram escolhidos no repositório Github e foram analisados pelo *UMLsmell*, para determinar quais deles possuiam mais *bad smells*. Neste processo foram escolhidos o *Picasso* e *JSLP*. O fato desses softwares serem pequenos viabiliza a inspeção manual pelos avaliadores.

Devido a carência de diagramas de classes para realizar os experimentos, os diagramas dos softwares utilizados para esta avaliação foram obtidos a partir de engenharia reversa usando a ferramenta Enterprise-Architect [2013] para ler os códigos fonte e gerar os diagramas de classes.

Como a ferramenta Enterprise-Architect [2013] não é capaz de criar todos os relacionamentos existentes no código fonte, utilizou-se o software DependencyFinder [2013] que a partir do *bytecode* de um software é capaz de gerar um arquivo XML contendo os relacionamentos de dependência para cada classe do software. Para gerar o diagrama de classes completo automaticamente, a partir das informações obtidas na engenharia reversa do Enterprise Architect e do resultado reportado pelo Dependency Finder, neste trabalho foi desenvolvida uma ferramenta para criar automaticamente todos os relacionamentos de dependência existentes no código fonte. Essa ferramenta intermediária foi desenvolvida na linguagem Java e recebe como entrada o arquivo XMI do Enterprise-Architect [2013] e o arquivo XML do DependencyFinder [2013]. A ferramenta realiza uma análise sintática do arquivo XML e os relacionamentos são criados no arquivo XMI da engenharia reversa feita pelo Enterprise-Architect [2013].

Com isso, obtém-se um arquivo XMI resultante que contém os dados de todass as classes do software e dos relacionamentos existentes entre elas.

5.2 Análise de Softwares Reestruturados

Esta análise busca responder a questão de pesquisa RQ1. Para cada software avaliado, foi necessário gerar dois diagramas de classes: um deles representando a versão usada para avaliar os *bad smells* e o outro, a versão refatorada do mesmo software. Esse detalhe é fundamental para esse experimento, pois dado um software B que representa a refatoração de um software A, espera-se que B apresente menos *bad smells* que A, visto que a refatoração propõe melhorar os aspectos do projeto.

A Figura 5.1 mostra como os resultados da *UMLsmell* foram organizados para cada uma das duas versões do sistema avaliado, considerando V1 a versão do software avaliado e V2 sua versão refatorada. Para cada software usado neste experimento foram geradas três tabelas como esta, uma para cada *bad smells* avaliado. Na primeira coluna da tabela ilustrada na Figura 5.1 aparecem as classes presentes na Versão V1. As demais colunas, cujos conteúdos são relatados a seguir, mostram o resultado da avaliação dos *bad smells*.

	Bad Smell Avaliado				
Software Avaliado	V1	V2	Falha Real	Falso Positivo	Caso Especial
Classe 1	Regular	Regular		x	
Classe 2	Regular	Bom	x		
...	...	...	...	...	...
Classe N	Bom	Ruim			x

Figura 5.1. Tabela com os resultados de *UMLsmell*.

As Colunas 2 e 3 contêm os resultados do *bad smell* avaliado para V1 e V2. O resultado pode ser um dos seguintes valores: *bom*, que indica que os valores das métricas que avaliam o *bad smell* estão dentro da faixa *bom* de seus valores referência, ou seja, uma faixa de valores referência considerados frequentes na maioria dos softwares; *regular*, quando todas as suas métricas estão com valores referência *regulares*; *ruim*, quando ao menos uma métrica está com o valor referência *ruim*; e nos outros casos os *bad smells* estão ausentes, portanto o limiar é considerado *bom*.

A coluna denominada *Falha Real* informa se houve melhora na condição de um *bad smell* de V1 para V2, ou seja, se em relação ao *bad smell* avaliado a classe avaliada foi refatorada de V1 para V2.

A coluna denominada *Falso Positivo*, informa se de V1 para V2 uma classe não sofreu melhoria em relação ao *bad smell* avaliado, por exemplo, uma classe possuía o valor *ruim* para um *bad smell* em V1 e continuou *ruim* em V2. Como o software foi refatorado, o esperado é que o *bad smell* não esteja presente na Versão V2.

A coluna denominada *Caso Especial* apresenta o caso no qual o *bad smell* piorou de V1 para V2 em uma determinada classe. Esses casos são considerados como especiais pois a expectativa é que um software refatorado não apresente resultados piores que na versão não refatorada.

O resultado esperado nesse experimento é que a coluna *Falha Real* possua mais ocorrências do que a coluna *Caso Especial*, pois isso significa que V2 possui menos ocorrências de *bad smells* que V1, que é o que se espera de um software refatorado.

5.2.1 Resultados da Avaliação Automática via *UMLsmell*

Nas próximas seções são descritos os resultados gerados pela *UMLsmell* para os softwares usados neste experimento. Também foi feita uma análise destes resultados para verificar se as estratégias de detecção sugeridas foram eficazes na identificação automática dos *bad smells*.

5.2.1.1 Resultados da Avaliação do *Framework* JHotdraw

O *JHotdraw* é um *framework* para criação de editores gráficos. Neste experimento foram avaliadas as Versões 5.2 e 7.0.6 do software *JHotdraw* correspondendo a V1 e V2, respectivamente. Essa escolha se deu pelo fato de Soares et al. [2011] verificarem que nesse período de tempo o software passou por diversas refatorações que provavelmente afetaram diversas classes.

Foram identificadas 171 classes na Versão V1 e 310 na Versão V2. Todas as classes de V1 mantiveram-se presentes na Versão V2, o que talvez indique que a Versão V2 tenha criado novas funcionalidades e refatorado a Versão V1 criando novas classes.

Os quantitativos de classes de acordo com a classificação (*bom*, *regular* e *ruim*) para cada *bad smell* são ilustrados nas Tabelas 5.1 e 5.2.

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	101	57	13
God Class	144	13	14
Indecent Exposure	148	22	1

Tabela 5.1. *Bad smells* no *JHotdraw* Versão 5.2

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	293	17	0
God Class	294	7	9
Indecent Exposure	233	68	9

Tabela 5.2. *Bad smells* no *JHotdraw* Versão 7.0.6

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	66	104	1
God Class	23	141	7
Indecent Exposure	14	127	30

Tabela 5.3. Comparação dos *bad smells* no *JHotdraw* Versões 5.2 e 7.0.6

Observa-se que mesmo a quantidade de classes tendo aumentado consideravelmente de uma versão para outra, a equipe de desenvolvimento do *JHotdraw* reduziu consideravelmente o nível de acoplamento entre as classes e também a concentração da inteligência em um pequeno grupo de classes, uma vez que a quantidade de classes com os *bad smells God Class* e *Shotgun Surgery* na Versão V2 é menor do que na Versão V1. Porém, o mesmo não ocorreu com o encapsulamento das classes. Na Versão V1 foi identificada apenas uma classe com o *bad smell Indecent Exposure*, enquanto a Versão V2 possui 9 classes com esse *bad smell*.

