PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n - Ondina
Salvador, BA, Brasil, 40170-110

http://pgcomp.dcc.ufba.br
pgcomp@ufba.br

Padrões de projeto são soluções reutilizáveis que podem ser aplicadas para resolver problemas específicos em projetos de software. No entanto, tais padrões podem ser mal aplicados e dar origem a code smells. Code smells ou smells são fragmentos no código-fonte que indicam possíveis falhas de design. Existem situações em que os padrões de projeto podem coocorrer com code smells dentro do mesmo módulo de código (e.g., arquivos .java de projetos Java). Esse fenômeno, caso ocorra com frequência, pode resultar em uma oportunidade para avaliar se existe algum desvio ou má prática de aplicação dos padrões que possam ser evitados. Possivelmente, ao refinar a forma como os padrões são adicionados a projetos, o surgimento e existência de smells possam ser mitigados. Este trabalho apresenta dois estudos experimentais com o objetivo de investigar as relações de coocorrência entre padrões de projeto e smells. Ambos demandaram a aplicação de ferramentas de detecção automatizadas de padrões de projeto e smells em sistemas de código-fonte aberto desenvolvidos na linguagem Java. No primeiro estudo busca-se compreender com que frequência os code smells coocorrem com padrões de projeto, bem como determinar as coocorrências mais comuns. Para esse fim, fora identificadas instâncias de smells em métodos de padrões de projeto de 25 sistemas. Também foram manualmente inspecionados fragmentos do código-fonte dos projetos para obter informações sobre a relação entre pares específicos padrão-smell. Entre outras descobertas, este estudo revelou que os métodos que fazem parte do padrão Adapter são mais propensos a conter smells, especialmente o Feature Envy. O segundo estudo é aplicado sobre o histórico de versão proveniente de repositórios de software Open Source. Essa análise consiste em entender a relação entre as duas variáveis (padrões de projeto e code smells) ao longo da evolução do software. Os resultados encontrados sugerem que o padrão Template Method está mais propenso a ser introduzido simultaneamente com smells ao longo do tempo.

Palavras-chave: Padrões de projeto, Code Smells, Design de Software, Mineração de Repositórios de Software.

MSC | 120 | 2021

Investigando a incidência de code smells em métodos de padrões de projeto

Ederson Assunção

UFBA

Investigando a incidência de code smells em métodos de padrões de projeto

Ederson dos Santos Assunção

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em Ciência da Computação

Setembro | 2021

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

INVESTIGANDO A INCIDÊNCIA DE CODE SMELLS EM MÉTODOS DE PADRÕES DE PROJETO

Ederson dos Santos Assunção

DISSERTAÇÃO DE MESTRADO

Salvador
1º de setembro de 2021

EDERSON DOS SANTOS ASSUNÇÃO

INVESTIGANDO A INCIDÊNCIA DE CODE SMELLS EM MÉTODOS DE PADRÕES DE PROJETO

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Rodrigo Rocha Gomes e Souza

Salvador
1º de setembro de 2021

Sistema de Bibliotecas - UFBA

Assunção, Ederson dos Santos.

Investigando a incidência de *code smells* em métodos de padrões de projeto / Ederson dos Santos Assunção – Salvador, 2021.

69p.: il.

Orientador: Prof. Dr. Rodrigo Rocha Gomes e Souza.

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Computação, 2021.

1. Padrões de Projeto. 2. Code Smells. 3. Qualidade de Software. I. Souza, Rodrigo Rocha Gomes e. II. Universidade Federal da Bahia. Instituto de Computação. III Título.

CDD – XXX.XX

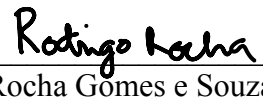
CDU – XXX.XX.XXX

"*Investigando a incidência de code smells em métodos de padrões de projeto*"

Ederson dos Santos Assunção

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora



Profº. Drº. Rodrigo Rocha Gomes e Souza(Orientador-UFBA)

Profº. Drº. Ricardo Terra Nunes Bueno Villela(UFLA)

Profº. Drº. Cláudio Nogueira Sant'Anna(UFBA)

Dedico este trabalho aos meus pais que sempre estiveram ao meu lado e por tudo que representam para mim.

AGRADECIMENTOS

Agradeço a Deus, as bênçãos e vitórias as quais tem me conduzido, e por sempre colocar pessoas corretas na minha vida.

Aos meus pais e irmão, Maria Loreta, Fernando e Egbert, que sempre me incentivaram e me deram o apoio necessário para alcançar meus objetivos.

A minha esposa, Tais, e ao meu filho Rafael, a paciência, apoio e carinho que contribuíram para suavizar a trajetória até aqui.

Ao meu orientador, Prof. Dr. Rodrigo Rocha Gomes e Souza, a confiança, parceria, amizade e ensinamentos transmitidos.

Aos amigos da Pós-Graduação Luis e Silvio, o compartilhamento de dificuldades, experiências e inspirações para a pesquisa.

"O maior bem que você pode fazer ao próximo não é apenas compartilhar suas riquezas, mas revelar à ela as suas próprias."

—BENJAMIN DISRAELI

RESUMO

Padrões de projeto são soluções reutilizáveis que podem ser aplicadas para resolver problemas específicos em projetos de software. No entanto, tais padrões podem ser mal aplicados e dar origem a *code smells*. *Code smells* ou *smells* são fragmentos no código-fonte que indicam possíveis falhas de *design*. Existem situações em que os padrões de projeto podem coocorrer com *code smells* dentro do mesmo módulo de código (e.g., arquivos .java de projetos Java). Esse fenômeno, caso ocorra com frequência, pode resultar em uma oportunidade para avaliar se existe algum desvio ou má prática de aplicação dos padrões que possam ser evitados. Possivelmente, ao refinar a forma como os padrões são adicionados a projetos, o surgimento e existência de *smells* possam ser mitigados. Este trabalho apresenta dois estudos experimentais com o objetivo de investigar as relações de coocorrência entre padrões de projeto e *smells*. Ambos demandaram a aplicação de ferramentas de detecção automatizadas de padrões de projeto e *smells* em sistemas de código-fonte aberto desenvolvidos na linguagem Java. No primeiro estudo busca-se compreender com que frequência os *code smells* coocorrem com padrões de projeto, bem como determinar as coocorrências mais comuns. Para esse fim, fora identificadas instâncias de *smells* em métodos de padrões de projeto de 25 sistemas. Também foram manualmente inspecionados fragmentos do código-fonte dos projetos para obter informações sobre a relação entre pares específicos padrão-*smell*. Entre outras descobertas, este estudo revelou que os métodos que fazem parte do padrão *Adapter* são mais propensos a conter *smells*, especialmente o *Feature Envy*. O segundo estudo é aplicado sobre o histórico de versão proveniente de repositórios de software *Open Source*. Essa análise consiste em entender a relação entre as duas variáveis (padrões de projeto e *code smells*) ao longo da evolução do software. Os resultados encontrados sugerem que o padrão *Template Method* está mais propenso a ser introduzido simultaneamente com *smells* ao longo do tempo.

Palavras-chave: Padrões de projeto, *Code Smells*, *Design* de Software, Mineração de Repositórios de Software.

ABSTRACT

Design patterns are reusable solutions that can be applied to solve specific problems of software projects. However, developers may not implement patterns adequately on some occasions. This can lead to the appearance of code smells. Code smells (or smells) are fragments of code that can indicate potential design flaws. There are situations in which design patterns and smells occur simultaneously in the same code unit (e.g., classes and methods of Java-based software projects). In case of such phenomenon occurs frequently it can reveal opportunities to investigate if a bad practice in the application of patterns is the cause of the occurrence of smells. As a consequence, refining the way how developers apply design patterns may be a good measure to avoid code smells. In this context, we present two experimental studies that aim at examining cases of co-occurrence of patterns and smells. Both studies required the application of detection tools to evaluate Java-based software projects. With our first study, we seek to comprehend the frequency by which code smells co-occur with design patterns. We also report the most common cases of co-occurrence. To achieve this, we identified instances of smells in methods of design patterns of 25 software systems. We also manually inspected source code fragments to obtain useful information about pairs of patterns and smells. Among other findings, we perceived that methods that take part in the implementation of the Adapter pattern are prone to contain smells, e.g., the Feature Envy smell. Our second study has the purpose of analyzing the evolution of open-source software projects. We wanted to provide insights about the co-occurrences along with the evolution of the projects. As a result, we found out that the Template Method pattern is more inclined to appear concomitantly with smells through time.

Keywords: Design Patterns, Code Smells, Software Design, Software Repository Mining.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ATFD	*Access to Foreign Data*
CC	*Changing Classes*
CDISP	*Coupling Dispersion*
CINT	*Coupling Intensity*
CM	*Changing Methods*
CYCLO	*Cyclomatic Number*
FDP	*Foreign Data Providers*
LAA	*Locality of Attribute Accesses*
LOC	*Lines of Code*
MAXNESTING	*Maximum Nesting Level*
NOAV	*Number of Accessed Variables*
CINT	*Coupling Intensity*
GoF	*Gang of Four*

.

Capítulo 1

Este capítulo apresenta as principais motivações, justificativas e objetivos de pesquisa que levaram a realizar esta dissertação. Apresenta uma breve introdução dos principais conceitos relacionados a padrões de projeto e code smells e, finalmente, apresenta a dissertação.

INTRODUÇÃO

Na criação de software é comum reutilizar modelos de referência para resolver problemas que ocorrem com frequência. No contexto de *design* de software, um dos modelos é conhecido como padrões de projeto (GAMMA et al., 1995). Padrões de projeto são soluções testadas e comprovadas para resolver problemas conhecidos de *design* de software com a finalidade de tornar os projetos orientados a objetos mais flexíveis e reutilizáveis (GAMMA et al., 1995). Dessa forma, os padrões de projeto ajudam engenheiros de software a definir mais rapidamente um *design* adequado baseado no reaproveitamento de experiências (SOMMERVILLE, 2011).

Os padrões de projetos tendem a apresentar desvantagens se não forem aplicados corretamente, e consequentemente podem elevar a complexidade do sistema (SUBBURAJ; JEKESE; HWATA, 2015; PRECHELT et al., 2001). Contudo, quando bem aplicados, incorporam boas práticas de projeto e promovem a qualidade do *design* por meio de uma arquitetura mais simples, compreensível, manutenível e padronizada (SOMMERVILLE, 2011; GAMMA et al., 1995). Esses atributos tendem aumentar efetivamente a qualidade interna do software.

Qualidade interna está intrinsecamente associada às propriedades estáticas do código-fonte e pode ser obtida observando as seguintes características: manutenibilidade, flexibilidade, reusabilidade, legibilidade, escalabilidade e testabilidade (MCCONNELL, 2002). Essas características podem ser alcançadas através de padrões de projeto; contudo, durante o ciclo de desenvolvimento de software, as soluções que empregam padrões podem sofrer alterações capazes de impactar de maneira negativa os atributos de qualidade para os quais eles são criados e aplicados. Ou seja, pode ocorrer a incidência de alguma prática inadequada de codificação nas mesmas classes que compõem o padrão. Como consequência, mesmo com aplicação de uma técnica visando a melhoria da qualidade do software, há possibilidade de que ocorra a incidência de algum fator que pode ser contrário a essa qualidade, como por exemplo *code smells* (FOWLER; BECK, 1999).

Code smells são fragmentos no código-fonte que indicam possíveis problemas de *design*. Se encontrados em sistemas orientados a objetos, podem deixá-los mais difíceis de entender, manter e evoluir. Como *code smells* tendem a diminuir efetivamente a qualidade interna do software, eles podem servir de referência para avaliar o *design* de um software. Ou seja, os *code smells* podem ser utilizados como pontos de partida para uma análise mais profunda de problemas que afetam a qualidade de sistemas. Embora não sejam necessariamente prejudiciais, podem atuar como uma indicação de que algo pode estar em desacordo com as boas práticas e princípios de programação (FOWLER; BECK, 1999).

Trabalhos anteriores já mostraram que a presença de *code smells* em projetos pode levar a problemas como alto acoplamento, propensão a mudanças e falhas (YAMASHITA; MOONEN, 2013; KHOMH et al., 2012; JAAFAR et al., 2013), além de incremento em manutenções e aumento da complexidade (FONTANA et al., 2013). Considerando esse último especificamente, engenheiros e desenvolvedores podem sofrer, por exemplo, com seus efeitos sobre a compreensão de unidades de código-fonte (ABBES et al., 2011).

1.1 MOTIVAÇÃO

Conforme as definições de padrões de projeto e *code smells*, percebe-se que tais conceitos indicam ideias contrárias relacionadas à qualidade do software. Todavia, podem surgir situações em que é criada uma zona de interseção entre os itens de promoção de qualidade (padrões) e os itens que podem indicar perda dessa mesma qualidade (*smells*). Dessa forma, é preciso entender como eles estão relacionados.

Embora existam estudos que avaliam o impacto de *code smells* sobre projetos de software, poucos investigaram o seu relacionamento com padrões de projeto do catálogo *Gang of Four* (GoF) definidos por Gamma et al. (1995). Uma revisão sistemática da literatura (SOUSA; BIGONHA; FERREIRA, 2018) realizada em 2018 detectou quatro estudos com o objetivo de identificar relações de coocorrência entre padrões de projeto e *code smells*. Nos anos seguintes, 2019 e 2020, mais dois trabalhos foram publicados (ALFADEL; ALJASSER; ALSHAYEB, 2020; SOUSA; BIGONHA; FERREIRA, 2019). Portanto, existe algum interesse em estudar a coocorrência entre padrões de projeto e *code smells*. Nesses estudos, uma das questões importantes se concentra em entender se os padrões estão propensos a conter *code smells*.

As pesquisas realizadas até o presente momento apresentam conclusões relevantes e sugerem pesquisas futuras que busquem replicar resultados. Também recomendam uma análise mais aprofundada desse assunto. Com o intuito de contribuir para uma investigação mais detalhada e específica acerca da aplicação do padrão de projeto e, especialmente, preencher uma lacuna não contemplada pelas pesquisas até então citadas, decidimos investigar a incidência de *code smells* a nível de métodos. Isto é, em métodos que fazem parte da estrutura de padrões de projeto especificados pela GoF (GAMMA et al., 1995). Os estudos anteriores realizaram análises a nível de classe, no qual incluíram também os métodos que não fazem parte da especificação definida pela GoF.

Também houve motivação por ainda haver pouco conhecimento sobre a ocorrência de *code smells* em padrões de projeto. Esse conhecimento é fundamental para: (i) apoiar os desenvolvedores a atuarem com estratégias na aplicação de padrões para melhorar

a qualidade dos projetos de software e (ii) revelar informações que possam ajudar os fornecedores de ferramentas de detecção de *code smells* a desenvolverem uma nova geração de ferramentas, levando em consideração a implementação de padrões de projetos.

Conhecer as coocorrências entre padrão-*smell* pode evitar uma situação contraditória: ao tentar melhorar a qualidade do código-fonte com a inclusão de padrões de projeto, essa qualidade pode ser diminuída pela introdução de *code smells*.

1.2 OBJETIVO

Este trabalho apresenta dois estudos experimentais com o objetivo de investigar as relações de coocorrência entre padrões de projeto *GoF* definidos por Gamma et al. (1995) e os *code smells*, cujas estratégias de detecção foram definidas por Lanza e Marinescu (2007). Especificamente, visam identificar se os métodos individuais que fazem parte da implementação de padrões de projeto coocorrem com *code smells*. A análise é feita no âmbito de métodos em classes de programas orientados a objetos, pois pretende-se avaliar, sob esse contexto específico, se a implementação de métodos de padrões leva à incidência de *code smells*.

Foram adotados os mesmos procedimentos para ambos os experimentos, uma vez que seus objetivos finais são idênticos: entender a relação entre as duas variáveis. Ambos demandaram a aplicação de ferramentas de detecção de padrões de projeto e *code smells* em sistemas desenvolvidos na linguagem Java. O primeiro estudo utiliza dados de uma versão específica de cada sistema, ou seja, uma versão final dos sistemas examinados. A meta é identificar as associações mais significativas, seguida por uma inspeção manual no código-fonte a fim de identificar as possíveis causas da coocorrência. O segundo estudo é aplicado sobre o histórico de versão proveniente de repositórios de software *open source*. Essa análise consiste em entender a relação entre as duas variáveis ao longo da evolução do software.

1.3 QUESTÕES DE PESQUISA

Para alcançar o objetivo estabelecido, elaborou-se um conjunto de questões de pesquisa que servem como guia para nosso estudo. As questões se encontram definidas a seguir.

1. Questão de Pesquisa 1 (**QP1**) – **Existe associação entre a ocorrência de padrões e a ocorrência de *code smells* em um mesmo método?** A finalidade é verificar se existe interseção entre os métodos que fazem parte de um padrão e os métodos que apresentam *code smells*.

    1.1. Questão de Pesquisa 1.1 (**QP1.1**) – **Quais padrões de projeto estão mais propensos a coocorrerem com *code smells*?** Respondendo a essa questão, busca-se identificar os padrões de projeto que estão mais associados com os *code smells*, bem como aqueles menos associados com os *smells*. Os resultados dessa investigação poderão ser utilizados para alertar desenvolvedores sobre padrões de projeto que merecem mais atenção ao serem introduzidos ou modificados no código-fonte, devido ao seu potencial de introduzir *code smells*.

1.2. Questão de Pesquisa 1.2 (**QP1.2**) – **Quais as coocorrências mais frequentes nos sistemas analisados?** Pretende-se com esta questão de pesquisa identificar os pares padrão–*smell* significativamente associados. A meta é utilizar o resultado para inspecionar o código-fonte e verificar alguma possível causa para existência destes pares. A partir desta análise é possível estabelecer possíveis implicações de uma eventual correlação e (re)pensar práticas para evitar situações que deem origem a determinados *smells*.

