ESTUDO DE COOCORRÊNCIAS DE PADRÕES DE PROJETO E *BAD SMELLS* USANDO MÉTRICAS DE SOFTWARE

BRUNO LUAN DE SOUSA

ESTUDO DE COOCORRÊNCIAS DE PADRÕES DE PROJETO E *BAD SMELLS* USANDO MÉTRICAS DE SOFTWARE

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
Coorientadora: Kecia Aline Marques Ferreira

Belo Horizonte
Julho de 2017

.

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Sousa, Bruno Luan de.

S725e Estudo de coocorrências de padrões de projeto e bad smells usando métricas de software. / Bruno Luan de Sousa. – Belo Horizonte, 2017.
xxi, 108 f.: il.; 29 cm.

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

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

1. Computação – Teses. 2. Padrão de projeto de software. 3. Métricas de software. 4. Bad smell. I. Orientadora. II. Coorientadora. III. Título.

CDU 519.6*32(043)

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

FOLHA DE APROVAÇÃO

Estudo de coocorrências de padrões de projeto e bad smells usando métricas de software

BRUNO LUAN DE SOUSA

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. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

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

Belo Horizonte, 07 de julho de 2017.

Agradecimentos

A trajetória de um curso de mestrado é uma tarefa árdua e desafiadora. Contudo, quem vivenciou essa trajetória sabe que é experiência enriquecedora e de plena superação. A conclusão desta dissertação marca o fim desta importante etapa na minha vida. Portanto, gostaria de agradecer à todos aqueles que compartilharam e caminharam comigo durante toda a trajetória deste curso.

Agradeço primeiramente à Deus por ser sempre o meu guia, por iluminar os meus caminhos e principalmente nunca me deixar desistir mesmo nos momentos mais difíceis que encontrei ao longo desta caminhada.

Agradeço à toda minha família. Em especial aos meus pais, Paulo e Irani, por todo amor, apoio e incentivo dedicado durante toda a minha vida. Eles são para mim exemplos de dedicação e perseverança, e me deram o alicerce necessário para minha educação.

Agradeço às professoras Mariza Bigonha e Kécia Ferreira, pela excelente orientação, dedicação e incentivo na condução deste trabalho. Durante esse período do curso de mestrado elas acompanharam minha evolução e forneceram o suporte necessário para superar minhas dificuldades no curso.

Agradeço a todos os meus professores que se dispuseram a doar conhecimento, tempo e paciência à mim para que eu pudesse evoluir.

Agradeço a todos os meus amigos com os quais compartilhei momentos incríveis durante toda essa caminhada do curso de mestrado.

Agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) e à CAPES pela oportunidade de realizar um curso com alto nível de excelência e pelo suporte financeiro prestado durante todo o curso.

Por fim, agradeço a todas as pessoas que contribuíram, de forma direta ou indireta, na conclusão deste trabalho e me deram o incentivo para sempre levá-lo adiante.

"Não haveria criatividade sem a curiosidade que nos move e que nos põe pacientemente impacientes diante do mundo que não fizemos, acrescentando a ele algo que fazemos."

(Paulo Freire)

Resumo

Bad Smells são sintomas que aparecem no código fonte de um software e que possivelmente indicam a presença de um problema estrutural que demanda refatoração do código. Padrões de projeto são soluções conhecidas como boas práticas de desenvolvimento de software e auxiliam na produção de software com qualidade e estrutura flexível. Alguns trabalhos na literatura destacam o uso dos padrões de projeto para remoção de *bad smells*. Contudo, outros trabalhos identificam relações de coocorrência entre padrões de projeto e *bad smells*. Tendo em vista a relevância de ambos conceitos no contexto de Engenharia de Software, é importante conhecer as situações em que o uso de padrões de projeto podem estar associadas ao surgimento de *bad smells*. Baseado nisso, o objetivo deste trabalho é estudar as ocorrências de *bad smells* em software desenvolvidos com padrões de projeto. Para atingir esse objetivo, inicialmente foi realizada uma revisão sistemática da literatura, a fim de se compreender o atual estado da arte referente a padrões de projeto e *bad smells*. Além disso, foi realizado um estudo com cinco projetos Java, a fim de (i) investigar se o uso de padrões de projeto reduzem a ocorrência de *bad smells*, (ii) identificar coocorrências entre padrões de projeto com *bad smells*, e (iii) identificar as principais situações que levaram os sistemas de software a apresentarem essas relações de coocorrência. Os resultados do estudo indicam que apesar de alguns padrões de projeto apresentarem baixa relação de coocorrência com *bad smells*, essas soluções não necessariamente evitam o surgimento de *bad smells*. Além disso, a análise das situações de coocorrência de *bad smells* e padrões de projeto indica que o mal planejamento de padrões de projeto e seu uso inadequado impactam diretamente no surgimento de *bad smells*.

Palavras-chave: Padrões de Projeto, *Bad Smell*, Métricas de Software, Coocorrência entre Padrões de Projeto e *Bad Smell*.

Abstract

Bad Smells are symptoms that appear in the source code of a software and possibly indicate the presence of a structural problem that requires code refactoring. Design patterns are solutions known as good software development practices and help in the production of software with quality and flexible structure. Some papers in the literature highlight the use of design patterns for removal of bad smells. However, other studies identify relations of co-occurrence between design patterns and bad smells. Given the relevance of both concepts in the context of Software Engineering, it is important to know the situations in which the use of design patterns may be associated with the emergence of bad smells. Based on this, the goal of this work is to study the occurrences of bad smells in software developed with design patterns. To achieve this goal, a systematic review of the literature was initially undertaken in order to understand the current state of the art regarding design patterns and bad smells. In addition, we perfomed a study with five Java projects in order to (i) investigate whether the use of design patterns reduces the occurrence of bad smells, (ii) identify co-occurrences between design patterns with bad smells, and (iii) identify the main situations that led software systems to present these relations of co-occurrence. The results of the study indicate that although some design patterns have a low co-occurrence relationship with bad smells, these solutions do not necessarily prevent the emergence of bad smells. In addition, analysis of co-occurrence situations of bad smells and design patterns indicates that poor planning of design patterns and their misuse directly impact the emergence of bad smells.

Keywords: Design Pattern, Bad Smell, Software Metrics, Co-Occurrence between Design Pattern and Bad Smell.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A Engenharia de software surgiu com o intuito de organizar o desenvolvimento e ainda fornecer métodos e ferramentas para auxiliarem na criação de software. Padrão de projeto é um tipo de solução que se encaixa dentre esses métodos na Engenharia de Software. Gamma et al. [1994] definem padrão de projeto como soluções recorrentes para um problema em um dado contexto. Eles descrevem um conjunto de padrões de projeto referenciados amplamente como "padrões *GOF*[1]". O principal objetivo dessa solução é garantir a criação de softwares flexíveis e extensíveis, com uma estrutura reusável e de fácil manutenção.

Padrões de projetos são conhecidos como boas práticas de programação e seguem a linha de reúso. Essas práticas, quando aplicadas de forma correta, podem auxiliar na redução de *bad smells* em software, apesar de não terem sido propostas com esse intuito [Speicher, 2013]. *Bad smells* são sintomas que aparecem no código fonte de um software e que possivelmente indicam um problema mais profundo que demanda refatoração do código [Fowler & Beck, 1999]. As regiões do software que apresentam esses sintomas não são consideradas erradas, porém a presença de *bad smells* prejudicam sua qualidade e violam princípios da Engenharia de Software como modularidade, legibilidade e reúso. Kerievsky [2004] destaca o uso de padrões de projetos para a remoção desses sintomas do código. Por outro lado, há estudos que identificam coocorrência de padrões de projeto e *bad smells* [Cardoso & Figueiredo, 2015; Jaafar et al., 2013, 2016; Walter & Alkhaeir, 2016].

Muito embora haja trabalhos na literatura associando padrões de projetos com *bad smells*, [Cardoso & Figueiredo, 2015; Jaafar et al., 2013, 2016; Walter & Alkhaeir, 2016], esse tema é pouco explorado no meio científico.

Além disso, embora os resultados desses estudos sejam promissores, tais estudos

[1] *Gang of four*

não avaliam possíveis situações ou causas que culminam nessas relações. Uma investigação sobre as razões que originam as coocorrências pode servir como um arcabouço teórico para pesquisadores e desenvolvedores, além de alertá-los para cenários nos quais a aplicação de um padrão de projeto em um dado contexto, resulta em estruturas problemáticas que elevam a complexidade de um sistema de software, prejudicando sua legibilidade e entendimento.

Diante do exposto, a pesquisa proposta investiga a relação entre padrões de projeto e *bad smells* sob dois aspectos: verificação da efetividade dos padrões de projeto na redução de *bad smells*, e identificação de possíveis coocorrências e situações que influenciam o surgimento de tais relações.

1.1 Objetivos

O objetivo geral desse trabalho é investigar e estudar as relações de coocorrências entre padrões de projeto e *bad smells* em sistemas de software orientado por objetos.

Para atender esse objetivo geral, o autor desta dissertação se propôs a:

- identificar se a aplicação de padrões de projeto em sistemas de software auxiliam na redução de ocorrências de *bad smells*;
- identificar padrões de projeto que apresentam relações de coocorrências com *bad smells*;
- identificar situações, por meio de uma análise do código fonte de sistemas de software, que contribuem para as relações de coocorrência entre padrões de projeto e *bad smells*.

Nesse sentido, foi realizado um Estudo de Caso com cinco *bad smells*, e 14 padrões de projeto que compõe o catálogo *GOF*, exibidos na Tabela 1.1.

Bad Smell	Padrões de Projeto		
Data Class	*Adapter*	*Bridge*	*Command*
Feature Envy	*Composite*	*Decorator*	*Factory Method*
Large Class	*Observer*	*Prototype*	*Proxy*
Long Method	*State*	*Strategy*	*Singleton*
Refused Bequest	*Template Method*	*Visitor*	

Tabela 1.1. Relação dos padrões de projeto e *bad smells* utilizados neste estudo.

A fim de avaliar a relação de coocorrência entre os *bad smells* e os padrões de projeto da Tabela 1.1 e atender os objetivos propostos nesse trabalho, foram propostas as seguintes Questões de Pesquisas (QP):

- **QP1:** Os padrões de projeto definidos no catálogo *GOF* evitam a ocorrência de *bad smells* em software?
- **QP2:** Quais padrões de projeto do catálogo *GOF* apresentaram coocorrência com *bad smells*?
- **QP3:** Quais são as situações mais comuns em que *bad smells* aparecem em sistemas de software que aplicam os padrões de projeto *GOF*?

1.2 Contribuições

Como principais resultados do estudo conduzido pelo autor desta dissertação, destacam-se as seguintes contribuições.

1. Revisão Sistemática da Literatura com foco em estudos sobre padrões de projeto e *bad smell*. Os resultados obtidos proporcionam à comunidade acadêmica uma visão geral do estado da arte nesse campo. Ao mesmo tempo, essa Revisão Sistemática da Literatura revela que existe uma baixa quantidade de estudos dedicados à coocorrência entre padrões de projeto e *bad smells*.
2. *Design Pattern Smell*, uma ferramenta que suporta detecção de coocorrência entre padrões de projeto e *bad smell* baseada em informações computadas de sistemas de software.
3. Avaliação quantitativa e qualitativa das relações de coocorrência entre padrões de projeto e *bad smells*, por meio de um Estudo de Caso com cinco sistemas de software Java, que mostrou como resultado casos de coocorrências e situações reais que levaram a existência dessas relações nos sistemas analisados.

1.3 Artigo Publicado

Como resultado da pesquisa realizada para esta dissertação, até o momento, foi gerada a seguinte publicação:

- Sousa, B. L.; Bigonha, M. A. S.; Ferreira, K. A. M. *Evaluating Co-Occurrence of GOF Design Patterns with God Class and Long Method Bad Smells.* In proceedings of the Brazilian Symposium on Information Systems, Lavras, Brazil, pages 396–403, 2017.

1.4 Organização da Dissertação

Esta dissertação está organizada conforme descrito a seguir.

Capítulo 2 apresenta os principais conceitos que fundamentam este trabalho.

Capítulo 3 descreve a condução de uma revisão sistemática para investigar como a literatura tem abordado a relação entre padrões de projetos e *bad smells*.

Capítulo 4 apresenta a metodologia usada nesta pesquisa para a condução de um Estudo de Caso, enfatizando suas etapas principais, como: as estratégias de detecção de *bad smells*, o *data set*, a coleta de dados, as regras de associação e por fim, o método usado para a análise dos resultados.

Capítulo 5 descreve a ferramenta *Design Pattern Smell (DPS)*, desenvolvida para suportar a identificação de coocorrências entre padrões de projeto e *bad smells*. Nesse capítulo é descrita a abordagem utilizada para a implementação dessa ferramenta, suas principais funcionalidades, sua arquitetura, detalhes técnicos de sua implementação e sua avaliação.

Capítulo 6 apresenta o Estudo de Caso conduzido nesta pesquisa e relata os resultados desse Estudo de Caso, bem como a análise e discussão dos resultados.

Capítulo 7 conclui esta dissertação e apresenta algumas propostas de trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo apresenta os principais conceitos utilizados nesta dissertação. São eles: métricas de software orientado por objetos, valores referência para métricas de software orientado por objetos, padrões de projetos e *bad smells*.

2.1 Métricas de Software

Medição de software é uma prática da Engenharia de Software para avaliação de processo, projeto e produto. Pressman [2006] define três conceitos importantes para a realização dessa prática: métrica, medição e medida. Métrica é um padrão de medição utilizado para avaliar atributos internos de um software. De acordo com Sommerville [2011] os atributos internos podem ser usados para avaliar atributos externos tais como: modularidade, facilidade de manutenção, portabilidade, facilidade de uso, dentre outros. Medição é o ato ou efeito de medir algo de acordo com a métrica, ou seja, é o processo de realizar a coleta de uma determinada métrica em determinado software. Medida consiste no resultado obtido após o processo de medição. Uma medida indica quantitativamente a presença de um atributo interno em um determinado software.

Na literatura, as métricas de software têm sido utilizadas principalmente para identificar desvios de código que tornam o software mais complexo e que pode levar a ocorrência de falhas. Esses desvios são definidos por Fowler & Beck [1999] como *bad smells*. Alguns trabalhos têm focado nesse tema, disponibilizando expressões quantificáveis formadas por métricas de software para auxiliarem na detecção desses desvios em software orientados por objetos, as chamadas estratégias de detecção [Marinescu, 2002; Filó et al., 2015; Souza, 2016].

Muitos trabalhos propuseram métricas orientadas por objeto, e seus valores referência formando, assim, um extenso arcabouço na literatura. A seguir, o conceito

de valores referência de métrica de software é apresentado e algumas das principais métricas de software orientado por objetos propostas na literatura são descritas. Parte dessas métricas e seus respectivos valores referência são aplicados neste trabalho.

2.1.1 Valores Referência

Assim como as métricas, valores referência têm um papel muito importante no processo de medição de software. De acordo com Lanza & Marinescu [2006], valor referência consiste em dividir os valores de uma métrica em regiões, estabelecendo limites e gerando um sentido útil para a mesma. Crespo et al. [1999] abordam valor referência como uma forma de determinar a relação de uma métrica com um *bad smell*, a fim de identificar a ocorrência desses em sistemas de software.

Vários trabalhos têm dado atenção para derivações desses valores referências para metricas de software (Rosenberg et al. [1999]; Benlarbi et al. [2000]; Shatnawi et al. [2010]; Chhikara & Khatri [2011]; Ferreira et al. [2012]; Kaur et al. [2013]; Oliveira et al. [2014]; Filó et al. [2015]; Vale & Figueiredo [2015]). A principal diferença entre esses trabalhos é a metodologia utilizada para a derivação dos valores referências para as métricas de software.

Filó et al. [2015] definem um catálogo de valores referência para 18 métricas de software (Vide Seção 4.1). As estratégias de detecção de *bad smell* utilizadas neste trabalho são baseadas nesse catálogo.

2.1.2 Métricas de Software Orientado por Objetos

A seguir são descritas as principais métricas de software orientado por objetos na literatura.

- **Métricas CK:** esse conjunto de métricas orientadas por objetos foram propostas por Chidamber & Kemerer [1994]. Essas métricas têm como principal objetivo medir os atributos internos de um software em relação aos atributos externos de qualidade como manutenção e reutilização de software. Esse conjunto é formado por um total de seis métricas, sendo que há valores referência definidos para 4 delas [Filó et al., 2015].

  - WMC (*Weighted Methods per Class*): essa métrica está relacionada com a complexidade da classe levando em consideração os seus métodos. O cálculo dessa métrica considera a atribuição de pesos em cada método de uma respectiva classe. Esse peso pode considerar, por exemplo, tanto a

quantidade de linhas de código quanto a complexidade ciclomática de cada um dos métodos.

- DIT (*Depth of Inheritance*): essa métrica está relacionada com a estrutura de herança em um software orientado por objetos. Ela indica o nível em que uma respectiva classe está em uma árvore de herança. O cálculo dessa métrica é realizado em função da raiz da árvore, assim, quanto mais distante uma classe está da raiz da árvore, maior é o valor dessa métrica. Árvores de herança muito profundas são indicadores de estruturas complexas que prejudicam o entendimento de um módulo e que podem demandar refatoração.
- NOC (*Number of Children*): essa métrica indica a quantidade de classes filhas que uma classe possui no sistema.
- LCOM (*Lack of Cohesion Of Methods*): essa métrica está relacionada com coesão em software orientado por objetos. Ela calcula a ausência de coesão entre os métodos de uma classe. Coesão para Chidamber & Kemerer [1994] consiste na similaridade entre métodos. Pares de métodos que compartilham o mesmo atributo de sua respectiva classe são considerados similares. O cálculo dessa métrica é realizado pela diferença entre o número de pares de métodos que compartilham atributos e o número de pares de métodos que não compartilham atributos. Quanto maior o valor dessa métrica, menor é considerada a coesão interna da classe.
- CBO (*Coupling Between Object Classes*): essa métrica calcula a quantidade de classes que estão sendo chamadas, por meio de um relacionamento de associação, por uma classe. Para Chidamber & Kemerer [1994], um acoplamento entre duas classes pode acontecer quando métodos de uma delas usa métodos ou variáveis de instância da outra.
- RFC (*Response for Class*): essa métrica indica o número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe. O resultado dessa métrica é dado pela soma de métodos de uma classe e a quantidade de métodos invocados por cada método dela.

- **Métricas MOOD:** o conjunto de métricas MOOD foi proposto por Abreu & Carapuça [1994]. Essas métricas têm como principal objetivo avaliar aspectos em sistemas de software orientado por objeto, tais como: herança, encapsulamento, acoplamento, polimorfismo e reusabilidade de um software. Esse conjunto é composto por um total de oito métricas: Fator Herança de Método (*MIF - Method Inheritance Factor*), Fator Herança de Atributo (*AIF - Attribute Inheritance Fac-*

tor), Fator Acoplamento (*COF - Coupling Factor*), Fator Agrupamento (*CLF - Clustering Factor*), Fator Polimorfismo (*PF - Polymorphism Factor*), Fator Ocultação de Método (*MHF - Method Hiding Factor*) , Fator Ocultação de Atributo (*AHF - Attribute Hiding Factor*) e Fator Reúso (*RF - Reuse Factor*).

Apesar de possuir um conjunto formado por 8 métricas para software orientado por objetos, apenas uma métrica possui valor referência derivado na literatura, COF [Ferreira et al., 2012], descrita a seguir:

- COF (*Coupling Factor*): essa métrica é responsável pela avaliação de acoplamento. Abreu & Carapuça [1994] consideram essa métrica como relação cliente-servidor, onde existem duas classes, das quais uma é responsável por prover serviços e a outra é responsável por consumir serviços. O cálculo dessa métrica é realizado pela razão entre o número total de conexões existentes entre as classes dos software e o maior número possível de conexões para o software. Em caso de um software totalmente conectado, o valor dessa métrica é 1.

- **Métricas de Complexidade:** o conjunto de métricas de complexidade, como o próprio nome diz, está relacionado com a complexidade de projetos de softwares. Dentre as métricas existentes nessa categoria, destacam-se aquelas que possuem valores referências já definidos na literatura [Filó et al., 2015]. São elas:

  - MLOC (*Method Lines of Code*): define a quantidade de linhas de código existentes em cada método de uma classe.
  - SIX (*Specialization Index*): essa métrica consiste na razão do número de métodos sobrescritos, ponderados pela profundidade da classe na árvore de herança (DIT), sobre o número de métodos totais existentes na classe.
  - NBD (*Nested Block Depth*): essa métrica mede a profundidade de blocos aninhados em um método. Blocos aninhados ocorrem quando estruturas de controle como: *if*, *while*, *for* ocorrem uma dentro da outra. Essa métrica é um indicativo de complexidade, uma vez que o aumento de blocos aninhados torna o código mais difícil de ser compreendido.
  - VG (*McCabe Cyclomatic Complexity*): essa métrica tem como principal objetivo medir a quantidade de caminhos de execução independentes em um código fonte. Com isso, a cada estrutura de controle existente no código, *if*, *while*, *for*, dentre outras, essa métrica é acrescida em 1.

- **Métricas de Martin:** esse conjunto de métricas, proposto por Martin [1994], tem como principal objetivo medir o acoplamento existente entre pacotes em um projeto. Esse conjunto é composto por cinco métricas: Acoplamento Aferente, Acoplamento Eferente e Distância Normalizada. As três métricas descritas a seguir possuem valores referência definidos na literatura [Filó, 2014; Filó et al., 2015].

  - AC (*Afferent Coupling*): essa métrica indica o número de classes externas a um pacote que utilizam classes existentes dentro desse pacote.
  - EC (*Efferent Coupling*): essa métrica indica o número de classes dentro de um pacote que utilizam classes externas a ele.
  - RMD (*Normalized Distance*): essa métrica faz um balanceamento entre instabilidade e abstração, e indica, por meio de uma distância normalizada, o quão longe a relação está do ponto mais próximo de balanceamente entre instabilidade e abstração.

- **Métricas de Lorenz e Kidd:** esse conjunto de métricas foi proposto por Lorenz & Kidd [1994] e tem como principal objetivo avaliar questões estáticas em um projeto, tais como, herança e responsabilidade das classes. Esse conjunto de métricas divide-se em três categorias: tamanho, herança e aspectos internos das classes, compreedendo um total de 10 métricas. As métricas que compõe esse conjunto são:

  - NCA (*Number of Afferent Connections*): mede a quantidade de classes que estão usando os serviços de uma classe avaliada.
  - NMP (*Number of Public Methods*): mede o número de métodos públicos que uma classe possui. Essa métrica permite avaliar o tamanho de uma classe e a quantidade de serviços que ela possui.
  - NAP (*Number of Public Attributes*): mede o número de atributos públicos pertencentes a uma classe.
  - NOF (*Number of Attributes*): mede a quantidade de atributos pertencentes a uma classe.
  - NOM (*Number of Methods*): mede o total de métodos existentes em uma classe.
  - NORM (*Number of Overridden Methods*): mede a quantidade de métodos de uma classe que foram sobrescritas por sua subclasse.

  - NSC (*Number of Children*): mede o número de subclasses diretas de uma respectiva classe.
  - NSF (*Number of Static Attributes*): mede a quantidade de atributos estáticos em uma classe.
  - NSM (*Number of Static Methods*): mede o total de métodos estáticos existentes em uma classe.
  - PAR (*Number of Parameters*): mede o número total de parâmetros existentes em cada método de um projeto.

Existem valores referência definidos para as dez métricas de Lorenz & Kidd [1994] citadas [Ferreira et al., 2012; Filó et al., 2015].

2.2 Padrões de Projeto

De acordo com Gamma et al. [1994], padrões de projeto são soluções gerais para um problema recorrente de projeto de software em um dado contexto. Essas soluções permitem a criação de um software flexível, extensível e com alto grau de reúso. Essas estruturas são consideradas boas práticas de programação, uma vez que facilita o entendimento de um software, deixando-o menos complexo.

Trabalhos na literatura têm definido e estendido padrões de projeto existentes. Porém, os padrões mais conhecidos e utilizados pela comunidade acadêmica para o desenvolvimento de software são os que foram propostos por Gamma et al. [1994].

Gamma et al. [1994] construíram um catálogo formado por vinte e três padrões de projeto que ficou conhecido por GOF (*Gang of Four*). Os padrões de projeto existentes nesse catálogo são classificados em três categorias: criação, estrutura e comportamento.

A seguir, é apresentado uma descrição de cada uma dessas categorias e de cada padrão que as compõem.

- Criação: os padrões de projeto de criação têm como principal objetivo abstrair o processo de criação de objetos, ou seja, sua instanciação, de modo que o sistema não precise se preocupar com questões sobre como o objeto é criado, composto, e qual a sua representação real.
  - *Abstract Factory*: fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

  - *Builder*: separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.
  - *Factory Method*: define uma interface para criar um objeto, mas deixa as subclasses decidirem que classe instanciar. O *Factory Method* adia a instanciação para subclasses.
  - *Prototype*: especifica tipos de objetos a serem criados usando uma instância protótipo e cria novos objetos pela cópia desse protótipo.
  - *Singleton*: garante que uma classe tenha somente uma instância e fornece um ponto global de acesso a ela.

- Estrutura: os padrões estruturais se preocupam em como as classes e objetos são compostos, ou seja, como é a sua estrutura. Esses padrões têm como objetivo facilitar o projeto do sistema identificando maneiras de realizar o relacionamento entre entidade, deixando o desenvolvedor livre dessa preocupação.

  - *Adapter*: converte a interface de uma classe em outra interface, esperada pelo cliente. O objetivo desse padrão é permitir que interfaces imcompatíveis trabalhem em conjunto.
  - *Bridge*: desacopla, ou seja, separa a parte de abstração da implementação em um sistema, de modo que as duas possam variar independentemente.
  - *Composite*: compõe objetos em estruturas de árvore para representar hierarquias do tipo partes-todo. Esse padrão permite que os objetos sejam tratados de maneira uniforme e individual.
  - *Decorator*: dinamicamente, agrega responsabilidades adicionais a objetos. As classes que exercem o papel de *Decorators* nesse padrão fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.
  - *Façade*: fornece uma interface unificada para um conjunto de interfaces em um subsistema. Esse padrão de projeto define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.
  - *Flyweight*: usa compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina.
  - *Proxy*: fornece um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto.

- Comportamento: os padrões de projetos comportamentais atribuem responsabilidades a entidades. Este padrões facilitam a comunicação entre objetos, distribuindo a responsabilidade e definindo a comunicação interna.
  - *Chain of Responsibility*: evita o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Esse padrão une os objetos receptores, passando uma solicitação ao longo da cadeia até que um objeto a trate.
  - *Command*: encapsula uma solicitação como objeto, permitindo parametizar cliente com diferentes solicitações, enfileirar ou fazer registro de solicitações e suportar operações que podem ser desfeitas.
  - *Interpreter*: dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem.
  - *Iterator*: fornece um meio de acessar, sequencialmente, os elementos de um objeto agregado sem exportar sua representação subjacente.
  - *Mediator*: define um objeto que encapsula a forma como um conjunto de objetos interage. O mediador promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentes.
  - *Memento*: sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de forma que o objeto possa ser restaurado para esse estado mais tarde.
  - *Observer*: define uma dependência do tipo um para muitos entre objetos, de maneira que quando um objeto muda, todos os seus dependentes são notificados e atualizados automaticamente.
  - *State*: permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.
  - *Strategy*: define uma família de algoritmos, encapsula cada uma delas e torna-as intercambiáveis. O padrão de projeto *Statregy* permite que o algoritmo varie independentemente dos clientes que o utilizam.
  - *Template Method*: define um esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Esse padrão permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura dele.

– *Visitor*: representa uma operação a ser executada nos elementos de uma estrutura de objetos. Esse padrão permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Este trabalho considera 14 padrões de projeto *GOF*. São eles: *Adapter*, *Bridge*, *Command Composite*, *Decorator*, *Factory Method*, *Observer*, *Prototype*, *Proxy*, *State*, *Strategy Singleton*, *Template Method* e *Visitor*. Esses padrões de projeto foram escolhidos, porque eles podem ser coletados de forma automática com auxílio de ferramentas de detecção de padrões de projeto, minimizando assim, o esforço na coleta dessas informações e o risco de falsos positivos e negativos comparado a uma detecção manual.

2.3 *Bad Smell*

De acordo com Fowler & Beck [1999], o termo *bad smell* refere-se às características encontradas na estrutura de um software e que são indicativo de que ele pode estar com algum problema e necessita ser refatorado. Os *bad smells* não são considerados diretamente falhas, mas sua presença no código fonte de um software deixa-o suscetível a apresentar futuras falhas. Essas estruturas prejudicam a evolução e manutenção de um software, dado que o código fonte se torna mais complexo e sua legibilidade é afetada negativamente compromentendo o entendimento do software. Por esse motivo, é necessário saber como identificar essas estruturas para que possam ser removidas. Geralmente, técnicas de refatoração são utilizadas como forma de eliminação desses sintomas do código fonte.

Fowler & Beck [1999] descrevem em seu livro um conjunto de 22 *bad smells* existentes que podem afetar o código fonte de uma aplicação. Os autores expõem algumas características principais de cada sintoma que auxiliam na sua identificação, contudo, não são relatadas possíveis metodologias para realizar tal tarefa.

Os *bad smells* identificados por Fowler & Beck [1999] são:

- *Duplicated Code:* essa anomalia ocorre quando existem fragmentos de códigos idênticos presentes em mais um local no código fonte de um software.
- *Long Method:* são métodos longos que implementam várias funcionalidades tornando-se complexos e díficeis de serem entendidos. Esses métodos tendem a centralizar as funcionalidades uma classe.
- *Large Class:* são classes que realizam muitas tarefas e possuem muitas responsabilidades no sistema. Essa anomalia pode ser caracterizada como um objeto que sabe muito e possui muitas instâncias de variáveis.

- *Long Parameter List:* consiste em métodos com uma lista longa de parâmetros definidos como argumentos. É importante que um método possua poucos parâmetros para que seja fácil compreendê-lo e usá-lo.

- *Divergent Change:* define uma classe que muda frequentemente por muitos motivos. Geralmente essa anomalia é causada devido ao fato de a classe estar conectada a várias outras classes do sistema.

- *Shotgun Surgery:* são classes que ao serem alteradas impactam diversas outras classes do sistema, propagando mudanças para as demais.

- *Feature Envy:* essa anomalia ocorre quando métodos de uma classe estão mais interessados em métodos de outras classes e os utilizam em excesso. Quando esse sintoma ocorre é sinal de que existem métodos que deveriam ser removidos da classe.

- *Data Clumps:* são conjuntos de atributos ou parâmetros que são usados com frequência em conjunto.

- *Primitive Obsession:* refere-se ao uso de diversas variáveis primitivas para representação de tipos complexos de dados. Quando isso ocorre, deve-se usar classes para representação desses tipos complexos.

- *Switch Statements:* consiste em um uso excessivo de comandos *switchs* no código fonte que acaba se propagando e gerando duplicações.

- *Parallel Inheritance Hierarchies:* essa anomalia consiste no fato de que toda vez que é feita uma subclasse para uma classe do sistema, é necessário fazer uma subclasse para outra classe.

- *Lazy Class:* são classes pequenas que possuem poucas funcionalidades do sistema. Classes com esse sintoma geralmente têm suas funcionalidades incorporadas em outras classes e são eliminadas do sistema.

- *Speculative Generality:* essa anomalia ocorre quando é criada estrutura genérica para tratamento de problemas que possivelmente nunca ocorrerão no funcionamento do sistema.

- *Temporary Field:* são atributos existentes em um objeto que são acessados somente em algumas circunstâncias.