A comparação entre as Versões V1 e V2 quanto ao número de *falhas reais*, falsos positivos e *casos especiais* são apresentados na Tabela 5.3. Esses dados permitem verificar a quantidade de refatorações de uma versão para outra, referentes aos *bad smells* avaliados. As *falhas reais* permitem quantificar quantas classes foram melhoradas de uma versão para outra, as ocorrências de falso positivo, as classes não refatoradas entre as versões, e, *casos especiais*, as pioras de uma versão para outra.

Para o *bad smell Shotgun Surgery*, de todas as 171 classes, 104 não sofreram alterações, apenas uma teve uma piora na versão posterior e 66 classes passaram por melhoria.

Em relação ao *bad smell God Class*, foi identificada uma situação similar ao *Shotgun Surgery*. Do total de 171 classes, apenas 7 pioraram em relação a esse *bad smell*, 141 se mantiveram no mesmo nível, e foram identificadas 23 classes com melhorias no *bad smell God Class*.

Para o *bad smell Indecent Exposure*, na Versão V2, as classes pioraram na avaliação em relação a esse aspecto.

Analisando a Tabela 5.1, é possível ver que na Versão V1 o número de classes problemáticas era 23, sejam elas com valores *regulares* ou *ruins*. Analisando a Tabela 5.3, é possível perceber que 14 dessas classes melhoraram, ou seja, 60% delas foram refatoradas. Porém, 30 classes da versão V1 pioraram no aspecto do *bad smell Indecent Exposure*. Soares et al. [2011] não descrevem se foram feitas, ou não, refatorações para reduzir a quantidade de atributos públicos. Uma possível explicação para que a refatoração não tenha ocorrido é que essas características mostram que essas 14 melhorias provavelmente são resultados da distribuição de inteligência e redução do acoplamento do sistema, ou seja, reduziu-se a quantidade de atributos públicos em uma classe que as concentrava e, por outro lado, os aumentou em outras que assumiram novas responsabilidades.

Os resultados encontrados para o software *JHotDraw* mostram que a estratégia de detecção do método proposto encontrou mais ocorrências de *bad smells* na versão não reestruturada, conforme esperado, o que indica que a estratégia proposta é capaz de identificar automaticamente a presença de falhas de projeto automaticamente.

5.2.1.2 Resultados da Avaliação do *Framework* Struts

Struts é um *framework* de desenvolvimento da camada controladora para arquitetura MVC (modelo-visão-controle) em Java. As versões avaliadas do software *Struts* foram as seguintes: V1 referindo-se à Versão 1.1 e V2, à Versão 1.2.7. Há 281 classes na versão V1 e 283 na Versão V2. Todas as classes de V1 mantiveram-se presentes na Versão V2, e apenas duas classes foram acrescentadas, portanto, é possível inferir que, se ocorreu a refatoração, talvez ela tenha ocorrido apenas internamente nas classes, ou então, houve apenas uma troca de responsabilidades entre as classes.

Os resultados de V1 e V2 estão resumidos nas Tabelas 5.4 e 5.5.

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	205	64	12
God Class	243	22	16
Indecent Exposure	221	54	6

Tabela 5.4. *Bad smells* no *Struts* Versão 1.1

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	166	101	16
God Class	234	29	20
Indecent Exposure	225	52	6

Tabela 5.5. *Bad smells* no *Struts* Versão 1.2.7

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	5	221	55
God Class	5	258	18
Indecent Exposure	6	272	3

Tabela 5.6. Comparação dos *bad smells* no *Struts* Versões 1.1 e 1.2.7

Analisando o *bad smell Indecent Exposure* percebeu-se que o encapsulamento de algumas classes passou por melhorias, enquanto o de outras piorou. Porém, houve mais melhorias do que pioras. Isso significa que o encapsulamento provavelmente foi um aspecto que passou por refatoração.

De acordo com os resultados das Tabelas 5.4 e 5.5, percebe-se que durante a refatoração, o nível de acoplamento entre as classes aumentou, assim como as classes que concentram inteligência do sistema, pois para os *bad smells Shotgun Surgery* e *God Class*, os valores *ruim* e *regular* aumentaram, enquanto o valor *bom* reduziu.

Os resultados de comparação entre V1 e V2 podem ser vistos na Tabela 5.6. De acordo com os resultados, dos itens avaliados, grande parte piorou da Versão V1 para Versão V2. Os *bad smells Shotgun Surgery* e *God Class* tiveram uma quantidade muito elevada de classes que pioraram na versão posterior, enquanto a quantidade de classes que melhoraram foi muito baixa. No caso do *bad smell Indecent Exposure* foi diferente, mais classes melhoraram em vez de piorar. Porém a quantidade de melhorias e pioras é de 6 para 3, respectivamente, um valor baixo se for considerado que o *Struts* possui 271 classes. Desta forma, possivelmente as refatorações realizadas não foram no sentido de eliminar ou amenizar os *bad smells God Class* ou *Shotgun Surgery*.

Um ponto que deve ser observado diz respeito aos tipos de alterações que foram feitas pela equipe de desenvolvimento durante as refatorações. Dig & Johnson [2005] relatam em seus estudos que a refatoração do *Struts* priorizou mover métodos, mover campos, remover métodos, mudar tipos de argumentos e renomear os métodos. Com isso, as maiores mudanças foram nos métodos e nos campos, de forma que refletiu

em melhoria no aspecto de ocultação de informação, referente ao *bad smell Indecent Exposure*, mas não dos aspectos dos *bad smells God Class* e *Shotgun Surgery*.

5.2.1.3 Resultados da Avaliação do Servidor de Banco de Dados HSqlDB

O *HSqlDB* é um servidor de banco de dados escrito em linguagem Java. As versões avaliadas neste experimento foram as *HSqlDB* 1.8.1, referenciadas aqui como V1, e *HSqlDB* 2.3.1, que é a versão refatorada e referenciada aqui como V2. Na Versão V1 foram identificadas 378 classes e na Versão V2, 703 classes. O aumento de classes de uma versão para outra foi de quase o dobro, o que possivelmente se deu devido à criação de novas funcionalidades, além da refatoração para melhoria das classes.