2. Questão de Pesquisa 2 (**QP2**) – **Quais padrões, quando adicionados em projetos, possuem maior tendência de incluir algum *code smell*?** Estudos sobre a evolução de *code smells* em sistemas orientados a objeto mostraram que a maioria dos *code smells* são introduzidos no projeto assim que uma entidade é criada (TUFANO et al., 2017; CHATZIGEORGIOU; MANAKOS, 2014). Consequentemente, os *smells* são efeito de análises e atividades de *design* iniciais ineficientes e, não, efetivamente, resultado de sua evolução. Caso se comprove na **QP1** que existe associação entre as variáveis analisadas, não é possível responder se o *smell* foi introduzido durante a incorporação do padrão ou no decorrer de sua evolução. Assim, foi realizada na **QP2** uma análise mais aprofundada que permita responder isso. Portanto, pretende-se com essa pergunta identificar coocorrências que aparecem com a criação do padrão de projeto, ou seja, a partir do ponto em que o método do padrão de projeto é introduzido no sistema. Se identificada a relação, engenheiros de software devem ficar alertas para determinadas situações, pois uma escolha de *design* eventualmente ruim pode exigir uma análise mais criteriosa para evitar problemas piores no futuro.

1.4 CONTRIBUIÇÕES

De acordo com os objetivos estabelecidos, as principais contribuições deste trabalho estão descritas abaixo:

1. apresentação do resultado de um estudo empírico que revela que: (i) métodos de padrões de projeto estão mais propensos a conter *code smell* do que outros métodos; (ii) a prevalência de *code smells* em métodos de padrões apresenta-se distribuída de modo não uniforme entre os padrões; (iii) os padrões *Adapter* e *State* estão mais significativamente associados a *smells* do que outros padrões; (iv) os pares padrão–*smell* mais significativamente associados são: *Template Method-Shotgun Surgery*, *Adapter-Feature Envy* e *Factory Method-Shotgun Surgery*; e (v) os métodos do padrão *Template Method* estão mais propensos a surgir simultaneamente com *code smells* do que outros métodos de padrões.

2. disponibilização do conjunto de dados gerados pelas ferramentas de detecção de padrões de projeto e *code smells*, de modo que novas pesquisas possam replicar o estudo ou analisar a relação de coocorrência a partir de uma perspectiva diferente do contexto aplicado neste trabalho.

3. publicação de um artigo (ASSUNÇÃO; SOUZA, 2019) com os resultados parciais desta dissertação no XIII *Brazilian Symposium on Software Components, Architectures, and Reuse* (SBCARS 2019) que fez parte da Conferência Brasileira de Software: Teoria e Prática (CBSoft 2019) promovido pela Sociedade Brasileira de Computação (SBC).

1.5 ESTRUTURA DO TRABALHO

Este trabalho está organizado em quatro capítulos, além do Capítulo 1 que apresenta os objetivos, o problema, a justificativa, a metodologia e as fontes:

- Capítulo 2: destina-se à revisão da literatura e provê o referencial teórico pertinente aos padrões de projeto e *code smells*;
- Capítulo 3: apresenta o primeiro estudo experimental realizado: detalha os objetivos, o método, os artefatos envolvidos no plano experimental, assim como os resultados obtidos e as análises de coocorrências detectadas no estudo;
- Capítulo 4: descreve o segundo estudo experimental: apresenta a metodologia seguida para condução e desenvolvimento do estudo, detalhando cada atividade realizada, as análises e resultados encontrados;
- Capítulo 5: apresenta a conclusão, descrevendo de forma sucinta os principais resultados, contribuições e sugestões para futuras pesquisas.

Capítulo 2

Este capítulo apresenta uma revisão da literatura em relação às principais definições e conceitos fundamentais sobre os temas abordados: (i) code smells e (ii) padrões de projeto.

REVISÃO BIBLIOGRÁFICA

2.1 CODE SMELLS

A princípio, o software deve ser simples, fácil de compreender e mudar. No entanto, isso nem sempre ocorre, pois, sob pressão de cronogramas de entrega, os desenvolvedores não conseguem alcançar uma qualidade de codificação adequada. Dessa forma, durante o ciclo de desenvolvimento de software, é possível que ocorra a introdução de algum fragmento de código, na estrutura interna do software, com potencial capacidade de violar os princípios de *design* da boa orientação a objetos, gerando *code smells* (SOMMERVILLE, 2011; FOWLER; BECK, 1999).

Code smells, segundo Fowler e Beck (FOWLER; BECK, 1999), indicam possíveis problemas de *design* estrutural em sistemas orientados a objetos que podem levar a dificuldades na manutenção e reutilização do software (ABBES et al., 2011; YAMASHITA; MOONEN, 2013). Tais fragmentos são também conhecidos na literatura como *bad smells* (FOWLER; BECK, 1999), desarmonias de *design* (LANZA; MARINESCU, 2007), anomalias de código (OIZUMI et al., 2016) e *design flaws* (MARINESCU, 2004).

Os *code smells* não são necessariamente prejudiciais, mas uma forte indicação de que algo pode estar em desacordo com as boas práticas e princípios de *design*. De acordo com Fowler et al. (1999), os *smells* não surgem instantaneamente, mas acumulam-se ao longo do tempo à medida que o software evolui. Sua identificação pode ser difícil, pois depende diretamente da interpretação, conhecimento e habilidade do desenvolvedor. Quando apropriado, ferramentas de medição podem ajudar desenvolvedores com a identificação de *smells*.

Após a identificação, é importante que a remoção dos *code smells* seja realizada o mais cedo possível no ciclo de desenvolvimento de software. Caso os *code smells* não sejam identificados e removidos do sistema, podem se propagar e levar à introdução e perpetuação de problemas. A incidência de *bugs* é um tipo de problema conhecido, com probabilidade de ser ocasionado por *smells*, e que podem causar diminuição da confiabilidade e da

qualidade do software (FOWLER; BECK, 1999; LANZA; MARINESCU, 2007; KHOMH et al., 2012). Dessa forma, uma maneira possível de prevenir *bugs* pode se basear na identificação e remoção de casos de *code smells* no código.

Vários tipos de *code smells* foram definidos na literatura. Fowler e Beck (1999) descrevem 22 *code smells* e suas estratégias de refatoração. Lanza e Marinescu (2007) descrevem 11 desarmonias de *design*, agrupados nas três seguintes categorias:

- desarmonias de identidade: são falhas que afetam entidades do *design* simples, como classes e métodos. A particularidade dos *smells* deste grupo é que eles podem ser encontrados ao se examinar cada entidade isoladamente. Fazem parte do grupo os *code smells: God Class, Brain Class, Feature Envy, Brain Method, Data Class* e *Significant Duplication*;

- desarmonias de colaboração: são falhas de *design* que afetam o modo como várias entidades colaboram para executar uma funcionalidade específica. O grupo é formado por *Dispersed Coupling, Intensive Coupling* e *Shotgun Surgery*;

- desarmonias de classificação: são falhas que afetam a hierarquia de classes. É formado pelo *Refused Parent Bequest* e *Tradition Breaker*.

Como exemplo de *code smell*, podemos citar o *Feature Envy*, que se refere a métodos que parecem estar mais interessados em dados de outra classe do que da classe em que se encontram. Segundo Fowler e Beck (1999), é um indício de que o fragmento de código detectado pode estar mal localizado e deveria ser movido para a outra classe. O *Feature Envy* prejudica a coesão e o acoplamento de classes, o que é considerado um problema para o desenvolvimento de software, uma vez que são propriedades essenciais para o reúso de software (LANZA; MARINESCU, 2007). A Seção 2.1.2 apresenta em detalhe os *code smells* que serão abordados neste trabalho.

2.1.1 Métricas e Valores Limites

Com o aumento da complexidade dos sistemas e a crescente demanda por software de qualidade, foi preciso pensar em um mecanismo capaz de medir e validar produtos de software. Predominantemente, métricas de software passaram a ser utilizadas para controlar, orientar e entender os resultados reais do processo de desenvolvimento e para realizar estimativas sobre o andamento de projetos de software (SOMMERVILLE, 2011).

Segundo Lanza e Marinescu (2007), uma métrica é o mapeamento de uma característica particular de uma determinada entidade para um valor numérico. Para Grady (1992), as métricas são usadas para medir atributos específicos de um produto de software ou o processo de desenvolvimento de software com o intuito de ajudar a tomar decisões melhores.

SOMMERVILLE (2011) argumenta que as métricas ajudam a avaliar a complexidade, a compreensibilidade e a manutenibilidade em componentes de software. Por exemplo, Lanza e Marinescu (2007) propuseram a combinação de métricas de código-fonte para identificar possíveis problemas de *design*. Ou seja, identificar componentes de software

em que a qualidade não atingiu os padrões adequados, com o propósito de melhorar sua estrutura, reduzir sua complexidade ou torná-lo mais compreensível.

Quando se trabalha com métricas é importante saber que existem valores limiares ou pontos de referências que delimitam a fronteira da entidade em valores métricos, onde, dependendo da região em que o valor métrico se encontra, é possível fazer uma análise a respeito do mapeamento da característica da entidade medida (MARINESCU, 2004). A definição de valores limites inferiores e superiores para uma métrica é um meio de associar um valor numérico à semântica intrínseca ou à característica de algum elemento de código-fonte (exemplo: classe e método). Através desse valor, é possível identificar se a relação entre cada elemento e algum atributo medido é muito alto ou muito baixo, muito ou pouco, bom ou ruim (LANZA; MARINESCU, 2007).

Com o avanço nos estudos sobre *code smells*, métricas e estratégias de detecção, várias ferramentas foram desenvolvidas com o objetivo de auxiliar os engenheiros de software a melhorar a qualidade do software durante o ciclo de desenvolvimento. A maioria é capaz de detectar os vários tipos de *code smells* presentes na literatura. Contudo, elas podem fornecer diferentes resultados ao analisarem o mesmo conjunto de artefatos, considerando a configuração dos valores limites. Por isso, é importante saber quais são os valores limites utilizados pelas ferramentas e usá-los corretamente (VIDAL et al., 2015; TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2008).

Os valores limites ajudam a controlar a qualidade interna do software, pois, através deles, é possível detectar componentes com características que se desviam da norma. Se os valores identificados estiverem dentro dos limites estabelecidos, eles podem ser aprovados. Caso contrário, os componentes detectados com *code smell* devem ser revelados pois podem ter problemas de qualidade. Oportunamente, o gerente de projeto pode usar esses dados para tomar decisões de planejamento e alocar recursos para que estes componentes sejam analisados em profundidade ou podem ser priorizados na fase de testes (SOMMERVILLE, 2011).

A seguir são listadas as estratégias de detecção dos *code smells* estabelecidas por Lanza e Marinescu (2007) que farão parte deste estudo. Tais estratégias se baseiam nas métricas previamente citadas.

2.1.2 Estratégias de Detecção de Code Smells

Lanza e Marinescu (2007) estabeleceram estratégias de detecção baseadas em métricas de qualidade de código-fonte para identificar potenciais problemas de *design* em componentes de software. Assim, uma estratégia de detecção é uma regra de *design* que segue um método sistemático, formado por expressões lógicas, que obedecem a heurísticas particulares, codificadas com métricas, capaz de revelar estruturas com características específicas que podem indicar possíveis falhas em sistemas orientado a objetos. O objetivo é tornar essas regras de *design* quantificáveis e, assim, detectar os *code smells* (LANZA; MARINESCU, 2007; FOWLER; BECK, 1999).

Com o passar do tempo e com as constantes manutenções evolutivas e corretivas, se tornou comum a introdução de *code smells* no *design* do software. Desse modo, as estratégias de detecção são utilizadas para identificá-los e permitir tomadas de decisões

por parte dos engenheiros de software, como remover os *code smells* da estrutura do código-fonte para promover atributos de qualidade como confiabilidade, manutenibilidade e usabilidade. Para isso, Fowler e Beck (1999) recomenda a utilização do processo de refatoração.

A refatoração é uma estratégia utilizada para redefinir a estrutura interna do software de tal modo que não altere seu comportamento externo. Isso ajuda a manter o código com bons princípios de *design*, mais confiável e com menor propensão a falhas, além de reduzir o custo e o esforço envolvido na atividade de manutenção (FOWLER; BECK, 1999).

A seguir são apresentados os *code smells* que fazem parte deste trabalho e descritas suas estratégias de detecção específicas. Os *smells* descritos neste trabalho são: *Feature Envy, Brain Method, Intensive Coupling, Dispersed Coupling* e *Shotgun Surgery*. Esses *smells*, classificados a nível de método, foram escolhidos devido: (i) ao escopo de investigação definido, que tem como objetivo analisar os métodos de padrões de projeto e (ii) pela ferramenta de detecção de *code smells* utilizada, identificar esses *smells* de métodos.

2.1.2.1 ***Feature Envy*** Para um bom projeto arquitetural orientado a objetos os dados e os métodos relacionados entre si devem ficar o mais próximo possível. Essa proximidade ajuda a evitar efeitos indesejáveis, como o desencadeamento de várias alterações em cascata. O *code smell Feature Envy* refere-se a métodos que parecem estar mais interessados em acessar dados de outra classe do que da sua própria.

Tais métodos acessam excessivamente dados de outra classe a fim de executar algum cálculo ou tomar alguma decisão. Quando isso ocorre, afeta a flexibilidade do *design*, pois esse tipo de acesso aos dados conduz a uma relação de dependência. Portanto, pode ser um sinal de que o método está na classe errada e deve ser encaminhado para outra classe. É, portanto, um caso próprio de acoplamento com impacto negativo na capacidade de reutilização (LANZA; MARINESCU, 2007; FOWLER; BECK, 1999).

A estratégia de detecção na Figura 2.1 é baseada na contagem do número de atributos referenciados, direta ou indiretamente, que estão em classes externas à classe do método sob análise.

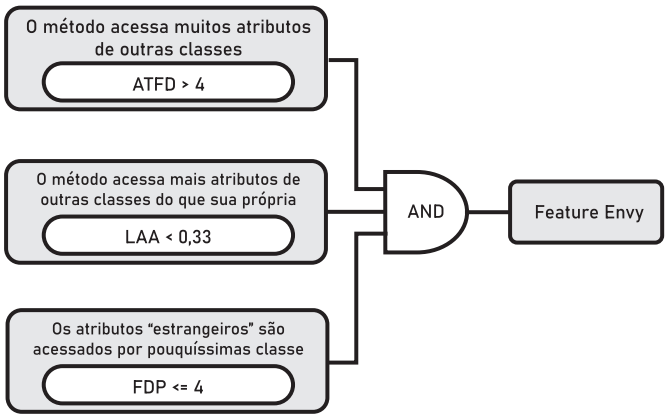


Figura 2.1 Estratégia de detecção do *Feature Envy*
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 85).

A estratégia na Figura 2.1 é composta pelas seguintes heurísticas:

1. **O método acessa muitos atributos de outras classes:** É usada a métrica ATFD (*Access To Foreign Data*) que conta o número de atributos de classes não relacionadas acessadas diretamente ou por meio de métodos de acesso. O objetivo é identificar métodos que acessam uma grande quantidade de atributos de outras classes;

2. **O método acessa mais atributos de outras classes do que da própria classe:** É usada a métrica LAA (*Locality of Attribute Accesses*) que calcula o número de atributos de métodos de classe dividido pelo número total de variáveis acessadas. Pretende-se identificar e comparar a quantidade de atributos acessados internamente com atributos de outras classes;

3. **Os atributos "estrangeiros" são acessados por pouquíssimas classes:** Utiliza-se a métrica FDP (*Foreign Data Providers*) para contar o número de classes que possuem atributos acessados. Utiliza-se essa métrica pelo fato de que é necessário identificar métodos que usam dados de poucas classes. Pois a natureza deste *code smell* está no fato de que ele reflete uma dependência bem direcionada.

Refatoração: O *code smells Feature Envy* pode ser resolvido movendo o método para a classe à qual ele está mais acoplado. Se apenas uma parte do método faz referência a outro recurso, pode ser necessário mover apenas essa parte para a classe invejada, isto é, a classe da qual acessa muitos atributos. Se o método invejar duas classes diferentes, o ideal é que seja movido para a classe que ele mais usa.

2.1.2.2 *Brain Method* São métodos que tendem a centralizar a inteligência da classe. Em geral, no princípio, o método não apresenta problema, porém à medida que mais funcionalidades são adicionadas, mais longo, mais difícil de entender e reutilizar eles ficam. As principais características do *Brain Method* são: excessivamente grande, muitos ramos condicionais com níveis de aninhamento profundo e usam muitas variáveis (LANZA; MARINESCU, 2007; FOWLER; BECK, 1999). Para detectar esse tipo de *smell*, é proposta a estratégia ilustrada na Figura 2.2.

A estratégia de detecção é baseada na convergência de três *smells* descritos por Fowler e Beck (1999): (i) métodos longos: não são desejáveis, pois afetam a compreensão e a testabilidade do código. Métodos longos tendem a implementar mais de uma funcionalidade e, portanto, utilizam muitas variáveis e parâmetros internos, tornando-os mais propensos a erros; (ii) ramificação excessiva: o uso intenso de instruções *switch* ou *if-else-if*, em muitos casos, é um sintoma claro de um design não orientado a objetos, no qual o polimorfismo é ignorado; e (iii) muitas variáveis utilizadas: o método utiliza muitas variáveis locais, como também muitas variáveis de instância. A estratégia de detecção é composta pelas seguintes heurísticas:

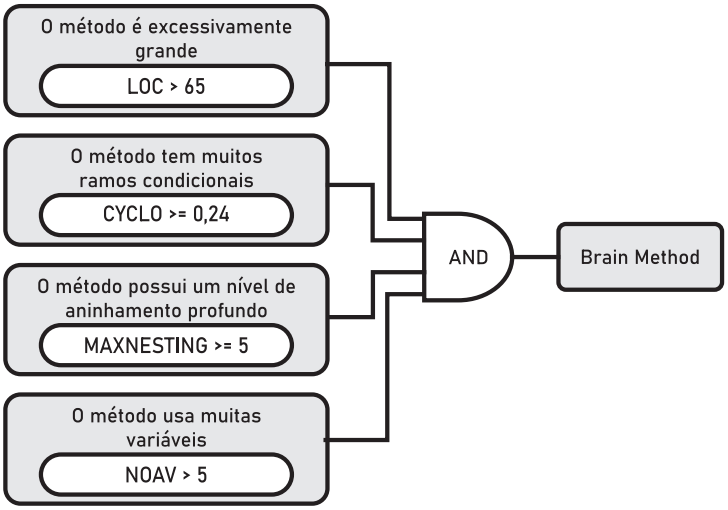


Figura 2.2 Estratégia de detecção do *Brain Method*
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 93).

1. **O método é excessivamente grande:** É utilizada a métrica LOC (*Lines of Code*) que calcula o número de linhas do método excluindo linhas em branco e comentários. Através desta métrica é possível calcular se o método é excessivamente grande.