- *Message Chains:* refere-se à presença de diversas chamadas semelhantes no código fonte.
- *Middle Man:* ocorre quando um objeto é chamado e delega a maioria de suas funções para um objeto interno.
- *Inappropriate Intimacy:* essa anomalia ocorre quando uma classe utiliza atributos e métodos internos de outras classes.
- *Alternative Classes with Different Interfaces:* refere-se a métodos que possuem a mesma funcionalidade, porém estão definidos com assinaturas diferentes.
- *Incomplete Library Class:* ocorre quando programadores não consultam a documentação de uma biblioteca e acabam criando os seus próprios algoritmos.
- *Data Class:* consiste em uma classe que apenas armazena dados e não possui funcionalidades. Classes com a presença desse *bad smell* geralmente possuem apenas métodos *gets* e *sets*.
- *Refused Bequest:* ocorre em estruturas de herança, quando uma subclasse não utiliza funcionalidades e atributos que foram herdados de sua superclasse. Isso é um indício de que há algo de errado com a estrutura de hierarquia das classes.
- *Comments:* ocorre quando comentários são usados em um código fonte com o intuito de tentar melhorar os nomes dos atributos e métodos que não estão suficientemente expressivos.

Neste trabalho são considerados os *bad smells* para os quais há estratégias de detecção definidas e avaliadas na literatura [Souza, 2016]. São eles: *Data Class*, *Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*.

2.4 Considerações Finais

As métricas possuem um papel importante na avaliação da qualidade de software. Por meio delas, é possível realizar medições em um projeto de software para que seus atributos internos, tais como quantidade de linhas de código, complexidade ciclomática, dentre outros, possam ser avaliados. Por meio da medição dos atributos internos de software, é possível também avaliar atributos externos, tais como, modularidade, reusabilidade, dentre outros.

Apesar de possuir uma grande importância, os resultados das métricas por si só não são informações suficientes. É necessário utilizar uma interpretação adequada deles que indique o que cada valor significa. Os valores referência possuem essa finalidade. Eles dividem o espaço de valores de uma métrica em faixas de valores, os quais geram um sentido para o resultado de uma métrica. Métricas e seus valores referência utilizados em conjunto permitem uma melhor avaliação da qualidade de um software e podem direcionar esforços de testes, manutenções e refatorações para pontos críticos do projeto, garantindo uma boa qualidade ao produto final.

Padrões de projetos são soluções gerais disponibilizadas na literatura para problemas de contextos comum. Essas estruturas são consideradas boas práticas de programação e proporcionam ao software uma estrutura flexível e extensível permitindo um alto grau de reúso. Gamma et al. [1994] construíram um catálogo conhecido como *GOF* que é muito disseminado na literatura e na indústria. Esse catálogo é composto por 23 padrões de projetos, os quais são os mais conhecidos e utilizados em processos de desenvolvimento de software.

Bad Smell são conhecidos na literatura como sintomas presentes no código fonte que indicam a presença de alguma anomalia. O uso de métricas e valores referência têm sido muito utilizado para a detecção dessas estruturas. Em uma atividade de detecção de *bad smells*, por meio de medição, são utilizadas expressões lógicas compostas por uma ou mais métricas, pertinentes às características do *bad smell* a ser identificado, com um limiar atribuído a cada uma dessas métricas. Essas expressões são chamadas de estratégias de detecção.

Neste trabalho são investigadas coocorrências de *bad smells* e padrões de projeto. A identificação de *bad smells* é realizada por meio de estratégias de detecção propostas e avaliadas na literatura. Tais estratégias são baseadas em métricas de software e seus respectivos valores referência.

O Capítulo 3 apresenta uma revisão sistemática da literatura que aborda como as relações entre padrões de projetos e *bad smells*, em especial as relações de coocorrência, foram exploradas em trabalhos anteriores.

Capítulo 3

Revisão Sistemática

Este capítulo apresenta uma revisão sistemática da literatura com o objetivo de pesquisar e analisar estudos primários relevantes que envolvem relações entre padrões de projeto e *bad smells*. Essa revisão sistemática é utilizada como uma forma de entender o atual estado da arte sobre relações de padrões de projeto e *bad smells* e identificar possíveis desdobramentos e focos para outros trabalhos de pesquisa.

A Revisão Sistemática da Literatura surgiu como um mecanismo de agregação e construção de conhecimento para pesquisadores da área de Engenharia de Software [Kitchenham & Charters, 2007]. Kitchenham & Charters [2007] definem Revisão Sistemática da Literatura como um método de identificar, analisar e interpretar pesquisas referentes a um determinado assunto ou área específica ou tema de interesse. Além disso, Kitchenham & Charters [2007] ainda apresentam algumas motivações para realização desse estudo. São elas:

- resumir a tecnologia ou área em questão estudada, a fim de conhecer suas limitações e benefícios;
- descobrir algumas lacunas nos estudos de alguma tecnologia ou área de pesquisa que ainda não foi preenchida, e sugerí-las para pesquisas futuras;
- fornecer novas estruturas a fim de encaminhar novas áreas de investigação;
- estudar tecnologias e teorias, a fim de confirmar essas teses ou levantar novas hipóteses para estudos.

Em contraste com o processo não-sistemático, a revisão sistemática é feita de maneira formal e rigorosa [Biolchini et al., 2005]. Isso significa que o processo de

condução da pesquisa deve seguir um protocolo bem definido que, dentre outras coisas, estabelece uma sequência clara dos passos.

A revisão sistemática é definida a partir de uma questão central, que relata o objetivo principal da investigação e que é expressa por meio de termos e conceitos específicos referente a questão explorada [Biolchini et al., 2005]. Devido a esse rigor, uma das principais vantagens da revisão sistemática é permitir que outros pesquisadores consigam reproduzir o protocolo definido e, assim, julgar a adequação das decisões escolhidas para a pesquisa.

Os estudos que contribuem para a condução de uma revisão sistemática são chamados de estudos primários, e são utilizados para investigar uma questão de pesquisa específica [Kitchenham & Charters, 2007]. Um estudo secundário revisa os estudos primários procurando identificar e estabelecer conclusões por meio de registros resumidos que são comuns entre eles. Desse modo, uma revisão sistemática é considerada um tipo de estudo secundário.

O processo de uma revisão sistemática pode ser sumarizado em três etapas: planejamento, execução e análise dos resultados. A etapa de planejamento consiste na definição do objetivo e construção do protocolo com os procedimentos básicos a serem seguidos durante a execução do processo. A etapa de execução consiste na aplicação de todos os procedimentos definidos durante a etapa planejamento colocando-os em prática durante essa parte do processo. É nessa etapa que os estudos primários são buscados e filtrados por meio de critérios de inclusão e exclusão. Por fim, na etapa de análise dos resultados deve-se extrair e sumarizar os dados a fim de publicar os resultados.

Biolchini et al. [2005] sugerem um modelo para a condução da revisão sistemática. Esse modelo foi proposto com o objetivo de guiar pesquisadores de Engenharia de Software na construção de um planejamento sólido, e auxiliá-los a definir de forma clara cada etapa do processo e o conteúdo existente em cada seção do protocolo. Esse modelo consiste nas 5 etapas descritas a seguir.

1. Formulação de questões de pesquisa: consiste em definir, de forma clara, questões de pesquisa que retratem o objetivo da pesquisa. No fim, essas questões devem ser respondidas estabelecendo uma conclusão sobre o assunto estudado.

2. Seleção de fontes: consiste na definição do repositório em que a busca dos estudos primários serão realizadas, bem como na definição da *string* de busca, do idioma dos estudos primários, dentre outros aspectos.

3. Seleção dos estudos: consiste em definir os critérios de inclusão e exclusão e a forma como os estudos primários serão selecionados.

4. Extração de informação: consiste em aplicar os critérios de inclusão e exclusão nos estudos primários e detalhar o processo de seleção.

5. Sumarização dos resultados: consiste em apresentar e analisar as informações extraídas.

O restante do capítulo está organizado da seguinte forma: Seção 3.1 descreve o planejamento realizado para a condução deste estudo. Seção 3.2 relata a execução da revisão sistemática, e discute em detalhes o processo de busca e filtragem dos artigos identificados. Seção 3.3 analisa os resultados e resume os documentos selecionados nas etapas de filtragem. Seção 3.4 discute os trabalhos encontrados que servirão para responder as questões de pesquisa propostas neste estudo. Seção 3.5 apresenta as principais ameaças à validade desta revisão sistemática. Seção 3.6 conclui este capítulo destacando os principais achados desta revisão.

3.1 Planejamento da Revisão Sistemática

O planejamento é uma das fases mais importantes em uma revisão sistemática. De acordo com Biolchini et al. [2005], é nessa fase que ocorre a definição dos objetivos da pesquisa e a forma como a revisão será conduzida. Isso inclui a formulação de questões de pesquisa, definição da *string* de busca, dentre outros aspectos importantes para execução desse estudo.

Esta seção é destinada a discussão do planejamento da revisão sistemática realizada neste trabalho. A seção está subdividida da seguinte forma: Seção 3.1.1 aborda as questões de pesquisa utilizadas nesse trabalho com foco no tema de interesse; Seção 3.1.2 relata o processo de construção da *string* de busca; Seção 3.1.3 apresenta e justifica os repositórios utilizados na busca dos estudos primários; Seção 3.1.4 é dedicada à exposição dos critérios de inclusão e exclusão usados para a seleção dos estudos.

3.1.1 Questões de Pesquisa

As Questões de Pesquisa (QPn) propostas visam identificar e entender como a literatura tem abordado as relações entre padrões de projeto e *bad smells*, mais especificamente as relações de coocorrências entre eles.

Inicialmente foram propostas duas questões de pesquisa de propósito geral. Para diferenciá-las das questões de pesquisa globais deste trabalho enunciada no Capítulo 1, elas serão denominadas QP1(RSL) e QP2(RSL), onde RSL é uma referência a Revisão Sistemática da Literatura.

QP1(RSL): Como a literatura tem abordado a relação entre padrões de projeto e *bad smells*?

QP2(RSL): A literatura tem explorado coocorrências entre padrões de projeto e *bad smells*?

Durante a execução da revisão sistemática, estudos que abordam coocorrência entre padrões de projeto e *bad smells* foram encontrados. Por esse motivo, a questão de pesquisa QP2(RSL) foi subdividida em outras duas questões de pesquisa específicas, conforme mostrado a seguir.

QP2.1(RSL): Quais *bad smells* são abordados pela literatura na identificação de co-ocorrências com padrões de projeto?

Com essa questão de pesquisa pretende-se identificar e catalogar os *bad smells* utilizados por estudos anteriores em avaliações empíricas de coocorrências com padrões de projeto. Assim, por meio desse catálogo é possível identificar outros tipos de *bad smells* que podem ser investigados.

QP2.2(RSL): Quais padrões de projeto são usados pela literatura na identificação de coocorrências com *bad smells*?

Para responder essa questão de pesquisa pretende-se identificar os padrões de projeto explorados em coocorrências com *bad smells*. Desse modo, os padrões listados nessa questão de pesquisa podem ser usados para identificação de coocorrências com *bad smells* não listados na QP2.1(RSL), e vice versa.

3.1.2 *String* de Busca

Uma revisão sistemática da literatura exige uma ampla busca de estudos relevantes ao tema explorado. Para realizar essa busca, Kitchenham & Charters [2007] sugerem uma abordagem que consiste na divisão das questões de pesquisa em características individuais, e buscar sinônimos, abreviaturas e grafias alternativas para cada uma dessas características. Todos os termos obtidos referentes às questões de pesquisa

são ligados por expressões booleanas *AND* e *OR* resultando em uma expressão para pesquisa, chamada de *string* de busca.

String de busca é utilizada em base de dados eletrônicas, a fim de encontrar estudos primários. Ao executar tal expressão em bases eletrônicas, *engines* de busca identificam estudos que possuem em seus metadados, os termos existentes na expressão.

Apoiado nessa abordagem sugerida por Kitchenham & Charters [2007], uma *string* de busca foi formulada para busca de estudos primários nessa revisão. Inicialmente, as palavras chaves "padrão de projeto" e "*bad smell*" foram definidas como termos principais da expressão. Logo em seguida, foi realizada uma pesquisa de sinônimos desses termos, a fim de sofisticar essa expressão e identificar estudos relevantes e coerentes com as questões de pesquisa propostas. Os termos e seus sinônimos selecionados para a composição da *string* de busca são:

- *bad smell*: *bad smell*, *bad smells*, *code smell*, *code smells*, *anti pattern*, *antipatterns* e *anti-pattern*
- padrão de projeto: *design pattern* e *design patterns*

Os termos definidos foram agrupados pelas expressões booleanas *AND* e *OR*, a fim de finalizar a composição da *string* e gerar uma semântica que auxilie as bases eletrônicas na identificação de estudos que compreendam os termos estipulados. Nesta revisão sistemática, deseja-se localizar estudos que possuam em seu metadados, tanto termos referentes a padrão de projetos quanto a *bad smells*. Por esse motivo, o operador `OR` foi utilizado para conectar todos os sinônimos de cada um dos termos principais, enquanto o operador `AND` foi utilizado para conectar as próprias palavras chaves principais. No fim desse processo, foi gerada a seguinte *string* de busca.

> ("code smell" OR "code smells" OR "bad smell" OR "bad smells" OR "anti pattern" OR "antipatterns" OR "anti-pattern") AND ("design patterns" OR "design pattern")

Como um dos critérios de inclusão definido para esse estudo, Seção 3.1.4, foi a seleção de documentos escritos em inglês, a *string* de busca engloba apenas termos em inglês.

3.1.3 Fontes de Pesquisa

A coleta dos estudos primários foi realizada em bases de dados eletrônicas, disponibilizadas pelo portal de periódicos Capes[1]. As bases eletrônicas utilizadas para a pesquisa

[1] `http://www.periodicos.capes.gov.br`

dos estudos estão listadas na Tabela 3.1.

Base de Dados	Endereço
ACM Digital Library	`http://dl.acm.org/`
Compendex (Engineering Village)	`https://www.engineeringvillage.com`
IEEE	`http://ieeexplore.ieee.org/`
Science Direct	`http://www.sciencedirect.com/`
Scopus	`http://scopus.com/`
Springer	`http://link.springer.com/`
Web of Science	`http://webofknowledge.com/`

Tabela 3.1. Bases de dados eletrônicas utilizadas na Revisão Sistemática.

O motivo da escolha dessas bases é justificada pelo fato delas representarem uma espécie de biblioteca virtual com um grande acervo de trabalhos completos e metadados, registrados em *BibTex*, de pesquisas publicadas tanto em conferências quanto revistas de grande importância para a comunidade acadêmica.

A busca baseiou-se na extração de arquivos *BibTex* dos estudos retornados pelas bases eletrônicas. Esses arquivos foram utilizados nas etapas de filtragem (Seção 3.2.2) e na aplicação dos critérios de inclusão e exclusão, (Seção 3.1.4).

3.1.4 Critérios de Inclusão e Exclusão

Os critérios de inclusão e exclusão têm como principal objetivo auxiliar na condução do processo de filtragem dos estudos primários. Desse modo, é possível avaliar a relevância de um estudo em relação ao tema explorado e, consequentemente, eliminá-lo ou selecioná-lo para uma análise mais profunda.

Biolchini et al. [2005] destacam que as *engines* de busca utilizadas para pesquisa dos estudos primários nas bases eletrônicas podem retornar uma grande quantidade de artigos que não respondem as questões de pesquisa. Isso ocorre porque as palavras chaves utilizadas podem estar presente em artigos que não lidam com o tópico explorado, ou podem ser empregadas com significados diferentes. Assim, os critérios de inclusão e exclusão são aplicados para delimitar os trabalhos aos objetivos da revisão e filtrar aqueles que realmente são relevantes para o estudo. Os critérios de inclusão e exclusão usados na condução desta revisão sistemática são mostrados na Tabela 3.2.

Critérios de Inclusão
Documentos publicados em inglês.
Documentos completos.
Documentos publicados em Ciência da Computação.
Documentos disponíveis em formato eletrônico.
Documentos publicados em conferências e revistas.
Documentos relacionados aos termos da *string* de busca.
Critérios de Exclusão
Documentos classificados como tutoriais, pôsters, painéis, palestras, mesas redondas, oficinas, teses, dissertações, capítulos de livro e relatório técnico.
Documentos duplicados.
Documentos que não podem ser localizados.

Tabela 3.2. Critérios de Inclusão e Exclusão da Revisão Sistemática.

3.2 Execução da Revisão Sistemática

Esta seção é destinada a descrever as etapas da fase de execução, bem como os resultados finais retornados nessa fase. Esta seção está dividida da seguinte forma. Seção 3.2.1 descreve o processo de execução da *string* de busca nas bases eletrônicas. Seção 3.2.2 relata o processo de filtragem dos estudos primários retornados na busca inicial, assim como os resultados obtidos em cada etapa de filtragem realizada. Seção 3.2.3 apresenta uma lista de artigos obtidos após as etapas de filtragem. Tais estudos representam o resultado final dessa fase de execução e constituem a amostra de estudos analisada nesta revisão sistemática.

3.2.1 Processo de Busca

Esta etapa da fase de execução consiste na busca de estudos primários existentes nas bases de dados eletrônicas, via execução da *string* de busca.

As bases eletrônicas selecionadas nessa revisão sistemática (Seção 3.1) possuem uma espécie de *engine* de busca própria que automatizam a pesquisa de artigos. Desse modo, durante a pesquisa dos estudos primários, a *string* de busca foi fornecida a cada uma dessas bases por meio de uma interface gráfica *web*. Os resultados retornados foram exportados em arquivos *BibTex* para que pudessem passar por algumas etapas de filtragem (Seção 3.2.2). A base eletrônica *Springer* não permitiu a exportação de seus resultados em formato *BibTex*. Em função disso, seus resultados foram exportados em um *CSV (Comma-Separeted Values)* e, logo após, convertidos para o formato *BibTex*,

via um programa chamado `SPRINGER_CSV2BIB`[2].

O processo de busca foi realizado no período de 15 a 20 de janeiro de 2017. Portanto, estudos posteriores a essa data não foram considerados nesta revisão sistemática.

O resultado final dessa primeira pesquisa é exibido na Tabela 3.3. Ao todo, 795 documentos foram obtidos após a realização desse processo.

Base de Dados	**Estudos Retornados**
ACM Digital Library	12
Compendex (Engineering Village)	57
IEEE Xplore	0
Science Direct	176
Scopus	86
Springer	433
Web of Science	31
Total	**795**

Tabela 3.3. Estudos obtidos após o processo de busca.

A base `IEEE Xplore` não retornou resultados. Por esse motivo, estudos referentes a essa fonte de pesquisa não foram incluídos nessa revisão.

3.2.2 Processo de Seleção dos Estudos

Após finalizar a etapa de busca dos estudos primários, iniciou-se uma etapa destinada à filtragem desses documentos.

Como há um grande número de artigos identificados na fase de busca, o processo de filtragem é composta por cinco etapas. Cada etapa foi proposta com foco nos critérios de inclusão e exclusão e relevância do estudo de acordo com o seu conteúdo. As cinco etapas propostas para esse processo de filtragem estão descritas a seguir.

Etapa 1: consiste na eliminação de estudos duplicados. Como as bases eletrônicas funcionam como bibliotecas virtuais e possuem um grande acervo de trabalhos publicados em revistas e conferências importantes, bases diferentes podem retornar um mesmo estudo. Nesse sentido, trabalhos redundantes são descartados, reduzindo a quantidade de documentos para as próximas etapas.

Após a execução da Etapa 1, foram eliminados um total de 138 artigos, resultando em uma quantidade de 657 documentos a serem analisados na Etapa 2.

[2] `https://sourceforge.net/projects/springer-csv2bib/`

Etapa 2: consiste na eliminação de documentos que não sejam artigos. Assim, documentos classificados como tutoriais, pôsters, painéis, palestras, mesas redondas, oficinas, teses, dissertações, capítulos de livro e relatório técnico foram removidos nessa etapa.

Durante a análise dos 657 artigos iniciais, foi identificada uma grande quantidade de trabalhos que não são considerados artigos, em específico 384 documentos. Esses trabalhos foram removidos, e um total de 273 artigos foi, então considerado na Etapa 3.

Etapa 3: as duas etapas anteriores foram realizadas como forma de reduzir a quantidade de artigos para análise dos metadados. Feito isso, a Etapa 3 consiste em um tipo de filtragem destinada à seleção de artigos, por meio do conteúdo presente em seu título e resumo.

Os 273 títulos e resumos referentes a quantidade inicial dessa etapa foram analisados. Apesar dessas duas estruturas terem como objetivo fornecer ao leitor uma ideia do conteúdo abordado no documento, em alguns trabalhos não é possível perceber, por meio dessas duas estruturas, se eles são relevantes para a condução da revisão sistemática. Por esse motivo, ao analisar alguns estudos, houve uma certa dúvida sobre a seleção ou exclusão do mesmo. Portanto, como forma de evitar decisões precipitadas, estudos que apresentaram essa característica foram classificados como "duvidosos" e repassados à Etapa 4 para uma leitura mais aprofundada.

No fim dessa etapa, dos 273 estudos iniciais, 6 foram considerados relevantes para esta revisão sistemática e foram selecionados. No entanto, 17 estudos foram classificados como duvidosos e repassados à Etapa 4 para uma análise mais aprofundada.

Etapa 4: consiste em uma análise mais profunda dos artigos identificados como duvidosos na Etapa 3. Assim, foi realizada uma leitura diagonal desses estudos. Leitura diagonal consiste em uma leitura da introdução, tópicos e conclusão dos artigos, com o intuito de verificar se ele realmente está relacionado com as questões de pesquisa e garantir que seja um estudo relevante para a condução da revisão sistemática.

Essa etapa contou com uma quantidade inicial de 17 documentos. Ao término da análise, 5 artigos foram identificados como relevantes e integrados aos trabalhos já selecionados. Dos 17 artigos inicialmente submetidos à Etapa 4, 12 não se mostraram relevantes e foram excluídos.

Etapa 5: nesse momento do processo de seleção, é possível observar que todos os 795 estudos primários, obtidos no processo de busca, foram filtrados resultando em um

total de 11 estudos relevantes. Contudo, a pesquisa em bases de dados eletrônicas não garante retornar todos os estudos relevantes sobre um determinado tema. A fim de amenizar isso, um processo de *snowballing* foi realizado nesta etapa. De acordo com Wohlin [2014], *snowballing* é uma abordagem de busca que usa citações em documentos de uma revisão sistemática da literatura como lista de referência para identificar outros documentos que ainda não foram encontrados.

Via abordagem *Snowballing*, nessa fase foram analisados um total de 380 citações referentes aos 11 estudos já selecionados. Dessas citações, 5 foram identificadas como relevantes e integradas aos estudos já selecionados.

Sumarizando, as Etapas 1, 2, 3, 4 receberam como entrada, nessa ordem, 795, 657, 273 e 17 estudos primários; e a Etapa 5 recebeu 380 citações referentes aos 11 estudos já selecionados até o momento. A Figura 3.1 apresenta os resultados de cada uma das etapas do processo de seleção dos estudos, considerando os estudos primários recebidos como entrada em cada uma delas. No fim do processo de seleção, um total de 16 artigos foram considerados relevantes. Esses trabalhos foram analisados nesta revisão sistemática a fim de responder as questões de pesquisa propostas.

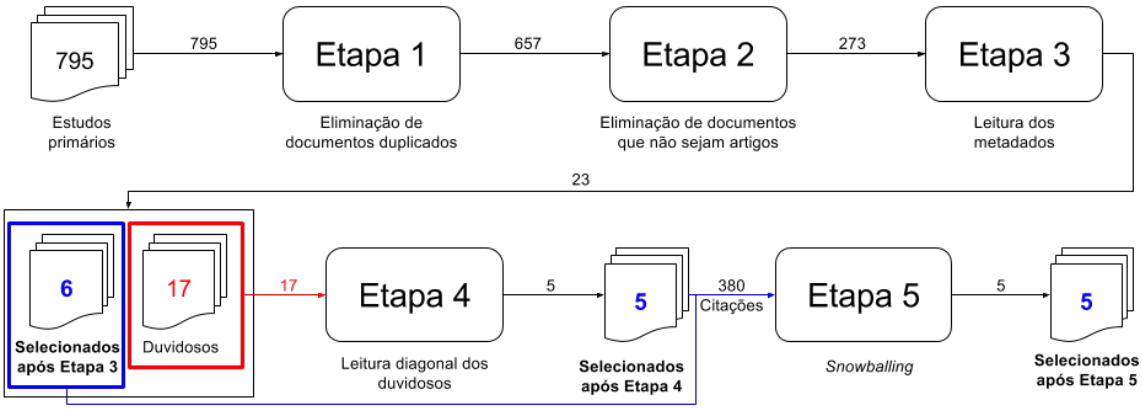

Figura 3.1. Processo de filtragem conduzido para seleção dos estudos.

3.2.3 Sumarização dos Resultados

Como mostrado na Seção 3.2.2, no fim da fase de seleção, foram encontrados um total de 16 estudos que abordam a relação entre padrões de projeto e *bad smells* e que se encaixaram no objetivo dessa revisão sistemática.

A Tabela 3.4 lista todos os 16 estudos primários selecionados. Os estudos estão ordenados pelo ano de publicação. Para cada estudo, foram extraídos título, autores,

local de publicação e a relação entre padrões de projeto e *bad smells* estabelecida neles. Durante a sumarização dos dados, foi percebido três tipos de relações estabelecidas: coocorrências, refatoração e impacto em qualidade de software.

3.3 Análise dos Resultados

Esta seção apresenta as principais ideias dos estudos selecionados. Nessa fase, todos artigos foram analisados e as suas principais e mais relevantes informações que auxiliam nas respostas das questões de pesquisa propostas foram extraídas.

Uma descrição de cada artigo é apresentada a seguir de acordo com as relações exploradas em cada um dos trabalhos. Esta seção está organizada da seguinte forma: Seção 3.3.1 descreve os estudos que exploram as relações de coocorrências entre padrões de projeto e *bad smells*; Seção 3.3.2 relata os estudos que aplicam os padrões de projeto como soluções de refatoração para determinadas estruturas com presença de *bad smells*; Seção 3.3.3 é destinada aos estudos que analisam o impacto dos padrões de projeto em relação a qualidade de software.

3.3.1 Relação de Coocorrência entre Padrões de Projeto e *Bad Smells*

Quatro dos 16 estudos estabelecem uma relação de coocorrência entre padrões de projeto e *bad smells*. Essa seção apresenta uma descrição desses estudos, seguindo a mesma ordem em que aparecem na Tabela 3.4.

Jaafar et al. [2013]

Jaafar et al. [2013] ressaltam que sistemas de software estão em constante mudanças. Em meio a essa mudança, o conhecimento inapropriado dos desenvolvedores pode ser um fator chave para introdução de *anti-patterns*. Sistemas grandes e de longa duração podem apresentar tanto presença de *anti-patterns* quanto de padrões de projeto no código fonte. Além disso, podem existir casos em que entidades dessas duas estruturas acabam tendo algum relacionamento. A partir desses fatos, Jaafar et al. [2013] analisaram a existência, a evolução e o impacto dos relacionamentos estáticos entre *anti-patterns* e padrões de projeto em sistemas de software. Durante a investigação, um estudo de caso foi realizado, a partir de *snapshots* de três sistemas Java de código aberto: ArgoUML, JFreeChart e XercesJ. Foram extraídas informações de padrões de projeto, *bad smells* e relações estáticas entre ambos. Por meio de uma tabela

Título	Autor	Local de Publicação	Relação
1. Assessment of Design Patterns During Software Reengineering: Lessons Learned from a Large Commercial Project	[Wendorff, 2001]	European Conference on Software Maintenance and Reengineering	Impacto em qualidade de software
2. Coupling of Design Patterns: Common Practices and Their Benefits	[McNatt & Bieman, 2001]	International Computer Software and Applications Conference	Impacto em qualidade de software
3. Defect frequency and design patterns: An empirical study of industrial code	[Vokac, 2004]	IEEE Transactions on Software Engineering	Impacto em qualidade de software
4. Do Design Patterns Impact Software Quality Positively?	[Khomh & Gueheneuce, 2008]	European Conference on Software Maintenance and Reengineering	Impacto em qualidade de software
5. Automated refactoring to the Strategy design pattern	[Christopoulou et al., 2012]	Information and Software Technology	Refatoração
6. Analysing Anti-patterns Static Relationships with Design Patterns	[Jaafar et al., 2013]	Electronic Communications of the EASST	Coocorrência
7. A multiple case study of design pattern decay, grime, and rot in evolving software systems	[Izurieta & Bieman, 2013]	Software Quality Journal	Impacto em qualidade de software
8. Code Quality Cultivation	[Speicher, 2013]	Communications in Computer and Information Science	Impacto em qualidade de software
9. Automated pattern-directed refactoring for complex conditional statements	[Liu et al., 2014]	Journal of Central South University	Refatoração
10. Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory	[Nahar & Sakib, 2015]	Central Europe CEUR Workshop Proceedings	Refatoração
11. A proposal of software maintainability model using code smell measurement	[Wagey et al., 2015]	International Conference on Data and Software Engineering	Impacto em qualidade de software
12. Co-Occurrence of Design Patterns and Bad Smells in Software Systems: An Exploratory Study	[Cardoso & Figueiredo, 2015]	Brazilian Symposium on Information Systems	Coocorrência
13. ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-patterns	[Nahar & Sakib, 2016]	IEEE International Conference on Software Analysis, Evolution and Reengineering	Refatoração
14. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults	[Jaafar et al., 2016]	Empirical Software Engineering	Coocorrência
15. The relationship between design patterns and code smells: An exploratory study	[Walter & Alkhaeir, 2016]	Information and Software Technology	Coocorrência
16. Automated refactoring of super-class method invocations to the Template Method design pattern	[Zafeiris et al., 2017]	Information and Software Technology	Refatoração

Tabela 3.4. Resultado final da fase de execução da Revisão Sistemática.

de contingência, teste exato de *Fisher* e *Odds ratio* (Sheskin [2007]), os dados foram analisados e a significância estatística das relações investigadas foram verificadas.

Jaafar et al. [2013] concluíram que (i) padrões de projeto podem ter diferentes proporções de relações estáticas com *anti-patterns*, (ii) o padrão de projeto *Command* foi apontado como aquele que apresentou maior relação com os *anti-patterns* investigados, (iii) as relações existentes entre essas duas estruturas não são casuais, visto que estão em constante crescimento no decorrer da evolução do projeto, e por fim, (iv) classes que participam de relações estáticas entre *anti-patterns* e padrões de projeto são mais propensas à mudanças e menos propensas a falhas em relação a classes que possuem a presença de *anti-pattern*, mas não participam dessa relação.

Cardoso & Figueiredo [2015]

Cardoso & Figueiredo [2015] relatam a importância dos padrões de projeto na literatura, e destacam que essas soluções são consideradas boas práticas de programação e encorajam a construção de estruturas flexíveis e reusável. Contudo, os autores identificaram estudos anteriores que apontam relações dessas soluções com *bad smells*, originadas principalmente da aplicação inapropriada dos padrões de projeto. Baseado nesses indícios, os autores realizaram um estudo exploratório, a fim de identificar coocorrências entre padrões de projeto e *bad smells*. Esse estudo concentrou-se em dois tipos de análise. A primeira delas apoia-se na identificação de possíveis padrões de projeto que podem coocorrerem com *bad smells*. Para conduzir essa primeira análise, Cardoso & Figueiredo [2015] extraíram dados referentes a padrões de projeto e *bad smells* de cinco projetos de software, e aplicaram regras de associação para detectar possíveis relações entre essas estruturas. A segunda análise fundamenta-se na descoberta de fatos que explicam o surgimento dessas relações.