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	320	57	1
God Class	352	19	7
Indecent Exposure	314	58	6

Tabela 5.7. *Bad smells* no *HSqlDB* Versão 1.8.1

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	595	95	13
God Class	622	41	40
Indecent Exposure	499	172	32

Tabela 5.8. *Bad smells* no *HSqlDB* Versão 2.3.1

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	32	321	25
God Class	11	343	24
Indecent Exposure	32	314	32

Tabela 5.9. Comparação dos *bad smells* no *HSqlDB* Versões 1.8.1 e 2.3.1

As Tabelas 5.7 e 5.8 mostram o resumo dos resultados gerados pela *UMLsmell* para as Versões V1 e V2 do *HSqlDB*, respectivamente.

De acordo com a Tabela 5.9 para o *bad smell Shotgun Surgery*, observa-se que houve melhoria na Versão V2 em relação a Versão V1. Por outro lado, observa-se

que para o *bad smell God Class* houve uma piora. Uma possível explicação para isso é que o aumento de classes na Versão V2 pode ser resultado do acréscimo de novas funcionalidades e também do aumento da inteligência de cada classe. A preocupação em relação ao encapsulamento das classes continuou a mesma, visto que as quantidades de melhorias e pioras são as mesmas.

Soares et al. [2011] afirmam que houve refatorações no software entre as versões analisadas aqui, mas não detalham os tipos de refatorações que foram realizadas. A análise dos resultados desse software mostram que, dentre os *bad smells* avaliados por *UMLsmell*, somente no aspecto de *Shotgun Surgery* houve melhora.

5.2.1.4 Resultados da Avaliação do Protocolo JSLP

O JSLP é uma implementação pura de SLP (*Service Location Protocol*) em Java. As versões avaliadas nos experimentos realizados foram 0.7 e 1.0, que correspondem respectivamente às Versões V1 e V2. Esse é um software pequeno, a Versão V1 possui 33 classes e V2 possui 35 classes. O fato do software ser pequeno e existir a diferença de apenas 2 classes entre as versões, indica que poucas mudanças devem ter ocorrido de uma versão para outra, o que pode resultar também em poucas alterações nos *bad smells*.

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	14	18	1
God Class	30	2	1
Indecent Exposure	31	2	0

Tabela 5.10. *Bad smells* no *Jslp* Versão 0.7

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	15	20	0
God Class	33	2	0
Indecent Exposure	33	2	0

Tabela 5.11. *Bad smells* no *Jslp* Versão 1.0

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	1	32	0
God Class	1	32	0
Indecent Exposure	0	0	0

Tabela 5.12. Comparação dos *bad smells* no *Jslp* Versões 0.7 e 1.0

Observando o resumo dos resultados gerados pela *UMLsmell* apresentados nas Tabelas 5.10 e 5.11, percebe-se que da Versão V1 para Versão V2 houve um aumento de classes na faixa *bom* para todos os *bad smells* avaliados. Também é possível ver que na Versão V2 já não existem classes com valores na faixa *ruim* para nenhum *bad smell*. Isto posto, é possível deduzir que o software passou por refatorações referentes aos *bad smells* analisados por *UMLsmell*.

Os resultados apresentados nas Tabelas 5.10 e 5.11 indicam que houve refatoração da Versão V2 em relação a Versão V1. Portanto, espera-se que sejam identificados mais *bad smells* em V1 do que em V2. Esses resultados são confirmados pelos resultados mostrados na Tabela 5.12. Tanto para o *bad smell Shotgun Surgery* quanto para o *God Class*, houve a melhoria de uma classe em cada um.

Os resultados dessa análise mostram que de fato JSLP passou por modificações que geraram melhorias em sua estrutura nos aspectos dos *bad smells* considerados neste trabalho. Isso indica que essas melhorias foram percebidas pelo método proposto, ou seja, ele se mostrou capaz de identificar os problemas estruturais na Versão V1, bem como se mostrou sensível às melhorias realizadas na Versão V2.

5.2.1.5 Resultados da Avaliação da API Log4j

Log4j é uma API (*Application Programming Interface*) para fazer *log* de dados em aplicações Java. As versões avaliadas neste experimento foram as Versões 1.2 e 1.3 alpha 6, respectivamente referenciadas aqui como Versões V1 e V2. A Versão V1 possui 123 classes e a Versão V2, 215 classes. O aumento do número de classes foi significativo de uma versão para outra.

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	119	4	0
God Class	121	1	1
Indecent Exposure	106	16	1

Tabela 5.13. *Bad smells* no *Log4j* Versão 1.2

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	207	8	0
God Class	213	1	1
Indecent Exposure	194	20	1

Tabela 5.14. *Bad smells* no *Log4j* Versão 1.3 alpha 6

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	0	121	0
God Class	0	121	0
Indecent Exposure	9	106	6

Tabela 5.15. Comparação dos *bad smells* no *Log4j* Versões 1.2 e 1.3a6

Observando as Tabelas 5.13 e 5.14, percebe-se que a quantidade de classes na faixa *bom* para todos os *bad smells* aumentou consideravelmente, enquanto a quantidade de classes na faixa *regular* aumentou suavemente para os *bad smells Shotgun Surgery* e *Indecent Exposure*, e a quantidade de classes na faixa *ruim* se manteve a mesma para todos os *bad smells*.

Do ponto de vista das melhorias das classes de V1 em V2, observando a Tabela 5.15, percebe-se que os *bad smells Shotgun Surgery* e *God Class* não sofreram alterações entre as Versões V1 e V2, não havendo nenhuma melhora e nenhuma piora. Isso sugere que, no aspecto da distribuição da inteligência concentrada das classes e do acoplamento das mesmas, as refatorações realizadas ou as funcionalidades incluídas preservaram a qualidade estrutural do projeto do software. Em relação ao *bad smell Indecent Exposure*, nota-se mais melhorias do que pioras da Versão V1 para Versão V2.

5.2.1.6 Resultados da Avaliação do Sweethome 3D

O *Sweethome 3D* é uma ferramenta para projetos de interiores. As versões avaliadas nesse experimento foram as 2.5 e 3.5 referenciadas aqui como V1 e V2, respectivamente. A Versão V1 possui 160 classes, enquanto a Versão V2 possui 178 classes. Poucas classes foram acrescentadas entre as duas versões tão distantes, o que sugere que refatorações realmente ocorreram da versão V1 para V2, conforme afirmam Soares et al. [2011].