2. **O método tem muitos ramos condicionais:** É usada a métrica CYCLO (*Cyclomatic Number*) para verificar as ramificações condicionais do método, pois com ela é possível obter a quantidade de caminhos linearmente independentes de uma operação.

3. **O método possui um nível de aninhamento profundo:** Utiliza-se MAXNESTING (*Maximum Nesting Level*) pelo fato de que é necessário contar o nível de aninhamento do método, ou seja, o nível máximo de aninhamento de estruturas de controle do método;

4. **O método usa muitas variáveis:** É usada a métrica NOAV (*Number Of Accessed Variables*) que realiza a contagem do número de variáveis acessadas diretamente pelo método, incluindo variáveis locais, parâmetros, atributos e variáveis globais.

Refatoração: O *Brain Method*, na maioria dos casos, deve ser dividido em um ou mais métodos simples. Fowler e Beck (1999) explica que uma boa técnica para descobrir o "ponto de corte" é identificar os fragmentos em que há a necessidade de comentar algo.

2.1.2.3 ***Intensive Coupling*** São métodos que estão vinculados a diversas outras operações no sistema, no qual essas operações estão dispersas em uma ou algumas classes. Ou seja, a dispersão das chamadas de método é baixa, enquanto a intensidade da chamada é muito alta, onde um único método de cliente está fortemente acoplado em apenas algumas classes provedoras. As principais características são: possui operações

condicionais aninhadas e chama muitos métodos do sistema, mas que pertencem a um número relativamente pequeno de classes. (LANZA; MARINESCU, 2007).

A estratégia de detecção de casos de *Intensive Coupling* (acoplamento intensivo) se encontra ilustrada em duas partes, nas Figuras 2.3 e 2.4.

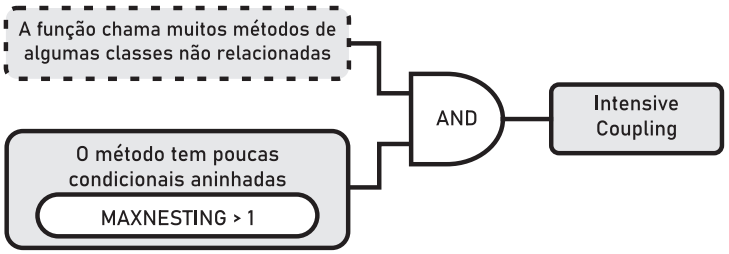


Figura 2.3 Estratégia de detecção do *Intensive Coupling*
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 121).

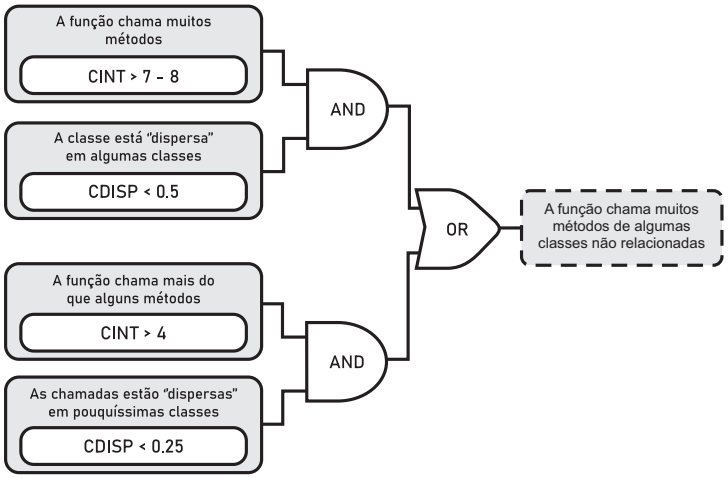


Figura 2.4 A função chama muitos métodos de algumas classes não relacionadas
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 122).

A estratégia de detecção é composta por duas condições principais que devem ser atendidas simultaneamente: a função deve invocar muitas operações de poucas classes e o seu nível de aninhamento deve ser baixo para evitar detectar casos de operações de configuração, por exemplo, métodos inicializadores ou de configuração de interface de usuário, pois revelam uma forma de acoplamento menos prejudicial (Figura 2.3). A estratégia de detecção é composta pelas seguintes heurísticas:

1. **A função chama muitos métodos de algumas classes não relacionadas:** A condição básica para que um método seja considerado com acoplamento intensivo é chamar muitos métodos pertencentes a algumas classes não relacionadas. Utiliza-se a métrica CINT (*Coupling Intensity*) para contar o número de métodos distintos chamados pelo método sob investigação e a dispersão é calculada pela métrica

CDISP (*Coupling Dispersion*) que conta o número de classes usadas chamados pelo método sob investigação dividido pelo resultado da métrica CINT (Figura 2.4);

2. **O método tem condicionais aninhados:** Utiliza-se MAXNESTING para verificar o nível de aninhamento do método que, neste caso, deve ser baixo.

Refatoração: Para o *code smells Intensive Coupling*, uma possível ação consiste em criar um novo serviço, mais complexo, na classe provedora e substituir as múltiplas chamadas por uma única chamada ao novo serviço criado.

2.1.2.4 ***Dispersed Coupling*** São métodos que estão vinculados a muitas outras operações dispersas em várias classes. O *Dispersed Coupling* é complementar ao *Intensive Coupling*, em que um único método cliente comunica-se com uma quantidade excessiva de classes em que a comunicação com cada uma dessas classes não é muito intensa. Ou seja, a operação chama um ou poucos métodos de cada classe (LANZA; MARINESCU, 2007).

A estratégia de detecção de casos de *Dispersed Coupling* se encontra ilustrada em duas partes, nas Figuras 2.5 e 2.6.

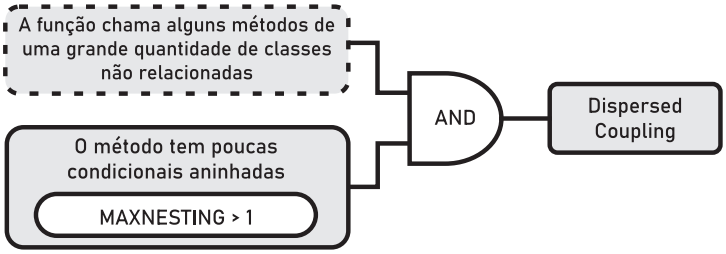


Figura 2.5 Estratégia de detecção do *Dispersed Coupling*
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 128).

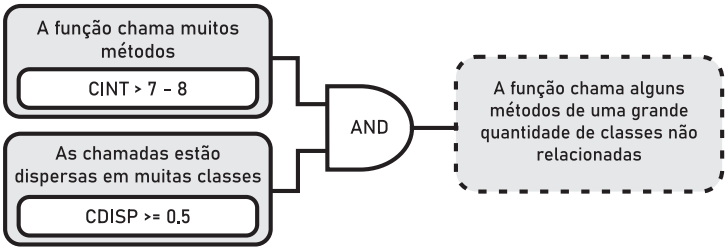


Figura 2.6 São operações que chamam poucos métodos dispersos em várias classes
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 128).

A regra de detecção segue o mesmo processo da regra definida para o *Intensive Coupling*, se diferenciando em apenas um aspecto: neste caso, captura-se as operações que apresentam alta dispersão de seu acoplamento. Ou seja, identifica os métodos que invocam operações de muitas classes diferentes (Figura 2.5). A estratégia de detecção em detalhes é:

1. **A função chama alguns métodos de uma grande quantidade de classes não relacionadas**: As métricas utilizadas são as mesmas já utilizadas para detectar Acoplamento Intensivo. Utiliza-se a métrica CINT para identificar a chamada de muitos métodos de outras classes, e a métrica CDISP para identificar a grande dispersão entra as classes das operações invocadas.

2. **O método tem poucas condicionais aninhados**: Utiliza-se MAXNESTING exatamente como no *Intensive Coupling* para ignorar casos irrelevantes.

Refatoração: É comum que o método com o *code smells Dispersed Coupling* seja também um *Brain Method*. Por isso a técnica de refatoração é baseada principalmente em dividir em mais métodos.

2.1.2.5 ***Shotgun Surgery*** São métodos que possuem um forte acoplamento aferente (entrada), através do qual muitas classes dependem dele. Isso indica que uma mudança em tais métodos pode provocar muitas mudanças em outros métodos e classes. Portanto, uma alteração realizada no método pode causar erros e consequentemente problemas de manutenção (LANZA; MARINESCU, 2007).

A Figura 2.7 exibe a regra de detecção do *smell*, *Shotgun Surgery*.

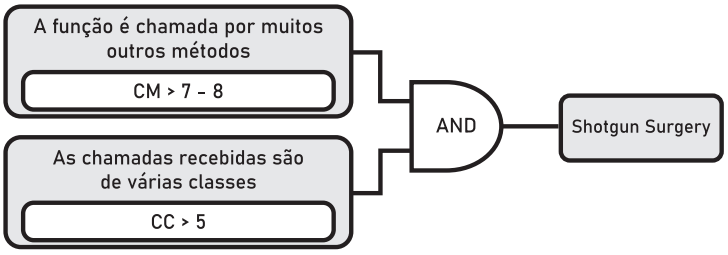


Figura 2.7 Estratégia de detecção do *Shotgun Surgery*
Fonte: Adaptado de (LANZA; MARINESCU, 2007) (p. 128).

A regra de detecção é definida com base nas operações que possuem uma grande quantidade de chamadas recebidas, com forte impacto de mudança, e na quantidade de diferentes tipos de chamadas, isto é, de funções que não pertencem à mesma classe nem à mesma hierarquia de classes da operação sob investigação. A estratégia de detecção (na Figura 2.7) é composta pelas seguintes heurísticas:.

1. **A função é chamada por muitas outras funções:** Um número alto de dependências aferentes pode causar problemas, como o anteriormente citado (uma mudança em um método implica em muitas mudanças em muitos outros métodos e classes). Um limite sugerido para o número de dependências aferentes é a capacidade da memória humana de curto prazo. Para identificá-lo usa-se a métrica CM (*Changing Methods*) que conta o número de métodos que chamam o método sob investigação.

2. **As chamadas recebidas são de várias classes:** Utiliza-se a métrica CC (*Changing Classes*) que quantifica o número de classes que chamam o método que está sob investigação, de modo que apenas os métodos que causam mais problemas de manutenção sejam detectados.

Refatoração: Os métodos com o *code smell Shotgun Surgery* que são grandes e complexos tendem a se tornar um *Brain Method*. Neste caso, a técnica de refatoração utilizada é baseada na divisão do método.

2.2 PADRÕES DE PROJETO

Projetar software reutilizável orientado a objetos é uma atividade difícil e complicada que exige anos de experiência na prática de desenvolvimento de sistemas. Isso vem da necessidade de identificar os objetos pertinentes ao domínio do projeto, separá-los em classes, definir as interfaces, as hierarquias de herança e estabelecer suas relações. Os engenheiros de software, quando encontram uma solução boa para um determinado problema, passam a utilizá-la frequentemente, pois são soluções testadas e comprovadas. Estas soluções são conhecidas como padrões de projeto (GAMMA et al., 1995).

Segundo Gamma et al. (1995), padrões de projeto descrevem soluções de projeto simples e elegantes para resolver problemas específicos de *design* que ocorrem com frequência em sistemas orientados a objetos. Esses padrões trazem qualidade ao projeto, pois são soluções que funcionaram e evoluíram a partir de resultados testados e comprovados de que tornaram o código mais flexível, manutenível e reutilizável. Portanto, os padrões de projeto são soluções comprovadas baseadas na experiência adquirida em projetos anteriores. Assim, eles podem ser aplicados imediatamente na prática visando trazer qualidade ao desenvolvimento de uma nova solução. No entanto, a utilização de padrões não garante que a solução esteja livre de todos os problemas relacionados à arquitetura e *design* do software.

Gamma et al. (1995) sintetizam o propósito e os impactos da aplicação de padrões:

> Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização. (GAMMA et al., 1995)

Vários tipos de padrões de projeto foram definidos na literatura. Gamma et al. (1995), em seu livro *"Design Patterns: Elements of Reusable Object-Oriented Software"*, descrevem vinte e três padrões de projeto divididos em três categorias: de criação, estrutural e comportamental, como mostra a Tabela 2.1. A Tabela 2.1 introduz a definição das categorias e seus conjuntos de padrões de projeto.

1. **Padrões de criação:** Abstraem a forma como os objetos são instanciados. Assim, eles tornam o sistema mais independente e flexível, pois reduzem a preocupação de desenvolvedores sobre como objetos são criados, compostos e representados:

Tabela 2.1 Classificação dos padrões nas três categorias

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method (112)	Adapter (classe) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (objeto) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Fonte: (GAMMA et al., 1995) (p. 26).

(a) *Abstract Factory*: Fornece uma única interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas;

(b) *Builder*: Separa a construção de um objeto complexo da sua representação de maneira a criar diferentes representações com o mesmo processo;

(c) *Factory Method*: Fornece uma interface para criar um objeto, mas delega para as subclasses decidirem qual classe será instanciada. Ou seja, o padrão permite que uma classe postergue a instanciação das suas subclasses;

(d) *Prototype*: Especifica os tipos de objetos a serem criados, definindo um protótipo que, usando uma instância prototípica, cria novos objetos a partir deste protótipo;

(e) *Singleton*: Garante que uma classe seja instanciada apenas uma vez e fornece apenas um único ponto global de acesso ao objeto.

2. **Padrões estruturais:** Se preocupam sobre como ocorre a composição das classes e objetos para formação de estruturas mais complexas, de modo a permitir o baixo acoplamento e flexibilidade na criação de sistema:

(a) *Adapter*: Converte a interface de uma classe em outra interface suportada pelo cliente. O padrão permite que classes com interfaces incompatíveis trabalhem juntas;

(b) *Bridge*: Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente;

(c) *Composite*: Compõe um objeto a partir de um estrutura de árvore para representar hierarquias do tipo partes-todo. O padrão permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme;

(d) *Decorator*: Atribui comportamentos adicionais a um objeto dinamicamente. O padrão fornece uma alternativa flexível a subclasses para extensão da funcionalidade;

(e) *Façade*: Provê uma interface unificada para um conjunto de interfaces de um subsistema. O padrão define uma interface de nível mais alto que torna o subsistema mais fácil de usar;

(f) *Flyweight*: Usa compartilhamento de objetos para suportar grandes quantidades de informações de maneira eficiente;

(g) *Proxy*: Fornece um objeto substituto ou um marcador de outro objeto para controlar o seu acesso.

3. **Padrões comportamentais:** atuam na interação de classes e objetos e distribuem responsabilidades:

(a) *Chain of Responsibility*: Evita o acoplamento entre o objeto remetente de uma solicitação e o objeto destinatário, dando a chance de mais de um objeto de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate;

(b) *Command*: Encapsula uma solicitação como um objeto, desta forma permitindo a parametrização de outros objetos que, através de diferentes solicitações, enfileirem ou registrem solicitações e operações que podem ser desfeitas;

(c) *Interpreter*: Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem;

(d) *Iterator*: Fornece uma maneira de acessar os elementos de uma coleção de objetos sem expor sua representação interna;

(e) *Mediator*: Define um objeto que encapsula a maneira como um conjunto de objetos interagem. O padrão promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente o que permite variações em suas interações;

(f) *Memento*: Captura e guarda o estado interno de um objeto em um determinado momento sem violar o encapsulamento, de maneira que o mesmo possa posteriormente ser restaurado para este estado;

(g) *Observer*: Define uma dependência um-para-muitos entre objetos, de maneira que, quando um objeto mudar o estado atual, todos os seus observadores sejam automaticamente notificados e atualizados;

(h) *State*: Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe;

(i) *Strategy*: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O padrão permite que o algoritmo varie independentemente dos clientes que o utilizam;

(j) *Template Method*: Define o "esqueleto"de um algorítimo em uma operação, postergando a definição de alguns passos para subclasses. Como consequência, o padrão permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura;

(k) *Visitor*: Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O padrão permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

As soluções de *design* especificadas pela GoF (GAMMA et al., 1995) mostram como as responsabilidades devem ser distribuídas entre as classes e como essas classes estabelecem relações para solucionar problemas específicos de *design*. As responsabilidades definem o papel que cada classe e método deve desempenhar na solução do problema. As soluções descrevem estruturas básicas de projeto que podem ser adaptadas para resolver problemas que ocorrem com frequência em sistemas orientados a objetos. Por exemplo, o padrão de projeto *Observer* é formado por uma estrutura elementar de classes que desempenham os papéis de Observador (*Observer*) e Assunto (*Subject*) e métodos que cumprem o papel de notificadores (*notify*). Outro exemplo é o padrão de projeto *Decorator*, cujo o diagrama de classes exibido na Figura 2.8 apresenta a estrutura básica do padrão com seus papéis, métodos e relacionamentos (GAMMA et al., 1995).

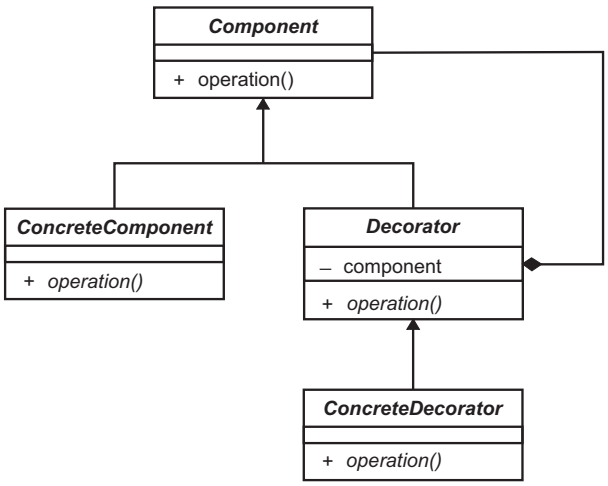


Figura 2.8 Diagrama de classe do padrão *Decorator*
Fonte: (GAMMA et al., 1995) (p. 128).

Dos 23 padrões de projetos descritos pela GoF (GAMMA et al., 1995), este trabalho analisa 14: *Factory Method*, *Prototype*, *Singleton*, *Adapter*, *Composite*, *Decorator*, *Proxy*, *Chain of Responsibility*, *Command*, *Observer*, *State*, *Strategy*, *Template Method* e *Visitor*. A seleção deste conjunto é motivada pelo fato de que existem ferramentas que automatizam o processo de detecção destes padrões e também como consequência das ferramentas disponíveis reconhecerem quase que exclusivamente os padrões de projeto *GoF* (WANG; ZHANG; WANG, 2018).