No fim do estudo, Cardoso & Figueiredo [2015] identificaram coocorrências entre (i) *Command* com *God Class* e (ii) *Template Method* com *Duplicate Code*. Ao analisar as entidades que apresentaram essas duas estruturas, os autores concluíram que o uso excessivo de uma simples classe receptora na aplicação do padrão *Command* para diferentes interesses ocasionou o surgimento do *bad smell God Class*. No caso do padrão de projeto *Template Method*, as várias duplicações de implementações foram responsáveis pela sua coocorrência com o *bad smell Duplicate Code*.

Este trabalho de dissertação realiza uma extensão do estudo de Cardoso & Figueiredo [2015], explorando coocorrências dos padrões de projeto *GOF* com outros tipos de *bad smells*. Além disso, este trabalho de dissertação utiliza uma abordagem diferente para detecção dos *bad smells* nos sistemas de software analisados.

Jaafar et al. [2016]

Jaafar et al. [2016] discutem que estudos anteriores têm apontado que padrões de projeto podem correlacionar-se com estruturas complexas e resultarem em ocorrências de falhas. Além disso, existem relatos de propagação de problemas em classes com padrões de projeto e *bad smells* que possuem dependências estáticas ou de comudança com outras classes. Dependência de comudança é definida pelos autores como alteração em uma classe que impacta diretamente na alteração de outra. Diante desse achado, os autores decidiram investigar o impacto de tais dependências em sistemas orientados por objetos e analisar a relação delas com (i) propensão a falha, (2) tipos de trocas, e (iii) tipos de falhas que as classes exibem.

Para conduzir essa investigação, foram extraídos dados referentes a padrões de projeto, *anti-patterns*, relações estáticas de padrões de projeto e *anti-patterns*, relações de comudança de padrões de projeto e *anti-patterns* e falhas em classes. Tais informações foram obtidas de *snapshots* de três sistemas Java de código aberto: ArgoUML, JFreeChart e XercesJ por meio do teste estatístico de *Fischer* e *Odds ratio* [Sheskin, 2007]. Jaafar et al. [2016] examinaram a significância das relações entre a ocorrência de uma dependência estática ou comudança em classes com padrões de projeto ou *anti-patterns* e o risco de falha.

Como principais resultados, os autores concluíram que (i) classes que possuem relacionamento estático ou de comudança com classes *anti-patterns* têm significativamente mais falhas que outras classes, (ii) classes que têm dependências de comudança com classes de padrões de projeto possuem significativamente mais falhas que outras classes, porém, menos falhas quando o relacionamento é estático, (iii) mudanças estruturais são mais propensas a ocorrerem em classes que têm dependências com *anti-patterns* do que em outras classes, (iv) adições de código são mais propensas a ocorrerem em classes que possuem dependências com padrões de projeto do que em outras classes, e (v) tipos específicos de falhas são mais predominantes em certos *anti-patterns*, como por exemplo, *Blob Class* e *Complex Class*, que propagam principalmente falhas lógicas.

Walter & Alkhaeir [2016]

Walter & Alkhaeir [2016] realizaram um estudo exploratório com o objetivo de (i) determinar se e como a presença dos padrões de projeto estão relacionados à presença de *bad smells*, (ii) investigar se e como a presença dessas relações mudam ao longo da evolução do código, e (iii) identificar relações entre padrões de projeto e *bad smells*. Para atingir esses objetivos, os autores realizaram uma análise da evolução de dois sistemas, Apache Maven e JFreeChart. No total, foram analisadas 87 versões, sendo

32 referentes ao primeiro sistema e 55 referentes ao segundo. Além disso, o estudo foi conduzido com 7 tipos diferentes de *bad smells* e nove padrões de projeto do catálogo GOF [Gamma et al., 1994]. As informações referentes a instâncias de padrões de projeto e *bad smells* nesses projetos foram extraídas e, por meio do teste de hipótese, teste de tendência não paramétrica (*Mann-Kendall*) e regras de associação, os autores realizaram a análise das informações extraídas, a fim de chegar a uma conclusão para os objetivos i, ii e iii, respectivamente.

Os principais resultados obtidos nesse estudo mostraram que a presença de padrões de projeto está relacionada com a ausência de *bad smells* em uma mesma classe. Em outras palavras, uma classe que faz uso de padrão de projeto tende a não apresentar *bad smells*. Além disso, alguns padrões de projeto foram apontados como mais propensos a não apresentarem *bad smells*. É o caso dos padrões: *State-Strategy*, *Adapter-Command*, *Factory Method* e *Singleton*. No entanto, o padrão *Composite* apresentou uma relação mais forte com a presença de *bad smells*, mostrando-se uma exceção a essa tendência. Segundo os autores, os resultados obtidos em relação ao padrão *Singleton* foram surpreendentes, devido ao fato de outras pesquisas apontarem o contrário.

Por fim, sobre o objetivo ii, os autores concluem que durante a evolução de um projeto, a presença de *bad smells* em classes com padrões de projeto não é maior do que na fase de criação do software. Portanto, esse resultado implica no fato de que sistemas de software já são projetados com *bad smells*.

3.3.2 Relação de Refatoração

Cinco dos 16 estudos estabelecem uma relação de refatoração entre padrões de projeto e *bad smells*. Nesses estudos, padrões de projeto são propostos como possíveis soluções para eliminar ocorrências de *bad smells* ou estruturas complexas que prejudiquem a qualidade de um software. Essa seção apresenta uma descrição desses estudos seguindo a mesma ordem em que aparecem na Tabela 3.4.

Christopoulou et al. [2012]

Christopoulou et al. [2012] propuseram uma abordagem de refatoração de código com o *bad smell Complex Conditional Statements*, a partir da aplicação do padrão de projeto *Strategy*. O *bad smell Complex Conditional Statements* foi relatado por Fowler & Beck [1999] como estruturas condicionais complexas que se propagam ao longo do código fonte de um software, reduzindo sua legibilidade e tornando-o mais complexo. Essas estruturas condicionais podem ser substituídas por uma hierarquia de herança, na qual é feito o uso de polimorfismo, deixando o código mais modularizado e flexível a

modificações. O padrão de projeto *Strategy*, proposto por Gamma et al. [1994], utiliza essa hierarquia em sua estrutura e, por isso, foi uma escolha de Christopoulou et al. [2012] como forma de combate a esse *bad smell*.

Na implementação proposta, Christopoulou et al. [2012] criaram dois tipos de algoritmos. O primeiro consiste na identificação das estruturas anômalas em um código fonte Java. O segundo consiste na aplicação do padrão de projeto *Strategy* para transformar as estruturas condicionais complexas em uma hierarquia de herança, substituindo o uso de condicionais por polimorfismos. Essa abordagem foi implementada no *plugin JDeodorant* da *IDE Eclipse* como suporte à refatoração automática em projetos Java.

Após a implementação, os autores realizaram um experimento envolvendo 8 projetos de software Java, dos quais seis possuem código aberto e dois possuem código proprietário. Para avaliação dessa abordagem, foram utilizadas as métricas: *precision* e *recall*. Os resultados da avaliação mostraram uma boa eficácia em relação a qualidade, dado que quase metade das refatorações sugeridas foram significativas. Além disso, a eficiência quanto ao tempo foi outro fator relevante e bastante satisfatório. O tempo de processamento do algoritmo não excedeu 30 segundos para projetos de tamanho médio e 2,5 minutos para projetos de grande porte.

Liu et al. [2014]

Liu et al. [2014] discutem sobre a importância de evitar o uso de declarações condicionais complexas no código fonte de um projeto. Se um programa utiliza essas declarações em grande escala, há uma grande probabilidade do *bad smell Switch Statements* [Fowler & Beck, 1999] estar presente na estrutura desse programa. Uma forma de eliminá-lo é via a aplicação de técnicas de refatoração, na qual uma hierarquia de herança é criada e polimorfismos são introduzidos no lugar das estruturas condicionais.

Apesar de muitos desenvolvedores terem conhecimento da importância da refatoração para a garantia de qualidade, ainda há subuso dessa técnica. Assim, Liu et al. [2014] propuseram uma abordagem de refatoração usando os padrões de projeto, *Factory Method* e *Strategy*, que consiste em (i) identificar oportunidades de refatoração no código, ou seja, pontos em que há estruturas condicionais, e (ii) aplicar a refatoração nesses pontos usando o padrão *Factory Method* ou *Strategy*. Nessa abordagem foram construídos 4 algoritmos, sendo dois para identificação de oportunidades de refatoração e dois para aplicação da refatoração.

Na avaliação dessa abordagem, foram utilizadas as seguintes métricas: *precision*, *recall* e *accuracy*. Pelos resultados, a abordagem de refatoração tanto pelo padrão *Factory Method* quanto pelo padrão *Strategy*, mostraram-se eficientes em relação à

redução de complexidade ciclomática e quantidade de linhas de código de métodos. Porém, foram encontrados alguns casos de falsos positivos e falsos negativos, originados de algumas deficiências na abordagem. Os autores pretendem corrigir esses casos em trabalhos futuros.

Nahar & Sakib [2015]

Nahar & Sakib [2015] acreditam que a escolha do padrão de projeto poderia ser facilitada caso fosse realizada em conjunto com uma ferramenta de recomendação. Em função disso, os autores criaram uma ferramenta, baseada em diagramas de classes *UML*, que incorpora tanto a detecção de *anti-patterns* quanto a recomendação de padrões de projeto na fase de projeto do software. *Anti-pattern based Design Pattern Recommender (ADPR)* é composta de duas fases. A primeira fase consiste na análise de possível existência de *anti-patterns* no software. A segunda fase realiza a detecção dos *anti-patterns* existentes e recomenda soluções compostas por padrões de projeto que eliminem os *anti-patterns* detectados.

Os autores realizaram um estudo de caso para avaliar recomendação de soluções com o padrão projeto *Abstract Factory*. O estudo de caso, foi utilizado um software de código aberto desenvolvido na linguagem Java, chamado *Painter*. Esse software foi escolhido porque a má aplicação do padrão de projeto *Abstract Factory* levou a ocorrência de *anti-patterns*. Nessa avaliação, a efetividade da abordagem proposta foi comprovada, o que lhe permitiu a execução de um experimento com um número maior de software e a comparação de ADPR com outros tipos de ferramentas.

Em uma segunda avaliação, os autores compararam a ferramenta proposta com uma ferramenta baseada em código. Foram utilizados cinco projetos Java de código aberto, os quais se encontram hospedados no *Github*. Na comparação dos resultados, os autores concluíram que a ADPR atingiu uma alta precisão, identificando todos os pontos em que a aplicação do padrão *Abstract Factory* foi degradada e culminou na presença de *anti-patterns*. A ferramenta baseada em código não teve um bom desempenho e apresentou ocorrências de falsos negativos.

Nahar & Sakib [2016]

Nahar & Sakib [2016] propuseram uma ferramenta para recomendação de padrões de projeto de criação na fase de projeto de um software. Essa ferramenta utiliza características de *anti-patterns* para realizar as possíveis recomendações. *Anti-patterns* nesse estudo são considerados pelos autores como estruturas de código fonte nas quais houve perda de efetividade dos padrões de projeto. Por exemplo, uma estrutura com um grupo de classes instanciadas diretamente sem uso de uma fábrica para sua instanciação

é considerada uma perda do padrão de projeto *Abstract Factory*. Cada padrão de projeto de criação possui características que impulsionam sua indicação como possível candidato a refatoração. Um sistema de pontuação foi incluso na abordagem para classificar as estruturas passíveis de refatoração e indicar um tipo de padrão de projeto mais propenso a ser aplicado nessa atividade.

A ferramenta foi implementada em Java. Os autores utilizaram um *data set* composto por 21 projetos de código aberto desenvolvido também na linguagem de programação Java para avaliação da ferramenta. Para essa avaliação, foram utilizadas as métricas: *precision*, *recall* e *f-measure*, para analisar a eficácia das recomendações. Os resultados desse experimento foram considerados satisfatórios, dado que a ferramenta não apresentou casos de falsos positivos e apenas um único caso de falso negativo foi retornado.

Zafeiris et al. [2017]

Zafeiris et al. [2017] destacam que a herança é o mecanismo utilizado na programação orientada por objetos e suporta a implantação de polimorfismo no desenvolvimento de um projeto de software. Esses recursos permitem a criação de um software modular e flexível. Contudo, a aplicação desse mecanismo deve ser especificada e implementada com cuidado para não levar a ocorrência de um *bad smell* denominado por Fowler [2015] como *Call Super*. Esse *bad smell* consiste em uma extensão de comportamento de um método concreto, por meio do uso da palavra chave `super`. Gamma et al. [1994] recomendam o uso do padrão *Template Method* para extensão controlada do comportamento de um método.

Assim, os autores sugerem uma técnica de refatoração que visa utilizar o padrão de projeto *Template Method* para eliminar estruturas com o sintoma de *Call Super* existente no código fonte. Inicialmente, os autores propuseram um algoritmo que realiza análise estática no código fonte de projetos Java e identifica oportunidades de refatoração. Esse algoritmo converte todas as classes de um projeto em uma árvore sintática abstrata (AST) e analisa as instâncias de métodos que incluem invocações de `super`. Após essa fase inicial de identificações, outro algoritmo é aplicado, a fim de realizar as transformações no código fonte. Esse segundo algoritmo é composto de sete passos que especificam a introdução do padrão de projeto *Template Method* nas oportunidades identificadas. A abordagem proposta foi implementada como uma extensão para o *plugin JDeodorant* da *IDE Eclipse*.

Para a avaliação, Zafeiris et al. [2017] utilizaram um conjunto composto de 12 projetos de código aberto, desenvolvidos em Java. A abordagem proposta foi testada nesse *benchmark*, e por meio de uma análise experimental, os autores constataram

que o algoritmo de identificação sugeriu 20,5% de instâncias candidatas a refatoração. Segundo os autores, grande parte das refatorações rejeitadas era referente a casos triviais ou semanticamente irrelevantes, tornando, assim, seu comportamento satisfatório. Além disso, a aplicação das refatorações nas oportunidades identificadas impactou na redução do Índice de Especialização (SIX) nas subclasses afetadas. Por fim, a abordagem mostrou-se escalável, consumindo um tempo de execução entre 7 a 25 segundos para projetos de pequeno a médio porte, e não excedendo 1 minuto para projetos de grande porte.

3.3.3 Relação de Impacto em Qualidade de Software

Sete dos 16 estudos estabelecem uma relação de impacto em qualidade de software. Esses estudos realizam análises empíricas nas quais são avaliados aspectos e questões relacionadas a aplicação de padrões de projeto, que podem melhorar a qualidade de um software ou gerar impactos que degradem a estrutura do padrão. Esta seção apresenta uma descrição desses estudos, seguindo a mesma ordem em que aparecem na Tabela 3.4.

Wendorff [2001]

Wendorff [2001] avaliou a aplicação dos padrões de projeto em um software comercial a fim de descobrir possíveis impactos negativos gerados por essa técnica. A condução dessa análise foi realizada por um engenheiro de software com experiência profissional de oito anos. Foi utilizado um software proprietário desenvolvido na linguagem C++. No estudo, Wendorff [2001] percebeu que essa técnica pode gerar alguns impactos negativos. Segundo ele, os primeiros resultados extraídos dessa análise refere-se ao mau uso dessas soluções por parte dos desenvolvedores, devido à não compreensão da lógica envolvida na sua implementação. Outros resultados interessantes consistem no fato da aplicação de padrões de projeto não se enquadrarem aos requisitos exigidos pelo projeto. Em outras palavras, o não enquadramento dessas técnicas pode ser representada por situações como: superestimativa e mudança de requisitos, bem como a aplicação da solução sem necessidade.

No fim do estudo, o autor desenvolveu um procedimento simples para amenizar os impactos negativos causados e guiar o desenvolvedor na remoção de padrões inapropriados do código fonte. Esse guia é composto de sete passos: (i) identificação de atributos de qualidade relevantes para o software, (ii) identificação dos padrões utilizados no código, (iii) tentativa de reconstrução do raciocínio existente por trás do padrão, (iv) avaliação do benefício concreto, (v) avaliação do custo extra concreto, (vi) avaliação do esforço necessário para remoção e (vii) aplicação de uma decisão para

remoção do padrão. Esse roteiro foi avaliado durante o processo de reengenharia do software analisado. De acordo com o autor, as decisões ficaram mais objetivas, bem documentadas e mais sólidas. No entanto, a decisão de remoção ainda permaneceu subjetiva em alguns lugares.

McNatt & Bieman [2001]

McNatt & Bieman [2001] destacam que no processo de desenvolvimento de um software, padrões de projeto podem relacionar-se entre si gerando acoplamentos de padrões. Esse tipo de acoplamento é definido pelos autores como dois padrões de projeto diferentes que possuem pelo menos uma classe em comum.

Nesse estudo, os autores decidiram avaliar o impacto dessas estruturas no contexto de qualidade de software por meio de trabalhos que relatam ocorrências de acoplamento de padrões. O conjunto de artigos selecionados para análise contou com 16 documentos, distribuídos entre aplicações industriais e estudos analíticos. Esses documentos foram divididos em quatro categorias: estudo analítico puro, estudo analítico com exemplos sintéticos, estudos de caso da indústria considerando código existente e estudo de caso da indústria de novos projetos. Os tipos de acoplamento também foram classificados em: *loosely*, referentes a padrões com poucas conexões e mais simples de serem modificados no futuro, e *tightly*, referente a padrões fortemente conectados e com muitas dependências, onde pequenas mudanças em um padrão podem gerar impacto nos outros.

Ao analisar o impacto dessas relações, os autores concluíram que: (i) o tipo de acoplamento *tightly* resulta em um projeto difícil de ser modularizado, dificultando a sua modificação; (ii) o tipo de acoplamento *loosely* é mais flexível e uma possível modificação nesse tipo de estrutura não gera um impacto grande comparado ao acoplamento *tightly*; (iii) o padrão de projeto *Singleton* apresentou uma tendência prejudicial ao atributo de qualidade modularidade, devido ao fato de terem sido encontradas instâncias de acoplamento *tightly* envolvendo esse padrão.

Vokac [2004]

Vokac [2004] argumenta que, apesar de existir uma grande aceitação dos padrões de projeto por parte de pesquisadores e profissionais, alguns estudos mostram que essas soluções podem gerar defeitos em um software. Em função disso, o autor se propôs a investigá-los. Assim, foi realizado um estudo de caso com um software proprietário, desenvolvido na linguagem C++, chamado *SuperOffice CRM5*. O proprietário desse produto forneceu ao autor do estudo acesso completo ao código fonte e ao histórico de versões do software. Durante a investigação, *snapshots* referentes ao período de três

anos foram analisados. Para a extração das instâncias de padrões de projeto, Vokac [2004] construiu uma ferramenta própria que suporta a identificação de cinco padrões de projeto proposto por Gamma et al. [1994]: *Singleton*, *Template Method*, *Decorator*, *Observer* e *Factory Method*. Os defeitos existentes no projeto foram extraídos por meio de uma integração do sistema controle de versão (CSV) com o sistema de detecção de defeitos, e uma análise textual dos comentários no CSV para recuperação de defeitos que não possuíam ligação direta entre os dois sistemas.

A análise dos dados foi realizada com um modelo de regressão logística [Kleinbaum, 1994]. O autor descobriu algumas correlações significativas para os padrões explorados. O padrão *Factory* apresentou uma taxa mais baixa de defeitos. Por outro lado, o padrão *Observer* foi correlacionado com maiores taxas de ocorrências de defeitos. O padrão *Template Method* apresentou uma característica inconclusiva no estudo, dado que o mesmo ocorreu em diversos contextos diferentes. O padrão *Decorator* não apresentou resultados estatisticamente significativos, devido a sua baixa frequência de ocorrência. Outro resultado interessante desse estudo foi que a combinação do padrão *Singleton* com *Observer* tendem a ser utilizados em áreas complexas, com mais código e maior frequência de defeitos. Por isso, o autor concluiu que tanto *Singleton* quanto *Observer* tendem a associar-se com complexidade dentro de um software.

Khomh & Gueheneuce [2008]

Khomh & Gueheneuce [2008] levantam uma hipótese de que padrões de projeto podem não melhorar os atributos de qualidades como esperado e, além disso, podem gerar impactos negativo. A fim de confirmar a hipótese levantada, os autores elaboraram um *survey* com o objetivo de fornecer evidências sobre o impacto dos padrões de projeto na qualidade de software. Nesse *survey*, foram considerados os atributos: expansibilidade, simplicidade, reusabilidade, aprendizado, compreensibilidade, modularidade, generalização, modularidade em tempo real, escalabilidade e robustez. Os padrões de projeto utilizados nesse estudo foram os 23 padrões propostos por Gamma et al. [1994]. O questionário utilizado para aplicação do *survey* foi construído com base nesses dados. Assim, para cada padrão de projeto, a pessoa deveria classificar o impacto que aquele padrão de projeto tem sobre cada um dos atributos de qualidade. A escala utilizada na classificação é composta por 6 tipos de respostas diferentes: muito positivo, positivo, não significante, negativo, muito negativo e não aplicável. Esse questionário foi aplicado durante o período de Janeiro a Abril de 2007 e, após esse período, os autores iniciaram a análise das respostas e aplicaram um teste de hipótese e um teste estatístico chamado distribuição de Bernoulli.

Khomh & Gueheneuce [2008] concluíram que nem sempre o uso de padrões

de projeto melhora a qualidade dos sistemas. Alguns padrões de projeto, como o *Flyweight*, por exemplo, decrementam alguns atributos, impactando de forma negativa na qualidade do software. Por esse motivo, é necessário certo cuidado no uso dessas soluções durante o desenvolvimento, dado que eles podem prejudicar a manutenção e evolução do software.

Izurieta & Bieman [2013]

Izurieta & Bieman [2013] relatam que padrões de projeto podem degradar-se com o tempo. A demanda de requisitos e as manutenções realizadas no projeto podem implicar na inclusão de responsabilidades para as classes e em modificações na estrutura do projeto, que aumentam a complexidade do código fonte e fazem com que possíveis padrões de projeto existentes no código fonte percam sua efetividade. Devido a essas possibilidades, os autores desse estudo propuseram uma investigação para entender até que ponto, na evolução de um software, os padrões de projeto mantém sua estrutura flexível e fácil de manter. Ainda, foi investigado se os sistemas mantêm os níveis iniciais de qualidade.

Um estudo de caso múltiplo foi realizado com três projetos de software de código aberto desenvolvido em Java. Nesse estudo de caso, foram obtidas versões de certo período de tempo desses projetos para que a efetividade dos padrões de projeto pudesse ser avaliada no decorrer da evolução de cada projeto. Para extração dos padrões de projeto, foram utilizadas as ferramentas: *Design Pattern Finder* e *PatternSeeker Tool*. Os padrões de projeto avaliados foram os seguintes: *Factory Method*, *Adapter*, *Singleton*, *State*, *Iterator*, *Proxy* e *Visitor*. Na condução desse estudo, os autores (i) identificaram as instâncias dos padrões de projeto existentes em cada uma das versões dos projetos de software, (ii) realizaram a engenharia reversa do diagrama *UML* dos padrões de projeto para verificar as relações das entidades, (iii) mineraram as versões dos software, a fim de capturar informações sobre a perda ou não da efetividade dos padrões de projeto, (iv) analisaram os resultados obtidos e (v) avaliaram doze preposições elaboradas no início dessa avaliação.

Izurieta & Bieman [2013] não identificaram evidências de decomposição da integridade estrutural dos padrões de projeto nesses sistemas. Contudo, foi encontrado uma considerável evidência de decadência de padrões devido ao acúmulo de artefatos não relacionados a classes que desempenham papéis nos padrões. As dependências entre componentes tiveram um aumento, reduzindo assim, a modularidade, a testabilidade e a adaptabilidade dos sistemas. Tais ocorrências estão diretamente ligadas ao aumento do acoplamento na evolução desses projetos.

Speicher [2013]

Speicher [2013] argumenta que padrões de projeto são considerados como boas práticas de programação e incentivam a produção de software flexível e extensível. Apesar disso, essas estruturas ainda podem acarretar na existência de *bad smells* no código fonte do projeto. Uma motivação a respeito do padrão de projeto *Visitor* é discutida pelo autor. Esse padrão tende a apresentar o *bad smell Feature Envy*. Speicher [2013] apresenta um exemplo em que esse padrão separa as funcionalidades dos dados, a fim de auxiliar a construção de objetos complexos e estendê-los ao uso de novas funcionalidades. Contudo, o modo como o padrão *Visitor* acessa os dados dos elementos colabora para uma falsa identificação de *Feature Envy*. Outros padrões como: *Flyweight*, *Interpreter*, *Mediator*, *Memento*, *State* e *Strategy*, são discutidos e o autor sugere que, além da separação dos objetos e dados, algumas propriedades dos padrões podem ser responsáveis pelas ocorrências de *bad smells* no código.

Baseado nessa discussão, uma abordagem para identificação automatizada de *bad smells* é proposta, na qual é levado em conta algumas decisões dos desenvolvedores que podem ser consideradas falsos indícios de *bad smells*. Essa abordagem foi implementada baseada em uma meta-programação lógica, onde o código Java foi representado como fatos *Prolog*, e as estratégias de detecção e os padrões de projeto foram definidos como predicados. Em meio a modelagem dessa abordagem, possíveis decisões dos desenvolvedores que poderiam ser consideradas falsos indícios de *bad smells* foram modeladas como verificações em seus respectivos predicados. Por exemplo, no padrão de projeto *Visitor*, o acesso da classe *Visitor* a dados de outros objetos pode ser considerada uma característica invejosa, mesmo isso sendo uma responsabilidade dessa classe. Tal ocorrência representa uma espécie de odor natural e deve ser desconsiderada na identificação de *bad smells*.

Como forma de avaliação da abordagem, foi realizado um estudo de caso, no qual é exibido um exemplo para identificação de *bad smells* levando em consideração intenções de desenvolvedores. Nesse estudo de caso, foi utilizado um software aberto desenvolvido na linguagem Java, chamado *ArgoUML*. Após a execução desse estudo de caso, o autor conclui que a abordagem proposta para identificação de *bad smell* obteve uma boa precisão.

Wagey et al. [2015]

Wagey et al. [2015] destacam a importância que a fase de manutenção tem em um software. Eles enfatizam que 66% do ciclo de vida de um software e 60% dos custos

são gastos justamente nessa fase. Essas informações mostram que no planejamento de um projeto é necessário levar em consideração a manutenibilidade dos mesmos para garantir uma redução de custos. Algumas soluções, como padrões de projeto, por exemplo, tende a gerar um impacto positivo no projeto de aplicações, evitando o surgimento de estruturas complexas.

A partir disso, os autores propuseram um novo modelo para medição do atributo externo manutenibilidade em uma aplicação de software, examinando *bad smells*. Esse modelo foi construído baseado em um grafo de dependência de atributos constituído de três níveis. Esses níveis foram classificados pelos autores da seguinte forma: baixo, médio e alto. O nível baixo é responsável por englobar as métricas utilizadas nesse modelo. Foram utilizadas 11 métricas diferentes conforme as características dos *bad smells* a serem identificados nesse modelo. O nível médio é composto de cinco *bad smells*, propostos por Fowler & Beck [1999]. O nível alto contém características de manutenibilidade: modularidade, reusabilidade, analisabilidade, modificabilidade e testabilidade. Durante a criação desse modelo de qualidade, os autores derivaram escalas de métricas para cada um dos *bad smells*. As escalas utilizadas variavam entre as seguintes características: *Very Good*, *Good*, *Medium*, *Bad*, *Very Bad*.

Para avaliar esse modelo, Wagey et al. [2015] realizaram um estudo de caso com seis aplicações Java de código aberto. Nessas aplicações, foram colhidos dados referentes a padrões de projeto, valores das métricas em cada software e a densidade de padrão de cada aplicação. Ao analisar os dados obtidos, os autores perceberam que o valor de densidade do padrão tem um impacto positivo na manutenibilidade de um software. Portanto, quanto maior for o valor da densidade do padrão, maior será a manutenibilidade de um projeto.

3.4 Discussão dos Resultados

Após analisar todos os artigos selecionados, esta seção é dedicada à resposta das questões de pesquisa. Seção 3.4.1 discute a Questão de Pesquisa 1 desta revisão sistemática, e Seção 3.4.2, a Questão de Pesquisa 2.

3.4.1 Abordagem da Relação entre Padrões de Projeto e *Bad Smells*

Essa seção apresenta e discute os resultados encontrados nessa revisão sistemática para as relações entre padrões de projeto e *bad smells*, por meio de respostas para a questão de pesquisa QP1(RSL).

QP1(RSL): Como a literatura tem abordado a relação entre padrões de projeto e *bad smells*?

Para responder essa questão de pesquisa, os 16 estudos selecionados foram analisados e extraídas as relações e focos de interesses que eles têm explorado. A Tabela 3.4, utilizada na Seção 3.2.3 para sumarizar os resultados, apresenta o tipo de relação extraída para cada um dos artigos selecionados. Foi possível identificar três categorias de relações: Coocorrência, Impacto em qualidade de software e Refatoração.

A categoria de coocorrência consiste em estudos que buscam avaliar as relações existentes entre os padrões de projeto e *bad smells*. Padrões de projeto são considerados soluções pontuais que incentivam a criação de estruturas de software modulares, flexíveis e extensível. Contudo, o mau uso ou aplicação errada de determinados padrões, podem elevar a complexidade de um software e gerar ocorrências de *bad smells*.

Jaafar et al. [2013] e Cardoso & Figueiredo [2015] exploraram as relações de coocorrências entre essas duas estruturas. Nesses dois estudos, uma forte relação entre o padrão *Command* com *bad smells* é indicada. Enquanto Jaafar et al. [2013] afirmam que o padrão *Command* foi aquele que apresentou maior relação com os *bad smells* investigados, Cardoso & Figueiredo [2015] é mais específico ao apontar a coocorrência desse padrão com o *bad smell God Class* e indicar coocorrências entre *Template Method* e *Duplicate Code*. Nessa mesma linha, Jaafar et al. [2016] investigam o impacto dessas relações sobre propensão a falha, tipos de trocas e tipos de falhas. Nessa investigação eles indicam que classes com dependências de co-mudança com classes que aplicam padrões de projeto tem significativamente mais falhas. Essas relações de co-mudanças ocorrem quando a alteração em uma classe impacta diretamente na alteração de outra. Além disso, Jaafar et al. [2016] indicam que mudanças estruturais e adições de código são mais propensos a ocorrer em classes que tem dependência com padrões de projeto.

Walter & Alkhaeir [2016] realizam um estudo semelhante ao de Jaafar et al. [2013] e ao de Cardoso & Figueiredo [2015]. Os resultados de Walter & Alkhaeir [2016] apontam que os padrões de projeto geram grandes impactos positivos no software, evitando ocorrências de *bad smells*. Contudo, o padrão *Composite* mostrou-se uma exceção a essa afirmação ao apresentar uma relação mais forte com a presença de *bad smells*.

A categoria de impacto em qualidade de software possui estudos que realizam avaliações empíricas da aplicação dos padrões de projeto em projetos de software. Nessa categoria, assim como na categoria de coocorrências, os estudos indicam que o mau uso dos padrões de projeto geram impactos negativos na qualidade de software. Wendorff [2001]; McNatt & Bieman [2001]; Izurieta & Bieman [2013] discutem algumas situações

que podem impactar negativamente na qualidade do software. Segundo eles, a aplicação de padrões de projeto que não se enquadram nos requisitos e acoplamento de padrões de projeto tornam a estrutura dos projetos mais complexas, prejudicando importantes atributos de qualidade como: modularidade, testabilidade e adaptabilidade. Nessa mesma linha, Vokac [2004]; Khomh & Gueheneuce [2008] apontam alguns padrões que têm se associado a complexidade. *Observer*, *Singleton* e *Flyweight* são alguns exemplos nessa linha, sendo que *Observer* foi indicado por Vokac [2004] com ocorrência de alta taxa de defeitos.