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	54	96	11
God Class	112	33	16
Indecent Exposure	116	38	7

Tabela 5.16. *Bad smells* no *Sweethome 3D* Versão 2.5

Bad Smell	*Bom*	*Regular*	*Ruim*
Shotgun Surgery	149	25	4
God Class	154	8	16
Indecent Exposure	124	40	14

Tabela 5.17. *Bad smells* no *Sweethome 3D* Versão 3.5

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	83	78	0
God Class	32	124	5
Indecent Exposure	7	146	8

Tabela 5.18. Comparação dos *bad smells* no *Sweethome 3D* Versões 2.5 e 3.5

Nas Tabelas 5.16 e 5.17 é possível observar que para os *bad smells Shotgun Surgery* e *Indecent Exposure* a melhoria da Versão V1 para V2 foi significativa, confirmando que realmente ocorreu a refatoração de tais aspectos. Para esses *bad smells* não ocorreram acréscimos nas faixas *regular* e *ruim*, apenas na faixa *bom*. Em relação ao *bad smell Indecent Exposure*, nota-se que as diferenças foram pequenas.

Os dados reportados na Tabela 5.18 mostram que a quantidade de *falhas reais* é alta, indicando que neste software foram realizadas refatorações que resultaram na

melhoria dos *bad smells Shotgun Surgery* e *God Class*. A quantidade de classes classificadas como *caso especial* são baixas, isso porque a refatoração considerou os aspectos avaliados.

5.3 Avaliação Manual

Esta análise busca responder a questão de pesquisa RQ2. Para a realização da análise manual foram escolhidos os softwares Picasso e JSLP. Picasso é uma biblioteca para o sistema operacional Android que permite baixar e armazenar imagens. Foi avaliada a Versão 1.1.1 do Picasso que possui 36 classes. JSLP é uma implementação pura de SLP (*Service Location Protocol*) em Java. Foi avaliada a Versão 1.0.0 RC5 do JSLP que possui 35 classes. A escolha desses softwares se deu pelo fato de *UMLsmell* identificar *bad smells* em algumas classes deles e também por que os softwares são de um tamanho viável para avaliação manual.

Para realizar a avaliação manual dos *bad smells*, 15 avaliadores, alunos de graduação de cursos da área de Computação, se voluntariaram. Os alunos possuem formação em Engenharia de Software. Para estes avaliadores foram disponibilizados: (1) em formato digital, os diagramas de classes dos softwares Picasso e JSLP (2) as definições dos *bad smells God Class*, *Shotgun Surgery* e *Indecent Exposure*, e, (3) para cada software foi distribuída uma lista com classes pré-selecionadas para que os avaliadores julgassem se uma classe possuia, um ou mais *bad smells*; ou se ela não possuia *bad smell*. Não foram apresentadas aos avaliadores as estratégias de detecção e as métricas de software utilizadas no método proposto para identificar os *bad smells*. O Apêndice A apresenta o material disponibilizado para os avaliadores para a realização da avaliação manual. O critério de escolha dos avaliadores foi a de que todos deveriam possuir conhecimento de desvios de projeto e engenharia de software.

Na análise dos resultados produzidos pelos avaliadores, as classes consideradas como problemáticas foram as que em mais de 40% dos questionários os avaliadores julgaram possuir um determinado *bad smell*.

5.3.1 Resultado da Avaliação Manual da Biblioteca Picasso

Das 36 classes do Picasso, 15 foram avaliadas. Foram escolhidas todas as classes com *bad smells* e algumas sem falhas de projeto, incentivando uma análise real dos avaliadores.

A Tabela 5.19 mostra para cada uma das 15 classes do Sistema Picasso e para cada um dos *bad smells*: *God Class*, *Shotgun Surgery* e *Indecent Exposure*, a porcentagem

de avaliadores que identificaram um determinado *bad smell* em uma dada classe. O resultado da avaliação do Picasso pela *UMLsmell* é apresentado na Tabela 5.20.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
Cache	20%	53%	33%
Loader	13%	67%	7%
Listener	0%	40%	13%
Picasso	100%	33%	7%
PicassoTest	80%	7%	13%
PicassoTransformTest	20%	0%	40%
RequestBuilder	33%	7%	13%
RequestBuilderTest	20%	0%	33%
StatsSnapshot	0%	0%	100%
TargetRequestTest	0%	0%	20%
TestTransformation	0%	7%	20%
UrlConnectionLoader	0%	13%	13%
ResponseCacheIcs	13%	7%	7%
UrlConnectionLoaderTest	0%	13%	13%
Utils	27%	7%	0%

Tabela 5.19. Avaliação manual do *Picasso*.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
Cache	*Bom*	*Regular*	*Bom*
Loader	*Bom*	*Regular*	*Bom*
Listener	*Bom*	*Regular*	*Bom*
Picasso	*Bom*	*Regular*	*Bom*
PicassoTest	*Ruim*	*Bom*	*Bom*
PicassoTransformTest	*Bom*	*Bom*	*Bom*
RequestBuilder	*Bom*	*Bom*	*Bom*
RequestBuilderTest	*Bom*	*Bom*	*Bom*
StatsSnapshot	*Bom*	*Bom*	*Ruim*
TargetRequestTest	*Bom*	*Bom*	*Bom*
TestTransformation	*Bom*	*Bom*	*Bom*
UrlConnectionLoader	*Bom*	*Bom*	*Bom*
ResponseCacheIcs	*Bom*	*Bom*	*Bom*
UrlConnectionLoaderTest	*Bom*	*Bom*	*Bom*
Utils	*Bom*	*Bom*	*Bom*

Tabela 5.20. Avaliação do *Picasso* pela *UMLsmell*.

Para o *bad smell God Class* a maioria dos avaliadores julgaram que duas classes possuíam tal problema, *Picasso* e *PicassoTest*. Dessas duas classes a *UMLsmell* julgou apenas uma delas como problemática, classe *PicassoTest*. A classe *Picasso*, apesar de ser considerada como problemática pelos avaliadores, não foi identificado o *God Class* pela *UMLsmell*. Todavia, no método proposto, são consideradas duas métricas para detectar *God Class*: NMP e NCA. *UMLsmell* reportou a métrica NMP como *regular* e NCA como *bom*. Desta forma, de acordo com a estratégia de detecção empregada no método, essa classe não foi indicada com esse *bad smell*.

As classes em que os avaliadores identificaram *God Class* e a *UMLsmell* não são visualmente grandes, devido a quantidade de atributos e métodos, porém a quantidade de relacionamentos aferentes é muito baixo nestas classes. Tal fato pode ter contribuído para que os resultados dos avaliadores e da *UMLsmell* fossem diferentes.

Em relação ao *bad smell Shotgun Surgery*, foram identificadas 4 classes com tal problema pela *UMLsmell*, *Cache*, *Loader*, *Listener* e *Picasso*. Essas classes foram também aquelas com maior porcentagem na identificação dos avaliadores.

O *bad smell Indecent Exposure* foi identificado por unanimidade na classe *StatsSnapshot* pelos avaliadores. A única classe identificada com *Indecent Exposure* pela

UMLsmell também foi a classe *StatsSnapshot*. Nas outras classes, menos de 40% dos avaliadores consideraram que elas possuem *Indecent Exposure*.