Capítulo 3

Este capítulo relata uma visão geral dos trabalhos relacionados conduzidos no campo de coocorrência entre padrões de projeto e code smells.

TRABALHOS RELACIONADOS

Durante o desenvolvimento deste trabalho foi encontrada uma revisão sistemática da literatura (SOUSA; BIGONHA; FERREIRA, 2018) que teve como objetivo pesquisar e analisar estudos primários que tratassem da relação entre padrões de projeto e *code smells*. A revisão sistemática foi realizada no ano de 2017 com o objetivo de entender o atual estado da arte sobre o relacionamento de padrões e *code smells* e identificar possíveis lacunas relativas ao tema para o desenvolvimento de novos trabalhos de pesquisa. Ao final de todo o procedimento de análise que uma revisão sistemática exige, foram considerados relevantes um total de 16 artigos.

Seguem elencadas abaixo as quatro questões de pesquisas definidas por Sousa, Bigonha e Ferreira (2018):

- (i) *"Como a literatura tem abordado a relação entre padrões de projeto e bad smells?"* Para esta questão de pesquisa concluíram que a literatura tem estudado as relações entre padrões de projeto e *code smells* de 3 formas diferentes:

    1. Impacto em qualidade de software: A maioria dos estudos encontram-se nesta categoria, com sete artigos. São trabalhos que realizam análises empíricas nas quais são avaliados aspectos e questões pertinentes à aplicação de padrões de projeto com o objetivo de melhorar a qualidade do projeto ou gerar impactos que degradem a estrutura do padrão;
    2. Refatoração: Esta categoria teve a segunda maior quantidade de estudos, com cinco artigos. São estudos que estabelecem uma relação de refatoração entre padrões de projeto e *code smells*. Nesses estudos, os padrões de projeto são propostos como prováveis soluções para eliminar as coocorrências com *code smells* ou fragmentos no código que prejudicam a qualidade do *design* do software;

3. Coocorrências: A categoria de coocorrências foi a que apresentou um menor número de estudos, com quatro artigos, com o objetivo de identificar relações de coocorrência entre padrões de projeto e *code smells* e apontar os motivos que geraram essas relações.

- (ii) *"A literatura tem explorado coocorrências entre padrões de projeto e bad smells?"* Para esta questão, os autores declararam que a resposta é afirmativa. No entanto, os resultados obtidos indicam que poucos estudos têm explorado essas relações de coocorrências entre padrões de projeto e *code smells*.

- (iii) *"Quais bad smells são abordados pela literatura na identificação de coocorrências com padrões de projeto?"* Mostraram que, nos trabalhos que avaliam coocorrência entre os padrões e *code smells*, foram identificados 18 *code smells*. A Tabela 3.1 mostra que os *code smells* descritos por Fowler e Beck (1999) têm sido utilizado na maioria dos estudos de coocorrência, 9 no total. Em menor quantidade, estão os *code smells* descritos por Brown et al. (1998), com o total de 6. Por último, 3 *code smells* descritos por Lanza e Marinescu (2007) foram identificados nos estudos dessa revisão sistemática.

- (iv) *"Quais padrões de projeto são usados pela literatura na identificação de coocorrências com bad smells?"* Para esta questão, concluíram que os padrões de projeto *Gang of Four* (GoF) são os mais utilizados pelos estudos analisados, como mostra a Tabela 3.2.

Tabela 3.1 *Code Smells* da Revisão Sistemática

***Bad smells* descritos por Brown *et al.* [1998]**
Anti Singleton, Blob, Class Data Should Be Private, Complex Class, Spaghetti Code e *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.*

Fonte: Adaptado de (SOUSA; BIGONHA; FERREIRA, 2018) (p. 07).

3.1 TRABALHOS

Segue abaixo a descrição dos artigos encontrados que fazem parte da relação coocorrência, que são os estudos que estão diretamente relacionados com o presente trabalho.

3.1.1 Cardoso e Figueiredo (2015)

Realizaram um estudo exploratório em cinco sistemas e, com base em regras de associação, analisaram duas situações: coocorrência de *Command* e *God Class* e coocorrência de

Tabela 3.2 Padrões de Projeto da Revisão Sistemática

[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.

Fonte: Adaptado de (SOUSA; BIGONHA; FERREIRA, 2018) (p. 07).

Template Method e *Duplicated Code*. Ao analisar o código-fonte, identificaram que o uso excessivo de uma classe do padrão *Command* para diferentes responsabilidades transformou a classe em uma *God Class*. Segundo os autores a melhor prática nesse caso seria criar diferentes instâncias do *Command* para diferentes interesses. Apontou-se também que o padrão *Template Method*, cujo objetivo é evitar duplicações, apresentou *code smell* do tipo *Duplicated Code*. Recomendaram mover o código duplicado para a superclasse e herdar a implementação nas subclasses. Com isto, mostraram que o uso inapropriado de um padrão pode conduzir ao surgimento de *smells*.

3.1.2 Walter e Alkhaeir (2016)

Realizaram uma análise na evolução de dois sistemas Java para identificar como a presença de padrões de projeto impacta a presença de *code smells* e como se dá a evolução desse vínculo ao longo do tempo. Com base nas regras de associação, identificaram que as classes que participam de um padrão tendem a possuir *code smells* com menos frequência do que outras classes. Com relação à evolução dos sistemas identificaram que a relação entre o número de *code smells* nas classes que participam de padrões e de *code smells* em outras classes é ligeiramente decrescente no tempo.

3.1.3 Sousa, Bigonha e Ferreira (2017)

Realizaram um estudo para investigar se o uso de padrões de projeto GoF evita o aparecimento dos *code smells God Class* e *Long Method* e identificaram as principais situações que levam os sistemas a apresentarem tais coocorrências. Com base em regras de associação, demonstram que o *Composite* e o *Factory Method* têm coocorrência baixa com os *smells God Class* e *Long Method*, e o *Template Method* e o *Observer* têm alta coocorrência com o *God Class* e o *Long Method*, respectivamente. Ao analisarem o código-fonte, encontraram classes com muitas responsabilidades, métodos complexos e repetição de código. Concluíram que o uso de padrões não necessariamente impede o surgimento de *smells*; pelo contrário, o uso incorreto contribui para o seu aparecimento.

3.1.4 Jaafar et al. (2014)

Realizaram um estudo empírico em três sistemas Java de código-fonte aberto. Foram analisadas a existência, a evolução e o impacto do relacionamento estático entre antipadrões e padrões de projeto na propensão à mudança e na propensão à falha. Aplicaram ferramentas de detecção e verificaram a significância estatística dos dados a partir de uma tabela de contingência e do teste exato de *Fisher* e *odds ratio*. Como resultado observaram que antipadrões apresentam relações estáticas com padrões de projeto, no entanto tais relacionamentos são temporários. Observaram também que classes com antipadrões que se relacionam com padrões de projeto são mais propensas a alterações e menos propensas a falhas do que classes com antipadrões que não estão correlacionados com padrões.

3.1.5 Sousa, Bigonha e Ferreira (2019)

Conduziram um estudo empírico em cinco sistemas *Java* com o objetivo de investigar a coocorrência entre padrões de projeto e *code smells* usando métricas de software e análise de código-fonte. Eles se concentraram em investigar se o uso de padrões de projeto reduz a ocorrência de *code smells*. Eles identificaram que a maioria dos padrões de projeto GoF não evitam necessariamente ocorrências de *code smells*. Os resultados mostraram que os padrões *Factory Method*, *Composite* e *Singleton* apresentaram baixa coocorrência com *code smells*, porém os padrões *Adapter-command*, *Proxy* e *State-strategy* apresentaram alta frequência de coocorrência com *code smells*. Após identificarem as coocorrências, realizaram um inspeção manual a fim de determinar as possíveis causas que possam ter contribuído para o surgimento das coocorrências. Descobriram que as coocorrências identificadas apareceram devido ao mau planejamento e aplicação inadequada de padrões de projeto.

3.1.6 Alfadel, Aljasser e Alshayeb (2020)

Com o objetivo de estudar a relação entre padrões de projeto e *code smells*, conduziram um estudo exploratório em dez sistemas *open source* desenvolvidos em *Java* que avaliou a propensão e a frequência de *code smells* entre classes que estão envolvidas em padrões de projeto e classes que não estão envolvidas. Eles analisaram a relação a nível de classe, a nível de categoria (criacional, estrutural e comportamental) e a nível individual de padrão de projeto. Semelhante aos estudos anteriores, associaram todas as instâncias de padrões de projeto e *code smells* à classe principal a que pertencem, ou seja, as instâncias de *code smells* do nível de método e os padrões de projeto implementados em classes internas foram associados à classe principal. No nível de classe, descobriram que as classes que participam de padrões de projeto exibem menos *smells* do que as classes que não participam de padrões de projeto. No nível de categoria, identificaram que quase não há diferenças significativas entre as categorias com relação a propensão a *code smells*. No nível de padrões de projeto individuais, comprovaram através da inspeção manual que alguns padrões de projeto estão associados a *smells*. Por exemplo, o padrão *Command* está associado aos *code smells Blob*, *External Duplication* e *God Class*; o padrão *Memento* está associado aos *code smells Blob* e *External Duplication*, e; o padrão *Decorator* não

está significativamente relacionado com *code smells*.

Os estudos anteriores demonstram a necessidade de se estudar o relacionamento de padrões de projeto e *code smells*. Compreender essa relação pode ajudar os engenheiros de software a refletir de forma crítica sobre a aplicação de determinados padrões de projeto e identificar mais facilmente classes que potencialmente precisam ser refatoradas. Este estudo tem como finalidade contribuir com o amadurecimento do tema dando ênfase ao estudo das correlações a nível de método.

Este trabalho, efetuou uma nova forma de identificação de coocorrência. O interesse é analisar métodos individuais que fazem parte da implementação de padrões de projeto nas situações em que tais métodos são afetados por *smells*. Foi identificado um único estudo que realizou a análise a nível de método, mas esse estudo considerou apenas o *code smell Long Method* (SOUSA; BIGONHA; FERREIRA, 2017). Os demais trabalhos não consideraram esse princípio e as suas análises englobam também métodos que não fazem parte da estrutura do padrão na forma prevista pela GoF (GAMMA et al., 1995). Como alternativa, o presente estudo analisa a relação procurando entender melhor as situações em que *smells* afetam os métodos previstos pelos padrões de projeto.

Capítulo

4

Esta seção descreve o método e os detalhes envolvidos no plano experimental para o estudo 1 (E1), assim como os resultados obtidos do experimento. O objetivo é entender o relacionamento estático entre padrões de projeto e code smells, evidenciando potenciais riscos associados à aplicação de determinados padrões.

COOCORRÊNCIA DE CODE SMELLS E PADRÕES DE PROJETO

4.1 OBJETIVO

O trabalho proposto foi desenvolvido por meio de 2 (dois) estudos experimentais com o objetivo de identificar as relações de coocorrência entre duas variáveis de pesquisa: (i) padrões de projeto *GoF* definidos por Gamma et al. (1995) e (ii) os *code smells* descritos por Lanza e Marinescu (2007). A meta é investigar se métodos que fazem parte da implementação de padrões de projeto coocorrem com *code smells*. A análise é feita através de métodos. Pretende-se avaliar, sob esse contexto, se a implementação desses métodos leva à incidência de *code smells*.

Os estudos seguiram o mesmo padrão metodológico, uma vez que os objetivos finais são os mesmos: visam entender a relação entre as duas variáveis citadas. Ambos demandaram a aplicação de ferramentas de detecção de padrões de projeto e *code smells* em sistemas desenvolvidos na linguagem Java. O primeiro estudo, Estudo 1 (E1), utiliza uma versão específica de cada sistema investigado, ou seja, uma versão final. E1 busca identificar as associações mais significativas e realizar uma inspeção manual no código-fonte, a fim de identificar as possíveis causas da coocorrência. O segundo estudo, Estudo 2 (E2), é aplicado sobre o histórico de versão (*commits*) proveniente de repositórios de software *open source*. E2 consiste em identificar coocorrências que se manifestam no momento da criação de padrões de projeto ao longo do tempo de evolução dos sistemas.

Com base na perspectiva anunciada, este capítulo objetiva apresentar o E1, a partir da exposição das questões de pesquisas, seguida da metodologia e análise dos dados, finalizando com os resultados e discussões que fundamentaram a formulação e a execução do E2, que será apresentado no próximo capítulo

4.2 QUESTÕES DE PESQUISA

Visando orientar os estudos, foram definidas duas questões de pesquisa gerais, QP1 e QP2. Com base na QP1 foram definidas mais duas questões específicas, QP1.1 e QP1.2. As questões QP1, QP1.1 e QP1.2 fazem parte do estudo experimental E1. Seguem abaixo as questões de pesquisa.

- (QP1) – Existe associação entre a ocorrência de padrões e a ocorrência de *code smells* em um mesmo método?

  Com essa questão de pesquisa, pretende-se verificar se existe interseção entre os métodos que fazem parte de um padrão e os métodos que apresentam *code smells*. Para isso, é investigada a existência de uma associação estatisticamente significativa entre as duas variáveis, e verifica-se se os métodos de padrões de projeto estão mais ou menos propensos a possuir *code smells* do que outros. Para tanto, foram utilizadas tabelas de contingência para registrar dados observados de um determinado experimento. Essa questão (QP1) é dividida em duas mais específicas, descritas a seguir.

  - (QP1.1) – Quais padrões de projeto estão mais propensos a coocorrerem com *code smells*?

    O objetivo da análise realizada para responder essa questão de pesquisa é identificar os padrões de projeto que estão mais propensos a ocorrerem em conjunto com *code smells*. Para isso, foram registrados os dados em tabelas de contingência e em seguida foram visualizados através de gráficos as associações estatisticamente significativas.

  - (QP1.2) – Quais as coocorrências mais frequentes nos sistemas analisados?

    Por meio dessa questão de pesquisa pretende-se verificar os pares padrão–*smell* significativamente associados. O objetivo é inspecionar manualmente o código-fonte e verificar alguma possível causa para existência desses pares.

A próxima seção (Seção 4.3) descreve a metodologia utilizada no experimento E1.

4.3 METODOLOGIA

Para responder às questões de pesquisa apresentadas anteriormente devemos: (i) selecionar as ferramentas de detecção de padrões de projeto e *code smells*, (ii) selecionar um conjunto de sistemas de software e (iii) viabilizar o processo de análise dos dados. A Figura 4.1 exibe as atividades conduzidas.

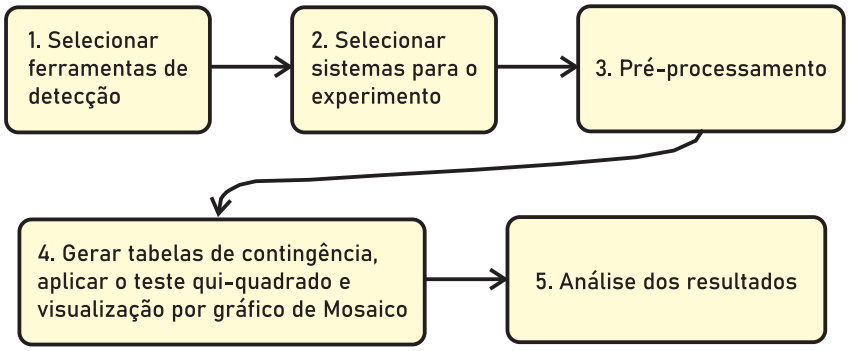


Figura 4.1 Metodologia de Pesquisa adotada no Trabalho.

A atividade **01** refere-se à seleção de ferramentas de detecção para o experimento. A escolha deverá seguir os seguintes critérios:

a) detectar a maioria dos padrões do catálogo GoF e dos *code smells* descritos por Lanza e Marinescu (2007) em projetos desenvolvidos na linguagem *Java* com código-fonte disponível online;

b) armazenar a saída dos resultados das análises de detecção em arquivos de texto. Isso permite realizar processamentos dos dados através de qualquer outra ferramenta habilitada a manipular dados em texto. Formatos proprietários ou binários de saída de dados poderiam dificultar a operação;

c) ser conhecida pela comunidade acadêmica e já ter sido usada em trabalhos anteriores. Tal critério possibilita avaliar a confiabilidade da ferramenta e de seus resultados.

A atividade **02** constitui-se da seleção dos sistemas que serão analisados. A escolha depende dos seguintes critérios:

a) devem possuir código-fonte aberto e disponível *online*;

b) serem escritos na linguagem *Java*. Esse critério foi escolhido por ser uma linguagem base de muitos projetos ativos.

A atividade **03** reúne os passos necessários para criação do conjunto de arquivos padronizados utilizados na análise. Os passos compreendem:

a) executar as ferramentas de detecção de padrões e *smells* em cada um dos sistemas e obter os arquivos resultantes da análise de detecção;

b) cruzar os dados com as informações de cada arquivo resultantes da análise de detecção e criar arquivos em um formato padronizado. O formato deve ser composto de: (i) o nome do padrão, (ii) o nome do sistema analisado e (iii) todas as instâncias encontradas de cada padrão associados aos *code smells*;

c) incluir apenas classes do sistema no cruzamento dos dados. Classes de teste e classes de bibliotecas externas não são incluídas no conjunto de dados. O entendimento é que classes de teste não fazem parte do conjunto (de classes) que implementam requisitos ou características primárias dos sistemas. Dessa forma, não são alvos comuns de aplicação de padrões de projeto. Além disso, existem *code smells* específicos para códigos de teste (DEURSEN et al., 2001), que estão fora do escopo deste trabalho. Portanto, foram removidos da análise os diretórios que armazenam as classes de testes, por exemplo *'fastjson/src/test/'* e *'commons-collections/src/test/'*. Classes de bibliotecas permanecem encapsuladas em seus arquivos binários (ou nos pacotes utilizados para injetá-las em sistemas), não sendo vistas no código-fonte dos sistemas que as usam. Dessa forma, não seriam detectadas e analisadas pelas ferramentas de detecção de padrões de projeto e *smells*.

A atividade **04** visa responder às Questões de Pesquisa. Para isso, devem ser criadas tabelas de contingência a partir das informações dos arquivos gerados anteriormente, o que permite tomar uma decisão entre duas hipóteses. As tabelas são utilizadas para gerar visualizações através de um gráfico de mosaico, destacando células em que a proporção observada é significantemente maior ou menor do que a proporção estimada sob o pressuposto de independência entre as variáveis. Na sequência, verifica-se se existe uma associação estatisticamente significativa entre as duas variáveis nominais através do teste do qui-quadrado. O tamanho do efeito foi medido através do V de Cramer.