Por outro lado, dois artigos dessa categoria discutiram alguns impactos positivos dos padrões na qualidade software. Speicher [2013] propôs uma abordagem que considera possíveis intenções dos desenvolvedores na hora da implementação, e a utiliza para identificar *bad smells* e poder melhorar essas estruturas. Wagey et al. [2015] discutem que os padrões de projeto auxiliam na melhora da manutenibilidade de um software, e relatam que quanto maior é a densidade de padrão em um software, maior é a sua manutenibilidade.

A categoria de refatoração corresponde a estudos que utilizam os padrões de projeto como soluções de refatoração para determinados tipos de *bad smells*. Nessa categoria, os estudos consideram a premissa de que na maioria das vezes a escolha dos padrões de projeto são realizadas de forma manual, a partir do próprio conhecimento do programador. Com isso, uma escolha errada pode impactar no surgimento de *bad smells* ou gerar uma alta complexidade na estrutura do software. Mas, se essa escolha é realizada e aplicada de forma automatizada, o padrão de projeto pode ser aplicado corretamente e proporcionar os impactos positivos esperados. Por isso, os estudos referentes a essa categoria decidiram investigar e propor tipos de refatorações para esse tema de interesse.

Christopoulou et al. [2012] e Liu et al. [2014] exploraram casos de refatoração para o *bad smell Complex Conditional Statements* ou *Switch Statements*, como definido por Fowler & Beck [1999]. Ambos autores utilizaram os padrões de projeto *Strategy* como solução de refatoração. Liu et al. [2014] ainda utilizou o padrão *Abstract Factory* como um outro tipo de solução para refatoração desse *bad smell*. Zafeiris et al. [2017] dedicou-se à refatoração do *Call Super*, [Fowler, 2015] e utilizou o padrão de projeto *Template Method* como solução. Por fim, Nahar & Sakib [2015, 2016] criaram uma ferramenta que analisa diagrama de classes *UML* e recomendam sugestões de refatorações com padrões de projeto, a partir dos *bad smells* identificados nos diagramas. Inicialmente, Nahar & Sakib [2015] criaram a ferramenta apenas com suporte para recomendação do padrão *Abstract Factory*, e depois, Nahar & Sakib [2016], estenderam a ferramenta para dar suporte à recomendação de todos os padrões de projeto de criação.

Sumário da QP1(RSL). Conclui-se pois, em resposta à QP1(RSL), que a literatura tem abordado as relações entre padrões de projeto e *bad smells* de 3 formas diferentes: Coocorrências, Impacto em qualidade e Refatoração. A maioria dos estudos tem se concentrado na categoria de impacto em qualidade, sete no total, e avaliado as consequências e impactos da aplicação dos padrões de projeto em um software. A categoria de refatoração apresentou a segunda maior quantidade de trabalhos, cinco no total, e tem proposto abordagens e ferramentas que aplicam padrões de projeto para eliminação de *bad smells*. A categoria de coocorrências foi a que apresentou um menor número de estudos, quatro no total, e tem buscado identificar relações de coocorrência entre padrões de projeto e *bad smells* e apontar os motivos geraram essas relações.

3.4.2 Coocorrências entre Padrões de Projeto e *Bad Smells*

Essa seção apresenta e discute os resultados encontrados nessa revisão sistemática acerca de coocorrência entre padrões de projeto e *bad smells*, por meio de respostas para a questãos de pesquisa QP2(RSL), QP2.1(RSL) e QP2.2(RSL).

QP2(RSL): A literatura tem explorado coocorrências entre padrões de projeto e *bad smells*?

A resposta dessa questão de pesquisa é afirmativa. No entanto, os resultados obtidos nessa revisão sistemática indicam que poucos estudos têm explorado essas relações de coocorrências entre padrões de projeto e *bad smells*. Em sua maioria, os trabalhos tem identificado algumas coocorrências que impactam de forma negativa na qualidade de software. O surgimento dessas relações tem sido atribuídas ao mau uso e aplicação dos padrões de projeto que aumentam a complexidade de suas entidades internas, ou geram relações de co-mudanças entre classes.

Sumário da QP2(RSL). O resultado desta revisão sistemática sugere que esse tema é pouco explorado na literatura. A QP2.1 e a QP2.2 visam mostrar o atual estado da arte em relação a esse tema e expor algumas novas oportunidades de investigações. As respostas dessas questões de pesquisa são discutidas a seguir.

QP2.1(RSL): Quais *bad smells* são abordados pela literatura na identificação de coocorrências com padrões de projeto?

Para responder esta questão de pesquisa, a Tabela 3.5 apresenta os *bad smells* identificados nos estudos referentes à categoria de coocorrência dessa revisão sistemática. Apesar de terem sido retornados poucos estudos, existe uma grande quantidade de *bad smells* que têm sido explorados. Ao todo, foram encontrados um total de 18 *bad smells*, organizados de acordo com o autor responsável por sua descrição na literatura.

***Bad smells* descritos por Brown et al. [1998]**
Anti Singleton, *Blob*, *Class Data Should Be Private*, *Complex Class*, *Spaghetti Code*, *Swiss Army Knife*.
***Bad smells* descritos por Fowler & Beck [1999]**
Data Class, *Data Clumps*, *Duplicate Code*, *Feature Envy*, *Long Method*, *Long Parameter List*, *Message Chains*, *Refused Parent Bequest* e *Speculative Generality*.
***Bad smells* descritos por Lanza & Marinescu [2006]**
External Duplication, *God Class* e *Schizophrenic Class*.

Tabela 3.5. Lista de *bad smells* identificados nesta revisão sistemática da literatura.

Sumário da QP2.1(RSL). Os estudos que avaliam coocorrência entre padrões de projeto e *bad smells*, foram identificados 18 *bad smells* nessa revisão sistemática. Na Tabela 3.5 é possível observar que os *bad smells* descritos por Fowler & Beck [1999] têm sido utilizado em maior quantidade para estudos de coocorrência. No total, dos 18 *bad smells* considerados nos estudos, 9 são pertencentes ao conjunto descrito por Fowler & Beck [1999]. Em menor quantidade, estão os *bad smells* descritos por Brown et al. [1998], com o total de 6. Por fim, 3 *bad smells* descritos por Lanza & Marinescu [2006] foram identificados nos estudos dessa revisão sistemática.

QP2.2 (RSL): Quais padrões de projeto são usados pela literatura na identificação de coocorrências com *bad smells*?

Para responder esta questão de pesquisa, a Tabela 3.6 exibe em detalhes os padrões identificados nos estudos que abordam relações de coocorrências. Foi identificado um total de 13 padrões de projeto, e todos eles pertencem ao catálogo *GOF*. Um fator importante que deve contribuir para a escolha desses padrões é a existência de ferramentas de análise estática de código fonte, que identificam as instâncias de padrões

de projeto em software. O uso dessas ferramentas proporcionam uma maior agilidade no processo de extração de informações e amenizam o surgimento de falsos positivos e falsos negativos.

[**Jaafar et al., 2013**]
Command, *Composite*, *Decorator*, *Factory Method*, *Observer*, *Prototype*.
[**Cardoso & Figueiredo, 2015**]
Adapter, *Command*, *Composite*, *Decorator*, *Factory Method*, *Observer*, *Prototype*, *Proxy*, *Singleton*, *Strategy*, *State*, *Template Method*, *Visitor*.
[**Jaafar et al., 2016**]
Command, *Composite*, *Decorator*, *Factory Method*, *Observer*, *Prototype*.
[**Walter & Alkhaeir, 2016**]
Adapter, *Command*, *Composite*, *Decorator*, *Factory Method*, *Observer*, *Prototype*, *Proxy*, *Singleton*, *Strategy*, *State*, *Template Method*, *Visitor*.

Tabela 3.6. Lista de padrões de projeto identificados nesta revisão sistemática da literatura.

Sumário da QP2.2(RSL). As informações exibidas na Tabela 3.6 indicam que os padrões de projeto *GOF* são os mais utilizados nos estudos analisados.

3.5 Ameaças à Validade

Esta seção apresenta algumas ameaças à validade desta revisão sistemática da literatura e discute algumas decisões que foram tomadas para minimizá-las.

A *string* de busca de uma revisão sistemática é algo que precisa ser muito bem definido a fim de retornar trabalhos que sejam relevantes para o tema de pesquisa. Neste estudo, vários sinônimos referentes aos termos principais do objetivo da revisão foram pesquisados. Algumas pesquisas piloto foram realizadas, a fim de localizar novos sinônimos para a *string* de busca. Portanto, espera-se que a *string* de busca definida tenha retornado o maior número possível de artigos relevantes. Contudo, não é possível afirmar que todos os trabalhos referentes a relações entre padrões de projetos e *bad smells* foram retornados.

A escolha das base eletrônicas é outro fator que pode impactar nos resultados de uma revisão sistemática. Neste trabalho, os estudos primários foram pesquisados em

sete bases eletrônicas diferentes. No entanto, outras bases que não foram utilizadas na pesquisa podem conter trabalhos relevantes para esta revisão. Para amenizar essa ameaça, uma estapa que consiste em um processo de *snowballing* [Wohlin, 2014] foi realizada durante a fase de filtragem dos artigos. Nesta etapa, as citações dos artigos selecionados foram verificados por meio de uma lista de referências, a fim de localizar outros estudos relevantes que não haviam sido retornados.

Esta revisão da literatura considerou apenas artigos escritos no idioma inglês. É possível que haja trabalhos relevantes escritos em outros idiomas. Contudo, como os principais meios de divulgação científica da área de Engenharia de Software aceitam trabalhos em inglês, considera-se que a filtragem usando esse idioma seja suficiente para filtrar os trabalhos mais relevantes no assunto.

Por fim, devido ao grande número de artigos retornados pelas bases eletrônicas, algumas etapas de filtragem contaram com uma leitura reduzida do conteúdo dos artigos para agilizar esse processo. Essa leitura reduzida pode influenciar diretamente na eliminação de trabalhos importantes. Para minimizar essa ameaça, foi executada uma etapa no processo de filtragem com o intuito de realizar uma análise mais aprofundada em artigos cujos objetivos não haviam ficado claros durante a leitura reduzida.

3.6 Considerações Finais

Este capítulo apresentou uma revisão sistemática da literatura, realizada com o objetivo de identificar como os estudos existentes na literatura têm abordado a relação entre padrões de projeto e *bad smells*, especificamente as relações de coocorrência entre eles.

Na fase de planejamento foi construída uma *string* de busca e definidas as bases eletrônicas e os critérios de inclusão e exclusão utilizados neste estudo. Esta revisão contou com uma pesquisa de estudos primários em sete diferentes e importantes bases eletrônicas para pesquisa de artigos, tais como: *ACM*, *Compendex (Engineering Village)*, *IEEE Xplore*, *Science Direct*, *Scopus*, *Springer* e *Web of Science*. Contudo, a *IEEE Xplore* não retornou estudo algum referente a *string* de busca utilizada.

A pesquisa nas bases eletrônicas retornou um total de 795 documentos. Esses documentos foram filtrados em quatro etapas diferentes e, no fim, na Etapa 5 foi realizado um processo de *snowballing* com os artigos selecionados das bases eletrônicas. A realização dessa técnica foi muito importante, pois identificou artigos relevantes que não foram retornados pelas bases eletrônicas. Esta revisão sistemática resultou em um total de 16 artigos que foram analisados para que as questões de pesquisas pudessem ser respondidas.

Os resultados desta revisão sistemática mostram que os trabalhos encontrados na literatura sugerem relações entre padrões de projeto e *bad smells* de três formas diferentes: avaliação do impacto dos padrões de projeto na qualidade de software, aplicação de técnicas de refatoração considerando padrões de projeto para eliminação de *bad smells*, e identificação de coocorrências entre padrões de projeto e *bad smells*. As análises realizadas nesses estudos indicam que quando os padrões de projeto culminam no surgimento ou ocorrência dos *bad smells*, essas relações impactam negativamente na qualidade de software, elevando a complexidade e prejudicando outros atributos externos importantes tais como, modularidade, flexibilidade, testabilidade, dentre outros. No entanto, quando a aplicação de padrões de projeto é bem projetada, como descrito nos estudos que tratam as relações de refatoração, os impactos gerados são positivos.

Percebe-se também nos resultados que poucos estudos tem investigado coocorrências entre padrões de projeto e *bad smells*. Os *bad smells* descritos por Fowler & Beck [1999] são os mais utilizados nessas investigações. No entanto, esses estudos também têm abordado alguns *bad smells* descritos por Brown et al. [1998] e Lanza & Marinescu [2006]. Em relação a padrões de projeto, os estudos tem abordado soluções integrantes do catálogo *GOF*, proposto por Gamma et al. [1994]. O uso desses padrões é justificado pela existência de ferramentas que realizam a extração das instâncias de forma automatizada. Essas ferramentas, além de agilizar o processo de extração, amenizam a ocorrência de falsos positivos e negativos. O surgimento de coocorrências entre padrões de projeto e *bad smells* são atribuídos à forma de implementação dos padrões e evolução dos projetos.

Na sequência, o Capítulo 4 apresenta a metodologia de pesquisa do Estudo de Caso e as principais etapas a serem seguidas para a condução dos experimentos deste trabalho.

Capítulo 4

Metodologia

Este capítulo apresenta a metodologia utilizada para a condução do Estudo de Caso realizado nesta dissertação. Seção 4.1 define a estratégia de detecção para *Data Class*, *Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*; Seção 4.2 descreve como foi realizada a escolha dos sistemas de software utilizado como *data set* no Estudo de Caso; Seção 4.3 detalha o processo de coleta de dados; Seção 4.4 apresenta as regras de associação utilizadas para identificação das coocorrências; e Seção 4.5 descreve o método usado para a análise dos resultados. Seção 4.6 apresenta as considerações finais deste capítulo.

As etapas da metodologia da condução do Estudo de Caso realizado nessa dissertação são representadas pela Figura 4.1.

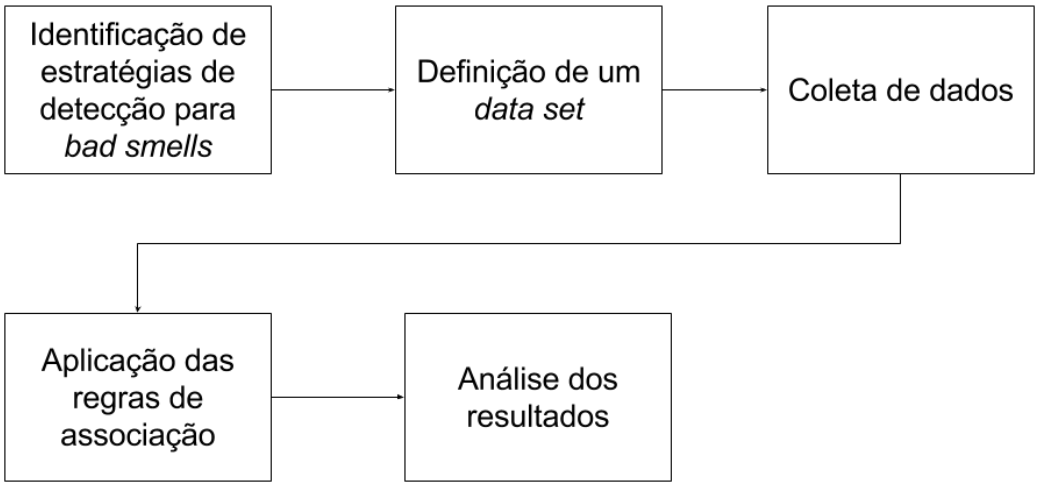

Figura 4.1. Etapas da metodologia da condução do Estudo de Caso.

4.1 Estratégia de Detecção de *Bad Smells*

De acordo com Marinescu [2002], estratégia de detecção é uma expressão quantificável de uma regra que avalia se fragmentos de um código fonte tem propriedades de um dado *bad smell*. Junto às estratégias de detecção, valores referências são utilizados para determinar a relação de um métrica com um *bad smell* e, assim, identificar entidades anômalas.

Nesta pesquisa de dissertação, foram considerados cinco *bad smells* descritos por Fowler & Beck [1999]: *Data Class*, *Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*. Esses *bad smells* foram escolhidos porque eles são especialmente problemáticos para a manutenção do software e estão relacionados a uma grande quantidade de informações e complexidade que podem tornar a compreensão do software difícil e aumentar o acoplamento entre os métodos e classes do sistema. Além disso, existem estratégias de detecção para eles previamente descritas e avaliadas na literatura.

As estratégias de detecção utilizadas neste estudo foram propostas por Souza [2016]. Essas estratégias foram escolhidas porque elas aplicam métricas de software bem conhecidas. Além disso, elas foram previamente avaliadas pela autora, e os resultados indicaram que elas possuem um boa precisão e são efetivas na identificação de *bad smells*. Os valores referência utilizados na composição dessas estratégias foram propostos por Filó et al. [2015].

Algumas ferramentas propostas na literatura tais como: *DECOR* [Moha et al., 2010], *JDeodorant* [Tsantalis et al., 2008], *JSpIRIT* Vidal et al. [2014] auxiliam pesquisadores e desenvolvedores na detecção automática de *bad smells*. No entanto, *DECOR* não utiliza uma abordagem baseada em estratégias de detecção. *JSpIRIT* e *JDeodorant* possuem uma abordagem baseada no uso de métricas [Fernandes et al., 2016], contudo, elas utilizam métricas e valores referências próprios e não permitem que eles sejam customizados. Por esse motivo, neste trabalho de dissertação foi utilizada uma ferramenta chamada *RAFTool* [Filó et al., 2015]. Ela suporta a implementação das estratégias de detecção proposta por Souza [2016] para identificação dos *bad smells*.

Filó et al. [2015] apresentaram um método de extração de valores referência para métricas de software orientados por objeto, e aplicaram esse método para identificação de valores referência para 18 métricas de software. Os valores referência propostos pelos autores classificam uma métrica em três faixa de valores: Bom/Frequente, Regular/Ocasional e Ruim/Raro. A faixa Bom/Frequente corresponde aos valores com alta frequência, caracterizando os valores mais comuns da métrica na prática. A faixa Ruim/Raro corresponde aos valores com baixa frequência, e a faixa Regular/Ocasional é intermediária, correspondendo aos valores que não são muito frequentes, mas também

não são muito raros. A Tabela 4.1 apresenta o catálogo proposto por Filó et al. [2015].

Metric	Bom/Frequente	Regular/Ocasional	Ruim/Raro
CA	$m \leq 7$	$7 < m \leq 39$	$m > 39$
CE	$m \leq 6$	$6 < m \leq 16$	$m > 16$
DIT	$m \leq 2$	$2 < m \leq 4$	$m > 4$
LCOM	$m \leq 0{,}167$	$0{,}167 < m \leq 0{,}725$	$m > 0{,}725$
MLOC	$m \leq 10$	$10 < m \leq 30$	$m > 30$
NBD	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NOC	$m \leq 11$	$11 < m \leq 28$	$m > 28$
NOF	$m \leq 3$	$3 < m \leq 8$	$m > 8$
NOM	$m \leq 6$	$6 < m \leq 14$	$m > 14$
NORM	$m \leq 2$	$2 < m \leq 4$	$m > 4$
NSC	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NSF	$m \leq 1$	$1 < m \leq 5$	$m > 5$
NSM	$m \leq 1$	$1 < m \leq 3$	$m > 3$
PAR	$m \leq 2$	$2 < m \leq 4$	$m > 4$
RMD	$m \leq 0{,}467$	$0{,}467 < m \leq 0{,}750$	$m > 0{,}750$
SIX	$m \leq 0{,}019$	$0{,}019 < m \leq 1{,}333$	$m > 1{,}333$
VG	$m \leq 2$	$2 < m \leq 4$	$m > 4$
WMC	$m \leq 11$	$11 < m \leq 34$	$m > 34$

Tabela 4.1. Catálogo de valores referência para métricas de softwares orientados por objetos [Filó, 2014; Filó et al., 2015]

.

Baseado nos fatos apresentados nesta seção, as estratégias de detecção propostas por Souza [2016] foram integradas a esta pesquisa. Cada estratégia de detecção é composta por uma sequência de cláusulas conectadas por operadores lógicos *AND* e *OR*. Uma cláusula consiste em uma composição de uma métrica com o seu valor referência. Na estratégia responsável pela identificação do *bad smell Data Class*, são utilizadas as seguintes métricas: profundidade da árvore de herança (DIT), número de atributos (NOF) e número de filhas (NSC). A Figura 4.2 mostra a estratégia de detecção do *bad smell Data Class*.

Para o *bad smell Feature Envy* a única métrica utilizada na estratégia de detecção é falta de coesão dos métodos (LCOM). A Figura 4.3 mostra a estratégia de detecção do *bad smell Feature Envy*.

Para o *bad smell Large Class*, as métricas utilizadas foram: falta de coesão dos métodos (LCOM), número de atributos (NOF), número de métodos (NOM) e métodos ponderados por classes (WMC). A Figura 4.4 mostra a estratégia de detecção do *bad smell Large Class*.

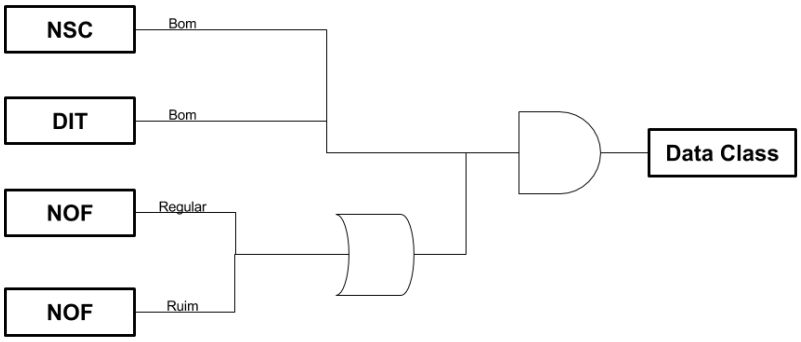

Figura 4.2. Estratégia de detecção para *Data Class* extraída de Souza [2016].

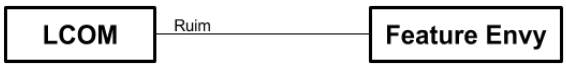

Figura 4.3. Estratégia de detecção para *Feature Envy* extraída de Souza [2016].

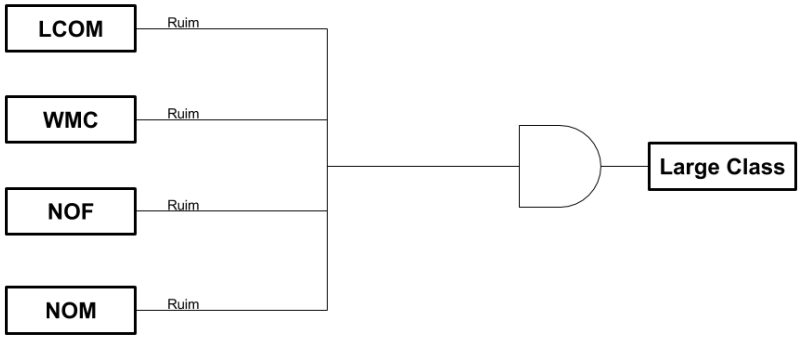

Figura 4.4. Estratégia de detecção para *Large Class* extraída de Souza [2016].

Para o *bad smell Long Method*, as métricas utilizadas foram: linhas de código por método (MLOC), profundidade de blocos aninhados (NBD) e complexidade de McCabe (VG). A Figura 4.5 mostra a estratégia de detecção do *bad smell Long Method*.

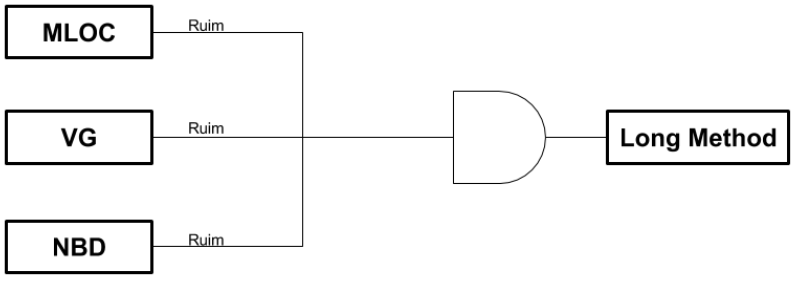

Figura 4.5. Estratégia de detecção para *Long Method* extraída de Souza [2016].

Por fim, para o *bad smell Refused Bequest*, é necessário apenas uma métrica: índice de especialização (SIX). A Figura 4.6 mostra a estratégia de detecção do *bad smell Refused Bequest*.

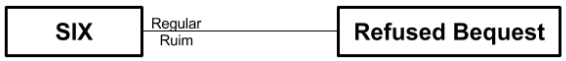

Figura 4.6. Estratégia de detecção para *Refused Bequest* extraída de Souza [2016].

4.2 *Data Set*

A segunda etapa baseia-se na definição da amostra de software a ser utilizada neste estudo. Os sistemas de software escolhidos para compor este *data set* foram extraídos do *Qualitas.class Corpus*, uma versão compilada do *Qualitas Corpus* proposto por Tempero et al. [2010] e disponibilizado por Terra et al. [2013]. O *Qualitas.class Corpus* contém métricas de software de 112 sistemas. O *Qualitas Corpus* é formado por uma coleção de software de código aberto desenvolvido em Java que são disponibilizados para estudos empíricos de artefato de código [Filó et al., 2015].

Como este estudo envolve inspeção manual, considerou-se uma amostra de cinco softwares: *Hibernate 4.2.0*, *JHotDraw 7.5.1*, *Kolmafia 17.3*, *Webmail 0.7.10* e *Weka 3.6.9*. Todos eles, exceto *Kolmafia*, são do *Qualitas.class Corpus*. O principal critério para a seleção desses projetos baseou-se em dois pontos: (i) eles usam padrões de projeto do catálogo GOF, e (ii) eles apresentam os *bad smells* considerados nesta dissertação de mestrado. Optou-se por incluir o *Kolmafia 17.3* nesta amostra, porque estudos prévios apontam valores de métricas considerados problemáticos nesse sistema de software [Ferreira et al., 2012], todavia, sem qualquer correlação desses valores com *bad smell* ou padrões de projeto.

4.3 Coleta de Dados

A terceira etapa implica na coleta dos dados analisados neste estudo. O *Qualitas.class Corpus* possui, além dos sistemas de software, arquivos em formato XML com métricas coletadas dos sistemas. Como a maioria dos sistemas usados neste trabalho foram extraídos do *Qualitas.class Corpus*, esses arquivos foram utilizados. Como os dados de *Kolmafia 17.3* não estão neste *corpus*, seu código foi baixado e suas métricas foram coletadas. As ferramentas utilizadas para coleta de métricas do *Kolmafia 17.3* foram:

IDE Eclipse 4.2 Juno [Eclipse, 2016] e o *plugin Metrics 1.3.6* [Metrics, 2016]. Após coletar as metricas, elas foram exportadas e salvas em um arquivo de formato *XML*.

Para verificar a existência dos padrões de projeto nos sistemas, foi utilizada uma ferramenta proposta por Tsantalis et al. [2006], chamada *Design Pattern Detection using Similarity Scoring 4 (DPDSS)*[1]. De acordo com os seus autores, essa ferramenta modela todos os aspectos dos padrões de projetos por meio de grafos direcionados, representados em matrizes quadráticas, e aplica um algoritmo chamado *Similary Scoring*. Esse algoritmo usa como entrada o sistema e o grafo do padrão, e a partir desses dados calcula a pontuações de similaridade entre os vértices. Segundo os autores, a principal vantagem dessa abordagem é a capacidade de detectar não apenas os padrões na sua forma de base, aquela normalmente encontrada na literatura, mas também versões modificadas dele. Essa ferramenta foi testada por seus autores em três sistemas: *JHotDraw 5.1*, *JRefactory 2.6.24* e *JUnit 3.7*, e falsos positivos não foram retornados para nenhum desses sistemas. Falsos negativos foram retornadas apenas para dois padrões de projetos: *Factory Method* e *State*. Os resultados apresentados por essa ferramenta indicaram que ela é eficiente na identificação de instâncias de padrões de projeto. Ademais, dentre as ferramentas estudadas, *DPDSS* foi identificada na literatura como aquela que consegue identificar uma maior quantidade de padrões de projetos do catálogo *GOF* (14 padrões no total), sendo que os padrões *Adapter*, *Command*, e *State*, *Strategy* não podem ser identificados separadamente por essa ferramenta. Por isso, ela identifica esses padrões como sendo *Adapter-Command* um tipo de instância, e *State-Strategy* sendo outro tipo de instância, totalizando 12 instâncias de padrões de projeto que podem ser identificados por essa ferramenta. Além disso, outro fator que influenciou na escolha dessa ferramenta foi o fato dela a mais utilizada em trabalhos que necessitam da extração de padrões de projeto em software.

Filó et al. [2014] desenvolveram uma ferramenta, *RAFTool*[2], que realiza a identificação de métodos, classes e pacotes com medições anômalas de métricas de software orientado por objetos. RAFTool foi usada com o propósito de implementar estratégias de detecção. A ferramenta recebe como entrada o arquivo XML do sistema alvo com suas métricas, e uma estratégia de detecção descrita por uma expressão lógica em um dado formato. A ferramenta relata as classes ou os métodos cujos valores de métricas se encaixam na estratégia de detecção.

Para realizar a coleta das informações de *bad smell* nos sistemas analisados, foi necessário transformar as estratégias de detecção, Figuras 4.2, 4.3, 4.4, 4.5 e 4.6, em expressões de filtragem para que pudessem ser utilizadas na *RAFTool*. Nessa ferramenta,

[1] `https://users.encs.concordia.ca/~nikolaos/pattern_detection.html`
[2] `http://homepages.dcc.ufmg.br/~tfilo/raftool/`

os valores referência das métricas que compõem as estratégias são representados pelas seguintes palavras chave: *COMMON*, que corresponde à faixa BOM/FREQUENTE das métricas; *CASUAL*, que corresponde à faixa REGULAR/OCASIONAL das métricas; e *UNCOMMON*, que corresponde à faixa RUIM/RARO das métricas. Além disso, após o uso de qualquer uma dessas palavras chave, é necessário informar a métrica que tal palavra se refere.

As expressões de filtragem utilizadas neste trabalho foram:

Exp1 COMMON[NSC] AND COMMON[DIT] AND (UNCOMMON[NOF] OR CASUAL[NOF])

Exp2 UNCOMMON[LCOM]

Exp3 UNCOMMON[LCOM] AND UNCOMMON[WMC] AND UNCOMMON[NOF] AND UNCOMMON[NOM]

Exp4 UNCOMMON[MLOC] AND UNCOMMON[VG] AND UNCOMMON[NBD]

Exp5 UNCOMMON[SIX]

A primeira expressão de filtragem refere-se ao *bad smell Data Class*. Ela é composta por uma combinação da faixa Bom (*COMMON*) para as métricas *NSC* e *DIT* e da faixa Regular (*CASUAL*) e Ruim (*UNCOMMON*) para a métrica *NOF* (Vide Figura 4.2).

A segunda expressão de filtragem refere-se ao *bad smell Feature Envy*. Ela é composta apenas pela métrica LCOM utilizando a faixa Ruim (*UNCOMMON*) do catálogo definido por Filó et al. [2015] (Vide Figura 4.3).