De forma geral, neste experimento, os resultados de *UMLsmell* foram coincidentes com aqueles reportados pelos avaliadores.

5.3.2 Resultado da Avaliação Manual do Protocolo JSLP

Na Tabela 5.21 são apresentados os resultados da avaliação manual do JSLP. Na Tabela 5.22 são apresentados os resultados da avaliação automática do JSLP feita por *UMLsmell*.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
AuthenticationBlock	13%	73%	27%
Filter	0%	73%	13%
ReplyMessage	0%	67%	0%
ServiceLocationException	27%	20%	60%
ServiceLocationManager	0%	73%	7%
ServiceType	7%	53%	27%
ServiceURL	40%	73%	40%
SLPConfiguration	80%	60%	53%
SLPManager	73%	67%	20%
SLPMessage	67%	60%	60%

Tabela 5.21. Avaliação manual do *JSLP*

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
AuthenticationBlock	*Bom*	*Regular*	*Bom*
Filter	*Bom*	*Regular*	*Bom*
ReplyMessage	*Bom*	*Regular*	*Bom*
ServiceLocationException	*Bom*	*Regular*	*Ruim*
ServiceLocationManager	*Bom*	*Bom*	*Bom*
ServiceType	*Regular*	*Regular*	*Bom*
ServiceURL	*Regular*	*Regular*	*Regular*
SLPConfiguration	*Bom*	*Regular*	*Bom*
SLPManager	*Bom*	*Regular*	*Bom*
SLPMessage	*Bom*	*Regular*	*Ruim*

Tabela 5.22. Avaliação do *JSLP* pela *UMLsmell*

Das duas classes identificadas com *God Class* nos resultados da *UMLsmell*, apenas uma foi considerada como problematica pelos avaliadores, a classe *ServiceURL*. Os avaliadores também julgaram que as três últimas classes que aparecem na Tabela 5.21 possuem *God Class*, enquanto a *UMLsmell* julgou que as mesmas não possuem tal problema. Essas três classes possuem muitos atributos e métodos, mas poucas conexões aferentes.

Das dez classes avaliadas, foram identificadas oito com o *bad smell Shotgun Surgery* tanto pelos avaliadores, quanto pela *UMLsmell*. Apenas a classe *ServiceLocationException* foi considerada como problemática pela *UMLsmell* mas não pelos avaliadores. Ocorreu outro caso em que os avaliadores consideraram a classe *ServiceLocationManager* como problemática, mas a *UMLsmell* não.

Dentre as quatro classes identificadas com o *bad smell Indecent Exposure* pelos avaliadores, três constam como problemáticas nos resultados da *UMLsmell*.

Neste experimento também, os resultados reportados por *UMLsmell* são muito próximos daqueles reportados pelos avaliadores.

5.4 Análise dos Resultados das Avaliações Automática e Manual

Para responder a questão de pesquisa RQ1: *As métricas de software e seus respectivos valores referência auxiliam a identificar bad smells em um software ?* foram realizados os experimentos apresentados na Seção 5.2.1. O objetivo da realização da avaliação

automática foi verificar se realmente é possível identificar *bad smells* automaticamente em diagramas de classes. Para cada software foram avaliados os resultados gerados pela *UMLsmell* para as Versões V1 e V2, onde a Versão V2 é tida na literatura como uma refatoração da Versão V1. Como a Versão V2 é resultante da refatoração da Versão V1 era esperado encontrar melhorias entre as duas versões em pelo menos um dos aspectos avaliados nesse estudo e, de fato, em quase todos os softwares foram identificadas ao menos uma melhoria de *bad smells* de uma versão para outra refatorada. Este resultado confirma a eficácia do método proposto, visto que foram encontrados mais falhas na versão não refatorada dos softwares avaliados.

Essas melhorias ficaram evidentes nos softwares *Jhotdraw* e *Sweethome 3D*. Nesses dois softwares, a quantidade de *bad smells* foi muito superior em V1 do que em V2, o que está de acordo com o que foi relatado na literatura que diz que a Versão V2 é uma refatoração de V1. Nesses softwares foram constatados que principalmente nos *bad smells Shotgun Surgery* e *God Class* a quantidade de classes que apresentavam esses problemas diminuiu muito mais do que cresceu, apontando que a estrutura do sistema realmente passou por melhorias. No entanto, nesses três casos, não se pode dizer o mesmo em relação ao *Indecent Exposure*, que em alguns casos a quantidade de classes que apresentavam esse *bad smell* foi mantida ou aumentou da Versão V1 para V2. Tal fato pode ter ocorrido porque esse *bad smell* pode não ter sido alvo de atenções durante as refatorações.

Também foram observadas melhorias mais discretas nos softwares *Struts*, *Log4j*, *HSqlDB* e *JSLP*. Neles, nos três *bad smells* avaliados, ocorreram menos melhorias da Versão V1 para a Versão V2, sendo que em alguns casos ocorreu aumento da quantidade de *bad smells*. Mas, ainda assim, foi possível detectar que a refatoração ocorreu em alguns casos nos quais os *bad smells* reduziram, por exemplo, o *Indecent Exposure* no software *Struts*.

Para responder a questão de pesquisa RQ2: *Os bad smells identificados pelo método em um software são correspondentes àqueles identificados por avaliadores com formação em Engenharia de Software?* foi realizada avaliação manual, apresentada na Seção 5.3. A avaliação manual feita apresentou resultados muito próximos dos apresentados pela *UMLsmell*. Para os *bad smells Shotgun Surgery* e *Indecent Exposure*, as classes identificadas como problemáticas pela *UMLsmell* foram as classes com maior frequência de identificação pelos avaliadores. Das classes em que *UMLsmell* acusou a presença do *bad smell God Class* a maioria também foi identificada com esses *bad smells* pelos avaliadores. Quatro classes identificadas como problemática pelos avaliadores, não foram identificadas como problemáticas pela *UMLsmell*. Esses resultados indicam que as estratégias de detecção definidas neste estudo, estão próximas dos resultados

dos avaliadores. No caso de *God Class*, houve uma diferença maior entre os resultados do método proposto e a análise manual. Uma explicação possível para isso é que os avaliadores provavelmente consideraram apenas o tamanho das classes, enquanto a estratégia de detecção do método proposto considera também a quantidade de classes que utilizam a classe avaliada.

5.5 Considerações Finais

Este capítulo mostrou os resultados dos experimentos realizados para avaliar o método proposto.

Durante a realização dos experimentos, observando seus resultados, foram identificadas algumas limitações. A principal delas está associada ao fato do método descrito neste trabalho considerar poucos *bad smells*. Esta limitação se deve ao fato de haver poucas métricas para os quais há valores referência propostos na literatura.