Finalmente, na atividade **05**, ocorre a conclusão e compilação dos dados a fim de apresentar os resultados da pesquisa.

As próximas seções detalham a realização dessas atividades, de **01** a **04**, executadas nesses experimentos.

4.4 EXPERIMENTO I

Este estudo experimental visa responder as questões de pesquisa QP1, QP1.1 e QP1.2. Para tanto, analisa-se apenas a versão principal, ou seja, uma versão específica de cada um dos sistemas. O intuito é identificar as associações mais significativas de padrões de projeto e *code smells* e realizar uma inspeção manual no código-fonte. A inspeção tem, como fim, entender as possíveis causas da coocorrência. As subseções seguintes detalham as atividades **01** a **04** executadas para este experimento.

4.4.1 Atividade 01: Definição das Ferramentas

Tendo em conta os critérios definidos na Atividade **1**, foi decidida a utilização da ferramenta *Design Pattern Detection using Similarity Scoring* para detecção de padrões de projeto. Ela foi proposta por Tsantalis et al. (2006) e tem a capacidade de detectar os padrões GoF, disponibilizar os resultados em arquivo XML (eXtensible Markup Language) e foi utilizada em estudos anteriores (WALTER; ALKHAEIR, 2016; SOUSA; BIGONHA; FERREIRA, 2017). A ferramenta emprega um algoritmo baseado na pontuação de similaridade entre os vértices de um grafo direcionado. Os grafos são representados por duas matrizes, uma contendo a representação do código e a outra a representação do padrão. Com esses

dados, o algoritmo calcula uma pontuação que indica o grau de similaridade das matrizes. Al-Obeidallah, Petridis e Kapetanakis (2016) relataram que a ferramenta pode detectar padrões de projeto com uma precisão de 100% e um *recall* entre 66,7% a 100%. A Tabela 4.1 apresenta os padrões de projeto e os papéis detectados pela ferramenta (TSANTALIS et al., 2018).

Tabela 4.1 *Padrões e Papéis* identificados pela ferramenta de detecção

Padrões	Papéis
Factory Method	Creator, Product
Prototype	Client, Prototype
Singleton	Singleton
Adapter	Target, Adapter, Adaptee
Composite	Component, Composite
Decorator	Component, Decorator
Proxy	RealSubject, Subject, Proxy
Chain of Responsibility	Handler
Command	Invoker, Receiver, Command
Observer	Subject, Observer
State	State, Context
Strategy	Strategy, Context
Template Method	Abstract Class
Visitor	Visitor, ConcreteElement

Para a detecção de *code smells* foi utilizada a ferramenta JSpIRIT, uma vez que ela utiliza as estratégias de detecção proposta por Lanza e Marinescu (2007), disponibiliza os resultados em arquivos texto e também foi utilizada em estudos anteriores (WALTER; ALKHAEIR, 2016; SOUSA; BIGONHA; FERREIRA, 2017). A Tabela 4.2 resume os *code smells* detectados pela ferramenta. Os parâmetros de configuração das ferramentas não foram modificados, permanecendo os valores padrão.

Tabela 4.2 *Code Smells* de métodos identificados pela JSpIRIT

Code Smells	Descrição
Brain Method	Método longo e complexo que centraliza a inteligência de uma classe.
Dispersed Coupling	Método que chama um ou poucos métodos de várias classes.
Feature Envy	Método que chama mais métodos de uma classe externa do que os métodos da sua própria classe.
Intensive Coupling	Método que chama vários métodos que são implementados em uma ou poucas classes.
Shotgun Surgery	Método chamado por muitos métodos que são implementados em classes diferentes.

Fonte: Adaptado de Vidal et al. (2015).

4.4.2 Atividade 02: Definição dos Sistemas

Foram selecionados 25 sistemas aleatoriamente da lista fornecida pelo *Qualitas.class Corpus* (TERRA et al., 2013), de código-fonte aberto (*open source*), escritos em Java, e que utilizam os padrões de projeto do catálogo GoF (GAMMA et al., 1995). O *Qualitas.class Corpus* é uma versão compilada do *Qualitas Corpus* proposto por Tempero et al. (2010), com 111 sistemas orientado a objetos e mais de 18 milhões de linhas de código-fonte, 1,5 milhões de métodos e 200 mil classes compiladas.

É importante mencionar que devido a um erro interno na ferramenta JSpIRIT alguns dos sistemas selecionados inicialmente não tiveram os *code smells* extraídos, isto é, a ferramenta não apresentou resultado de detecção para os sistemas *ant-1.8.2*, *antlr-3.4*, *argouml-0.34*, *aspectj-1.6.9* e *hibernate-4.2.0*. Assim, foi preciso substituí-los por outros projetos selecionados também de forma aleatória.

A Tabela 4.3 mostra a lista de sistemas selecionados para o experimento. Pode-se observar que os sistemas possuem tamanhos e domínios diferentes, sendo o *fitjava-1.1* o menor, com 3.457 linhas de código e o itext-5.0.3 o maior, com 78.348 linhas de código.

Tabela 4.3 Sistemas analisados

Sistema	Classes	Métodos	LOC	Domínio
axion-1.0-M2	248	3.082	24.163	database
collections-3.2.1	414	3.788	55.398	tool
colt-1.2.0	291	3.511	35.919	SDK
displaytag-1.2	171	1.500	20.498	data visualiz.
drawswf-1.2.9	309	2.877	27.674	graphics
emma-2.0.5312	320	2.021	21.492	testing
fitjava-1.1	83	542	3.457	testing
FreeCS-1.3.20100406	146	1.521	22.645	tool
itext-5.0.3	582	5.846	78.348	data visualiz.
jena-2.6.3	1.073	9.886	65.774	middleware
jext-5.0	757	4.681	60.160	data visualiz.
jgraph-5.13.0.0	295	3.113	31.818	tool
jgrapht-0.8.1	248	1.428	17.232	tool
jmoney-0.4.4	82	726	8.197	tool
jOggPlayer-1.1.4s	285	2.314	29.878	graphics
jparse-0.96	75	954	24.796	parsers
jpf-1.5.1	139	1.370	13.342	SDK
junit-4.10	118	796	6.580	testing
nekohtml-1.9.14	52	546	7.647	parsers
oscache-2.3	72	587	7.624	middleware
picocontainer-2.10.2	202	1.380	9.253	middleware
pooka-3.0-080505	487	4.612	44.474	tool
proguard-4.9	635	5.718	62.618	tool
quickserver-1.4.7	185	1.805	18.339	middleware
sablecc-3.2	242	2.354	28.394	parsers
Total	7.511	66.958	725.720	

4.4.3 Atividade 03: Pré-processamento

Para cada sistema selecionado, foram removidas as classes de teste e foi definida que a análise se daria em nível de método. Por causa disso, não foram considerados *code smells* de classes, como o *God Class*. A Tabela 4.2 mostra os *code smells* utilizados neste estudo.

Foi criada uma aplicação para ler os arquivos de saída das ferramentas de detecção e gerar novos arquivos padronizados, que foram utilizados na fase de análise para criação das tabelas de contingência e geração de gráficos. A aplicação executa os seguintes passos:

a) realiza um *parsing* de cada um dos projetos Java e obtém o nome completamente qualificado dos métodos de todos os sistemas, *i.e.*, o nome do método prefixado pelo nome da classe e pelo nome do pacote;

b) lê a saída da ferramenta de detecção de padrões de projeto e gera um novo arquivo contendo o nome completamente qualificado do método, o padrão encontrado e o papel desempenhado pelo método dentro do padrão;

c) lê a saída da ferramenta de detecção de *code smells* e gera um novo arquivo contendo o nome completamente qualificado do método e o *smell* encontrado no método. Com esses arquivos é possível cruzar os dados dos padrões de projeto com os dados dos *code smells* de cada sistema.

4.4.4 Atividade 04: Processamento e Análise

Essa fase foi realizada em três etapas: (etapa I) cruzamento dos dados, (etapa II) construção de tabelas de contingência e (etapa III) visualização por meio de gráficos de mosaico e aplicação de testes de hipótese.

Após coletar as informações necessárias de padrões de projeto e *code smells*, a aplicação realizou o cruzamento dos dados das entidades (etapa I) e gerou as tabelas de contingência (etapa II) para cada sistema.

Para aplicação de testes de hipótese e visualização por meio de gráficos de mosaico (etapa III) verificou-se se existe uma associação estatisticamente significativa entre as duas variáveis categóricas nominais, através do teste de qui-quadrado.

Finalmente, foi visualizada a tabela de contingência através de um gráfico de mosaico, destacando células em que a proporção observada é significantemente maior ou menor do que a proporção estimada sob o pressuposto de independência entre as variáveis.

4.5 ANÁLISE E RESULTADOS

A combinação dos dados de todos os sistemas analisados resultou em 7.511 classes, 66.958 métodos e 725.720 linhas de código. Desses métodos, 3.179 (4,75%) estão associadas a padrões de projetos e 4.272 (6,38%) estão associadas a *code smells*.

Os três padrões mais frequentes nos sistemas analisados foram *State*, *Adapter* e *Decorator*. Os três *code smells* mais frequentes foram *Feature Envy*, *Shotgun Surgery* e *Brain Method*.

4.5.1 QP1: Existe associação entre ocorrência de padrões e ocorrência de code smell?

Inicialmente, foi computada uma relação de contingência através da Tabela 4.4, que relaciona a ocorrência de padrões de projeto e a ocorrência de *code smells* existentes nas amostras. A interseção da linha e coluna representa o número de observações que correlacionam padrões com *code smells*. A linha (l) 1 e coluna (c) 1 (l1c1) exibe o número de métodos sem *code smells* e sem padrões, a l1c2 exibe o número de métodos com *smells* e sem padrões, l2c1 exibe o número de métodos sem *smells* e com padrões e l2c2 exibe o

número de métodos com *smells* e com padrões. A última coluna exibe o total de métodos com e sem padrões. Observa-se que a prevalência de *smells* em métodos que implementam padrões (12,05%) é maior do que naqueles que não implementam padrões (6,10%).

Tabela 4.4 Tabela de contingência que relaciona ocorrência de padrões e ocorrência de *smells*

	SMELL		Total
PADRÃO	AUSENTE	PRESENTE	
AUSENTE	59.890 (93,90%)	3.889 (6,10%)	63.779 (100%)
PRESENTE	2.796 (87,95%)	383 (12,05%)	3.179 (100%)

Para visualizar as diferenças, a Figura 4.2 apresenta a Tabela 4.4 através de um gráfico de mosaico. Células destacadas em azul representam resíduos de *Pearson* superiores a dois, indicando uma proporção muito maior do que a esperada caso as variáveis fossem independentes. Analogamente, células em vermelho representam uma proporção menor do que a esperada. No gráfico, cada retângulo representa uma frequência e sua área é proporcional às observações de cada célula da tabela de contingência.

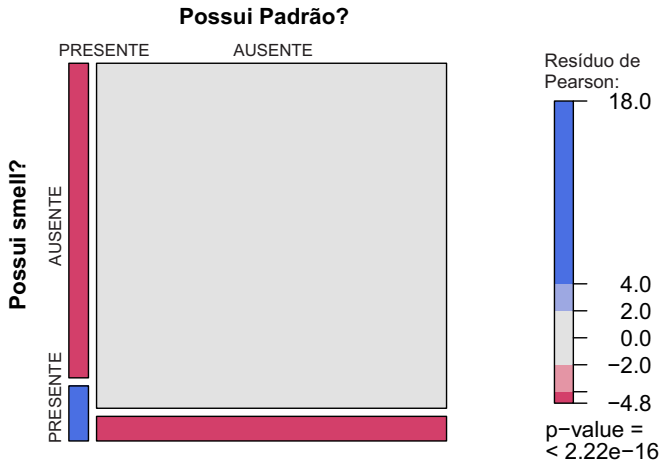


Figura 4.2 Propensão a apresentar *code smell* em padrões

Dentre os 3.179 métodos associadas a padrões de projeto, 383 (12,05%) possuem *code smells*. Dentre os 63.779 métodos que não estão associadas a padrões, há 3.889 (6,10%) métodos com *smells*. Desta forma, métodos associados a padrões de projeto estão 2,11 vezes mais propensos a conterem *code smells*.

Aplicando o teste do qui-quadrado, obtém-se $p < 2,2 \times 10^{-16}$. Portanto, a associação entre as variáveis é estatisticamente significativa (considerando um nível de significância de 5%). O V de Cramer foi 0,052, o que representa um tamanho de efeito pequeno. Assim, conclui-se que métodos que participam da implementação de padrões de projeto estão um pouco mais propensos a possuir *code smells*.

Considerando os dados estatísticos apresentados, para os 25 sistemas analisados, observa-se que os métodos de classes que participam da implementação de padrões de projeto mostraram-se associados à prevalência de *code smell*. Contraditoriamente, estudos anteriores demonstraram que classes que participam de um padrão tendem a possuir *code smells* com menos frequência do que outras classes. Observa-se, portanto, um resultado inesperado, pois a prevalência do *code smells* costuma ser baixa em classes de padrões de projeto de acordo com os estudos anteriores apresentados na Seção 3.

4.5.2 QP1.1: Quais padrões de projeto estão mais propensos a coocorrerem com code smells?

Para responder a essa questão de pesquisa, foi gerada a tabela de contingência mostrada na Tabela 4.5. A primeira coluna da tabela indica o padrão, a segunda aponta a quantidade de ocorrências do padrão sem *smells*, a terceira coluna apresenta a quantidade de ocorrências do padrão com *smells* e a quarta indica o total de métodos que implementam o padrão.

Tabela 4.5 Tabela de contingência que relaciona cada padrão com ocorrência de *smell*

	CODE SMELLS		
PADRÕES	AUSENTE	PRESENTE	total
Adapter	609	198	807
Bridge	112	6	118
Command	0	4	4
Composite	85	0	85
Decorator	561	28	589
Factory Method	162	11	173
Observer	23	2	25
Proxy	54	14	68
State	869	230	1099
Template Method	454	41	495
Visitor	130	1	131

A partir da Tabela 4.5 foi gerado o gráfico da Figura 4.3. Cada coluna indica um padrão e se subdivide na presença ou ausência de *smells*. Observa-se no gráfico que os métodos dos padrões *Adapter* e *State* de acordo ao tamanho da área, estão mais fortemente associados a *code smells* do que os demais padrões. O gráfico revelou que o padrão *Decorator* está mais propenso a não possuir *smells*. O padrão *Composite* não teve seus métodos associados a *code smells* e os métodos dos padrões *Command*, *Visitor* e *Observer* tiveram baixa incidência de *code smells*.

A diferença na proporção de métodos com *code smells* de um padrão para outro é estatisticamente significativa ($p < 2,2 \times 10^{-16}$). Além disso, o tamanho do efeito é grande (V de Cramer = 0,256, graus de liberdade = 10), revelando que há padrões muito mais ou

muito menos propensos a coocorrerem com *code smells* do que o que seria esperado caso tal propensão fosse independente do padrão.

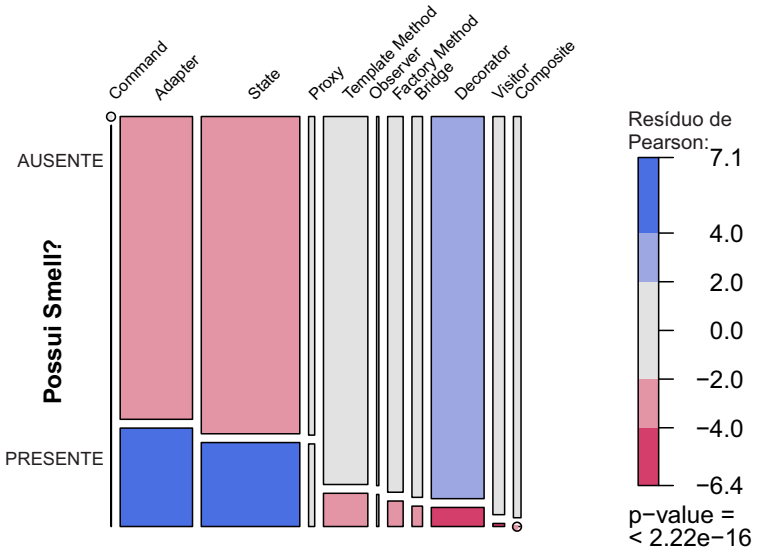


Figura 4.3 Propensão a apresentar *code smell*

O resultado encontrado na QP1 revelou que métodos de padrões de projeto estão associados à prevalência de *code smells*, no entanto, observa-se nesta análise que existe uma diferença significativa da presença de *code smells* entre os padrões de projeto. Assim, pode-se concluir que a prevalência de *code smells* em métodos de padrões de projeto apresenta-se distribuída de modo não uniforme, e os padrões *Adapter* e *State* são os que estão mais significativamente associados a *smells* do que outros padrões.

4.5.3 QP1.2: Quais as coocorrências mais frequentes, encontradas nos sistemas analisados?

Essa QP tem como objetivo obter resultados que apontem os padrões de projeto e os *code smells* que estão mais sujeitos a ocorrerem juntos. Para isso, foram relacionados todos os padrões de projeto com todos os *code smells*, como mostra a Tabela 4.6. Com esses dados, foi gerada a Figura 4.4 que apresenta o gráfico de mosaico relacionando, para cada método analisado, a presença de cada padrão de projeto e a ocorrência de cada *code smell*.