A terceira expressão de filtragem refere-se ao *bad smell Large Class*. Ela é composta por uma combinação da faixa Ruim (*UNCOMMON*) para as quatro métricas associadas a esse *bad smell*: LCOM, WMC, NOF e NOM (Vide Figura 4.4).

A quarta expressão de filtragem refere-se ao *bad smell Long Method*. Ela é uma combinação pela faixa Ruim (*UNCOMMON*) para as três métricas associadas a esse *bad smell*: MLOC, VG e NBD (Vide Figura 4.5).

A quinta expressão de filtragem refere-se ao *bad smell Refused Bequest*. Assim como a expressão construída para *Feature Envy*, ela é composta por uma única métrica, SIX, utilizando a faixa Ruim (*UNCOMMON*) do catálogo proposto por Filó et al. [2015] (Vide Figura 4.6).

4.4 Regras de Associação

A quarta etapa consiste na associação dos padrões de projeto e *bad smell* para identificação das coocorrências. Nesse processo, foram aplicadas regras de associação baseada no conceito de mineração de dados [Agrawal et al., 1993; Brin et al., 1997]. O motivo da escolha desse método advém do fato de que as regras de associação combinam itens de um *data set* para extrair conhecimento sobre os dados analisados. Reforçando essa escolha, estudos prévios no mesmo contexto deste, [Cardoso & Figueiredo, 2015; Walter & Alkhaeir, 2016] também aplicaram esse mesmo método para identificação de coocorrências entre essas duas estruturas.

Para aplicar as regras de associação, três métricas são utilizadas, Suporte [Agrawal et al., 1993], Confiança [Agrawal et al., 1993] e Convicção [Brin et al., 1997]. Essas métricas são baseadas nos seguintes conceitos:

- **Transação:** conjunto de itens.
- **Antecedente:** item que aparece do lado esquerdo da regra de associação.
- **Consequente:** item que aparece do lado direito da regra de associação.

Uma regra de associação possui a seguinte forma: Antecedente ⇒ Consequente.

A métrica Suporte (sup) em uma regra de associação indica a frequência que um item ocorre em uma transação (Equação 4.1).

$$sup(X \Rightarrow Y) = P(x, y) \tag{4.1}$$

Por exemplo, considere uma base de compras em um supermercado. Suponha que exista um *data set* com o registro de 1.000 transações, que são o conjunto de itens que foram comprados. Nesse *data set*, os itens `macarrão` e `tomate` aparecem juntos em 100 desses registros. Assim, o Suporte dessa relação é 0,1, ou seja, 10%.

A métrica Confiança (conf) expressa a probabilidade de um consequente ocorrer dado que o antecedente ocorre. Em outras palavras, ela indica a chance do lado direito da regra ocorrer, dada a ocorrência do lado esquerdo (Equação 4.2).

$$conf(X \Rightarrow Y) = \frac{sup(X \Rightarrow Y)}{sup(X)} \tag{4.2}$$

No exemplo mencionado, considere que o item `macarrão` seja encontrado sozinho em 200 das 1.000 transações do *data set*. Para calcular a Confiança da regra de associação `macarrão` ⇒ `tomate`, é necessário dividir o Suporte dessa regra, 0,1, pelo Suporte apenas do `macarrão` - Antecedente na regra de associação -, 0,2, resultando em um

valor de 0,5, ou seja, 50%. Apesar dessa métrica ser muito importante, é necessário ter cautela com seu uso, uma vez que a Confiança é muito sensível em relação à frequência do lado direito da regra. Portanto, um valor muito alto para o lado direito pode gerar uma alta Confiança, mesmo que os itens não possuam nenhum tipo de relação.

Para resolver esse problema da Confiança, Brin et al. [1997] propuseram a métrica Convicção. Essa métrica foi proposta com o objetivo de cobrir a falha deixada pela métrica Confiança. Ela utiliza o Suporte tanto do Antecedente como do Consequente (Equação 4.2).

$$conv(X \Rightarrow Y) = \frac{sup(X) * (1 - sup(Y))}{sup(X) - sup(X \Rightarrow Y)} \tag{4.3}$$

No exemplo dado, considere que o item `tomate` é encontrado sozinho em 300 das 1.000 transações do *data set*. Assim, o suporte do `tomate`, (sup(`tomate`)) é 0,3 e a confiança da regra, conf(`macarrão` ⇒ `tomate`) é 0,5. Aplicando esses valores na Equação 4.3, a Convicção da regra, conv(`macarrão` ⇒ `tomate`), é 1,4. Quando o valor da Convicção é 1, indica que o antecedente e o consequente não possuem relação. Quando o valor de Convicção é menor que 1, indica que se o antecedente ocorre, o consequente tende a não ocorrer. Quando o valor da Convicção é maior que 1, significa que o antecedente e o consequente possuem relação; quanto maior o valor da Convicção, maior a relação entre o antecedente e o consequente. Um resultado infinito indica que o antecedente não aparece em transação alguma.

Para a aplicação da regra de associação neste estudo, Antecedente, Consequente e Transação foram assim considerados:

- **Transação:** representa cada classe existente no sistema analisado.
- **Antecedente:** representa cada padrão de projeto explorado neste trabalho e pertencente ao catálogo GOF.
- **Consequente:** representa cada *bad smell*, explorado neste trabalho.

4.5 Método Utilizado para a Análise dos Resultados

A Figura 4.7 ilustra, via diagrama, o método usado para analisar os dados obtidos neste estudo.

A primeira etapa consiste na identificação dos padrões de projetos nos softwares utilizados neste estudo. Para identificá-los, foi usada a ferramenta *Design Pattern De-*

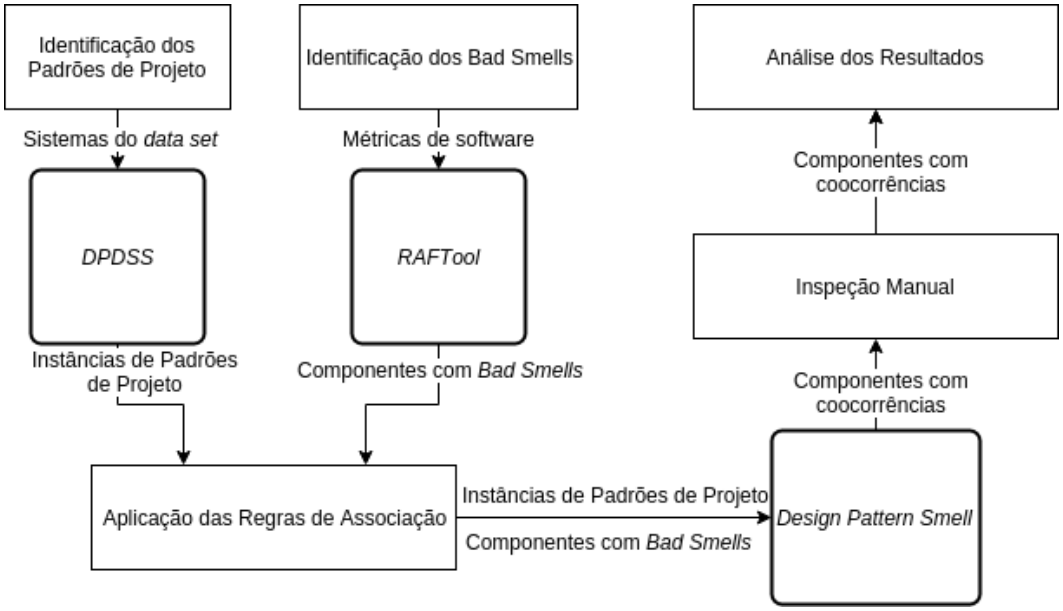

Figura 4.7. Método de análise dos resultados.

tection using Similarity Scoring 4 (DPDSS) e os resultados obtidos foram armazenados em uma tabela.

A segunda etapa consiste na aplicação das expressões de filtragem, **Exp1**, **Exp2**, **Exp3**, **Exp4** e **Exp5**, na ferramenta *RAFTool*. O objetivo dessa etapa é a identificação de classes e métodos com valores referência ruins e com a presença dos *bad smells* investigados neste estudo.

A terceira etapa consiste na identificação das coocorrências entre os padrões de projeto e *bad smells* e a aplicação das regras de associação. Para realizar essas duas tarefas, foi utilizada a ferramenta *Design Pattern Smell*, descrita no Capítulo 5.

A quarta etapa consiste na inspeção manual dos métodos e classes com coocorrência, a fim de identificar situações que favoreceram a presença dessas relações em tais componentes.

A quinta etapa consiste na análise dos dados a fim de responder as questões de pesquisa:

- **QP1:** Os padrões de projeto definidos no catálogo *GOF* evitam a ocorrência de *bad smells* em software?
- **QP2:** Quais padrões de projeto do catálogo *GOF* apresentaram coocorrência com *bad smells*?
- **QP3:** Quais são as situações mais comuns em que *bad smells* aparecem em sistemas de software que aplicam os padrões de projeto *GOF*?

Neste estudo, uma coocorrência foi considerada como sendo um componente que faz parte de uma instância de padrão de projeto e ao mesmo tempo possui a presença de um *bad smell*. Esses componentes foram identificados pela ferramenta *Design Pattern Smell*, via uma abordagem de cruzamento de dados. Identificadas as coocorrências e aplicadas as regras de associação, realizou-se uma inspeção manual nos componentes pertencentes a relação de maior intensidade para identificar situações presentes no código fonte dos projetos do *data set* que contribuíram para o surgimento dessas relações.

4.6 Considerações Finais

Este capítulo descreveu a metodologia de pesquisa utilizada na realização dessa dissertação de mestrado e as principais etapas a serem seguidas para a condução do Estudo de Caso. Na sequência, o Capítulo 5 descreve uma ferramenta que foi desenvolvida no decorrer deste trabalho, para identificar coocorrências entre padrões de projeto e *bad smell*. No Capítulo 6 é relatado os resultados do Estudo de Caso e realizada uma análise e discussão desses resultados.

Capítulo 5

Uma Ferramenta para Detecção de Coocorrência entre Padrões de Projeto e *Bad Smells*

Algumas pesquisas investigaram a relação entre padrões de projeto e *bad smells*. No entanto, nenhuma delas propõe ferramenta para identificação de coocorrências entre eles [Jaafar et al., 2013; Cardoso & Figueiredo, 2015; Jaafar et al., 2016; Walter & Alkhaeir, 2016]. Por exemplo, analisando os trabalhos descritos no Capítulo 3, percebeu-se que os autores desses estudos usaram seus próprios *scripts* ou processamento manual para identificar essas relações.

Baseado nesse contexto, este capítulo apresenta uma ferramenta, *Design Pattern Smell*, que suporta a detecção de coocorrência entre padrões de projeto e *bad smell* baseado em informações computadas de sistemas de software. Essa ferramenta recebe como entrada arquivos XML contendo as instâncias de padrões de projeto e um arquivo CSV contendo os artefatos de código[1] com *bad smells*. *Design Pattern Smell* analisa esses arquivos e realiza um cruzamento de dados (Vide Seção 5.3.1) para detecção as coocorrências. Além disso, ela permite o usuário aplicar regras de associação [Agrawal et al., 1993; Brin et al., 1997] nesses dados para identificar a intensidade dessas relações. *Design Pattern Smell* foi implementada na linguagem de programação Java.

O restante desse capítulo está organizado da seguinte forma. Seção 5.1 descreve a abordagem utilizada para o desenvolvimento da *Design Pattern Smell*. Seção 5.2 discute as principais funcionalidades implementadas na *Design Pattern Smell*. Seção 5.3 apresenta a arquitetura e organização interna da ferramenta. Seção 5.4 discute detalhes

[1]Artefato é uma classe ou método de um sistema

técnicos utilizados na implementação da *Design Pattern Smell*. Seção 5.5 fornece um exemplo de uso. Seção 5.6 conclui esse capítulo.

5.1 Abordagem Proposta

Design Pattern Smell é uma ferramenta de análise estática proposta para identificação de coocorrências de padrões de projeto com *bad smell* baseada em informações extraídas do código fonte. A decisão de construí-la foi baseada no propósito de auxiliar a identificação de fragmentos de código que apresentam ocorrências tanto de padrões de projeto quanto *bad smells*, além de apoiar uma análise exploratória a fim de entender os motivos que contribuíram para as coocorrências dessas duas estruturas. Embora *Design Pattern Smell* necessite de informações sobre padrões de projeto e *bad smell* previamente extraídas, tais informações podem ser facilmente obtidas por ferramentas como: *Design Pattern Detection using Similarity Scoring* [Tsantalis et al., 2006], *JDeodorant* [Tsantalis et al., 2008], *JSpIRIT* [Vidal et al., 2014] e *RAFTool* [Filó et al., 2014].

Assim, os principais objetivos da *Design Pattern Smell* é suportar a avaliação da qualidade de software por identificação de estruturas de código em que a presença de um *bad smell* pode degenerar a aplicação de um padrão de projeto. *Design Pattern Smell* suporta a detecção de coocorrências de *bad smells* a nível de classes e métodos com 14 padrões de projeto do catálogo *GOF* [Gamma et al., 1994].

5.2 Principais Funcionalidades

As principais funcionalidades da *Design Pattern Smell* estão descritas a seguir.

Importação das Instâncias de Padrões de Projeto Computadas. Para identificar as coocorrências entre padrões de projeto e *bad smells*, *Design Pattern Smell* requer que o usuário importe arquivos *XML* com instâncias de padrões de projeto de um dado sistema. Esse arquivo de entrada segue o mesmo padrão do formato de arquivo exportado pela *Design Pattern Detection usign Similarity Scoring* [Tsantalis et al., 2006] e está descrito no *website* da *Design Pattern Smell* [Sousa et al., 2016].

Importação dos Artefatos com *Bad Smells*. Outro requisito para identificação das coocorrências entre padrões de projeto e *bad smells* é um arquivo *CSV* contendo artefatos com a presença de *bad smell*. Depois da importação desse arquivo,

Design Pattern Smell realiza uma análise de seu conteúdo e então cruza as informações contidas nos arquivos *CSV* com aquelas do arquivo *XML*, identificando os artefatos que têm coocorrência dessas duas estruturas. O arquivo de entrada tem que ser escrito de acordo com formato especificado no *website* da ferramenta [Sousa et al., 2016].

Aplicação de Regras de Associação. O usuário pode aplicar regras de associação nos dados usados, a fim de analisar a intensidade das coocorrências entre padrões de projeto e *bad smell*. *Design Pattern Smell* fornece um módulo que aplica essas regras de associação automaticamente, evitando que o usuário tenha que fazer esse trabalho manualmente. Para a aplicação dessas regras, (i) é fornecido um campo onde o usuário entra com o número de transações do sistema analisado e (ii) é fornecido um painel com quatro tipos de regras: Suporte, Confiança, *Lift* e Convicção [Agrawal et al., 1993; Brin et al., 1997], que podem ser calculadas pela ferramenta. Por padrão, essas quatro regras são pré-selecionadas para ser computadas. No entanto, o usuário pode filtrar e calcular somente aquelas de seu interesse. Transação em regras de associação refere-se ao total de número de classes ou métodos pertencentes ao sistema, de acordo com a granularidade do *bad smell* usado.

Geração, Visualização e Exportação de Resultados. Após o cruzamento de dados, o usuário pode consultar os seguintes relatórios: número de instâncias de padrões de projeto no sistema e quantidade e informações sobre os artefatos que apresentaram coocorrências de padrões de projeto com *bad smell*. Além disso, depois da aplicação das regras de associação, o relatório com os resultados são exibidos para o usuário. Esses relatórios são apresentados na forma de tabela. *Design Pattern Smell* exporta esses resultados para um arquivo *CSV* a fim de facitar análises e manipulações futuras.

Gerenciamento de Dados. Essa funcionalidade permite o usuário usar dados de padrões de projeto já armazenados na *Design Pattern Smell* para um cruzamento de dados com outro *bad smell*. Nesse caso, o usuário deve limpar as informações referentes ao *bad smells* em análise e executar a importação do outro arquivo *CSV*. Esse processo também pode ser realizado para padrões de projeto quando o usuário deseja alterar o sistema em análise.

Ajuda. Para usuário não familiarizado com regras de associação, essa funcionalidade descreve esse métodos de análise, bem como as fórmulas usadas para calcular essas relações, uma visão geral de como analisar os resultados, e as definições dos termos técnicos usados para calcular as fórmulas. Além disso, essa funcionalidade apresenta informação geral sobre a versão dessa ferramenta, e fornece um *link* de um vídeo tutorial que apresenta um exemplo de execução da ferramenta.

5.3 Arquitetura

Esta seção descreve a arquitetura utilizada no desenvolvimento da *Design Pattern Smell*. Seção 5.3.2 apresenta uma visão geral da arquitetura da ferramenta, discutindo as responsabilidades de cada um de seus módulos internos. Seção 5.3.2 apresenta a organização interna dos componentes da ferramenta, via diagrama de pacotes, discutindo o propósito de cada um desses pacotes.

5.3.1 Visão Geral

A arquitetura da *Design Pattern Smell* é composta de seis módulos internos como mostrado na Figura 5.1.

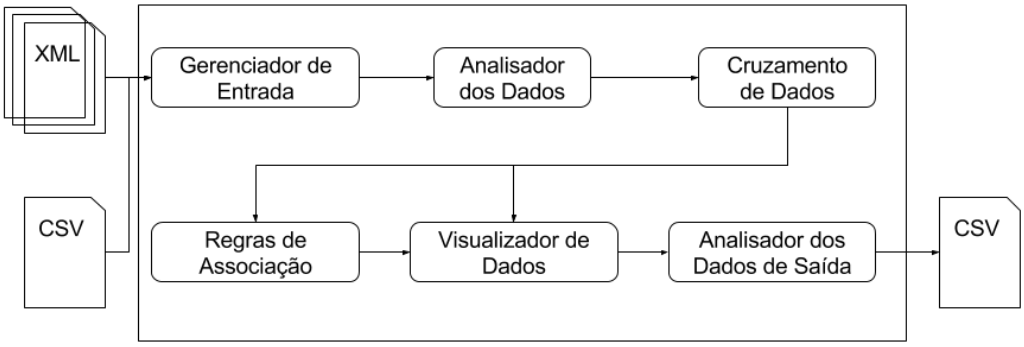

Figura 5.1. Arquitetura da *Design Pattern Smell Architecture*.

Gerenciador de Entrada. Esse módulo é responsável pelo gerenciamento dos arquivos de entrada bem como a realização da verificação e validação dos formatos requisitados pela *Design Pattern Smell*. Além disso, ele realiza uma limpeza de dados e prepara a ferramenta para o recebimento de novas informações de instâncias de padrões de projeto ou *bad smell*.

Analisador de Dados. Esse módulo é responsável por analisar as informações nos arquivos de entrada. Ele pode receber um ou mais arquivos *XML* contendo instâncias de padrões de projeto computadas. Cada instância pode ser composta por vários componentes que desempenham um determinado papel nessas instâncias. Nesse módulo, esses componentes são separados e classificados de acordo com sua granularidade: classe, método ou atributo. O arquivo *CSV* é analisado, e as informações em cada linha daquele arquivo é agrupado como um tipo de componente e ordenado de acordo com a granularidade especificada pelo usuário. Todas informações extraídas,

tanto dos arquivos *XML* quanto dos arquivos *CSV*, são persistidas em memória e utilizadas pelos outros módulos. O formato padrão para os arquivos de entrada está disponível no *website* da ferramenta [Sousa et al., 2016].

Cruzamento de Dados. Esse módulo realiza o cruzamento dos dados extraídos no módulo anterior, Analisador de Dados, para identificar artefatos que possuem coocorrência de padrão de projeto e *bad smell*. Nesse cruzamento de dados, é verificado se cada classe ou métodos com presença de *bad smell* é parte de alguma instância de padrão de projeto. Se sim, *Design Pattern Smell* lista esses artefatos como coocorrência.

Regras de Associação. Esse módulo implementa a aplicação das regras de associação nas informações fornecidas para a ferramenta, a fim de identificar a intensidade das coocorrências entre padrões de projeto e *bad smell*. Ele usa informações quantitativas, computadas no módulo Cruzamento de Dados, juntamente com a quantidade total de transações do sistema fornecido pelo usuário para calcular as regras.

Visualizador de Dados. Esse módulo permite a geração de relatórios em forma de tabela. A partir disso, o usuário pode navegar pela lista dos artefatos identificados com coocorrência. Além disso, é possível emitir outros tipos de relatórios com a informação computada, tanto para o módulo de Cruzamento de Dados quanto para o módulo de Regras de Associação. Os outros relatórios que podem ser emitidos pela *Design Pattern Smell* são: quantidade de instâncias de padrões de projeto em um sistema, taxa de artefatos afetadas, e intensidade da coocorrência identificada em cada padrão existente no sistema.

Analisador dos Dados de Saída. Esse módulo executa uma análise dos relatórios emitidos pelo módulo Visualizador de Dados e gera um arquivo de saída *CSV*, o qual é armazenado em um local definido pelo usuário em sua máquina. Esse arquivo contém as mesmas informações exibidas pelo módulo Visualizador de Dados e pode ser útil para analisar as coocorrências identificadas.

5.3.2 Organização Interna

A Figura 5.2 mostra a organização interna da *Design Pattern Smell*, via diagrama de pacotes. Nessa representação é possível ver os pacotes existentes na ferramenta e a dependência entre eles. Cada um dos pacotes representados na Figura 5.2 está descrito

a seguir:

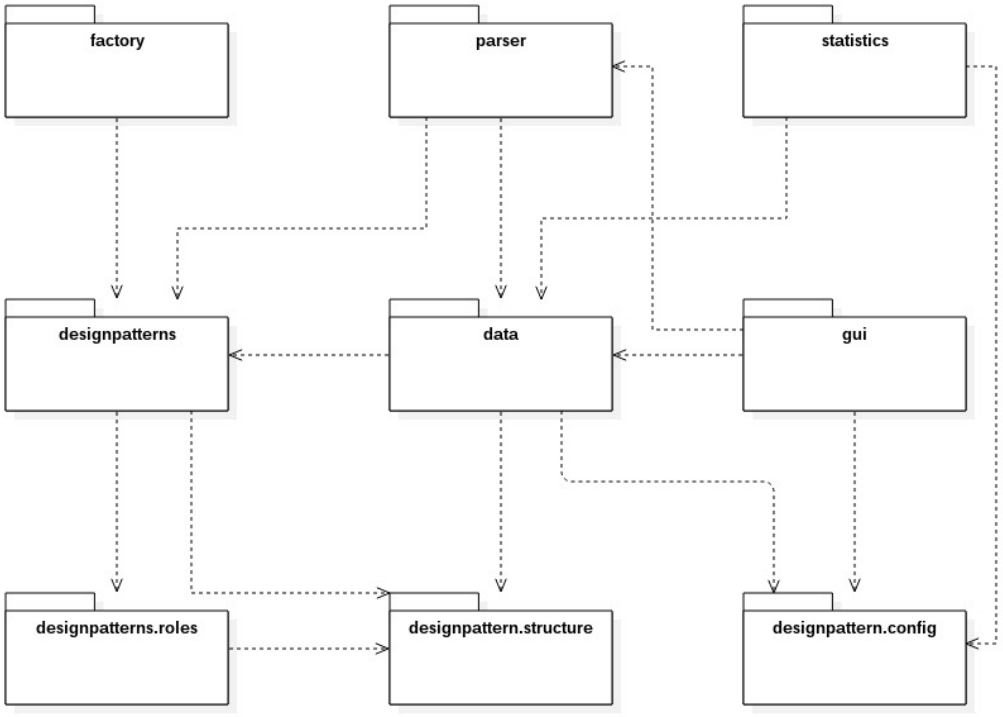

Figura 5.2. Diagrama de pacotes da ferramenta *Design Pattern Smell*.

- *parser:* esse pacote agrupa classes responsável por analisar os arquivos fornecidos como entrada para a *Design Pattern Smell*, bem como realizar análise e conversão dos dados existentes nesses arquivos.

- *designpatterns:* esse pacote agrupa classes que têm o objetivo de representar os padrões de projeto suportados pela *Design Pattern Smell*. Além dessas classes, esse pacote agrupa outros três pacotes: *config*, *role* e *structure*.

- *designpatterns.roles:* esse pacote engloba classes responsáveis pela representação dos papéis que os componentes de uma instância de padrão de projeto desempenham. Por exemplo, uma instância do padrão de projeto *Observer* é composto por classes que podem desempenhar o papel de Observador (*Observer*) ou Assunto (*Subject*) e métodos que exercem o papel de notificadores (*notify*). Esses papéis foram modelados como componentes e armazenados dentro do respectivo pacote.

- *designpatterns.structure:* esse pacote possui classes responsáveis por representar estruturas básicas para formação dos atributos internos de um padrão.

- *designpatterns.config:* esse pacote possui classes de configuração da ferramenta.
- *data:* esse pacote possui classes que armazenam as informações fornecidas para *Design Pattern Smell*.
- *factory:* esse pacote possui classes responsável pela criação de objetos na *Design Pattern Smell*.
- *statistics:* esse pacote possui classes responsável pelo cálculo e aplicação das regras de associação nos dados fornecidos para *Design Pattern Smell*.
- *gui:* esse pacote contém classes que representam as interfaces gráficas *GUI*, responsáveis por realizar as interações com o usuário.

5.4 Implementação

Design Pattern Smell foi desenvolvida na linguagem de programação Java com suporte da JDK 1.7 e a API Java Swing para criação da interface gráfica de usuário (GUI). Java foi escolhida devido a sua portabilidade, e além disso, é uma linguagem difundida tanto na academia quanto na indústria.

Para interpretar e realizar a conversão dos arquivos XML, foi utilizada a API JDOM[2] para interpretar e manipular os arquivos XML contendo métricas de software que são fornecidos para *Design Pattern Smell*.

Para apresentar as regras de associação, na opção *"Help"*, foi utilizada a API JLaTeXMath 1.0.3[3] para mostrar as fórmulas matemáticas usadas para cada uma das métricas.

Por fim, *Design Pattern Smell* foi construída por meio da IDE NetBeans 8.0.2[4], que fornece a funcionalidade *drag and drop* para construção da interface gráfica com usuário. *Design Pattern Smell* está disponível na Versão 1.0 no *website* da ferramenta [Sousa et al., 2016].

5.5 Exemplo de Uso

Esta seção apresenta um exemplo de uso das principais funcionalidades da *Design Pattern Smell*.

[2] `http://www.jdom.org/`
[3] `https://forge.scilab.org/index.php/p/jlatexmath/`
[4] `https://netbeans.org/`

Figura 5.3 mostra a tela principal da *Design Pattern Smell*. Essa tela fornece três tipos de funcionalidades para o usuário: (i) no painel *"Import XML Files with Design Pattern Instances"*, o usuário importa os arquivos de entrada *XML* referentes a um sistema alvo e insere o nome do referido sistema no campo *"Name of the System"*; (ii) no painel *"Data Crossing"*, o usuário importa um arquivo *CSV* com classes ou métodos do sistema alvo que tem *bad smell*. Quando ocorre a importação desse arquivo, o nome do *bad smell* e sua granularidade devem ser informados; (iii) no topo da tela principal, existe um menu com quatro opções: *"File"* permite ao usuário entrar com novas informações sobre outros *bad smells* e padrões de projeto; *"Results"* fornece informações e relatórios sobre as instâncias de padrões de projeto e os artefatos afetados; *"Statistics"* permite ao usuário aplicar regras de associação nos dados usados para identificar as coocorrências; e *"Help"* contém uma referência e descrição das regras de associação usada pela *Design Pattern Smell*, bem como informações sobre a versão da ferramenta, lista de desenvolvedores e *links* para o código fonte e um vídeo tutorial com um exemplo de execução.

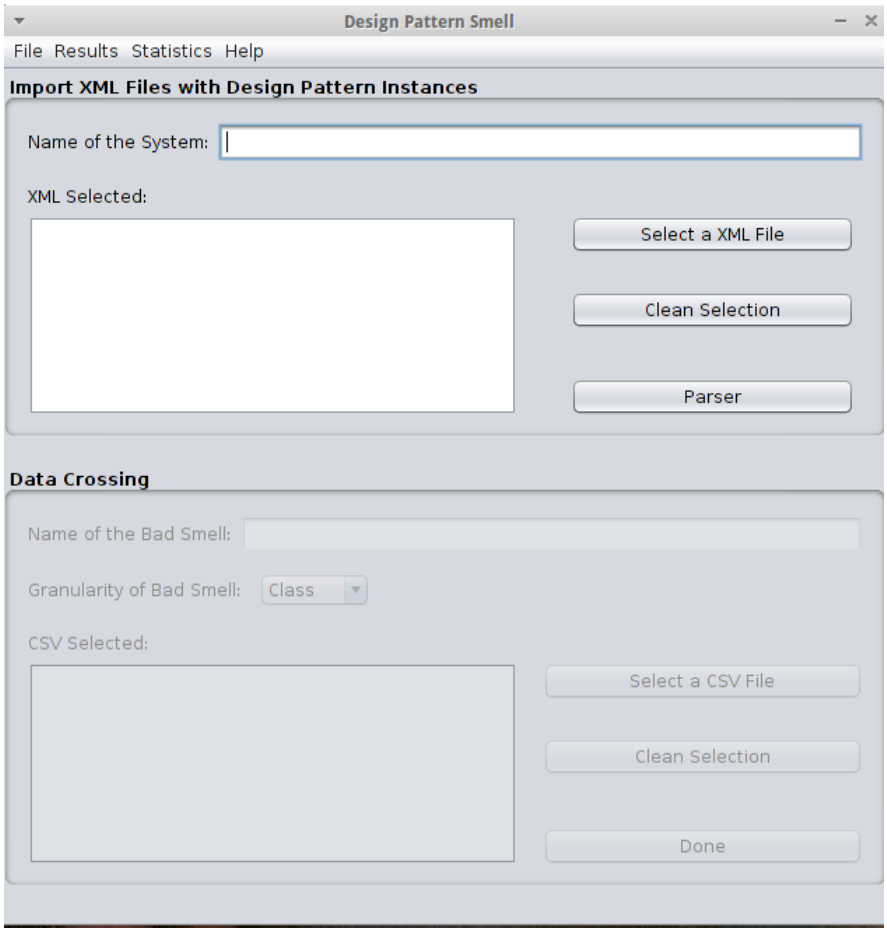

Figura 5.3. Tela principal para importação dos arquivos XML e CSV.

Figura 5.4 mostra a tela para aplicação das regras de associação. Nessa tela, o usuário entra com um valor indicando o número de transações existentes no sistema alvo. Transação nesse contexto refere-se ao número total de classes ou métodos que o sistema alvo possui. O usuário ainda pode filtrar as regras que serão calculadas. Por padrão, as quatro regras (*Support*, *Confidence*, *Lift* e *Conviction*) são pré-selecionadas para serem calculadas. No entanto, o usuário pode selecionar as regras para ser calculadas. Após o cálculo das regras de associação, os valores são exibidos no painel *"Association Rules Results"* em um formato de tabela que pode ser exportado para um arquivo *CSV*.