As demais limitações encontradas estão relacionadas com:

Diagramas de classes usados nos experimentos: o ideal seria a utilização de diagramas de classes produzidos durante a fase de projeto do software. Todavia, é difícil obter esse tipo de diagrama em projetos abertos e, em geral, há restrições por parte das organizações em fornecer qualquer dado sobre seus softwares proprietários. Devido a isso, os diagramas foram obtidos via engenharia reversa a partir do código fonte de softwares abertos.

Desconhecimento da arquitetura e do contexto dos softwares avaliados: em uma avaliação manual, para afirmar com precisão que as classes apontadas como problemáticas realmente possuem os *bad smells* acusados pela *UMLsmell*, seria necessário que um avaliador tivesse conhecimento das classes e seus elementos. Por exemplo, uma superclasse que utiliza o padrão de projeto *Strategy* e possui muitas subclasses não deve ser considerada como problemática, pois dependendo do contexto, tal estratégia exige muitas conexões aferentes

Apesar dessas limitações, os resultados da avaliação do método proposto indicam que ele é capaz de auxiliar, de forma automática, a identificação de *bad smells* em software orientado por objetos a partir de diagrama de classes, utilizando, para isso, métricas de software e seus respectivos valores referência.

Os resultados dos experimentos realizados em softwares tidos como refatorados mostram que em todos os softwares foram identificadas ao menos uma melhoria de *bad smells* de uma versão para outra refatorada. Esse resultado indica que o método

proposto é capaz de detectar automaticamente os *bad smells* considerados neste trabalho, visto que foram encontradas mais falhas na versão não refatorada dos softwares avaliados. Com isso, conclui-se que a resposta para RQ1 é positiva. Da mesma forma os resultados da avaliação manual se mostraram muito próximos daqueles produzidos por *UMLsmell*, aonde se conclui que a resposta para a RQ2 também é positiva.

Capítulo 6

Conclusão

Esse trabalho de dissertação propõe um método para identificar desvios de projeto de software nas fases iniciais do ciclo de vida do sistema, mais especificamente na fase de projeto. O método proposto se baseia em extrair métricas de diagramas de classes e usar os valores referência propostos na literatura para tais métricas para identificar *bad smells*.

Inicialmente, foram identificados os *bad smell* que podem ser extraídos de diagramas de classes. Depois foram identificadas as principais métricas que podem ser extraídas de diagramas de classes. Em relação às métricas, duas características foram levadas em consideração: primeiro, as métricas deveriam possuir valores referência propostos na literatura e segundo, que elas representassem melhor os *bad smells* que podem ser identificados em diagramas de classes. Os valores referência usados para relacionar as métricas aos *bad smells Shotgun Surgery*, *God Class* e *Indecent Exposure* foram definidos por Ferreira et al. [2012].

Neste trabalho, também foi projetada e implementada uma ferramenta, denominada *UMLsmell*, que permite aplicar o método proposto. A ferramenta identifica automaticamente *bad smells* em diagramas de classes.

Para avaliar esse método, foram conduzidos dois grupos de experimentos. O primeiro foi realizado com o objetivo de verificar se o método proposto identifica os *bad smells* considerados neste trabalho, a partir da comparação entre versões de softwares que, de acordo com relatos na literatura, passaram por refatorações. O segundo experimento teve por objetivo comparar os resultados reportados pelo método e a avaliação manual.

A análise automática consistiu em avaliar duas versões de softwares abertos, tal que a segunda versão do software em questão era tida como a refatoração da primeira versão. Era esperado que *UMLsmell* apontasse menos problemas na segunda versão se

comparada com a primeira. Os resultados dessa análise mostram que todos os softwares apresentaram algum tipo de melhoria da primeira para a segunda versão.

O segundo experimento foi realizado com um grupo de 15 avaliadores. Foram distribuídos os diagramas de classes dos softwares *Picasso* e *JSLP* para que os avaliadores julgassem quais classes possuem algum tipo de *bad smell*.

A comparação entre os resultados manuais dos avaliadores e dos automáticos gerados pela *UMLsmell* permitiu identificar que, na maioria dos casos, os resultados dos avaliadores e da *UMLsmell* foram próximos, principalmente para os *bad smells Shotgun Surgery* e *Indecent Exposure*. Isso indica que as métricas e valores referência utilizados no método definido neste trabalho permitem identificar automaticamente *bad smells* que seriam apontados de forma mais custosa e demorada de forma manual.

Os experimentos realizados evidenciam que o método proposto auxilia a identificar automaticamente a presença de *bad smells* em diagramas de classes.

6.1 Trabalhos Futuros

Com os resultados obtidos neste trabalho, identificam-se os seguintes trabalhos futuros:

- A identificação de valores referência para outras métricas, que será importante para viabilizar a detecção de outros *bad smells* com o método proposto.
- Realizar outros experimentos com pequenos ajustes nos valores referência utilizados com o objetivo de verificar se isso melhoraria a eficácia da detecção.
- Avaliar o uso de outras métricas para avaliar os *bad smells* considerados nesta dissertação a fim de refinar as estratégias de detecção sugeridas.
- Aplicar o método em diagramas de colaboração da UML, como o diagrama de sequência, a fim de verificar se é possível identificar outros *bad smells* além daqueles considerados neste trabalho.

6.2 Publicações

- Nunes, Henrique Gomes; Bigonha, Mariza Andrade da Silva; Ferreira, Kecia Aline Marques. *Identificação de Bad Smells em Softwares a partir de Modelos UML*. Publicado em WTDSoft 2013, Brasília, DF.
- Nunes, Henrique Gomes; Bigonha, Mariza Andrade da Silva; Ferreira, Kecia Aline Marques; Airjan, Flávio. *Um Método para Identificação Automática de Falhas de*

Projeto em Modelos UML. Esse artigo está sendo escrito para posteriormente ser submetido em CBSoft 2014.

- Nunes, Henrique Gomes; Bigonha, Mariza Andrade da Silva; Ferreira, Kecia Aline Marques; Airjan, Flávio. *UMLsmell: Uma ferramenta para Identificação de Falhas de Projeto em Modelos UML*. Esse artigo está sendo escrito para posteriormente ser submetido em uma sessão de ferramentas no CBSoft 2014.

Apêndice A

Avaliação Manual dos *Bad Smells*

A avaliação manual dos *bad smells* foi realizada por 15 alunos de graduação da área de Computação, com formação em Engenharia de Software. Para estes avaliadores foram disponibilizados:

1. em formato digital, os diagramas de classes dos softwares Picasso e JSLP;
2. as definições dos *bad smells God Class*, *Shotgun Surgery* e *Indecent Exposure*;
3. uma lista, para cada software, com classes pré-selecionadas para que os avaliadores julgassem se cada classe possuia, um ou mais *bad smells* ou se ela não possuia *bad smell*.