No gráfico da Figura 4.4, as colunas mostram os padrões de projeto e cada subdivisão indica a coocorrência com cada *code smells*. Ou seja, cada área do gráfico indica a interseção do padrão com o *code smells*. Sendo assim, observa-se no gráfico da figura que três células estão destacadas em azul, indicando uma proporção muito maior a coocorrerem: (i) coocorrência do padrão de projeto *Template Method* com o *code smell Shotgun Surgery*,

Tabela 4.6 Tabela de contingência que relaciona cada padrão com cada *smell*

	Adapter	Bridge	Command	Decorator	F. Method	Observer	Proxy	State	T. Method	Visitor	Total
Brain Method	19	2	2	3	0	0	1	40	6	0	73
Disp. Coupling	25	0	1	6	0	0	3	38	1	0	74
Feature Envy	133	3	0	16	0	2	9	112	10	0	285
Inten. Coupling	20	0	1	3	0	0	0	34	3	0	61
Shotgun Surgery	1	1	0	0	11	0	1	6	21	1	42
Total	198	6	4	28	11	2	14	230	41	1	-

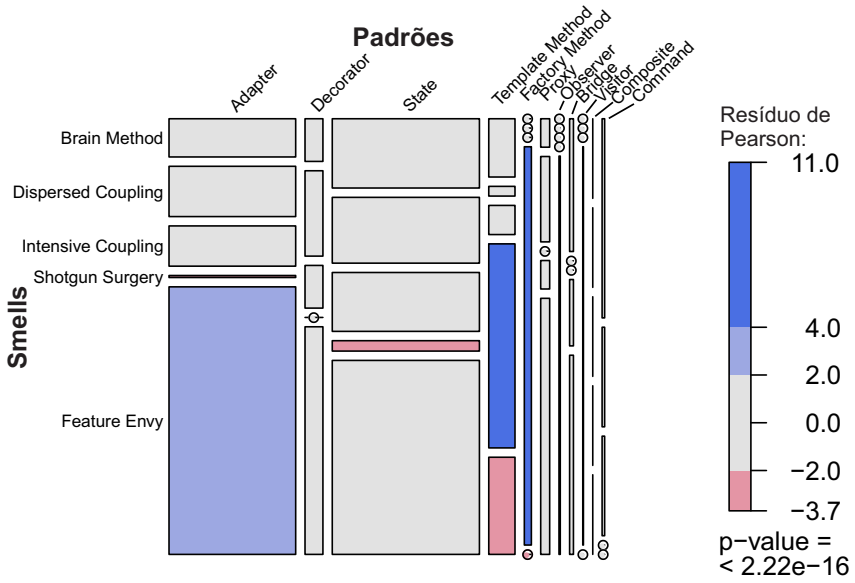


Figura 4.4 Cada padrão de projeto x cada *code smells*

(ii) coocorrência do padrão de projeto *Adapter* com o *code smell Feature Envy* e (iii) coocorrência do padrão de projeto *Factory Method* com o *code smell Shotgun Surgery*.

Por outro lado, o padrão de projeto *Template Method* possui uma propensão muito menor de possuir o *code smell Feature Envy* e o padrão de projeto *State* também possui uma menor propensão a possuir o *code smell Shotgun Surgery*. Para esses dados não foi possível realizar o teste do qui-quadrado, uma vez que muitas células da tabela de contingência possuem valor abaixo de 5.

Com essa questão de pesquisa foi possível identificar os pares padrão-*smell* mais propensos a coocorrerem: (i) coocorrência do padrão de projeto *Template Method* com o *code smell Shotgun Surgery*, (ii) coocorrência do padrão de projeto *Adapter* com o *code smell Feature Envy* e (iii) coocorrência do padrão de projeto *Factory Method* com o *code smell Shotgun Surgery*.

4.6 DISCUSSÃO

A aplicação de padrões de projeto revela, em tese, uma preocupação dos desenvolvedores com qualidade de projeto de software. Assim, esperava-se que métodos que participam de padrões de projeto raramente contivessem problemas de projeto. O resultado obtido na QP1, no entanto, foi inesperado: métodos que participam de padrões de projeto são mais propensos a conter *code smells*.

Isso não significa que todos os padrões de projeto são igualmente propensos a coocorrerem com *code smells*. Uma análise mais detalhada (QP1.1) revela que os padrões *Adapter* e *State* são os que mais contribuem para a associação encontrada entre padrões e *smells*.

Analogamente, os *smells* não ocorrem todos com a mesma frequência nos sistemas analisados. O *smell Feature Envy* é o mais frequente em métodos que participam de padrões de projeto (QP1.2). Em particular, métodos do padrão *Adapter* são 5,34 vezes mais propensos a conter o *Feature Envy* do que outros métodos. Assim, recomenda-se que desenvolvedores que desejam aplicar o padrão *Adapter* dediquem atenção especial a evitar introduzir o *smell Feature Envy* aplicando as técnicas de refatoração, se necessário (FOWLER; BECK, 1999).

Já o *Shotgun Surgery*, embora não seja tão comum, corresponde a boa parte dos *smells* encontrados em métodos que implementam os padrões *Template Method* e *Factory Method*. Em particular, todos os *smells* encontrados em aplicações do *Factory Method* foram do tipo *Shotgun Surgery*.

Considerando esses achados, foi decidido que seria vantajoso analisar as relações que estão mais propensas a coocorrerem. Assim, foi realizada uma inspeção manual no código-fonte a fim de entender as possíveis causas desses relacionamentos.

4.6.1 Template Method e Shotgun Surgery

A Tabela 4.7 lista os métodos encontrados na amostra associados ao padrão *Template Method* e ao *code smell Shotgun Surgery*. Uma dessas coocorrências encontra-se no sistema *colt-1.2.0*, no método *cern.colt.list.AbstractList.setSize*. Esse método foi escolhido de forma aleatória para a inspeção manual.

Como já explicado na Seção 2.2, o padrão *Template Method* define um esqueleto de um algoritmo dentro de um método e permite que parte de sua implementação seja realizada por subclasses através de métodos abstratos. A Figura 4.5 mostra um exemplo do diagrama do padrão identificado no sistema *colt-1.2.0*.

O diagrama contém a classe *AbstractList*, que encapsula o método *setSize()*. Esse método possui um corpo de código e foi detectado como sendo uma implementação do padrão Template Method. A classe também define os métodos abstratos *beforeInsertDummies()* e *removeFromTo()*. As subclasses *AbstractCharList* e *AbstractBooleanList* implementam os métodos abstratos *beforeInsertDummies()* e *removeFromTo()*, que são chamados no método *setSize()*. O Código 1 mostra uma parte da implementação da classe *AbstractList* com o método *setSize()* e as chamadas aos métodos abstratos. Em contrapartida, o *code smell Shotgun Surgery* (Tabela 4.2) representa métodos que são chamados por várias partes do sistema, e do qual muitas classes dependem. Ao realizar uma busca

Tabela 4.7 Coocorrência do *Template Method* com o *Shotgun Surgery*

Métodos
cern.colt.list.AbstractList.clear
cern.colt.list.AbstractList.quickSort
cern.colt.list.AbstractList.remove
cern.colt.list.AbstractList.setSize
cern.colt.matrix.DoubleMatrix1D.assign
cern.colt.matrix.DoubleMatrix1D.set
cern.colt.matrix.DoubleMatrix2D.assign
cern.colt.matrix.DoubleMatrix2D.like
cern.colt.matrix.DoubleMatrix2D.set
cern.colt.matrix.DoubleMatrix2D.viewColumn
cern.colt.matrix.DoubleMatrix2D.viewRow
cern.colt.matrix.DoubleMatrix2D.viewSelection
cern.jet.random.engine.RandomEngine.raw
gnu.regexp.REToken.next
jparse.stmt.StatementAST.getExceptionTypes
jparse.stmt.StatementAST.nextControlPoints
jparse.expr.ExpressionAST.retrieveType
jparse.expr.ExpressionAST.getExceptionTypes
jparse.expr.ExpressionAST.getValue
org.sablecc.sablecc.node.Node.cloneNode
org.sablecc.sablecc.node.Node.cloneList

pelas referências ao método *setSize()* no projeto, foram encontradas 86 referências em 41 classes espalhadas pelo código-fonte. Isso explica por que a ferramenta de detecção de JSpIRIT detectou este método como sendo um *Shotgun Surgery*.

```
1  public abstract class AbstractList {
2  ...
3   public void setSize(int newSize) {
4     if (newSize < 0)
5       throw IndexOutOfBoundsException(newSize);
6     if (newSize > currentSize)
7       beforeInsertDummies(int, int);
8     else if (newSize < currentSize)
9       removeFromTo(int, int);
10  }
11  abstract void beforeInsertDummies(int,int)
12  abstract void removeFromTo(int,int)
13 }
```

Código 1: Fragmento da classe *AbstractList* no colt-1.2.0.

Lanza e Marinescu (2007), no entanto, ao proporem estratégias de detecção para o *code smell Shotgun Surgery*, admitem que um número alto de dependências aferentes "também pode ser um bom sinal de reúso de *design* e funcionalidade", contanto que "as interfaces utilizadas sejam estáveis" (LANZA; MARINESCU, 2007) (tradução livre). Esse parece ser o caso de diversos métodos apresentados na Tabela 4.7, que representam operações elementares de tipos abstratos de dados como listas, matrizes e nós. Essas operações provavelmente possuem interface estável, variando apenas a implementação de alguns passos através da sobreposição (*override*) de métodos chamados por essas operações.

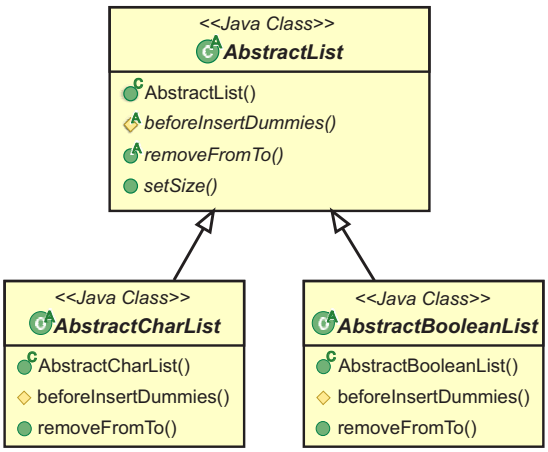


Figura 4.5 Fragmento do diagrama de classe do *Template Method* do sistema colt-1.2.0.

Além disso, pode-se argumentar que a aplicação do padrão *Template Method* contribui para aumentar tanto a estabilidade do método quanto a sua aplicabilidade. Ao definir pontos de variação na implementação de um método, é possível, em determinadas circunstâncias, alterar o comportamento do método sem alterar sua implementação ou sua assinatura através da implementação de métodos abstratos em subclasses. Dessa forma, pode-se adaptar o método a diferentes situações e, por isso, torná-lo útil em mais contextos, fazendo com que ele seja mais chamado.

Entende-se, então, que essa coocorrência não representa um caso de má aplicação do padrão de projeto, visto que o número excessivo de dependências aferentes, que caracteriza o *Shotgun Surgery*, é justificado pela estabilidade das interfaces. Portanto, as instâncias do *Shotgun Surgery* identificadas nesses casos podem ser consideradas como falso-positivos, uma vez que são irrelevantes para os desenvolvedores.

Diante do resultado obtido com a inspeção manual, recomendamos para os desenvolvedores de ferramentas de detecção de *code smells* que considerem o histórico de revisões dos métodos identificados com o *code smell Shotgun Surgery*. Caso os métodos sejam estáveis, eles não devem ser considerados instâncias do *code smell*. À vista disso, quanto menos falso-positivos forem reportados por essas ferramentas, menor será o trabalho de análise dos desenvolvedores durante o processo de manutenção, reduzindo o desperdício de tempo com análise de falso-positivos.

4.6.2 Adapter e Feature Envy

O código-fonte foi inspecionado para entender a coocorrência entre o padrão *Adapter* e o *smell*, *Feature Envy*. O padrão *Adapter* é utilizado quando há necessidade de que classes com interfaces distintas trabalhem respondendo a uma interface comum.

A Figura 4.6 mostra um fragmento do diagrama de classes do sistema *jena-2.6.3* que implementa o padrão *Adapter*. A classe *RETETerminal* é o adaptador e implementa o método *fire()*, da interface comum, *RETESinkNode*. Os objetos do tipo adaptador, *RETETerminal*, têm objetos da classe adaptada, *RETERuleContext*, sendo integrados por

meio de composição.

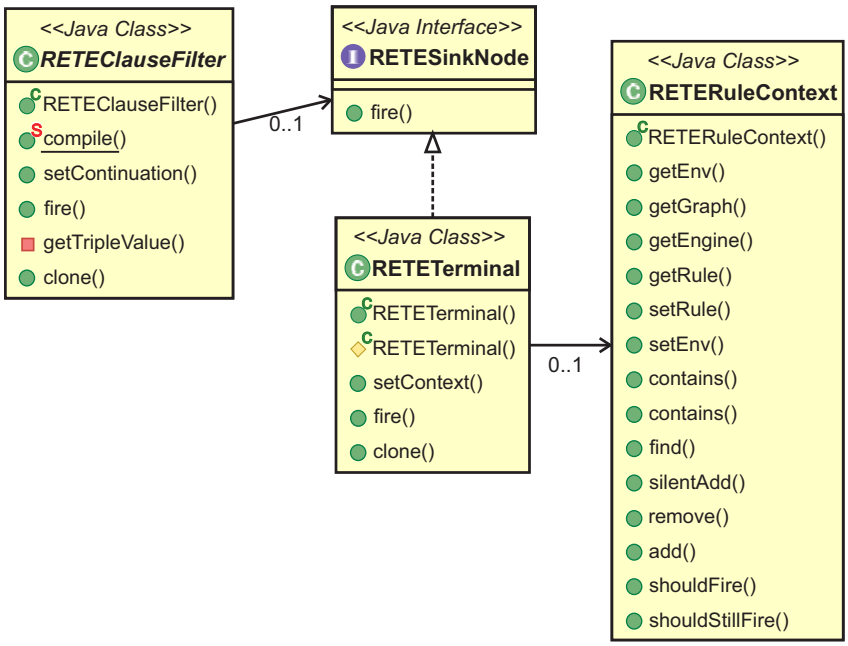


Figura 4.6 Fragmento do diagrama de classe do padrão *Adapter* do sistema jena-2.6.3.

O Código 2 mostra a implementação do método *fire()*, no qual, o JSpIRIT identificou esse método como *Feature Envy*, que refere-se a métodos que estão mais interessados em acessar dados de outra classe do que da sua própria classe.

Os exemplos do padrão *Adapter* comumente apresentados em materiais didáticos mostram o método do adaptador delegando funcionalidades para um único método da classe adaptada. O fragmento do sistema *jena-2.6.3*, no entanto, possui um método *fire()* com quatro chamadas a métodos da classe adaptada *RETERuleContext* através do objeto *context*. Possivelmente devido a essas chamadas, o JSpIRIT identificou esse método como *Feature Envy*.

```
1  public class RETETerminal implements RETESinkNode{
2  ...
3  //Context containing the specific rule...
4    protected RETERuleContext context;
5    public void fire(env, isAdd){
6      Rule rule = context.getRule();
7      context.setEnv(env);
8      if (! context.shouldFire(isAdd)) return;
9      context.getEngine().requestRuleFiring();
10   }
11 }
```

Código 2: Fragmento da classe *RETETerminal* no jena-2.6.3.

Para eliminar o *code smell Feature Envy*, nesse caso, seria necessário realizar a seguinte refatoração: (i) criar um novo método na classe adaptada (*RETERuleContext*), (ii) mover

o conteúdo do método *fire()* do adaptador (*RETETerminal*) para o novo método criado e (iii) realizar uma única chamada no método *fire()* ao método criado. No entanto, o uso do padrão *Adapter* sugere que houve uma decisão consciente de não modificar a classe *RETERuleContext*. Nesse caso é compreensível que a classe adaptador exiba os sintomas do *Feature Envy*, afinal, é característica da classe adaptador acessar membros da classe adaptada.

Fontana et al. (2016) cunharam o termo "*code smell* imposto" para se referir a *smells* que são efeitos colaterais de uma decisão de *design* consciente (por exemplo, a aplicação de um padrão de projeto), e argumentam que esses casos devem ser considerados falso-positivos. Eles citam o caso dos métodos *visit()* do padrão *Visitor*, que podem ser detectados como instâncias do *smell Feature Envy*, uma vez que eles acessam membros da classe visitada. Acreditamos que o mesmo raciocínio pode ser aplicado para o exemplo de coocorrência entre *Adapter* e *Feature Envy*. Portanto a detecção do *Feature Envy* pode ser considerada um falso-positivo.

Observa-se com esse resultado uma possível recomendação a ser seguida pelos desenvolvedores de ferramentas de detecção de *smells*: se um método implementar o padrão *Adapter* e, especialmente, se a classe adaptada não puder ser modificada (e.g., uma classe de componente de terceiros), talvez não faça sentido reportar o *smell Feature Envy*, pois a aplicação do padrão revela uma decisão de *design* consciente de criar uma classe que se interessa por outra, ainda que os métodos dessa outra classe não estejam organizados como esperado.

4.6.3 Factory Method e Shotgun Surgery

Inspecionamos a coocorrência entre o padrão *Factory Method* e o *smell Shotgun Surgery* com o objetivo de identificar uma possível causa para existência desse relacionamento. O padrão *Factory Method* consiste na definição de uma interface comum para a criação de objetos, mas deixa que as subclasses decidam que classe instanciar. O padrão permite adiar a instanciação para as subclasses que são responsáveis por criar os objetos solicitados pelo cliente. É muito comum a utilização desse padrão em *frameworks*, uma vez que eles encapsulam o conhecimento sobre a subclasse, mas deixa que a instanciação da classe seja realizada fora do *framework* (GAMMA et al., 1995).

A Figura 4.7 mostra um exemplo do diagrama do padrão identificado no sistema *jena-2.6.3* que é um *framework* em *Java* para construir aplicações para a web semântica. As interfaces *Creator* e *Product* formam a base do padrão e são representados respectivamente no diagrama pelas interfaces *ReasonerFactory* e *Reasoner*.