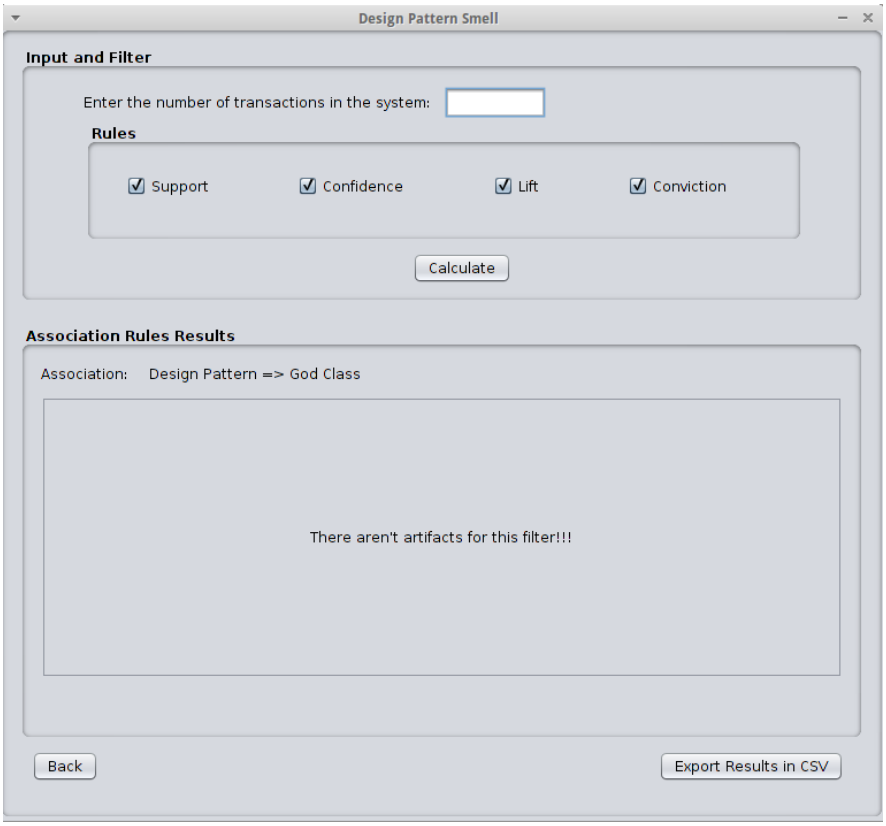

Figura 5.4. Tela para aplicação das regras de associação.

Após a execução do cruzamento dos dados importados na tela exibida pela Figura 5.3, *Design Pattern Smell* permite ao usuário visualizar o número de artefatos identificados com coocorrência de padrões de projeto e *bad smell*. Figura 5.5 ilustra a tela responsável pela exibição dessa informação. Essa tela é formada por uma tabela cujas linhas referem-se a um padrão de projeto específico. Para cada linha, é informado o total de artefatos que compõe o respectivo padrão de projeto, o número de artefatos que foram afetados por *bad smell*, e a porcentagem de artefatos afetados. Esses resul-

tados podem ser exportados em um arquivo *CSV* via o botão no canto inferior direito chamado *"Export Results in CSV"*.

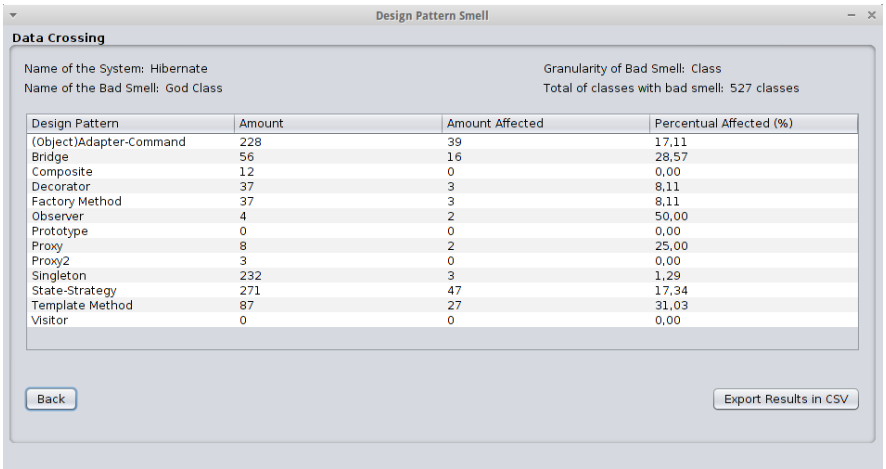

Figura 5.5. Visualização da quantidade de artefatos afetados por coocorrência.

Finalmente, depois de realizar o cruzamento dos dados, o usuário pode visualizar os artefatos nos quais as coocorrências foram identificadas. Para acessar essa informação o usuário deve selecionar a opção *"Artifacts with Co-occurrence"* dentro do menu *"Results"* e será redirecionado para a tela mostrada na Figura 5.6. Essa tela exibe uma tabela com a lista de artefatos afetados. Cada linha refere a um tipo de artefato e exibe informações tais como: nome, arquivo (no caso dos métodos), pacote, padrão de projeto que esse artefato é referente, e papel desempenhado dentro da instância do padrão de projeto. Além disso, o usuário pode restringir as informações exibidas nessa tela para um padrão de projeto específico. Para fazer isso, o usuário deve selecionar as opções desejadas e pressionar o botão *"Filter"*. Os resultados podem ser exportados para um arquivo *CSV*.

5.6 Conclusão

Este capítulo apresentou *Design Pattern Smell*, uma ferramenta para detecção de coocorrências entre padrões de projeto e *bad smells*. Essa ferramenta recebe como entrada arquivos *XML* com instâncias de padrões de projeto e um arquivo *CSV* com artefatos de código que tem presença de *bad smell*. *Design Pattern Smell* fornece uma interface simples e intuitiva para seleção dos arquivos de entrada e fornece agilidade na detecção dos artefatos com coocorrência. Além disso, *Design Pattern Smell* permite ao usuário aplicar regras de associação nos dados coletados para identificar a intensidade das coocorrências. A ferramenta permite também a geração e a exportação de relatórios.

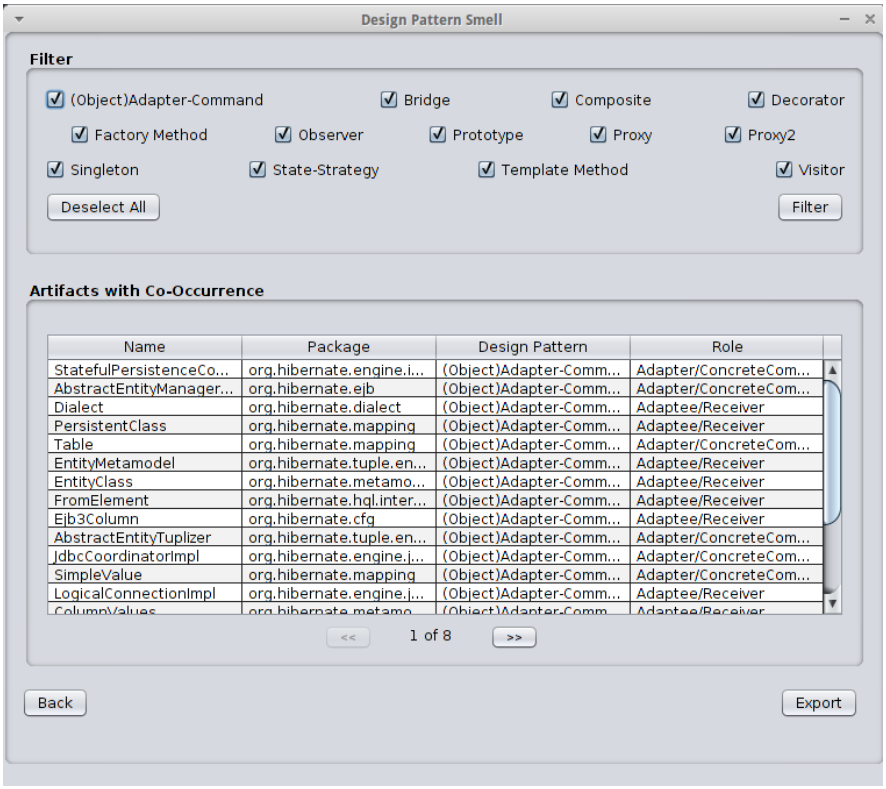

Figura 5.6. Tela para visualização dos artefatos afetados pelas coocorrências.

Design Pattern Smell atualmente suporta detecção de coocorrências de *bad smells* a nível de classe e método com 14 padrões de projeto *GOF*. A ferramenta foi aplicada neste trabalho para identificação dos artefatos que possuem coocorrência de padrões de projeto e *bad smells*.

Para instalação e uso da *Design Pattern Smell* é necessário apenas que a máquina virtual Java com versão igual ou superior a 1.7 esteja instalada. O executável dessa ferramenta está disponível em:

`http://llp.dcc.ufmg.br/Products/indexProducts.html`

O código fonte da *Design Pattern Smell* está disponibilizado no *GitHub*. O *link* de acesso ao código fonte da *Design Pattern Smell* é:

`https://github.com/BrunoLSousa/DesignPatternSmell`

Um vídeo tutorial com exemplo de uso da *Design Pattern Smell* foi disponibilizado *online*, por meio do seguinte *link*:

`http://llp.dcc.ufmg.br/Products/indexProducts.html`

Como proposta de trabalho futuro, deseja-se expandir o conjunto de padrões de projeto para todos os 23 padrões que fazem parte do catálogo *GOF*. Além disso, deseja-se adicionar um módulo de análise estática em código fonte de sistemas de software Java para identificação automática das instâncias de padrões de projeto, e adicionar outros métodos estatísticos para análise da intensidade de coocorrências.

Capítulo 6

Estudo de Caso

Este capítulo apresenta o estudo de caso realizado neste trabalho. Seção 6.1 descreve o Estudo de Caso. Seção 6.2 apresenta os resultados obtidos após a execução dos experimentos. Seção 6.3 discute a Questão de Pesquisa 1 (QP1). Seção 6.4 responde a Questão de Pesquisa (QP2) e sumariza os padrões de projeto que apresentaram coocorrência com cada um dos *bad smells* estudados. Seção 6.5 discute as situações que ocasionaram o surgimentos das coocorrências entre os padrões de projeto e *bad smells*. Seção 6.6 apresenta algumas ameaças à validade do estudo. Seção 6.7 discute algumas lições aprendidas após a condução do Estudo de Caso.

6.1 Descrição do Estudo de Caso

Foi conduzido um Estudo de Caso considerando um *data set* composto por cinco sistemas de software Java de código aberto. Esses sistemas passaram por processo de coleta de dados, no qual foram extraídas as informações de padrões de projeto e *bad smells*.

Inicialmente foram coletadas as informações referentes aos padrões de projeto dos sistemas. Esses dados foram extraídos por meio da ferramenta *DPDSS*. Após coletar os dados sobre padrões de projeto de cada um dos sistemas, realizou-se a filtragem dos métodos e classes que possuíam os cinco *bad smells* estudados neste trabalho. São eles: *Data Class*, *Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*. Para identificá-los, foram utilizadas as expressões de filtragem **Exp1**, **Exp2**, **Exp3**, **Exp4** e **Exp5** com a ferramenta *RAFTool* [Filó et al., 2014], conforme descrito na Seção 4.3.

Após implementar as estratégias de detecção com o suporte da *RAFTool*, os resultados obtidos foram validados manualmente pelo autor desta dissertação, a fim de eliminar ocorrência de falsos positivos. Essa validação foi realizada como base na implementação dos componentes retornados nos arquivos *CSV*, exportado pela *RAFTool*.

Os componentes identificados como falsos positivos foram removidos desses arquivos para identificação das coocorrências por meio da ferramenta *Design Pattern Smell*. As Tabelas 6.1, 6.2, 6.3, 6.4 e 6.5 mostram os resultados obtidos da coleta dos *bad smells*.

Software	# classes com *Data Class*	# total de classes	% classes com *Data Class*
`Hibernate`	825	7.711	10,70%
`JHotDraw`	70	1.061	6,60%
`Kolmafia`	441	3.225	13,67%
`Webmail`	14	129	10,85%
`Weka`	348	2.401	14,49%

Tabela 6.1. Resultados obtidos para *Data Class*.

Software	# classes com *Feature Envy*	# total de classes	% classes com *Feature Envy*
`Hibernate`	1.099	7.711	14,25%
`JHotDraw`	101	1.061	9,52%
`Kolmafia`	422	3.225	13,09%
`Webmail`	17	129	13,18%
`Weka`	421	2.401	17,53%

Tabela 6.2. Resultados obtidos para *Feature Envy*.

Software	# classes com *Large Class*	# total de classes	% classes com *Large Class*
`Hibernate`	79	7.711	1,02%
`JHotDraw`	15	1.061	1,41%
`Kolmafia`	103	3.225	3,19%
`Webmail`	2	129	1,55%
`Weka`	175	2.401	7,29%

Tabela 6.3. Resultados obtidos para *Large Class*.

Software	# métodos com *Long Method*	# total de métodos	% métodos com *Long Method*
`Hibernate`	331	48.234	0,69%
`JHotDraw`	133	7.633	1,74%
`Kolmafia`	1.015	28.214	3,60%
`Webmail`	24	1.091	2,20%
`Weka`	860	20.871	4,12%

Tabela 6.4. Resultados obtidos para *Long Method*.

Realizada a coleta dos dados de padrões de projeto e *bad smell*, no passo seguinte foram coletadas as coocorrências entre essas duas estruturas. Para identificação das

Software	# classes com *Refused Bequest*	# total de classes	% classes com *Refused Bequest*
`Hibernate`	2.050	7.711	26,59%
`JHotDraw`	411	1.061	38,74%
`Kolmafia`	960	3.225	29,77%
`Webmail`	29	129	22,48%
`Weka`	1.025	2.401	42,69%

Tabela 6.5. Resultados obtidos para *Refused Bequest*.

coocorrências, foi usada a ferramenta *Design Pattern Smell*, descrita no Capítulo 5. O restante deste capítulo apresenta e discute os resultados observados no Estudo de Caso.

6.2 Resultados

Os resultados da identificação das entidades com coocorrências podem ser visualizados nas Tabelas: 6.6, 6.7, 6.8, 6.9 e 6.10 . Em todas essas tabelas, a coluna "T" indica o número total de classes ou métodos que possuem padrão de projeto e a coluna "PP&BS" indica a quantidade de classes ou métodos identificados com relações de coocorrência entre padrões de projeto (PP) e *bad smells* (BS).

Padrão de Projeto	`Hibernate` 4.2.0		`JHotDraw` 7.5.1		`Kolmafia` 17.3		`Webmail` 0.7.10		`Weka` 3.6.9	
	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS
Adapter-Command	228	40	53	13	386	75	40	8	152	23
Bridge	56	2	40	0	14	1	6	3	0	0
Composite	12	0	12	1	8	0	0	0	0	0
Decorator	37	2	10	1	67	7	0	0	32	10
Factory Method	37	0	5	0	31	0	2	0	22	1
Observer	4	2	2	1	8	1	0	0	36	1
Prototype	0	0	21	4	0	0	0	0	0	0
Proxy	8	3	0	0	18	6	0	0	35	10
Singleton	232	3	13	1	77	9	1	1	34	0
State-Strategy	271	51	121	24	334	52	23	3	93	18
Template Method	87	5	16	2	54	3	4	2	22	1

Tabela 6.6. Quantidade total de classes com padrão de projeto e quantidade de classes com coocorrências entre padrão de projeto e *Data Class*.

Após a coleta das coocorrências, foram aplicadas as regras de associação, a fim de identificar a intensidade das relações de coocorrências entre os padrões de projeto e *bad smells*, e com isso, responder as questões de pesquisa. Sendo assim, as métricas mostradas na Seção 4.4 foram aplicadas nos dados mostrados nas Tabelas 6.6, 6.7, 6.8, 6.9 e 6.10, e como ponto de análise dos resultados, utilizou-se a métrica `Convicção` seguindo os limiares descritos também na Seção 4.4. As Figuras 6.1, 6.2, 6.3, 6.4 e 6.5

Padrão de Projeto	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS
Adapter-Command	228	36	53	14	386	74	40	6	152	64
Bridge	56	18	40	7	14	4	6	2	0	0
Composite	12	0	12	4	8	0	0	0	0	0
Decorator	37	3	10	2	67	5	0	0	32	9
Factory Method	37	2	5	0	31	3	2	0	22	3
Observer	4	1	2	1	8	2	0	0	36	11
Prototype	0	0	21	5	0	0	0	0	0	0
Proxy	8	1	0	0	18	6	0	0	35	16
Singleton	232	2	13	0	77	5	1	1	34	5
State-Strategy	271	41	121	31	334	54	23	2	93	43
Template Method	87	27	16	5	54	14	4	1	22	8

Tabela 6.7. Quantidade total de classes com padrão de projeto e quantidade de classes com coocorrências entre padrão de projeto e *Feature Envy*.

Padrão de Projeto	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS
Adapter-Command	228	13	53	6	386	33	40	2	152	35
Bridge	56	8	40	2	14	3	6	0	0	0
Composite	12	0	12	1	8	0	0	0	0	0
Decorator	37	2	10	1	67	2	0	0	32	7
Factory Method	37	1	5	0	31	2	2	0	22	0
Observer	4	1	2	0	8	1	0	0	36	7
Prototype	0	0	21	1	0	0	0	0	0	0
Proxy	8	1	0	0	18	1	0	0	35	9
Singleton	232	0	13	0	77	2	1	0	34	3
State-Strategy	271	21	121	7	334	30	23	1	93	29
Template Method	87	6	16	2	54	4	4	0	22	2

Tabela 6.8. Quantidade total de classes com padrão de projeto e quantidade de classes com coocorrências entre padrão de projeto e *Large Class*.

Padrão de Projeto	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS
Adapter-Command	271	12	73	0	703	44	50	0	222	20
Bridge	61	3	51	2	19	3	8	0	0	0
Composite	8	0	29	0	37	0	0	0	0	0
Decorator	115	1	31	0	255	2	0	0	61	6
Factory Method	58	0	23	0	45	0	2	0	27	0
Observer	8	0	2	0	7	1	0	0	24	0
Prototype	0	0	16	2	0	0	0	0	0	0
Proxy	6	0	0	0	31	1	0	0	37	1
Singleton	340	0	15	0	672	0	1	0	83	0
State-Strategy	343	24	227	21	974	61	19	0	173	31
Template Method	275	8	47	2	161	19	14	0	34	4

Tabela 6.9. Quantidade total de classes com padrão de projeto e quantidade de classes com coocorrências entre padrão de projeto e *Long Method*.

mostram os resultados da métrica `Convicção` para os *bad smells Data Class*, *Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*, nessa ordem. Esses resultados serão discutidos em detalhes nas seções seguintes deste capítulo.

Padrão de Projeto	Hibernate 4.2.0		JHotDraw 7.5.1		Kolmafia 17.3		Webmail 0.7.10		Weka 3.6.9	
	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS	T	PP&BS
Adapter-Command	228	22	53	5	386	30	40	5	152	39
Bridge	56	4	40	4	14	0	6	1	0	0
Composite	12	1	12	5	8	0	0	0	0	0
Decorator	37	5	10	3	67	0	0	0	32	22
Factory Method	37	1	5	0	31	1	2	0	22	6
Observer	4	0	2	0	8	0	0	0	36	0
Prototype	0	0	21	8	0	0	0	0	0	0
Proxy	8	4	0	0	18	9	0	0	35	23
Singleton	232	59	13	7	77	34	1	0	34	3
State-Strategy	271	23	121	31	334	19	23	2	93	31
Template Method	87	14	16	6	54	6	4	0	22	10

Tabela 6.10. Quantidade total de classes com padrão de projeto e quantidade de classes com coocorrências entre padrão de projeto e *Refused Bequest*.

6.3 Questão de Pesquisa 1

Nesta seção, é realizada uma análise e discussão com o intuito de responder a Questão de Pesquisa 1 (QP1).

QP1. Os padrões de projeto definidos no catálogo *GOF* evitam a ocorrência de *bad smells* em software?

A maioria dos padrões de projeto explorados neste estudo possuem estruturas modulares que podem contribuir para reduzir o acoplamento interno do software e, assim, deixar o código mais flexível e menos complexo.

Contudo, apesar de possuir uma estrutura pré-definida na literatura, um padrão de projeto pode conter algumas variações em sua implementação que muitas das vezes ocorrem como forma de adaptação da solução para um determinado contexto. Quando essa adaptação é realizada, o padrão pode ser degradado, resultando na presença de *bad smells*.

Analisando as Figuras 6.1, 6.2, 6.3, 6.4 e 6.5, percebe-se que dois padrões de projeto apresentam baixa coocorrências com *bad smells*: *Composite* e *Factory Method*. A inspeção manual relevou que esses dois padrões de projeto possuem uma estrutura modular que divide as tarefas entre várias classes. A ideia do padrão *Composite* é construir objetos complexos por meio de objetos mais simples. Esses objetos mais simples são definidos em módulos, de modo que a inteligência seja dividida entre eles, reduzindo a complexidade das classes. O padrão *Factory Method* simula a ideia de uma fábrica em que existe uma interface para criar objetos, mas a criação em si é realizada pela subclasse que implementa tal interface. Assim, é possível criar vários módulos, cada qual responsável por criar e gerenciar as informações de um conjunto

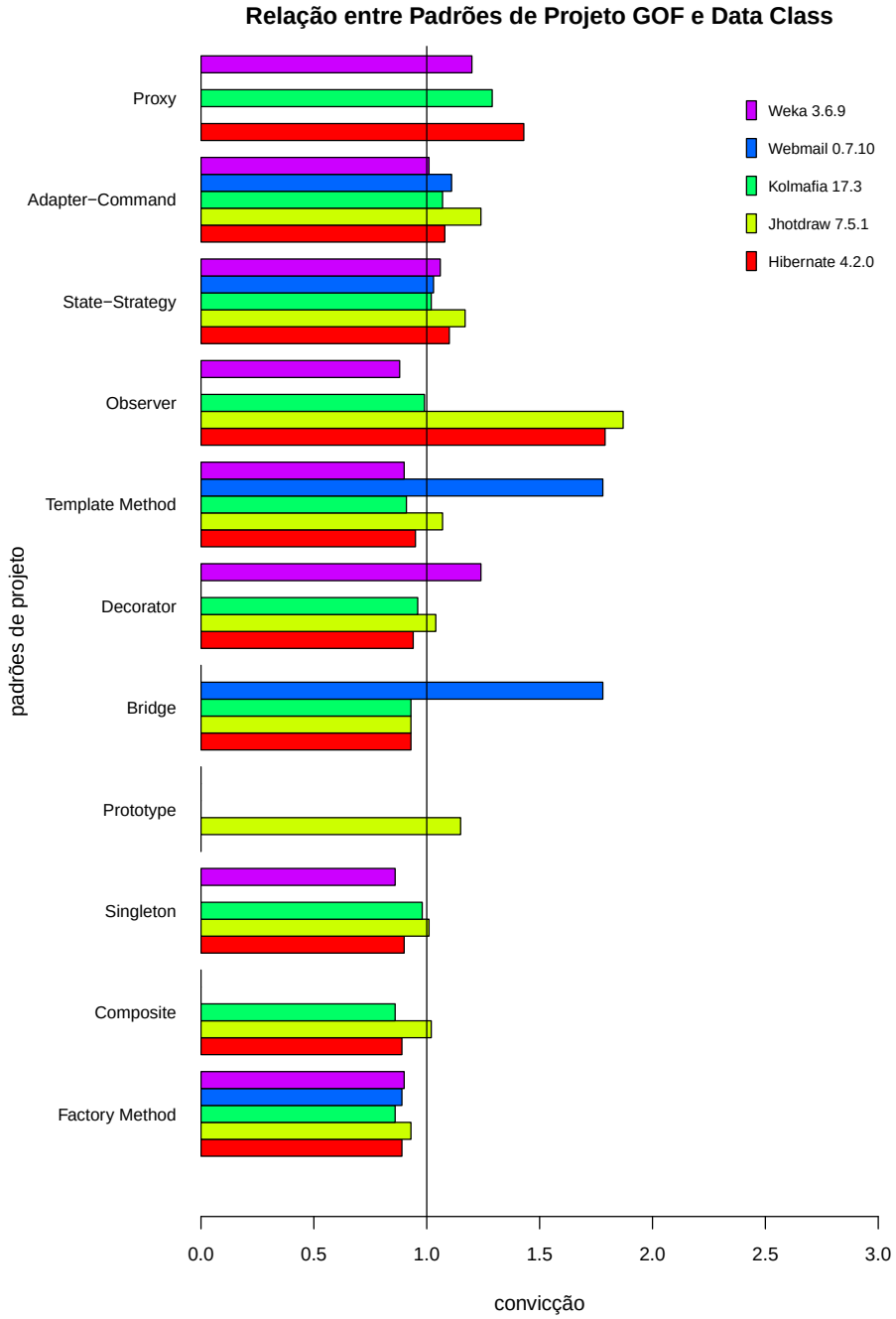

Figura 6.1. Resultado da associação (Padrão de Projeto ⇒ *Data Class*).

de objetos no sistema, removendo a carga de trabalho de uma única classe. Portanto, tanto *Composite* e *Factory Method* são padrões de projeto intrisicamente modulares.

Dos cinco *bad smells* avaliados, *Singleton* apresentou baixa coocorrência com três deles: *Data Class*, *Feature Envy* e *Large Class*. Esse resultado sugere que *Singleton* pode ser uma boa opção quando deseja-se evitar ocorrências de *bad smells* referentes

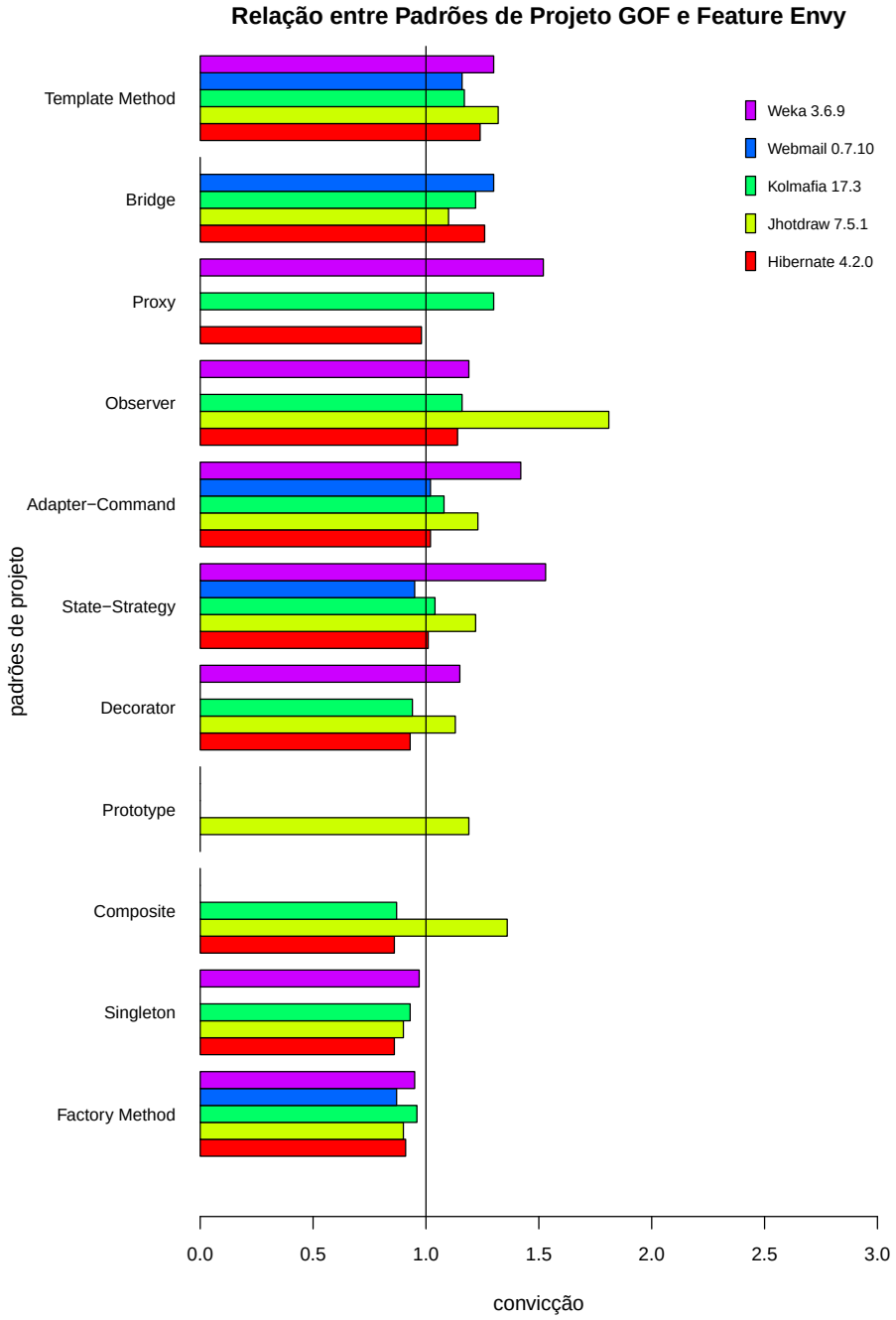

Figura 6.2. Resultado da associação (Padrão de Projeto ⇒ *Feature Envy*).

a complexidade a nível de classe em um software. Em relação ao *bad smell Long Method*, *Singleton* é um caso especial. Embora o resultado exibido pela Figura 6.4 mostre que há uma baixa coocorrência desse padrão de projeto com *Long Method*, ele é considerado um falso negativo. Esse resultado foi indicado como falso negativo porque as instâncias desse padrão de projeto identificadas pela *DPDSS* são baseadas apenas

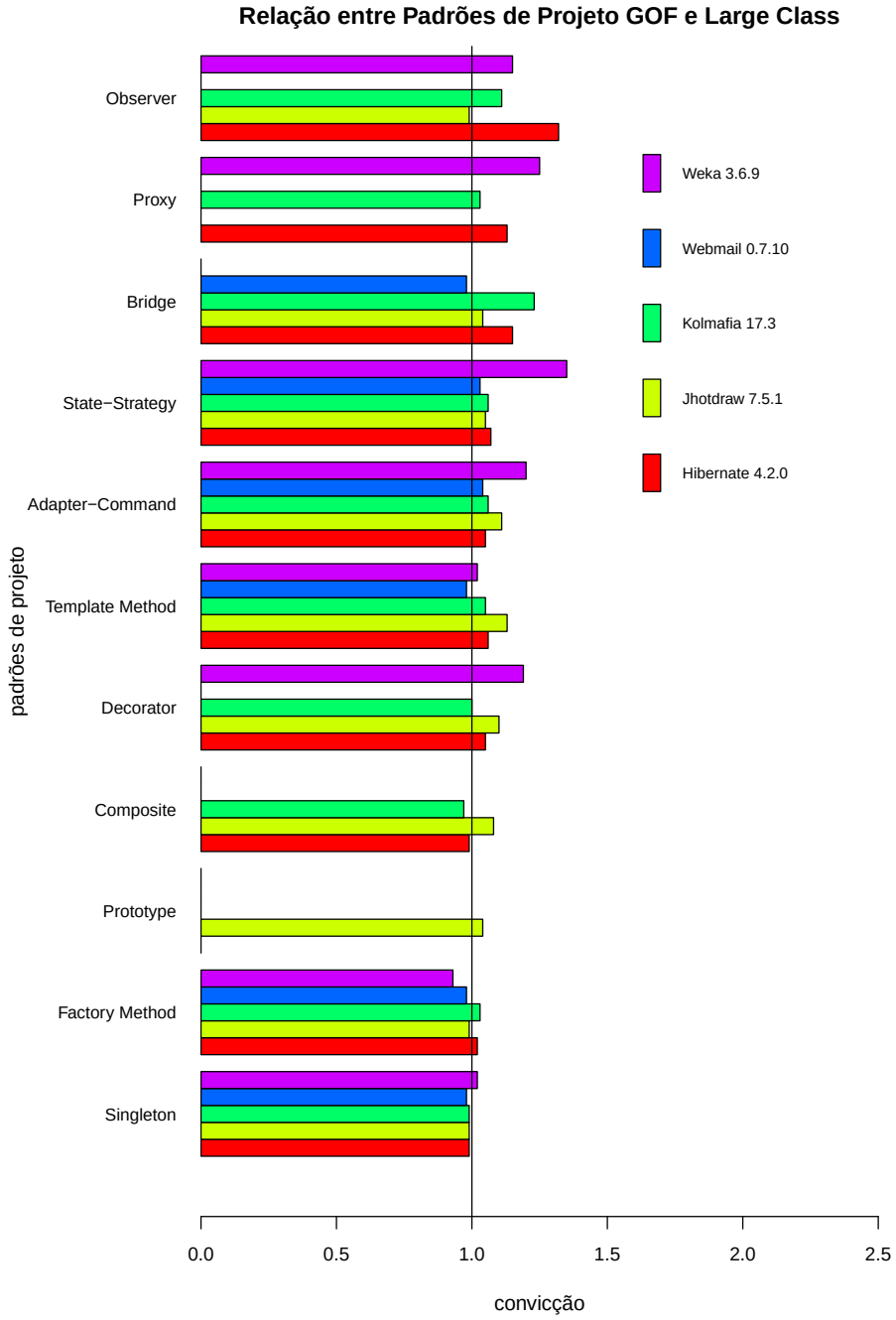

Figura 6.3. Resultado da associação (Padrão de Projeto ⇒ *Large Class*).

no atributo estático apresentado na classe. Essa ferramenta não considera método algum como característica desse padrão. Por esse motivo, ao realizar o cruzamento das informações de padrão de projeto e *bad smell*, ele retornou zero. Contudo, quando as classes *Singleton* foram manualmente inspecionadas, foram encontradas algumas ocorrências desse *bad smell* dentro dessas classes.