A seleção dessas classes se deu devido ao fato de as mesmas possuirem *bad smells* de acordo com a avaliação do *UMLsmell*. Também foram incluídas algumas classes sem *bad smells*. Não foram apresentadas aos avaliadores as estratégias de detecção e as métricas de software utilizadas no método proposto para identificar os *bad smells*. O questionário aplicado neste experimento é apresentado em A.3.

A.1 Definição dos *Bad Smells*

O autor Martin Fowler define *bad smell* como um indicador de possível problema estrutural em projetos de software, que pode ser melhorado via refatoração. Na literatura, podemos encontrar os seguintes exemplos de *bad smells*:
God Class: ocorre quando poucas classes tendem a centralizar grande parte da inteligência do sistema, fazendo com que muitas classes dependam dos serviços do pequeno conjunto dessas classes. Tal prática é indesejada, pois isso faz com que essas classes sejam grandes, complexas e de difícil manutenção.

Shotgun Surgery: ocorre quando mudanças em uma classe geram pequenas mudanças em muitas outras classes. A consequência do *Shotgun Surgery* é que a manutenção de um software será muitas vezes custosa, pois sempre que uma classe for alterada, provavelmente muitas outras também serão.
Indecent Exposure: ocorre quando uma classe é mal encapsulada, expondo seus dados que deveriam ser ocultos. O resultado do mal encapsulamento é a baixa segurança de um sistema.

A.2 Softwares Usados na Avaliação Manual

Os softwares escolhidos para a avaliação manual foram: Picasso e JSLP. Picasso é uma biblioteca para o sistema operacional Android que permite baixar e armazenar imagens. Para esta avaliação foi usada a Versão 1.1.1 do Picasso que possui 36 classes. JSLP é uma implementação pura de SLP (*Service Location Protocol*) em Java. Em relação a este software foi avaliada a Versão 1.0.0 RC5 que possui 35 classes. A escolha desses softwares se deu pelo fato de *UMLsmell* identificar *bad smells* em algumas classes deles e também por que os softwares são de um tamanho viável para avaliação manual.

A.3 Questionário Usado para a Avaliação Manual

Considerando as definições dos três *bad smells* apresentados, para cada classe a seguir, identifique quais *bad smells* cada classe possui. Uma classe pode possuir nenhum *bad smell* ou mais do que um *bad smell*.

A.3.0.1 JSLP

AuthenticationBlock	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
Filter	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ReplyMessage	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ServiceLocationException	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ServiceLocationManager	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ServiceType	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ServiceURL	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
SLPConfiguration	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
SLPManager	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
SLPMessage	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*

A.3.1 Picasso

Cache	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
Listener	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
Loader	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
Picasso	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
PicassoTest	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
PicassoTransformTest	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
RequestBuilder	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
RequestBuilderTest	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
ResponseCacheIcs	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
StatsSnapshot	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*
TargetRequestTest	[ ] *God Class*	[ ] *Shotgun Surgery*	[ ] *Indecent Exposure*

A.4 Exemplo de Classes Avaliadas

Nas Figuras A.1 e A.2 são apresentadas duas classes julgadas como problemáticas, tanto pelos alunos quanto pelo *UMLsmell*.

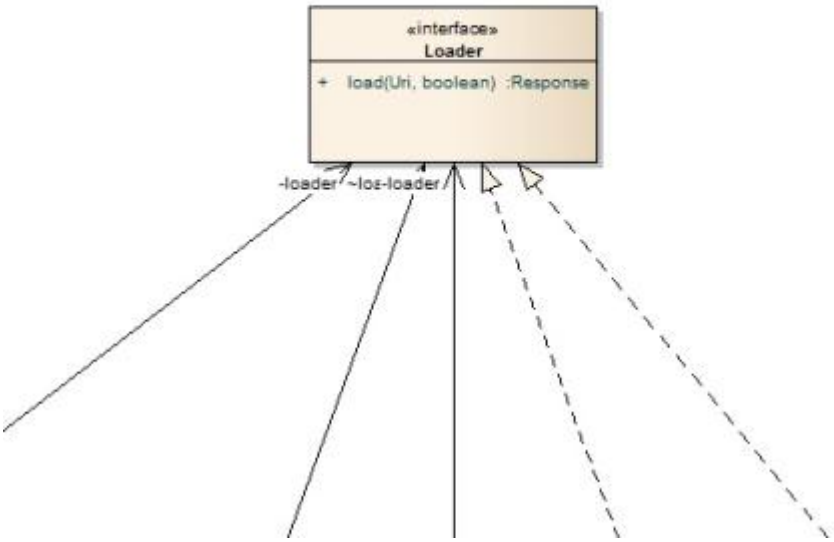

Figura A.1. Classe *Loader* do software *Picasso*

Na classe *Loader*, ilustrada na Figura A.1, foi identificado o *bad smell Shotgun Surgery* por 67% dos alunos e a mesma classe foi julgada julgada na faixa *regular* para o

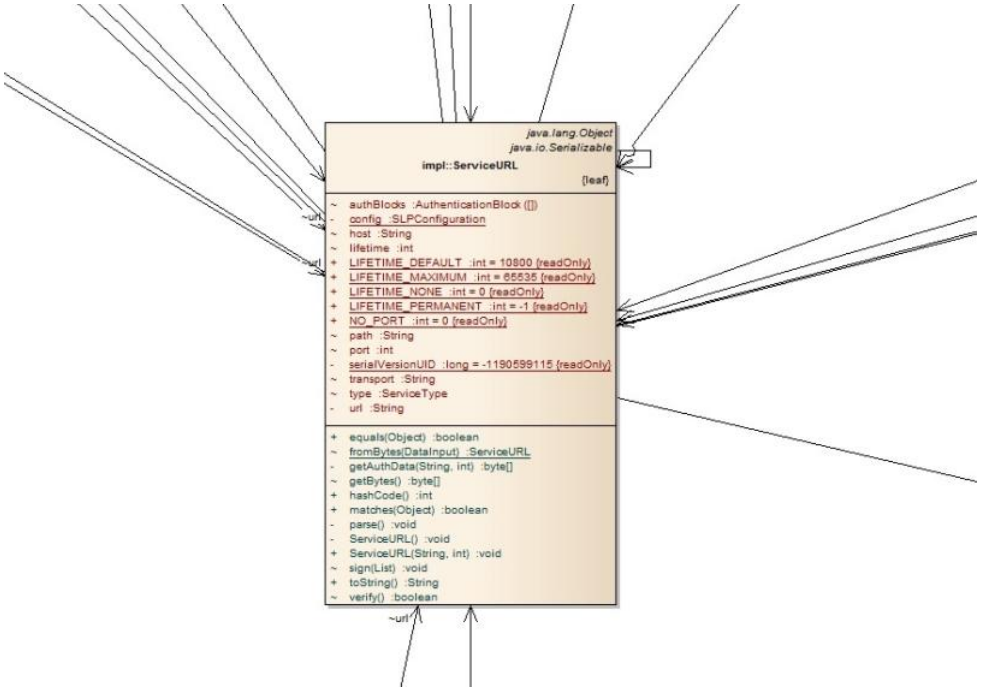

Figura A.2. Classe *ServiceURL* do software *JSLP*

mesmo *bad smell* pelo *UMLsmell*. Para os outros *bad smells* menos de 15% dos alunos identificaram problemas e o *UMLsmell* não apontou problemas.