De acordo com a Figura 4.7, a estrutura do padrão é formado pelos seguintes participantes:

- ***Product*** (*Reasoner*): interface comum dos produtos a serem criados;
- ***ConcreteProduct*** (*RDFSFBRuleReasoner* e *OWLFBRuleReasoner*): representam os produtos concretos que implementam a interface *Product* (*Reasoner*). Para o sistema Jena, cada *Reasoner* representa uma máquina de inferência;

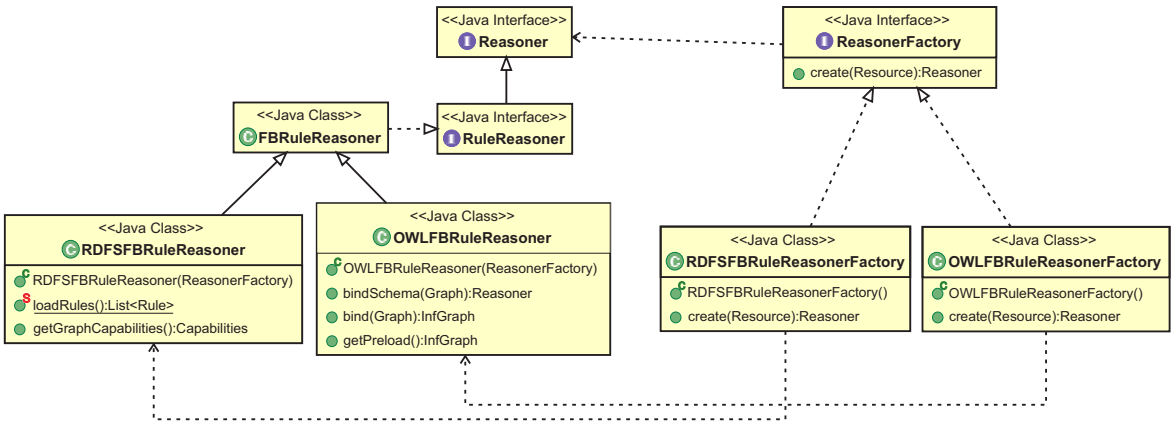


Figura 4.7 Fragmento do diagrama de classe do padrão *Factory Method* do sistema jena-2.6.3.

- ***Creator*** (*ReasonerFactory*): declara o método *factoryMethod* (*create(Resource): Reasoner*), que retorna um objeto *Product* (*Reasoner*).
- ***ConcreteCreator*** (*RDFSFBRuleReasonerFactory* e *OWLFBRuleReasonerFactory*): contém a implementação do método *factoryMethod* (*create(Resource):Reasoner*) que retorna uma instância de um *ConcreteProduct* (*RDFSFBRuleReasoner* ou *OWLFBRuleReasoner*) específico. Ou seja, para cada tipo de máquina de inferência existe uma classe de fábrica, que está em conformidade com a interface *ReasonerFactory*, usada para criar instâncias do *Reasoner* associado.

Analisando o código-fonte do sistema, foram identificadas 9 subclasses que implementam a interface (*ReasonerFactory*): *RDFSRuleReasonerFactory*, *OWLFBRuleReasonerFactory*, *OWLMiniReasonerFactory*, *TransitiveReasonerFactory*, *RDFSFBRuleReasonerFactory*, *AMLMicroReasonerFactory*, *GenericRuleReasonerFactory*, *WrappedReasonerFactory* e *OWLMicroReasonerFactory*. Para cada uma dessas fábricas, existe também um *ConcreteProduct* associado.

Com a análise, percebeu-se que o padrão *Factory Method* foi utilizado para desacoplar a construção das máquinas de inferência do código cliente. Desta forma, para obter uma instância de uma determinada máquina, é preciso apenas conhecer a fábrica que fornece os mecanismos para sua criação. Como consequência, o código cliente torna-se desacoplado da criação das máquinas de inferência. Esse desacoplamento possibilita alterar as máquinas de inferência sem precisar alterar outras partes do código. Isso reduz o acoplamento entre as classes e aumenta a flexibilidade e a reutilização do código-fonte.

A utilização do padrão também favoreceu a construção segura das máquinas de inferência, uma vez que elas são instanciadas apenas através da própria fábrica. Além disso, observa-se que, sempre que há necessidade de criar uma máquina, é preciso usar o método estático *theInstance()* implementado pelas fábricas, que retorna a interface *ReasonerFactory*. Só é possível construir as máquinas de inferência por meio dessa referência através do método (*create(Resource):Reasoner*). Dessa maneira, existe uma centralização na criação das máquinas, o que proporciona maior organização e manutenibilidade do código-fonte.

Com base na documentação oficial do Jena, é possível adicionar novas máquinas de inferência externas e a possibilidade de substituição de máquinas de forma facilitada, alterando o arquivo de configuração (*Resource*) na chamada do método (*create(Resource):Reasoner*). Como vantagem dessa abordagem, destaca-se o fornecimento de um meio de mapeamento das máquinas de inferência para suas instâncias em tempo de execução.

É provável que, devido à centralização do método *create(Resource):Reasoner* da interface *ReasonerFactory*, o polimorfismo promovido pelo uso do padrão torna o método útil, fazendo com que ele seja chamado de várias partes do sistema. Consequentemente, a ferramenta JSpIRIT o identificou como *Shotgun Surgery*. No entanto, a utilização do padrão *Factory Method* demonstra a intenção dos criadores do Jena de encapsular em um único método de interface a criação das máquinas de inferências. Deste modo, entende-se que essa coocorrência também não representa um caso de má aplicação do padrão de projeto, considerando os argumentos utilizados para justificar a coocorrência com o padrão *Template Method*. Portanto, as instâncias do *Shotgun Surgery* identificadas nesses casos podem ser consideradas também como falso-positivos.

Percebe-se, então, mais um caso em que é possível propor recomendação para os desenvolvedores de ferramentas de detecção de *smells*: se o método indicado com o *smell Shotgun Surgery* referir-se a um *Factory Method* e o método pertencer a interface ou ser um método abstrato, talvez não faça sentido reportar o *smell*, porque a aplicação do padrão revela uma decisão consciente de criar uma interface padrão que centraliza a criação de instâncias para uma família de classes.

4.7 AMEAÇAS À VALIDADE

Uma ameaça à validade externa ocorre devido a não ser possível dar maior extensão aos resultados encontrados, uma vez que as conclusões não podem ser generalizadas para outras linguagens orientadas a objetos. Isso vem como consequência de que esse trabalho analisou apenas sistemas desenvolvidos na linguagem Java. Para que seja possível generalizar os resultados é necessário ampliar o campo de pesquisa e analisar sistemas em outras linguagens orientadas a objetos. Uma outra questão a considerar é o fato de que apenas sistemas *open source* foram investigados. É provável que exista diferença entre o desenvolvimento de sistemas *open source* e sistemas proprietários. No entanto, para minimizar os efeitos, utilizamos projetos com diferentes tamanhos, propósitos e domínios, como também, os sistemas analisados foram desenvolvidos com variados estilos de *design*.

A detecção automática de padrões de projeto e *code smells* é outra ameaça à validade, devido a não ser um processo totalmente preciso. Para minimizar essa ameaça, buscou-se usar ferramentas que já foram utilizadas e avaliadas por outros estudos.

Outra questão é se a aplicação do padrões de projeto conduz ao surgimento de *smells*. Pode ser que a incidência de *code smells* em padrões de projeto esteja vinculada a outras características críticas do projeto, como o nível de experiência dos membros da equipe e vivacidade do projeto. Para minimizar esta ameaça, foram utilizados sistemas do *Qualitas.class Corpu*, devido a ele conter sistemas maduros e consolidados.

4.8 CONSIDERAÇÕES FINAIS

Foi apresentado um estudo experimental com o objetivo de investigar a relação entre *code smells* e padrões de projeto, a fim de entender com que frequência *code smells* ocorrem em métodos de padrões de projeto. Utilizamos ferramentas de detecção para identificar instâncias de padrões e *smells* em 25 projetos Java *open source* do *Qualitas.class Corpus*. Aplicamos testes estatísticos e visualizações para identificar as coocorrências.

O estudo apresentou uma investigação sobre a incidência de *code smells* em métodos individuais de padrões de projeto. Com esse nível de análise mais refinado, foi possível identificar situações nas quais a implementação de padrões de projeto levaram a ferramenta de detecção de *code smells* a identificar falsos-positivos e, portanto, foi possível propor recomendações para os fornecedores deste tipo de ferramenta mitigar a detecção desses *smells* em determinadas estruturas de padrões de projeto.

Os resultados encontrados sugerem que métodos que participam de padrões de projeto estão mais propensos a conter *code smells*. No entanto, o nível de significância estatística das relações varia entre os diferentes padrões de projeto. Ao analisar em detalhes cada um dos padrões, identificamos que o padrão *Adapter* e o *State* são os que mais contribuem para a associação. Os resultados também indicam que os pares de padrões-*smells* mais propensos a acontecerem são: *Template Method-Shotgun Surgery*, *Adapter-Feature Envy*, e *Factory Method-Shotgun Surgery*.

Ferramentas automatizadas foram desenvolvidas para identificar *code smells* em sistemas de software e medir a qualidade do código-fonte. No entanto, estudos vêm demonstrando que, em alguns casos, os *code smells* são justificáveis e nem sempre são prejudiciais para qualidade do software. Por exemplo, Fontana et al. (2016) propuseram um catálogo de falsos-positivos comuns para diversos *code smells*. Este trabalho demonstra que o número de falso-positivos ainda é, de fato, um problema para esse tipo de ferramenta e reforça a importância de analisar as instâncias de *code smells* por outras perspectivas, levando em consideração a implementação de padrões de projetos, uma vez que, a investigação manual realizada sugere que a existência do *code smell Shotgun Surgery* em métodos de padrões de projeto pode ser na verdade evidência de boas práticas de reúso de *design* e funcionalidade. Além disso, consideramos que é natural pensar que o padrão *Adapter* coocorra com o *smell Feature Envy*, dado o propósito do padrão. Assim, o próprio resultado de que métodos com padrões estão mais propensos a conter *smells* pode ser ameaçado por essas limitações das ferramentas de detecção de smells.

Capítulo

5

Esta seção descreve a metodologia adotada no plano experimental para o Estudo 2 (E2), os resultados obtidos e os instrumentos utilizados no experimento. O objetivo é entender o relacionamento entre padrões de projeto e code smell ao longo da evolução de projetos de software apresentando potenciais riscos na aplicação de padrões de projeto, como forma de mitigar possíveis efeitos negativos que podem afetar a qualidade do software.

EVOLUÇÃO DE CODE SMELL EM PADRÕES DE PROJETO

5.1 OBJETIVO

O resultado obtido no Estudo 1 (E1) demonstrou que há métodos de padrões de projeto que apresentam *code smells* (conforme visto no Capítulo 4). Contudo, o estudo apresentado não investigou em que momento essas correlações aparecem no código-fonte. Assim, o presente experimento visa investigar o histórico de versão (*commits*) de projetos de software, tendo por objetivo verificar se os métodos que fazem parte da implementação de padrões de projeto já nascem com algum *smell* ou se os *smells* são introduzidos à medida que o seu código é evoluído.

Chatzigeorgiou e Manakos (2014) investigaram a evolução de *code smells* e, como resultado, apontaram que uma porcentagem significativa deles são inseridos no projeto assim que uma entidade é criada. Ou seja, no instante em que o método ou classe em que *smells* existem são introduzidos no sistema. Além disso, relataram que, na maioria dos casos, os *smells* persistiram no código-fonte até a última versão analisada. Sob uma mesma linha de pesquisa, Tufano et al. (2017) realizaram uma investigação na evolução de 200 sistemas de software para entender quando e por que os *code smells* são introduzidos. De maneira geral, verificou-se que, por vezes, as entidades identificadas com *code smells*, já nascem com o *smell*, corroborando com os resultados de Chatzigeorgiou e Manakos (2014).

A partir dessas considerações, pressupõe-se que uma parte considerável de possíveis problemas de *design* pode ser uma consequência direta de análises e atividades de *design* iniciais ineficientes. Nesse contexto, pretende-se identificar casos em que o *code smell* é introduzido no código-fonte sincronicamente com a criação de métodos de padrão de

projeto. Tendo esse conhecimento mais aprofundado sobre essa relação, os engenheiros de software podem implementar medidas que ajudem a mitigar os possíveis efeitos negativos que podem afetar a qualidade do software. Vale salientar que, até o momento, nenhum outro trabalho avaliou a associação (padrão-*smell*) com esse objetivo.

Se identificada a relação, engenheiros de software devem ficar alertas para determinadas situações que podem exigir uma análise mais criteriosa, fundamental para a escolha correta da solução de *design*. O objetivo seria evitar problemas maiores no futuro decorrentes de uma possível ineficiência da análise. Além disso, entender esse relacionamento pode ajudar os desenvolvedores a (i) compreender melhor a formação dos *code smells* em padrões de projeto e o seu significado para a qualidade do *design*; (ii) identificar e evitar a introdução de *code smells* em métodos de padrões a partir de sua implementação inicial; e (iii) evitar a evolução de *code smells* em métodos de padrões de projeto.

Frente a esse contexto, realizou-se o presente estudo (E2), o qual foi orientado pela questão de pesquisa apresentada na próxima seção.

5.2 QUESTÃO DE PESQUISA

Para este estudo, formulamos a seguinte questão de pesquisa:

- (QP2) – Quais padrões, quando adicionados em projetos, possuem maior propensão a apresentar algum *code smell*?

  Essa pergunta procura identificar se a criação de um determinado padrão de projeto leva ao surgimento concorrente de *code smells*. Para esse fim, são utilizadas tabelas de contingência e visualizações baseadas em gráficos relacionados às associações estatisticamente significativas entre os dois conceitos considerados.

  A Seção 5.3 descreve a metodologia utilizada nos experimentos.

5.3 METODOLOGIA

Este estudo preservou os passos metodológicos utilizados em E1. No entanto, vale ressaltar duas diferenças entre E1 e E2: (i) o volume de dados analisados, significativamente superior em E2, no qual busca-se identificar as coocorrências entre padrões de projeto e *code smells* em cada *snapshot* (*commit*) dos sistemas analisados; e (ii) a substituição da ferramenta de detecção de *code smells*, uma vez que a ferramenta utilizada no E1 analisava uma versão dos sistemas de cada vez e só podia ser executada via interface gráfica. Esse ponto será discutido com mais detalhes na Seção 5.3.1 (*Definição das Ferramentas*).

Para atingir o objetivo proposto, foram analisados *snapshots* de projetos provenientes de repositórios *open source* do *GitHub*[1], uma das maiores plataformas públicas de hospedagem de código-fonte, possuindo mais de 50 milhões de usuários e 100 milhões de repositórios (GITHUB, 2020). Optou-se por utilizar o *GitHub* devido ao fato de que a maioria dos projetos utilizados no E1, dos quais foram selecionados um subconjunto para o E2, estão hospedados nessa plataforma.

As seguintes subseções detalham as atividades de **01** a **04** executadas nesse experimento.

[1] https://github.com/

5.3.1 Atividade 01: Definição das Ferramentas

A ferramenta utilizada para detecção de *code smells* no primeiro estudo só analisava uma versão dos sistemas de cada vez, o que inviabilizaria seu uso para analisar o histórico de alterações dos projetos de software de forma automatizada. Além disso, sua operação era feita via interface gráfica. A partir daí foi feita uma busca por uma ferramenta mais adequada, que fosse capaz de coletar informações de *code smells* em repositórios de software ao longo do histórico de evolução de sistemas. Para tanto, identificamos três ferramentas:

- *HistoryMiner*: analisa todo o histórico de alterações de projetos de software em ordem cronológica com o objetivo de detectar *commits* que introduzem *code smells*. A ferramenta minera cada *commit* do histórico do projeto e produz, para cada arquivo de código-fonte, uma lista de *commits* especificando se o código foi adicionado, modificado ou excluído e se foi afetado por algum *code smell*. Assim, é possível identificar o momento em que o *smell* foi introduzido no código-fonte (TUFANO et al., 2017).

- *HIST (Historical Information for Smell deTection)*: com uma abordagem semelhante ao *HistoryMiner*, a ferramenta analisa co-alterações que ocorrem entre os artefatos de código-fonte com base em informações de *commits* extraídos de sistemas de controle de versão. É capaz de identificar instâncias de cinco diferentes *code smells*: *Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, e Feature Envy* (PALOMBA et al., 2013, 2014).

- *RepositoryMiner*: *framework* desenvolvido na linguagem Java que tem como objetivo analisar repositório de software a fim de produzir informações relevantes sobre sua evolução. Através da extração e processamento de informações, é capaz de detectar *code smells* e permitir a integração com ferramentas de terceiros que tem por finalidade investigar a evolução do software. Suporta a detecção de 9 *code smells*: *Brain Class, Brain Method, Conditional Complexity, Data Class, Feature Envy, God Class, Long Method, Refused Parent Bequest* e *Depth of Inheritance Tree*. Após o processo de extração os dados são armazenados em um banco de dados (MENDES et al., 2017).

Baseado nesta pesquisa, optou-se pela substituição da ferramenta *JSpIRIT* pela ferramenta *RepositoryMiner*, uma vez que, além de detectar *smells* de maneira rápida, automatizada e com precisão, ela ajuda desenvolvedores e pesquisadores a minerar repositórios de software por coletar informações de *snapshot* de versões de software. (MENDES et al., 2017). Optou-se também utilizar o *RepositoryMiner* por: (i) não ter sido possível ter acesso às ferramentas *HistoryMiner* e *HIST*. Obteve-se apenas informações sobre elas, pois não encontram-se disponíveis para *download*. Por esse motivo foram desconsideradas; (ii) ser de código-fonte aberto e possuir uma arquitetura extensível que permitiu a completa integração com a ferramenta desenvolvida, sendo possível obter e armazenar os dados extraídos em arquivos padronizados utilizados na análise; (iii) utilizar as estratégias de

detecção de *smells* descritas por Lanza e Marinescu (2007); e (iv) ter sido utilizada em um quantidade razoável de estudos anteriores, e.g., (DIAS et al., 2019; GOMES et al., 2019; IBIAPINA et al., 2018; KHOMYAKOV et al., 2019; MENDES et al., 2015, 2019).

Não houve restrição quanto ao aproveitamento da ferramenta empregada no estudo E1 para detectar os padrões de projeto. Ela apresenta uma solução capaz de realizar a análise via linha de comando, o que permite detectar os padrões de projeto ao longo do histórico de alterações dos projetos de forma automatizada. Dessa forma, foi decidido utilizar a ferramenta *Design Pattern Detection using Similarity Scoring* pelos mesmos motivos/critérios explicados na Subseção 4.4.1 sobre Definição das Ferramentas do experimento E1. Além disso, outro ponto que embasa essa escolha é o fato de preservar a consistência dos dados de padrões de projeto do estudo anterior.

5.3.2 Atividade 02: Definição dos Sistemas

Depois de estabelecida a localização dos repositórios (GitHub) e de como obter suas informações (RepositoryMiner), em uma fase posterior foram selecionados os sistemas a serem submetidos à análise. Para isso, foram incluídos dois novos critérios além dos já definidos na Seção 5.3: (i) o sistema deve ser proveniente de repositórios *open source* do *GitHub*; e (ii) deve ser compilável de forma automática, através do Apache Maven[2] (FOUNDATION, 2002), devido à necessidade de compilação do código-fonte para execução da ferramenta de detecção de padrão de projeto. Optou-se utilizar o Maven por ser uma ferramenta de suporte para o gerenciamento automático de compilação a qual faz download das bibliotecas e *plug-ins* (FOUNDATION, 2002).