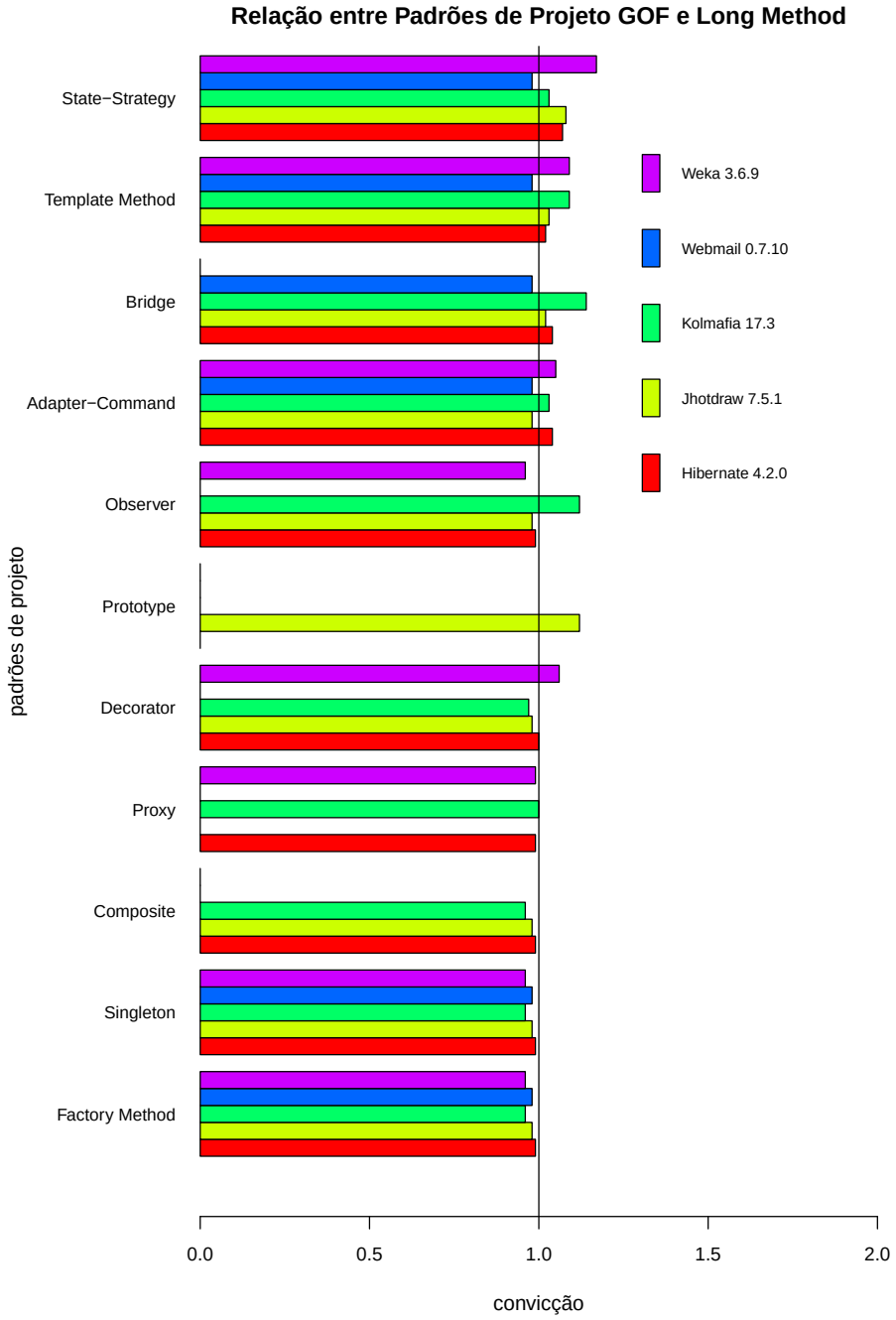

Figura 6.4. Resultado da associação (Padrão de Projeto ⇒ *Long Method*).

Sumário da QP1. Embora os padrões de projeto *Factory Method*, *Composite* e *Singleton* tenham apresentado baixa coocorrência com os *bad smells* estudados, a conclusão geral dessa análise é que a maioria dos padrões de projeto estão associados aos *bad smells* estudados. Portanto, a resposta da QP1 é: "Não, os padrões de projeto *GOF* não necessariamente evitam ocorrências de *bad smells*".

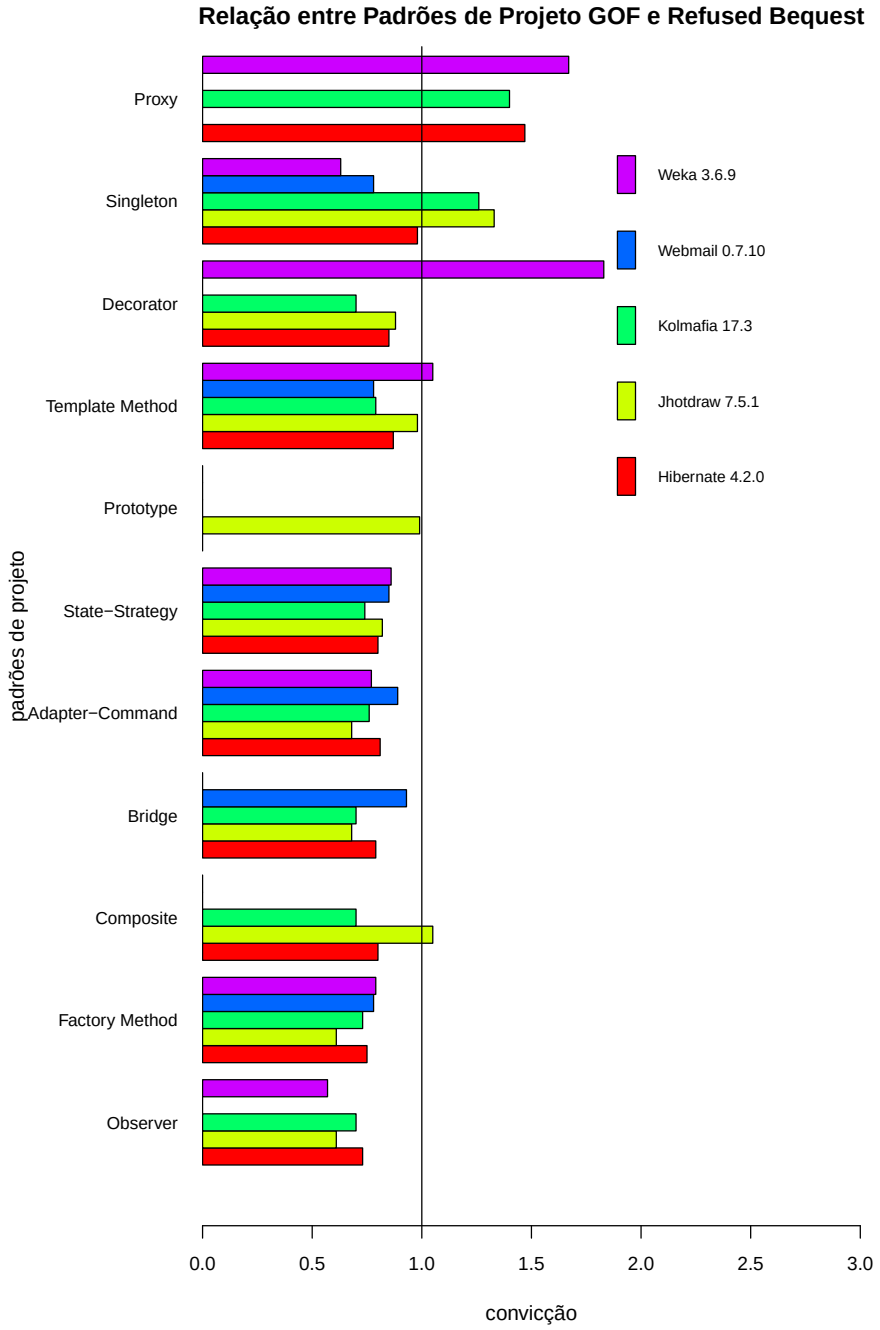

Figura 6.5. Resultado da associação (Padrão de Projeto ⇒ *Refused Bequest*).

6.4 Questão de Pesquisa 2

A análise realizada nesta seção tem como principal objetivo responder a Questão de Pesquisa 2 (QP2).

QP2. Quais padrões de projeto do catálogo *GOF* apresentaram coocorrência com *bad smells*?

A fim de analisar as associações entre padrões de projeto e *bad smells* foram considerados os valores da `Convicção`. A escolha dessa métrica ocorreu devido ao fato de ela ser capaz de estabelecer uma relação entre as métricas `Suporte` e `Confiança`. Além disso, `Convicção` possui uma melhor sensibilidade na identificação das relações entre `Antecedente` e `Consequente`. Assim, a fim de identificar as relações de coocorrências entre os padrões de projeto e *bad smells* foram considerados os limiares da métrica `Convicção`, mencionados na Seção 4.4. Como critério para identificação das coocorrências, foram observados casos prevalentes, ou seja, casos em que padrões de projeto apresentaram valor de convicção maior que 1 para a maior parte dos sistemas que possuíam instâncias do respectivo padrão de projeto.

Analisando os resultados da Figura 6.1, percebe-se que os padrões de projeto *Proxy*, *Adapter-Command* e *State-Strategy* foram aqueles que apresentaram maior relação de coocorrência com o *bad smell Data Class*. O padrão de projeto *Observer* apresentou um alto índice de coocorrência para apenas dois sistemas, *JHotDraw* e *Hibernate*. Contudo, pelo critério utilizado para identificação das coocorrências, não é possível afirmar que o *Observer* apresentou coocorrência com *Data Class*. *Template Method* e *Bridge* também apresentaram alta coocorrência para um único sistema, *Webmail*, porém no geral, as relações de coocorrência não foram muito intensas comparadas a *Proxy*, *Adapter-Command* e *State-Strategy*. Portanto, esses resultados sugerem que os padrões de projeto *Proxy*, *Adapter-Command* e *State-Strategy* foram aqueles que apresentaram maior relação de cocorrência com o *bad smell Data Class*.

A Figura 6.2 exibe um gráfico com os resultados da métrica `Convicção` calculada para o *bad smell Feature Envy*. Analisando esse gráfico, percebe-se que vários padrões de projeto possuem relação de coocorrência com esse *bad smell*. Dentre essas relações, é possível destacar seis padrões de projeto com alta coocorrência: *Template Method*, *Bridge*, *Proxy*, *Observer*, *Adapter-Command* e *State-Strategy*. No entanto a relação *Template Method* $\Rightarrow$ *Feature Envy* foi aquela que apresentou uma maior intensidade neste Estudo de Caso, dado que para todos os sistemas do *data set* utilizado, foi o padrão de projeto que apresentou maiores valores de `Convicção`.

A Figura 6.3 exibe um gráfico com os resultados da métrica `Convicção` calculada para o *bad smell Large Class*. Analisando esse gráfico percebe-se que, assim como para o *bad smell Feature Envy*, existem vários padrões de projeto que possuem relações de coocorrência mais intensa. Dentre as relações exibidas, é possível listar sete diferentes padrões de projeto que apresentaram relações de coocorrência: *Observer*, *Proxy*, *State-*

Strategy, *Adapter-Command*, *Bridge*, *Template Method* e *Decorator*. No entanto a relação *Observer* ⇒ *Large Class* foi aquela que apresentou uma maior intensidade em comparação com as demais coocorrências.

A Figura 6.4 apresenta os resultados obtidos para o *bad smell Long Method*, por meio de um gráfico com a métrica `Convicção`. Nesse gráfico é possível ver que os padrões de projeto não apresentaram uma relação tão alta comparada aos outros *bad smells*. Os padrões de projeto que apresentaram maior coocorrência com o *bad smell Long Method* foram: *State-Strategy*, *Template Method*, *Bridge* e *Adapter-Command*. Contudo, ao comparar a intensidade das coocorrências, percebeu-se que a relação *State-Strategy* ⇒ *Long Method* foi aquela que apresentou uma maior intensidade.

Por fim, os resultados obtidos para o *bad smell Refused Bequest*, Figura 6.5, sugerem que dois padrões de projeto, *Proxy* e *Singleton* apresentaram maior relação de coocorrência com esse *bad smell*. O padrão de projeto *Decorator* apresentou um alto valor de `Convicção` para um único sistema, *Weka*, contudo, para os outros sistemas esse valor manteve se baixo, e no geral, as relações de coocorrência desse padrão de projeto não foram tão intensas comparadas com as relações obtidas para *Proxy* e *Singleton*.

Sumário da QP2. As coocorrências identificadas neste Estudo de Caso estão sumarizadas na Tabela 6.11.

6.5 Questão de Pesquisa 3

A análise realizada nessa seção tem por objetivo responder a Questão de Pesquisa 3 (QP3).

QP3. Quais são as situações mais comuns em que *bad smells* aparecem em sistemas de software que aplicam os padrões de projeto *GOF*?

Para responder a Questão de Pesquisa 3 (QP3), essa seção está subdivida em outras 5 subseções, que discutem as situações que proporcionaram o surgimento de coocorrência dos padrões de projeto com cada um dos cinco *bad smells* abordados neste trabalho.

6.5.1 *Data Class*

Os resultados encontrados neste estudo indicam que os padrões de projeto *Proxy*, *Adapter-Command* e *State-Strategy* foram aqueles que apresentaram maior relação de

Bad Smell	**Coocorrência Identificada**
Data Class	*Proxy* *Adapter-Comand* *State-Strategy*
Feature Envy	*Template Method* *Bridge* *Proxy* *Observer* *Adapter-Command* *State-Strategy*
Large Class	*Observer* *Proxy* *State-Strategy* *Adapter-Command* *Bridge* *Template Method* *Decorator*
Long Method	*State-Strategy* *Template Method* *Adapter-Command* *Bridge*
Refused Bequest	*Proxy* *Singleton*

Tabela 6.11. Sumarização das coocorrências identificadas.

coocorrência com o *bad smell Data Class*. A fim de identificar o motivo de tais coocorrências, uma inspeção manual foi realizada nas classes que apresentaram a relação *Proxy* ⇒ *Data Class*.

O padrão de projeto *Proxy* é uma solução cujo propósito é fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso ao mesmo. Essa solução é formada por um conjunto de classes, das quais, a classe que exerce o papel de *Subjet* é responsável por padronizar a interface de acesso para as classes *RealSubject* e *Proxy*. A classe *RealSubject* representa o objeto real sobre o qual a classe *Proxy* exercerá um controle de acesso. A classe *Proxy* é aquela responsável por controlar o acesso a um objeto, nesse caso, o *RealSubject*.

Ao analisar as classes que apresentaram a relação *Proxy* ⇒ *Data Class*, percebeu-se que todas as classes com coocorrências exerciam o papel de *RealSubject*. Além disso, foi realizado uma inspeção manual nessas classes, a fim de identificar situações que oca-

sionaram o surgimento dessa relação. Foi constatado que essas classes possuem uma grande quantidade de atributos e muitas ocorrências de métodos *gets* e *sets* para atribuição e acesso dos valores armazenados nessas classes, configurando o papel figurativo de um banco de dados dentro dos sistemas.

A Figura 6.6 mostra um diagrama de classes extraído do *Hibernate*, contento uma classe que apresentou a coocorrência *Proxy* ⇒ *Data Class*. O padrão de projeto possui uma classe, *FromReferenceNode*, que exerce o papel de *Proxy*, e outra responsável por exercer o papel de *RealSubject*, *FromElement*. Como é possível observar, a classe *FromElement* possui uma grande quantidade de dados, caracterizando a ocorrência do *bad smell Data Class*. Baseado nisso, é possível concluir que a implementação da classe *RealSubject* dentro do padrão de projeto *Proxy* foi o fator que contribuiu para o surgimento da coocorrência *Proxy* ⇒ *Data Class*.

Sumário. A principal situação que contribui para o surgimento da coocorrência *Proxy* ⇒ *Data Class* é a implementação da classe *RealSubject* dentro do padrão de projeto *Proxy*, mantendo uma grande quantidade de dados e poucas funcionalidades sobre esses dados.

6.5.2 *Feature Envy*

Nos resultados encontrados neste estudo, seis padrões de projeto apresentaram coocorrência com o *bad smell Feature Envy*. Para identificar o motivo que ocasionou essa relação, foi realizada uma inspeção manual nas classes com a coocorrência *Template Method* ⇒ *Feature Envy*, dado que essa foi a relação de maior intensidade para esse *bad smell*.

Template Method é uma solução que define o esqueleto de um algoritmo por meio de uma operação, transferindo alguns passos para as subclasses, que possuem o poder de redefinir as características desse algoritmo sem precisar alterar sua estrutura. Esse padrão de projeto é formado por um conjunto de classes, das quais, uma classe representa a classe *Template* (*AbstractClass*) que define as operações primitivas e genéricas para todas as subclasses. Na *AbstractClass*, um método *template*, é definido, e nele é implementado o esqueleto do algoritmo desejado. As subclasses referentes a classe template, *ConcreteClass*, são responsáveis por redefinir as características de um objeto, utilizando o método *template* definido na *AbstractClass*. Esse padrão de projeto utiliza uma estrutura modular, na qual os comportamentos de um objeto são modelados em subclasses e atribuídos ao mesmo por meio de polimorfismos. A vantagem dessa

Figura 6.6. Diagrama de Classes que representa a relação *Proxy* ⇒ *Data Class*.

implementação é a redução da complexidade na superclasse, uma vez que as definições de estruturas condicionais como *if* e *switch* podem ser substituídas por polimorfismos.

Ao analisar as classes que apresentaram a relação *Template Method* ⇒ *Feature Envy*, percebeu-se que as classes que apresentaram essa relação de coocorrência exercem o papel de *AbstractClass* dentro desse padrão de projeto. Além disso, alguns dos métodos *template* dessas classes contêm uma alta quantidade de acesso a métodos alocados em outras classes, elevando o nível de acoplamento das *AbstractClass* e baixando sua coesão. Por isso, foi possível concluir que a implementação desses métodos em classes inadequadas foi a principal situação que contribuiu para o surgimento dessa característica de tais classes. Como uma forma de solucionar esse problema, os métodos *template* deveriam ser refatorados, via extração de métodos, e a implementação de caráter "invejoso" deveria ser realocada para sua respectiva classe. Com essa ação, acredita-se que essas coocorrências poderiam ser amenizadas nos sistemas.

A Figura 6.7 mostra um diagrama de classes extraído do *Webmail*, contento uma classe que apresentou a coocorrência *Template Method* ⇒ *Feature Envy*.

Nesse diagrama, a classe *WebmailServer* é responsável por exercer o papel de *AbstractClass* e disponibilizar o(s) método(s) *template* com um algoritmo pré-definido. Essa classe possui três diferentes tipos de métodos *template*: *doInit*, *restart* e *shutdown*. A classe *WebmailServlet* exerce o papel de *ConcreteClass*, a qual tem liberdade de definir uma característica específica para o algoritmo definido nos métodos *template* da *AbstractClass*. No entanto, durante a inspeção manual, foi observado que dois dos três métodos *template*, *restart* e *shutdown*, fazem uso somente de funcionalidades referentes ao objeto *storage* instanciado nessa mesma classe. Em virtude disso, acredita-se que a implementação desses métodos contribuíram para o surgimento da característica invejosa dessa classe, uma vez que eles deveriam ser alocados dentro da classe referente ao objeto usado, *Storage*.

Sumário. A principal situação que contribuiu para o surgimento da coocorrência *Template Method* ⇒ *Feature Envy* foi a implementação de métodos acessados pelo método *template* fora da classe *AbstractClass*. Tais métodos implementam funcionalidade de mesmo interesse da classe *AbstractClass* e por isso deveriam ter sido implementados dentro da referida classe.

6.5.3 *Large Class*

Nos resultados obtidos para *Large Class*, sete padrões de projeto apresentaram coocorrência com esse *bad smell*: *Observer*, *Proxy*, *Bridge*, *State-Strategy*, *Adapter-Command*,

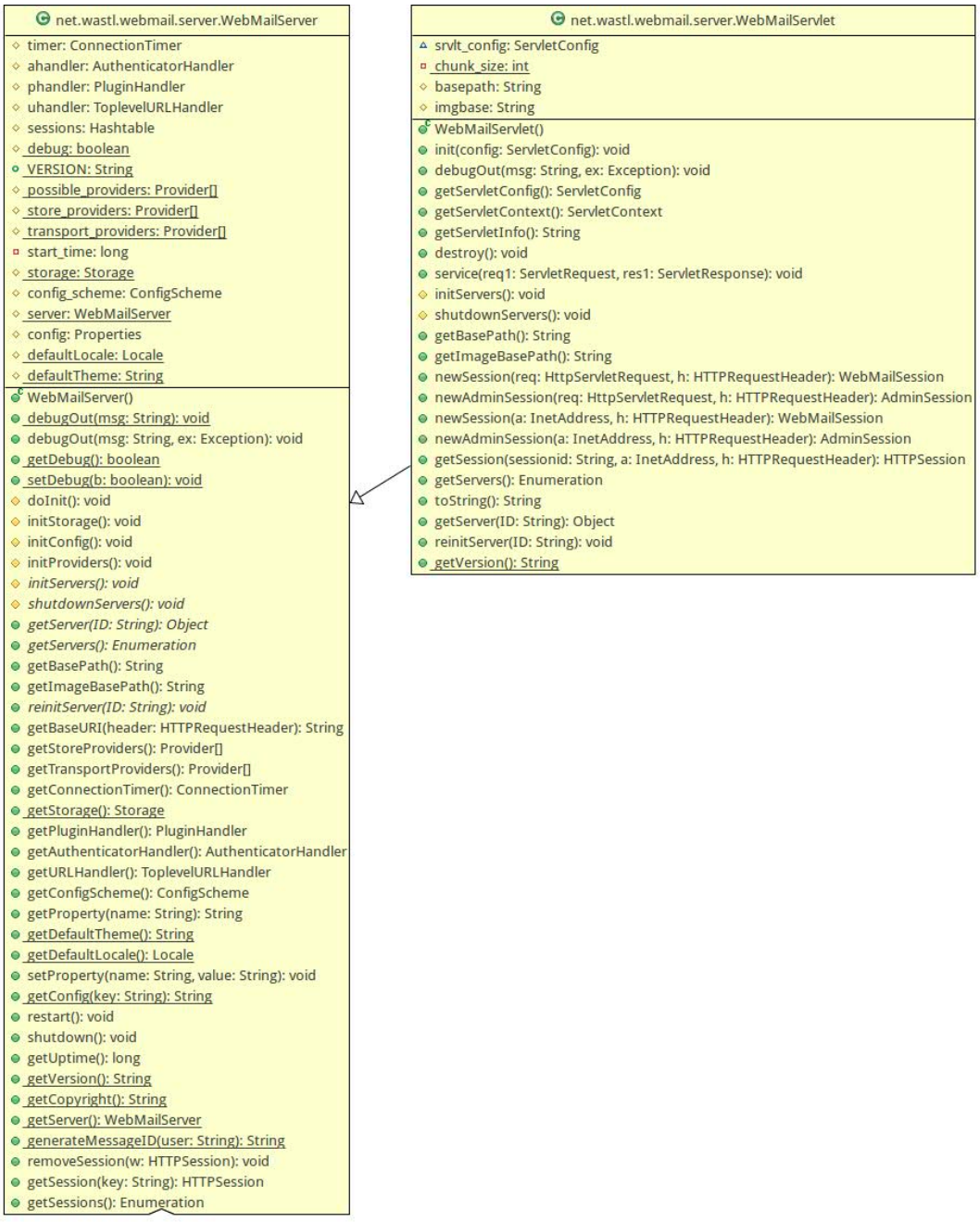

Figura 6.7. Diagrama de Classes que representa a relação *Template Method* ⇒ *Feature Envy*.

Template Method e *Decorator*. Para identificar as situações que ocasionaram essas relações, foi realizada uma inspeção manual nas classes que apresentaram a coocorrência *Observer* ⇒ *Large Class*.

O padrão de projeto *Observer* é uma solução que define uma dependência do tipo um para muitos entre os objetos. Esse padrão é composto por um conjunto de classes, das quais uma é responsável por exercer o papel de *Subject* e armazenar informações que são utilizadas pelas classes que exercem o papel de *Observer*. A classe *Subject* desse padrão de projeto possui uma lista com todas as classes *Observers* que utiliza seus dados, e quando alguma informação é alterada por qualquer classe *Observer*, a classe *Subject* é acionada, alterando os demais objetos *Observers*. O objetivo desse padrão de projeto é a sincronização de dados e atualização de objetos em tempo real. Essa atualização ocorre por meio do polimorfismo de inclusão, evitando o aumento da complexidade que geralmente ocorre com o uso de estruturas condicionais.

Ao analisar as classes que apresentaram a relação *Observer* ⇒ *Large Class*, percebeu-se que todas as classes com coocorrência exerciam o papel de *Subject* dentro de padrão de projeto. Além disso, durante a inspeção manual, foi possível identificar que a implementação inadequada dessa classe elevou a sua complexidade e, consequentemente, implicou no surgimento desta coocorrência.

A Figura 6.8 mostra um diagrama de classes extraído do *Weka*, contendo uma classe que apresentou a coocorrência *Observer* ⇒ *Large Class*.

Analisando o diagrama da Figura 6.8, é possível observar que a classe responsável por exercer o papel de *Subject*, representado pela classe *Classifier*, dentro do padrão de projeto *Observer*, possui implementação inadequada. Além de possuir uma grande quantidade de dados, observa-se pelo diagrama de classes, que ela trabalha com diversos tipos de observadores que provavelmente possuem responsabilidades diferentes. Portanto, a melhor prática nesse caso, seria criar diversos componentes *Subject*, um para cada tipo de interesse, e conectá-los com seus respectivos observadores. Essa ação evitaria que diversos interesses fossem tratados por um único *Subject* e reduziria a complexidade dessas classes. Por exemplo, deveria ser implementado um *Subject* para interagir com *GraphListener*, outro para interagir com *IncrementerClassifierListener*, e assim por diante. Assim, os métodos de notificação e os atributos e métodos utilizados por cada um dos observadores devem ser extraídos para outras classes, implicando na aplicação de uma refatoração de extração de classe.

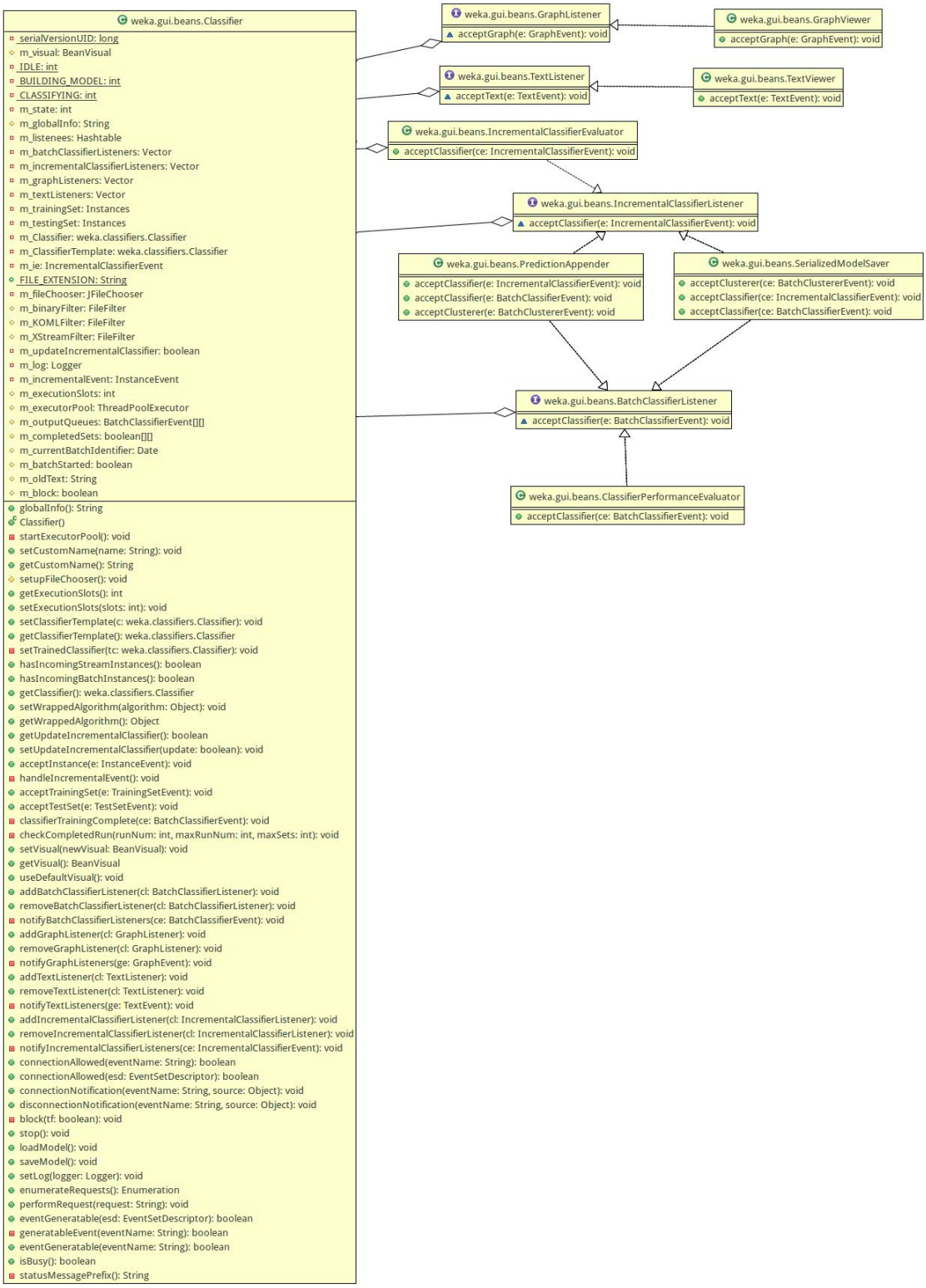

Figura 6.8. Diagrama de Classes que representa a relação *Observer* ⇒ *Large Class*.