Na classe *ServiceURL*, ilustrada na Figura A.2, mais de 40% dos alunos identificaram os três *bad smells* avaliados e o mesmo ocorreu na avaliação automática em que para todos os *bad smells* a classe se enquadrou na faixa *regular*.

Referências Bibliográficas

Abreu, F. B. & Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. Em *proceedings of the 4th International Conference on Software Quality*, volume 186.

ArgoUML (2013). http://argouml.tigris.org/, acessado em 20/11/2013.

Bansiya, J. & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4--17.

Bertran, I. M. (2009). Avaliação da qualidade de software com base em modelos uml. Dissertação da PUC-RJ. Rio de Janeiro, RJ, Brasil. Pontifícia Universidade Católica do Rio de Janeiro.

Briand, L.; Devanbu, P. & Melo, W. (1997). An investigation into coupling measures for c++. Em *Proceedings of the 19th international conference on Software engineering*, pp. 412--421. ACM.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476--493.

Crespo, Y.; López, C. & Marticorena, R. (1996). Relative thresholds: Case study to incorporate metrics in the detection of bad smells. Em *QAOOSE 2006 Proceedings*, pp. 109–118.

D'Ambros, M.; Bacchelli, A. & Lanza, M. (2010). On the impact of design flaws on software defects. Em *Quality Software (QSIC), 2010 10th International Conference on*, pp. 23--31. IEEE.

DependencyFinder (2013). http://depfind.sourceforge.net/, acessado em 08/12/2013.

Dig, D. & Johnson, R. (2005). The role of refactorings in api evolution. Em *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 389--398. IEEE.

Enterprise-Architect (2013). http://www.sparxsystems.com.au/, acessado em 20/11/2013.

ESS-Model (2013). http://essmodel.sourceforge.net/, acessado em 20/11/2013.

Ferreira, K. A.; Bigonha, M. A.; Bigonha, R. S.; Mendes, L. F. & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244--257.

Ferreira, K. M. (2011). Em *Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos. Tese de Doutorado*, Belo Horizonte, MG. Universidade Federal de Minas Gerais.

Fowler, M. (1999). *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2005). *UML Essencial: Um breve guia para a linguagem-padrão de modelagem de objetos. 3a ed.* Bookman, Porto Alegre.

Gamma, E.; Johnson, R.; Helm, R. & Vlissides, J. (2006). *Padrões de Projetos: Soluções Reutilizáveis.* Bookman, Porto Alegre, RS.

Genero, M. (2002). Defining and validating metrics for conceptual models. Ph.D. thesis. University of Castilla-La Mancha.

Genero, M.; Piattini, M. & Calero, C. (2005). A survey of metrics for uml class diagrams. *Journal of Object Technology*, 4(9):59--92.

Girgis, M.; Mahmoud, T. & Nour, R. (2009). Uml class diagram metrics tool. Em *Computer Engineering Systems, 2009. ICCES 2009. International Conference on*, pp. 423 –428.

Hamza, H.; Counsell, S.; Loizou, G. & Hall, T. (2008). Code smell eradication and associated refactoring. Em *Proceedings of the European Computing Conference (ECC)*.

Harrison, R.; Counsell, S. & Nithi, R. (1998). Coupling metrics for object-oriented design. Em *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pp. 150--157. IEEE.

IEEE (1990). Ieee standard glossary of software engineering terminology. *Office*, 121990(1):1.

Infusion (2012). http://www.intooitus.com/products/infusion, acessado em 25/11/2012.

iPlasma (2013). http://loose.upt.ro/reengineering/research/iplasma, acessado em 08/12/2013.

JHotDraw (2012). http://www.jhotdraw.org/, acessado em 25/11/2012.

Kerievsky, J. (2005). *Refactoring to patterns*. Pearson Deutschland GmbH.

Lanza, M.; Marinescu, R. & Ducasse, S. (2006). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Li, W. & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111--122.

Lorenz, M. & Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

MagicDraw (2013). http://www.nomagic.com/products/magicdraw.html, acessado em 20/11/2013.

Marchesi, M. (1998). Ooa metrics for the unified modeling language. Em *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on*, pp. 67--73. IEEE.

Marinescu, C.; Marinescu, R.; Mihancea, P. F. & Wettel, R. (2005). iplasma: An integrated platform for quality assessment of object-oriented design. Em *In ICSM (Industrial and Tool Volume*. Citeseer.

Marinescu, R. (2002). Em *Measurement and Quality in Object-Oriented Design*, Tese de Doutorado. Timisoara, Romênia. University of Timisoara.

Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. Em *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 350 – 359.

Microsoft (2013). http://www.microsoft.com/en-us/download/details.aspx?id=24023, acessado em 20/11/2013.

Moose (2013). http://moose.iinteractive.com/en/, acessado em 08/12/2013.

Nuthakki, M. K.; Mete, M.; Varol, C. & Suh, S. C. (2011). Uxsom: Uml generated xml to software metrics. *SIGSOFT Softw. Eng. Notes*, 36(3):1--6.

OMG (2012). http://www.omg.org/, acessado em 25/11/2012.

Soares, G.; Catao, B.; Varjao, C.; Aguiar, S.; Gheyi, R. & Massoni, T. (2011). Analyzing refactorings on software repositories. Em *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pp. 164--173. IEEE.

Soliman, T.; El-Swesy, A. & Ahmed, S. (2010). Utilizing ck metrics suite to uml models: A case study of microarray midas software. Em *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pp. 1 –6.

Sommerville, I. (2007). *Engenharia de Software 8a ed.* São Paulo: Pearson Addison Wesley.

UMLet (2013). http://www.umlet.com/, acessado em 20/11/2013.

Yi, T.; Wu, F. & Gan, C. (2004). A comparison of metrics for uml class diagrams. *SIGSOFT Softw. Eng. Notes*, 29(5):1--6.