Sumário. A principal situação que contribuiu para o surgimento da coocorrência *Observer* ⇒ *Large* foi a implementação inadequada do componente *Subject* dentro do padrão *Observer*, o que elevou a complexidade dessas classes e gerou a ocorrência do *bad smell Large Class*.

6.5.4 *Long Method*

Nos resultados obtidos para *Long Method*, dois padrões de projeto, *State-Strategy* e *Template Method* apresentaram uma maior taxa de coocorrência para esse *bad smell*. A fim de identificar algumas situações que ocasionaram essas coocorrências, a relação *State-Strategy* ⇒ *Long Method* foi inspecionada manualmente.

Como a ferramenta de detecção de padrão de projeto, *DPDSS*, utilizada neste estudo não diferencia os padrões *Strategy* e *State*, inicialmente não foi possível saber se um deles ou ambos coocorrem com o *bad smell Long Method*. No entanto, ao realizar a inspeção manual no código fonte dos sistemas, percebeu-se que o padrão de projeto *Strategy* aparece com muito mais frequência que o padrão de projeto *State*, e portanto possui uma maior taxa coocorrência com o *bad smell Long Method*.

O padrão de projeto *Strategy* foi proposto por Gamma et al. [1994] com o propósito de definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Além disso, esse padrão de projeto permite que o algoritmo varie independentemente dos clientes que o utilizam. Ele é composto por um conjunto de classes, das quais, uma é responsável por exercer o papel de *Strategy* e definir uma interface comum para a família de algoritmos de forma que a classe *Context* consiga utilizá-los. A classe *ConcreteStrategy* é responsável por realizar a implementação de cada um dos algoritmos definidos como estratégia. A classe *Context* é configurada como um objeto *ConcreteStrategy* e ela é responsável por repassar solicitações dos seus clientes para a estratégia configurada. A vantagem dessa solução é a flexibilidade proporcionada com o particionamento das regras de negócio de um sistema em pequenos componentes. Além disso, em caso de inclusão de novas regras ou algoritmos, essa solução permite que elas sejam adicionadas facilmente sem a necessidade de alterar o código fonte.

Ao analisar os métodos que apresentaram a relação *Strategy* ⇒ *Long Method*, percebeu-se que todos eles estavam situados dentro das classes *Context*, nos pontos em que eram definidas as estratégias a serem utilizadas pelos clientes. Além disso, durante a inspeção manual, constatou-se que esses métodos possuem excesso de código, devido ao fato das estratégias serem definidas por estruturas condicionais ao invés

do polimorfismo. Essa prática gerou uma alta complexidade para esses métodos e dificultou a legibilidade e entendimento do código.

A Figura 6.9 mostra um diagrama de classes extraído do *Hibernate*, contendo uma instância do padrão de projeto *Strategy* que apresentou métodos com coocorrência de *Long Method*. A *interface Mappings* e *ExtendedMappings* exercem o papel de *Strategy* e define uma interface de serviços comum para todas as estratégias. A classe *MappingsImpl* exerce o papel de *ConcreteStrategy* e realiza a implementação dos serviços definidos pelas interfaces Strategy, de acordo a estratégia atribuída a essa classe. A classe *SimpleValueBinder* exerce o papel de *Context* e realiza a implementação das estratégias definidas pelo cliente. No entanto, essa última classe possui ocorrência do *bad smell Long Method*. O método *setType* da classe *SimpleValueBinder*, responsável por realizar a implementação de uma estratégia solicitada pelo cliente, é um exemplo dessa ocorrência. Esse método possui uma grande quantidade de código e utiliza estruturas condicionais para definir escolhas de implementação. A melhor prática nesse caso, seria criar vários componentes menores, uma para cada estrutura condicional definida dentro do método, com uma interface padrão, e instanciá-las dentro do método conforme a necessidade de uso. Com isso, o código ficaria menos complexo e melhoria sua legibilidade e entendimento.

Sumário. A principal situação que contribui para o surgimento da coocorrência *Strategy* ⇒ *Long Method* é o excesso de código implementado nos métodos responsáveis pela definição de estratégias. Tal excesso se deve ao uso exagerado de estruturas condicionais utilizadas para definir os passos e comportamentos realizados por cada um dos algoritmos de estratégia implementados. Esse excesso de código prejudicou a legibilidade e entendimento do código fonte.

6.5.5 *Refused Bequest*

Nos resultados obtidos para *Refused Bequest*, dois padrões de projeto, *Proxy* e *Singleton* apresentaram uma maior taxa de coocorrência para esse *bad smell*. A fim de identificar algumas situações que ocasionaram essas coocorrências, a relação *Proxy* ⇒ *Refused Bequest* foi inspecionada manualmente.

O padrão de projeto *Proxy* é uma solução que fornece um objeto substituto ou marcador para controle de acesso a outro objeto. Essa solução é formada por um conjunto de classes, das quais, uma representa o objeto real que se deseja controlar o acesso, *RealSubject*, e as outras, *Proxy*, são responsáveis por realizar o controle sobre os objetos reais.

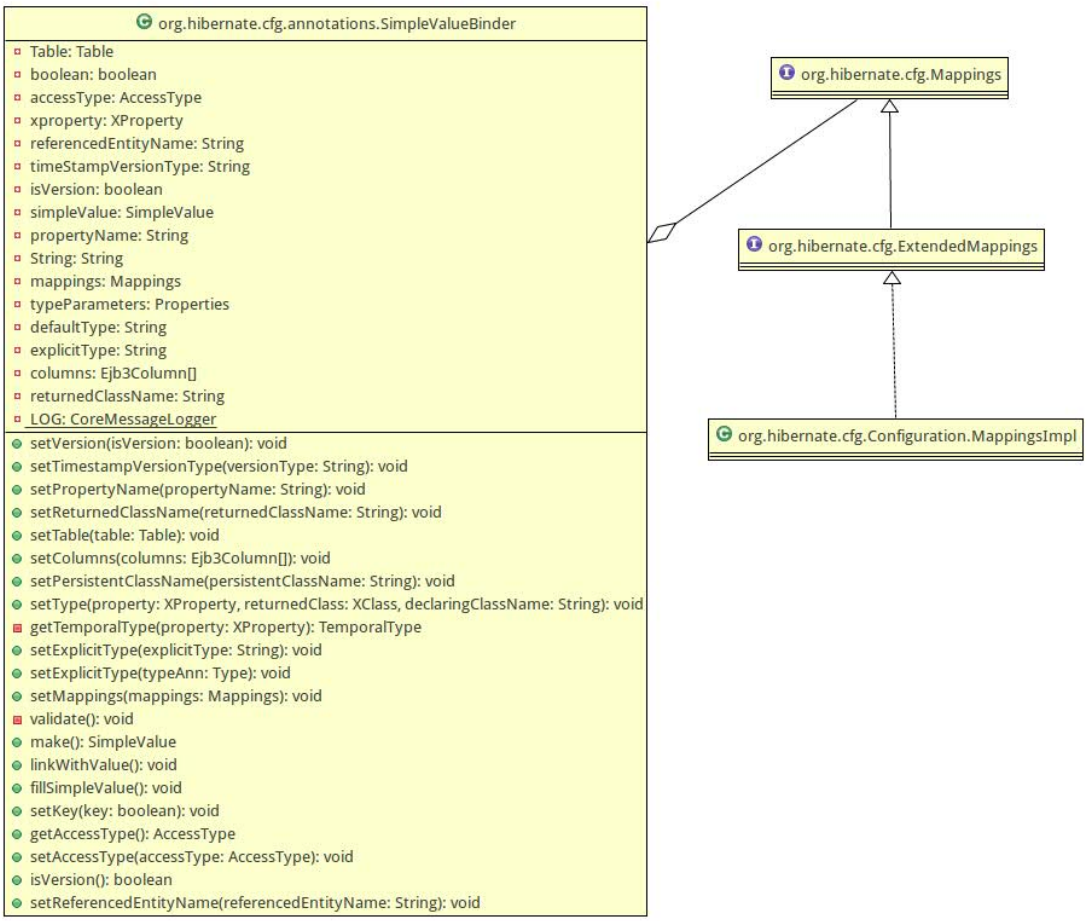

Figura 6.9. Diagrama de Classes que representa a relação *Strategy* ⇒ *Long Method*.

Ao analisar as classes que apresentaram a relação *Refused Bequest* ⇒ *Proxy*, percebeu-se que a maioria das classes que apresentaram coocorrência exerciam o papel de *Proxy* dentro do padrão de projeto. Ainda, foi possível identificar duas situações principais que ocasionaram o surgimento dessas relações. A primeira delas foi o uso da herança como mecanismo de reúso. Herança é um mecanismo que deve ser usado apenas quando superclasse e subclasse possuem um relacionamento "é um". No entanto, várias classes dos sistemas analisados estavam utilizando esse recurso para reusar códigos já implementados em outras classe, sem que houvesse o relacionamento "é um" entre as classes relacionados. A segunda situação identificada foi o uso inadequado da herança entre classes. Quando esse recurso não é utilizado adequadamente, a classe no nível mais baixo da hierarquia começa a sobrescrever uma grande quantidade de serviços, vindo de suas classes pai, algo que não deveria acontecer quando se utiliza esse mecanismo.

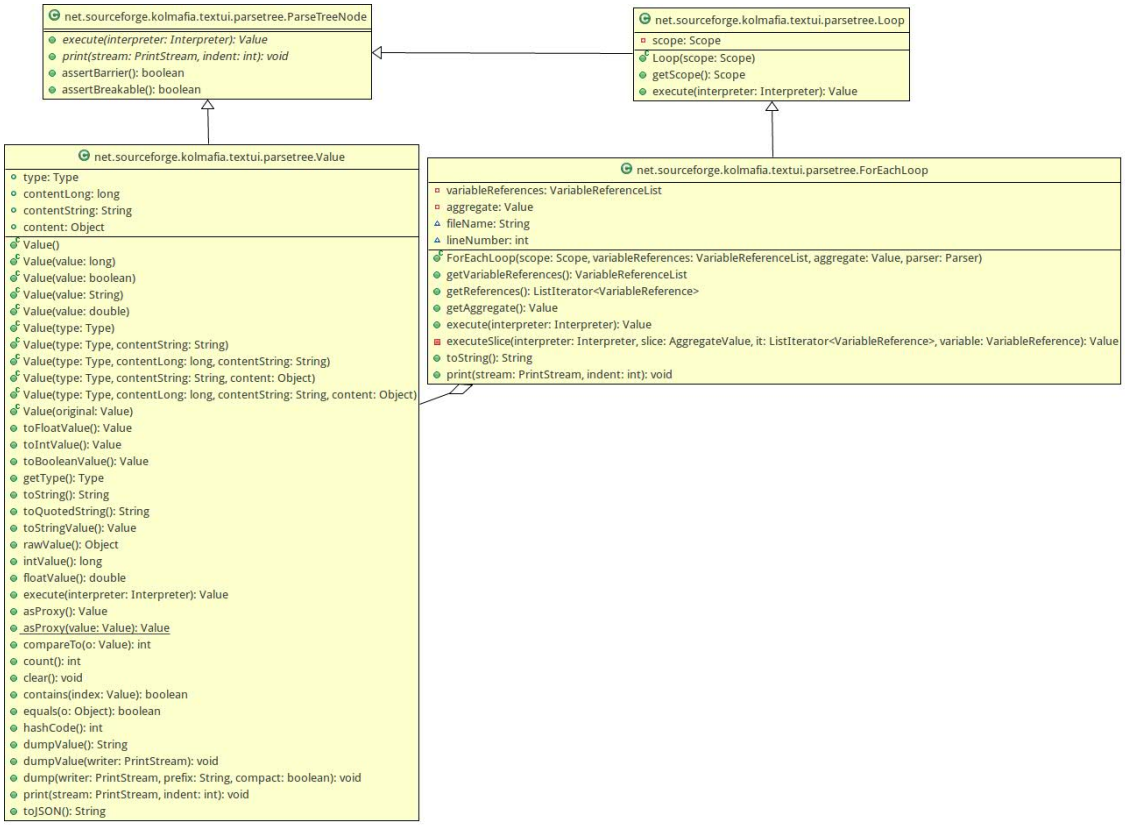

Figura 6.10. Diagrama de Classes que representa a relação *Proxy* ⇒ *Refused Bequest*.

Baseado nisso, a principal solução para remoção das coocorrências nessas classes, seria a aplicação de refatoração, trocando o relação de herança pelo relacionamento de composição. Essa ação seria uma boa prática para a redução das sobrescritas de métodos e rejeição de características.

A Figura 6.10 mostra um diagrama de classes extraído do *Kolmafia*, contendo uma instância do padrão de projeto *Proxy* que apresentou ocorrência do *bad smell Refused Bequest*.

Nesse diagrama, a classe *Value* exerce o papel de *RealSubject* dentro do padrão de projeto. A classe *ForEachLoop* exerce o papel de *Proxy*. Analisando o diagrama de classes apresentando, é possível ver que a classe *ForEachLoop* está posicionada no nível mais baixo da hierarquia de herança. Durante a inspeção manual, percebeu-se que essa classe está posicionada no nível errado. Ela sobrescreve um método já implementado pela sua classe pai, *Loop*, rejeitando, assim, uma característica já definida. Essa classe deveria ser refatorada e movida para o mesmo nível da classe *Loop* nessa hierarquia, ou seja, ela deveria ter um relacionamento de herança direto com a classe *ParseTreeNode*, dado que ela utiliza características apenas dessa classe, e ainda implementa um método

que é apenas definido como abstrato por essa classe.

Sumário. A principal situação que contribuiu para o surgimento da coocorrência *Proxy* ⇒ *Refused Bequest* é o uso da herança como mecanismo de reúso e o posicionamento de componentes em um nível inadequado na hierarquia de herança. Tais situações contribuíram para que componentes rejeitem funcionalidade providas pelas suas superclasses.

6.6 Ameaças à Validade

Este capítulo apresentou um Estudo de Caso realizado com cinco *bad smells* e um *data set* composto por cinco sistemas de software Java de código aberto. Desses cinco sistemas foram extraídas informações de padrões de projeto e *bad smells* para identificação das coocorrências. Os sistemas utilizados para compor o *data set* foram extraídos de um *data set* maior, chamado *Qualitas Corpus*, e conta com sistemas de pequeno, médio e grande porte. Entretanto, devido à quantidade pequena de sistemas utilizados na condução desse Estudo de Caso, não é possível generalizar os resultados encontrados. Ainda assim, os resultados obtidos são importantes porque mostram que o uso de padrões de projeto não necessariamente evitam *bad smells* em sistemas orientado por objetos.

A coleta de dados foi realizada com o auxílio de ferramentas. Para identificar os métodos e classes que compõem instâncias de padrões de projeto, foi utilizada a ferramenta *DPDSS*. Para identificar os métodos e classes que possuem ocorrência de *bad smells*, as estratégias de detecção apresentadas na Seção 4.1 foram implementadas na ferramenta *RAFTool* que filtrou os componentes com a presença dos *bad smells*. E para identificação das coocorrências, foi utilizada a ferramenta *Design Pattern Smell*, descrita no Capítulo 5. Embora todas essas ferramentas tenham sido avaliadas e apresentaram um bom resultado, não se pode garantir que os resultados delas estejam totalmente corretos.

Por fim, para identificar as situações que ocasionaram o surgimento das coocorrências entre padrões de projeto e *bad smells*, foi realizado um processo de inspeção manual nos métodos e classes envolvidos em tais relações. A inspeção manual foi realizada pelo autor desse trabalho de dissertação, contudo, por se tratar de atividade manual, ela é suscetível a erros. Para mitigar essa ameaça, a análise realizada foi conduzida em uma pequena quantidade de sistemas de software neste trabalho.

6.7 Lições Aprendidas

Analisando os resultados observados neste Estudo de Caso, constatou-se que a aplicação de padrões de projeto em sistemas de software não necessariamente evita a ocorrência de *bad smells*. No entanto, alguns padrões de projeto apresentaram uma baixa relação com os *bad smells* avaliados neste estudo. São eles: *Factory Method*, *Composite* e *Singleton*. Em relação aos padrões de projeto *Factory Method* e *Composite*, é esperado que eles apresentassem uma baixa coocorrência, dado que eles são padrões intrinsecamente modulares. Contudo, a indicação do *Singleton* como padrão de projeto que apresentou baixa coocorrência com os *bad smells* que lidam com grande quantidade de complexidade e dados foi uma surpresa. *Singleton* tem como propósito principal fornecer um ponto global de acesso para uma classe. De acordo com alguns trabalhos anteriores, Vokac [2004], esse padrão tende a centralizar a inteligência do sistema e consequentemente aumentar a complexidade de seus componentes internos gerando ocorrência de *bad smells*. Os resultados obtidos no presente estudo contradiz o achado de Vokac [2004].

Além disso, foram extraídas várias relações de coocorrência entre os padrões de projeto para cada um dos *bad smells* estudados. Analisando essas relações de forma geral (Tabela 6.11), percebe-se que os padrões de projeto *Proxy*, *Adapter-Command* e *State-Strategy* foram aqueles que se mostraram mais frequentes nas relações identificadas para cada *bad smell*. Eles aparecerem em quatro dos cinco sintomas analisados nesse estudo, sendo que para dois *bad smells*, *Data Class* e *Refused Bequest*, o padrão de projeto *Proxy* foi identificado como aquele que apresentou maior coocorrência. *Bridge* e *Template Method* aparecem logo após com uma frequência de três em cinco dos sintomas analisados. Isso indica que essas soluções merecem uma atenção especial no momento de sua aplicação, com o intuito de evitar o surgimento de *bad smell*.

Por fim, cada relação de maior coocorrência para cada um dos *bad smells* foram inspecionadas manualmente, com o intuito de identificar situações que impactaram no surgimento dessas relações. Embora essas situações sejam específicas de cada caso, é possível concluir que elas surgiram devido a um mal uso e implementação inadequada dos padrões de projeto nos sistemas analisados. Portanto, ao utilizar uma dessas soluções é necessário que haja uma atenção especial e um bom planejamento dessas soluções para que ocorrências de *bad smells* sejam evitadas.

Capítulo 7

Conclusão

Nesta dissertação, foi realizado um estudo com o objetivo de analisar a relação entre padrões de projeto e *bad smells*. Nesse sentido, foi feita uma revisão sistemática da literatura com o intuito de mapear estudos anteriores relacionados ao tema explorado e avaliar como a literatura tem abordado relações entre padrões de projeto e *bad smells*.

Por meio dessa revisão sistemática, foi possível identificar que a literatura tem abordado as relações entre padrões de projeto e *bad smells* de três formas diferentes: coocorrências, impacto em qualidade e refatoração. A maioria dos estudos tem-se concentrado na categoria de impacto em qualidade, sete no total, e avaliado as consequências e impactos da aplicação dos padrões de projeto em um software. A categoria de refatoração apresentou a segunda maior quantidade de trabalhos, cinco no total, e tem proposto abordagens e ferramentas que aplicam padrões de projeto para eliminação de *bad smells*. A categoria de coocorrências foi a que apresentou um menor número de estudos, quatro no total, e tem buscado identificar relações de coocorrência entre padrões de projeto e *bad smells* e apontar os motivos geraram essas relações.

A partir da revisão sistemática, foi possível observar que, embora o tema de relação entre padrões de projeto e *bad smells* seja considerado importante, ele ainda é pouco explorado na literatura. Em especial, verificou-se que poucos estudos têm investigado coocorrências entre padrões de projeto e *bad smells*. Os *bad smells* descritos por Fowler & Beck [1999] são os mais utilizados nessas investigações. No entanto, esses estudos também têm abordado alguns *bad smells* descritos por Brown et al. [1998] e Lanza & Marinescu [2006]. Em relação aos padrões de projeto, os estudos têm abordado soluções integrantes do catálogo *GOF*, proposto por Gamma et al. [1994]. O uso desses padrões é justificado pela existência de ferramentas que realizam a extração das instâncias de forma automatizada. Essas ferramentas, além de agilizar o processo de extração, amenizam a ocorrência de falsos positivos e negativos.

A condução do Estudo de Caso realizado neste trabalho foi realizado dentro das seguintes etapas. A primeira consistiu na detecção de estratégias de detecção para os *bad smells Feature Envy*, *Large Class*, *Long Method* e *Refused Bequest*. Foram utilizadas as estratégias de detecção proposta por Souza [2016], dado que elas possuem métricas de software conhecidas pela comunidade acadêmica e foram previamente avaliadas. Na segunda etapa, foi construído um *data set* para a realização do estudo com cinco sistemas Java de código aberto, de pequeno a grande porte, sendo quatro deles parte do *Qualitas.class Corpus* [Terra et al., 2013]. Na terceira etapa, os dados referentes a padrões de projeto e *bad smells* foram coletados desses sistemas, e as regras de associação foram aplicadas para extração das relações entre essas duas estruturas. Por fim, a análise dos resultados, foi realizada baseando-se no valor da métrica `Convicção` referente às regras de associação. Essa métrica foi utilizada como ponto de análise devido ao fato dela ser capaz de estabelecer uma relação entre as métricas `Suporte` e `Confiança`, e possuir uma melhor sensibilidade na identificação das relações entre padrões de projeto e *bad smells*.

Os resultados do Estudo de Caso indicam que a aplicação de padrões de projeto em sistemas de software não necessariamente evita a ocorrência de *bad smells*. Contudo, alguns padrões de projeto como *Factory Method*, *Composite* e *Singleton* apresentaram uma baixa coocorrência com os *bad smells* estudados.

A análise das situações que impactaram no surgimento das coocorrências entre padrões de projeto e *bad smells* indica que, apesar de elas serem específicas para cada um dos casos de maior coocorrência, elas surgiram devido a um mal planejamento e implementação inadequada dos padrões de projeto nos sistemas analisados. Por esse motivo, é necessário que haja uma atenção especial e um bom planejamento dessas soluções para se evitar o surgimento de *bad smells*.

Baseado nos principais resultados do estudo conduzido nesta dissertação, destacam-se as seguintes contribuições:

- uma Revisão Sistemática da Literatura com foco em estudos sobre padrões de projeto e *bad smell*. Os resultados obtidos proporcionam à comunidade acadêmica uma visão geral do estado da arte nesse campo. Essa Revisão Sistemática da Literatura revela que existem poucos estudos dedicados a coocorrência entre padrões de projeto e *bad smells*;

- uma ferramenta chamada *Design Pattern Smell* que suporta detecção de coocorrência entre padrões de projeto e *bad smell* baseada em informações computadas de sistemas de software;

- avaliação quantitativa e qualitativa das relações de coocorrência entre padrões de projeto e *bad smells*, por meio de um Estudo de Caso com cinco sistemas de software Java, que mostrou casos de coocorrências e situações reais que levaram a existência dessas relações nos sistemas analisados.

Como sugestões de trabalhos futuros, é importante (i) estender esta pesquisa a uma quantidade maior de amostra de software. Realizar uma análise com uma amostra maior, considerando o tipo e o tamanho dos sistemas de software, também ajudaria a melhorar a compreensão dos sistemas de software que aplicam padrões de projeto; (ii) investigar coocorrências de padrões de projeto com outros tipos de *bad smells*; (iii) investigar as relações de padrões de projeto, *bad smells* e falhas em software; e (iv) construir um sistema de recomendação de refatoração que auxiliem na eliminação de *bad smells* do código fonte de um sistema de software.

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, pp. 1--8.

Agrawal, R.; Imieliński, T. & Swami, A. (1993). Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207--216. ISSN 0163-5808.

Benlarbi, S.; Emam, K. E.; Goel, N. & Rai, S. (2000). Thresholds for object-oriented measures. Em *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE '00, pp. 24--, Washington, DC, USA. IEEE Computer Society.

Biolchini, J.; Mian, P. G.; Natali, A. C. C. & Travassos, G. H. (2005). Systematic review in software engineering. *System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES*, 679(05):45.

Brin, S.; Motwani, R.; Ullman, J. D. & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255--264. ISSN 0163-5808.

Brown, W. H.; Malveau, R. C.; McCormick, H. W. S. & Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1st edição. ISBN 0471197130, 9780471197133.

Cardoso, B. & Figueiredo, E. (2015). Co-occurrence of design patterns and bad smells in software systems: An exploratory study. Em *Proceedings of the Annual Conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1*, SBSI 2015, pp. 46:347--46:354, Porto Alegre, Brazil, Brazil. Brazilian Computer Society.

Chhikara, A. & Khatri, S. (2011). Evaluating the impact of different types of inheritance on the object oriented software metrics. Em *International Journal of Enterprise Computing and Business Systems ISSN: 223-8849 Volume 1 Issue*.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476--493. ISSN 0098-5589.

Christopoulou, A.; Giakoumakis, E. A.; Zafeiris, V. E. & Soukara, V. (2012). Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202--1214.

Crespo, Y.; López, C. & Marticorena, R. (1999). Relative thresholds: Case study to incorporate metrics in the detection of bad smells. Em *QAOOSE 2006 Proceedings*, pp. 109--118.

Eclipse (2016). Eclipse 4.2 juno. Disponível em: http://www.eclipse.org/downloads/packages/release/Juno/SR2. Acessado em Junho de 2016.

Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T. & Figueiredo, E. (2016). A Review-Based Comparative Study of Bad Smell Detection Tools. Em *Proc. of the 20th EASE*, pp. 1--12.

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

Filó, T. G. S. (2014). Identificação de valores referência para métricas de software orientado por objetos. Dissertação de mestrado, Universidade Federal de Minas Gerais, Departamento de Ciência da Computação, Curso de Pós-Graduação em Ciência da Computação, Minas Gerais.

Filó, T. G. S.; Bigonha, M. A. S. & Ferreira, K. A. M. (2014). Raftool - ferramenta de filtragem de métodos, classes e pacotes com medições incomuns de métricas de software. Em *Proceedings of the X Workshop Anual do MPS (WAMPS 2014)*, pp. 1–6.

Filó, T. G. S.; Bigonha, M. A. S. & Ferreira, K. A. M. (2015). A catalogue of thresholds for object-oriented software metrics. Em *International Conference on Advances and Trends in Software Engineering SOFTENG. Proceedings of International Conference on Advances and Trends in Software Engineering*, p. 1.

Fowler, M. (2015). Callsuper. `https://martinfowler.com/bliki/CallSuper.html`. Accessado em: 09-03-2017.

Fowler, M. & Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN 9780201485677.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-63361-2.

Izurieta, C. & Bieman, J. (2013). A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21(2):289--323.

Jaafar, F.; Guéhéneuc, Y.; Hamel, S. & Khomh, F. (2013). Analysing anti-patterns static relationships with design patterns. *ECEASST*, 59.

Jaafar, F.; Gueheneuc, Y.-G.; Hamel, S.; Khomh, F. & Zulkernine, M. (2016). Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21(3):896--931.

Kaur, S.; Singh, S. & Kaur, H. (2013). A quantitative investigation of software metrics threshold values at acceptable risk level. Em *International Journal of Engineering Research and Technology*. ESRSA Publications.

Kerievsky, J. (2004). *Refactoring to Patterns*. Pearson Higher Education. ISBN 0321213351.

Khomh, F. & Gueheneuce, Y.-G. (2008). Do design patterns impact software quality positively? Em *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pp. 274--278, Washington, DC, USA. IEEE Computer Society.

Kitchenham, B. & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Relatório técnico EBSE 2007-001, Keele University and Durham University Joint Report.

Kleinbaum, D. (1994). *Logistic Regression: A Self-Learning Text*. Springer New York.

Lanza, M. & Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 3540244298.

Liu, W.; Hu, Z.-G. b.; Liu, H.-T. & Yang, L. (2014). Automated pattern-directed refactoring for complex conditional statements. *Journal of Central South University*, 21(5):1935–1945. ISSN 20952899.

Lorenz, M. & Kidd, J. (1994). *Object-oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-179292-X.

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

Martin, R. (1994). Oo design quality metrics. *An analysis of dependencies*, 12.

McNatt, W. B. & Bieman, J. M. (2001). Coupling of design patterns: Common practices and their benefits. Em *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pp. 574--579, Washington, DC, USA. IEEE Computer Society.

Metrics (2016). Metrics. Disponível em: http://metrics.sourceforge.net. Acessado em Junho de 2016.

Moha, N.; Gueheneuc, Y. G.; Duchien, L. & Meur, A. F. L. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36.

Nahar, N. & Sakib, K. (2015). Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory. Em *CEUR Workshop Proceedings*, volume 1519, pp. 9--16.

Nahar, N. & Sakib, K. (2016). Acdpr: A recommendation system for the creational design patterns using anti-patterns. Em *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 4*, pp. 4–7.

Oliveira, P.; Valente, M. T. & Lima, F. P. (2014). Extracting relative thresholds for source code metrics. Em *Proceedings of the Conference on Software Maintenance, and Reengineering (CSMR)*, pp. 254--263.

Pressman, R. S. (2006). *Engenharia de software*. MacGraw Hill, Rio de Janeiro, 6ª edição edição.

Rosenberg, L.; Stapko, R. & Gallo, A. (1999). Risk-based object oriented testing. *24th SWE*.

Shatnawi, R.; Li, W.; Swain, J. & Newman, T. (2010). Finding software metrics threshold values using roc curves. *J. Softw. Maint. Evol.*, 22(1):1--16. ISSN 1532-060X.

Sheskin, D. J. (2007). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edição. ISBN 1584888148, 9781584888147.

Sommerville, I. (2011). *Engenharia de Software*. Person Education Brasil, 9th edição. ISBN 978-8579361081.

Sousa, B.; Bigonha, M. & Ferreira, K. (2016). Design pattern smell. `http://llp.dcc.ufmg.br/Products/indexProducts.html`. Accessed on October 17, 2016.

Souza, P. (2016). A utilidade dos valores referência de métricas na avaliação da qualidade de softwares orientados por objeto. Dissertação de mestrado, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais.

Speicher, D. (2013). Code quality cultivation. *Communications in Computer and Information Science*, 348:334–349. ISSN 18650929.

Tempero, E.; Anslow, C.; Dietrich, J.; Han, T.; Li, J.; Lumpe, M.; Melton, H. & Noble, J. (2010). The qualitas corpus: A curated collection of java code for empirical studies. Em *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pp. 336--345. IEEE.

Terra, R.; Miranda, L. F.; Valente, M. T. & Bigonha, R. S. (2013). Qualitas. class corpus: A compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5):1--4.

Tsantalis, N.; Chaikalis, T. & Chatzigeorgiou, A. (2008). JDeodorant: Identification and Removal of Type-Checking Bad Smells. Em *Proc. of the 12th CSMR*, pp. 329--331.

Tsantalis, N.; Chatzigeorgiou, A.; Stephanides, G. & Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896--909.

Vale, G. A. D. & Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. Em *2015 29th Brazilian Symposium on Software Engineering*, pp. 110–119.

Vidal, S. A.; Marcos, C. & Díaz-Pace, J. A. (2014). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23:501--532.

Vokac, M. (2004). Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904--917.

Wagey, B. C.; Hendradjaya, B. & Mardiyanto, M. S. (2015). A proposal of software maintainability model using code smell measurement. Em *2015 International Conference on Data and Software Engineering (ICoDSE)*, pp. 25–30.

Walter, B. & Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127--142.

Wendorff, P. (2001). Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. Em *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pp. 77--84, Washington, DC, USA. IEEE Computer Society.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Em *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pp. 1--10.

Zafeiris, V. E.; Poulias, S. H.; Diamantidis, N. & Giakoumakis, E. (2017). Automated refactoring of super-class method invocations to the template method design pattern. *Information and Software Technology*, 82:19--35.