A princípio, iniciou-se a busca pelos repositório dos sistemas a partir da documentação fornecida pelo *Qualitas.class Corpus* (TERRA et al., 2013) que contém os *links* para os sites oficiais dos 25 sistemas analisados no E1. Nessa análise, foram identificados nove sistemas hospedados no *Github*: *itext, jena, jgrapht, junit, collections, displaytag, proguard, quickserver e sablecc*. O segundo critério está relacionado à ferramenta Maven para compilação dos sistemas. Ao investigar os repositórios dos projetos, foram identificados seis sistemas que são compilados através dessa ferramenta: *itext, jena, jgrapht, junit, collections e displaytag*.

Por restrições de tempo/recurso e o número elevado de *commits* presentes em todos os projetos identificados (25.719 *commits*, como mostra a Tabela 5.1) que, em função do processo de compilação, inviabilizaria o processamento, não foi possível realizar o estudo com todos esses sistemas. Então foram selecionados aleatoriamente 3 projetos: *commons-collections*, *displaytag* e *itext*. No entanto, foram incluídos na amostra mais dois sistemas que foram analisados durante a fase de teste da ferramenta desenvolvida: *fastjson* e *commons-io*. Considerou-se incluir esses sistemas tendo em vista que atenderam a todos os critérios estabelecidos. Além disso, são sistemas ativos com mais de 5 anos e possuírem respectivamente 75,49% e 57,18% do total de suas revisões compiladas. Um histórico extenso de desenvolvimento permite obter uma análise estatística significativa sobre muitos *commits*. Isso permite ter evidências mais consistentes a fim de determinar a significância das correlações.

[2]https://maven.apache.org

Tabela 5.1 Projetos do E1 identificados no *Github* compilados com o Maven

Sistema	*Commits*
displaytag	1.639
junit	2.565
jgrapht	3.277
commons-collections	3.782
itext	5.288
jena	9.078
Total	25.719

A Tabela 5.2 mostra a lista de sistemas selecionados para o experimento. Pode-se observar que eles possuem diferentes domínios, quantidade de classes, número de linhas de código e autores. A última coluna 'Período' da Tabela 5.2 mostra o período em ano em que os *commits* utilizados na amostra foram realizados.

Observa-se na Tabela 5.3 que os sistemas possuem quantidades de *commits* diferentes, sendo o *displaytag* o menor com 1.639 *commits* e o *fastjson* o maior, com 6.913 *commits*. A Tabela 5.3 também mostra a quantidade de *commits* compilados e a quantidade de *commits* descartados por não possuírem *commits* adjacentes. Esses foram excluídos por não apresentarem um *commit* imediatamente superior (pai), o que impossibilita a identificação de padrão-*smell* introduzidos simultaneamente.

Tabela 5.2 Sistemas analisados

Sistemas	Domínio	Classes	Linha de código	Autores	Período
commons-io	middleware	315	67.409	77	2002-2020
commons-collections	tool	590	131.201	86	2001-2020
fastjson	parsers	2.978	242.192	193	2011-2020
displaytag	data visualiz.	309	44.594	17	2003-2020
itext	data visualiz.	2.131	480.840	70	2014-2021
Total	-	6.323	966.236	443	2001-2021

Tabela 5.3 Total de *commits* analisados

Sistema	*Commits*	Compilados	Descartados (sem pai)
commons-io	2.460	1.857 (75,49%)	21
commons-collec	3.782	907 (23,98%)	79
fastjson	6.913	3.953 (57,18%)	527
displaytag	1.639	534 (32,58%)	108
itext	5.288	5.055 (95,59%)	178
Total	20.082	12.306 (61,28%)	913

5.3.3 Atividade 03: Pré-processamento

O objetivo desta atividade é realizar a extração de dados a partir de um Sistema de Controle de Versão (SCV) e armazená-los em arquivos padronizados utilizados no processo de análise desse estudo. Os procedimentos aplicados para criação desses arquivos são realizados pela aplicação desenvolvida utilizando o *framework RepositoryMiner* e a ferramenta *Design Pattern Detection using Similarity Scoring*. Isto é, foi desenvolvida uma ferramenta baseada no *RepositoryMiner* e no *Design Pattern Detection using Similarity Scoring* visando automatizar a extração e análise do histórico de evolução dos sistemas descritos na Tabela 5.2.

Para entender como ocorre a extração de dados do histórico de alterações é preciso saber como é a relação entre *commits* e um SCV. Ao realizar um *commit*, uma cópia de todos os arquivos alterados e uma referência dos arquivos não alterados são salvos e associados a um *snapshot*. Cada um desses *snapshots* contém um ponteiro (ou referência) para o *snapshot* anterior, com exceção do primeiro. Assim, ao realizar o *checkout* de um sistema, ou seja, o download de uma cópia do código-fonte, é também realizado o *download* da estrutura de *commits* que registra informações sobre as ocorrências e eventos ocorridos no projeto, que são os dados de criação, alteração e remoção de arquivos. Através desses registros, é possível criar uma organização cronológica através da qual, a partir de um registro de *snapshot*, é possível encontrar os seus ancestrais e assim identificar as alterações realizadas ao longo do tempo. Como resultado, a ferramenta é capaz de rastrear e coletar informações dos *smells* e padrões de projeto de cada classe do sistema percorrendo as diversas alterações do código-fonte.

Como o objetivo deste experimento é identificar se um *smell* é introduzido na classe do sistema juntamente com a introdução do padrão de projeto, a ferramenta percorre toda ancestralidade dos *commits* até identificar o *snapshot* em que os objetos de estudo foram introduzidos no sistema. Para tal, definimos a seguinte abordagem: (i) a ferramenta faz o *checkout* do primeiro *commit* (C_0) do repositório Git em um diretório temporário. Nesse diretório deve existir o arquivo `pom.xml` (arquivo de configuração do Maven), e então, (ii) o projeto é compilado pelo Maven. Caso o diretório não contenha um arquivo `pom.xml`, a compilação não será realizada e o *commit* é excluído da amostra, não sendo considerado para análise; (iii) o procedimento seguirá para o próximo *commit* (C_1), se esse puder ser compilado; (iv) o RepositoryMiner detecta os *code smells* e logo após a ferramenta de detecção de padrão de projeto é executada; (v) depois que a detecção é concluída, os resultados são armazenados em arquivos padronizados; (vi) em seguida o processo de detecção se repete analisando as revisões subsequentes $\{C_1, C_2, ..., C_n\}$ até atingir a revisão final (C_n); (vii) quando todos os *commits* forem examinados, existirá um arquivo para cada um deles, contendo os *smells* e os padrões de projeto que foram encontrados; (viii) imediatamente, cada par de *commits* consecutivos (C_{i-1} e C_i) são comparados quanto às diferenças nas instâncias de *smells* e padrões, permitindo, assim, determinar o instante que eles são introduzidos no código-fonte; (ix) os resultados dessa análise são armazenados em dois arquivo padronizados que contêm todas as adições e remoções de *smells* e padrões de projeto de cada *snapshot*. Com esses arquivos é possível cruzar os dados dos padrões de projeto com os dados dos *code smells* de cada revisão.

Com o intuito de obter os dados de padrões de projeto, houve a necessidade de compilar cada revisão através do Maven (veja Seção 5.3.2). Consequentemente, as revisões que apresentaram erros de compilação foram excluídas do processo de análise.

Para identificar as coocorrências ocorridas ao longo da evolução de um sistema de software, é necessário extrair as mudanças ocorridas entre os *commits* adjacentes. Para isso, é necessário comparar cada *commit* examinado com seu *commit* pai do histórico de versões. Assim, para evitar a análise duplicada de coocorrência, foram desconsiderados os *commits* de *merge*. *Commit* de *merge* representa apenas a mesclagem de duas ou mais ramificações (GITHUB, 2020), logo, ele possui pelo menos dois pais, e a sua comparação com todos os *commits* adjacentes (pais) resultaria na detecção duplicada de coocorrência. Além disso, foram excluídos também mais 913 revisões do processo de análise devido a não terem um *commit* pai.

Nessa fase foram manipulados um total de 20.082 *commits*; desses, foram compilados com sucesso 12.306 (61,28%) já considerando a exclusão dos *commits* de *merge*. Na sequência, mais 913 (4,55%) revisões foram descartadas por não possuírem o *commit* pai. Sendo assim, houve um descarte total de 8.689 (43,27%) *commits* da análise. Em geral, o conjunto final de dados de todos os repositórios possui um total de 11.393 revisões, o que representa (56,73%) dos *commits* totais manipulados pela ferramenta.

É importante destacar que as ocorrências de classes de testes foram excluídas automaticamente da análise com o auxílio da ferramenta de detecção de padrões de projeto, pois a ferramenta não pesquisa a presença de padrões neste tipo de classe (TSANTALIS et al., 2018). Além disso, para este experimento, assim como no E1, utilizamos apenas os *smells* de método: *Brain Method, Conditional Complexity, Feature Envy* e *Long Method* detectados pelo *framework RepositoryMiner*. Decidimos incluir os *smells Long Method* e *Conditional Complexity* (MENDES et al., 2017) ao estudo E2 devido (i) à facilidade de inclusão através do *framework RepositoryMiner*, e (ii) por serem os *smells* que mais surgiram simultaneamente com padrões de projeto.

5.3.4 Atividade 04: Processamento e Análise

Essa atividade seguiu as mesmas etapas do estudo anterior: (i) coleta e cruzamento dos dados, (ii) construção de tabelas de contingência e (iii) visualização por meio de gráficos de mosaico e aplicação de testes de hipótese.

Como explicado na Seção 5.3.3 (anterior), utilizamos a aplicação desenvolvida para gerar o conjunto de dados padronizados (etapa I). Após coletar os dados, criou-se a tabela de contingência (etapa II) com o cruzamento de todos os dados das entidades. Em seguida, verificou-se se existe uma associação estatisticamente significativa entre as duas variáveis e aplicaram-se os testes de hipótese. Finalmente, foi utilizada a tabela de contingência para visualizar os dados através de um gráfico de mosaico (etapa III).

No geral, a extração de dados para responder à QP2 levou aproximadamente 15 semanas em uma máquina Windows 64 bits Intel(R) Core(TM) i7-5500U CPU 2.40GHz, 2 Núcleos, 4 Processadores Lógicos e 8 Gb de RAM.

5.4 ANÁLISE E RESULTADOS

Esta seção relata os resultados com o objetivo de responder à questão de pesquisa QP2. A partir do cruzamento dos dados de todos os sistemas e análise de conteúdo, foram considerados 11.393 *commits*, contendo 6.323 classes, 966.236 linhas de código, 3.431 métodos de padrões de projeto adicionados e 11.130 *code smells* introduzidos ao longo da evolução dos sistemas.

5.4.1 QP2: Quais padrões, quando adicionados em projetos, possuem maior propensão a apresentar algum code smell?

Para responder a essa questão de pesquisa, foi gerada uma relação de contingência mostrada na Tabela 5.4. Essa tabela detalha quantitativamente como a introdução de métodos de padrões de projeto é afetada pela manifestação de *code smells*.

A primeira coluna da tabela indica o nome dos padrões de projeto que foram adicionados durante a evolução dos sistemas. A segunda aponta quantas vezes o padrão foi introduzido sem ser acompanhado de *smell*. A terceira coluna apresenta a quantidade de métodos de padrões inseridos acompanhados de *code smell*. A quarta indica o total de métodos de padrões adicionados ao longo do tempo. Em cada coluna também é mostrado o percentual de ocorrências em que cada padrão de projeto nasce com ou sem *smell*. A última linha da tabela exibe o total de métodos de padrões introduzidos com e sem *smell*.

Tabela 5.4 Tabela de contingência que relaciona a introdução de padrão com a introdução de *smells*

	CODE SMELLS		
PADRÕES	AUSENTE	PRESENTE	total
Adapter	1.127 (99,38%)	7 (0,62%)	1.134 (100,0%)
Bridge	46 (100,0%)	0 (0,00%)	46 (100,0%)
Composite	27 (100,0%)	0 (0,00%)	27 (100,0%)
Decorator	231 (99,57%)	1 (0,43%)	232 (100,0%)
Factory Method	290 (100,0%)	0 (0,00%)	290 (100,0%)
Observer	15 (100,0%)	0 (0,00%)	15 (100,0%)
Prototype	14 (100,0%)	0 (0,00%)	14 (100,0%)
Proxy	165 (100,0%)	0 (0,00%)	165 (100,0%)
State	556 (98,23%)	10 (1,77%)	566 (100,0%)
Strategy	23 (100,0%)	0 (0,00%)	23 (100,0%)
Template Method	888 (96,63%)	31 (3,37%)	919 (100,0%)
Total	3.382 (98,57%)	49 (1,43%)	3.431 (100,0%)

Finalmente, a partir da Tabela 5.4, foram extraídos seus valores para produzir o gráfico representado na Figura 5.1. Utilizou-se gráfico de mosaico para identificar onde as frequências observadas se desviam das frequências esperadas se as variáveis forem independentes. Cada coluna do gráfico indica um padrão e se subdivide na presença ou ausência de *smells*. A largura de cada barra é proporcional ao tamanho da amostra de cada padrão. A altura das barras representa as proporções que indicam a presença ou ausência de *smells*.

É importante esclarecer que o número de métodos de padrões introduzidos concorrentemente com *smell* em cada revisão é relativamente pequeno, conforme podemos inferir observando a terceira coluna (PRESENTE) da Tabela 5.4, impossibilitando a extração de dados. Para ampliar o conjunto de dados da amostra, optou-se por agrupar todas as instâncias identificadas dos sistemas em uma única análise.

Observa-se no gráfico que os métodos dos padrões *Bridge*, *Composite*, *Factory Method*, *Observer*, *Prototype*, *Proxy* e *Strategy* não tiveram seus métodos criados concorrentemente com *code smells*. Os padrões *Decorator* e *State* apresentaram baixa frequência com ocorrência de *code smell*.

Os métodos do padrão de projeto *Adapter* apresentaram uma propensão muito menor do que o esperado de surgir com *code smell*. Por outro lado, os métodos do padrão *Template Method* foram os que apresentaram maior propensão de surgir juntamente com algum dos *code smells* analisados neste estudo. Para esses dados não foi possível realizar o teste do qui-quadrado, visto que não se obteve o valor mínimo esperado igual ou maior que 5, em pelo menos 80% das células da tabela.

Analisando os dados do cruzamento, é percebido que as instâncias dos padrões *State* e *Template Method* surgiram principalmente com os *code smells Brain Method* (20 ocorrências), *Conditional Complexity* (29 ocorrências) e *Long Method* (27 ocorrências). É interessante apontar que a maioria dos métodos identificados com esses *smells* possuem simultaneamente os três *code smells*, e vale destacar também que 85,71% das coocorrências identificadas persistiram no código-fonte até a última versão examinada por este estudo.

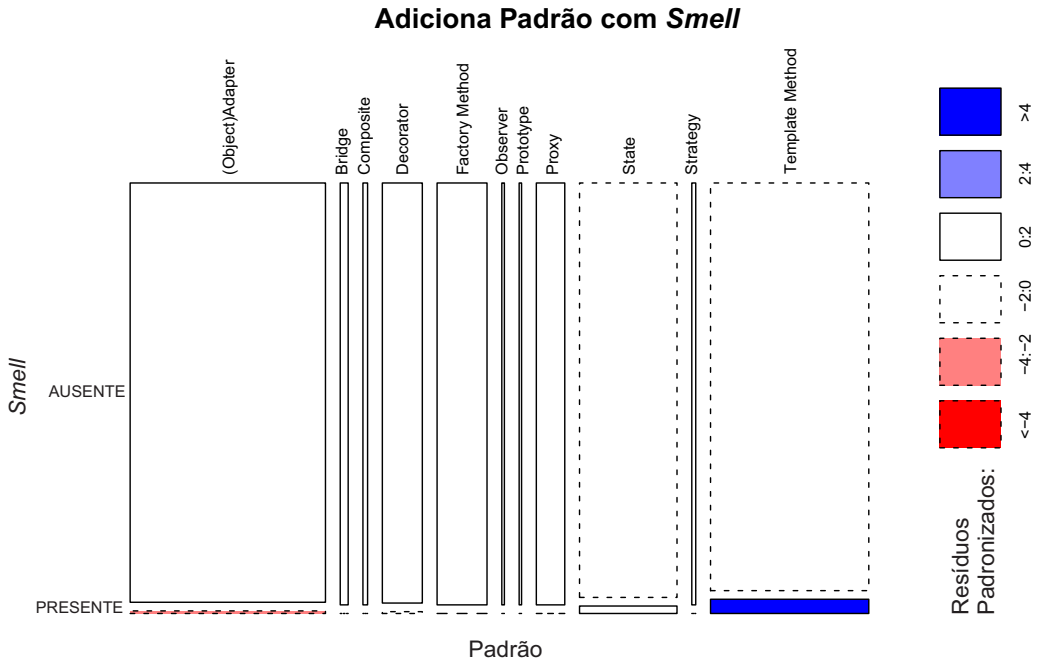


Figura 5.1 Propensão a ocorrência simultânea de Padrões x *Smells*

Como forma de conclusão para a QP2, os resultados evidenciaram que os métodos do padrão *Template Method* estão mais propensos a surgir com *code smells* do que outros métodos de padrões. Além disso, os métodos do padrão *Adapter* apresentaram uma propensão menor do que o esperado de surgir com *code smell*.

5.5 DISCUSSÃO

A fim de entender algumas possíveis causas para o surgimento simultâneo desses elementos, foi selecionado aleatoriamente um método com coocorrência entre o padrão *Template Method* e o *smell Long Method*, e então foi realizada a leitura do seu código-fonte. O primeiro está associado a um bom *design* de software e o segundo a um possível problema. Uma dessas coocorrências encontra-se no sistema *FastJSON*, no método *alibaba.fastjson.parser.JSONLexer.scanDecimal*. A escolha desse método para inspeção manual seguiu o mesmo critério dos métodos avaliados no E1, portanto sua escolha se deu de forma aleatória.

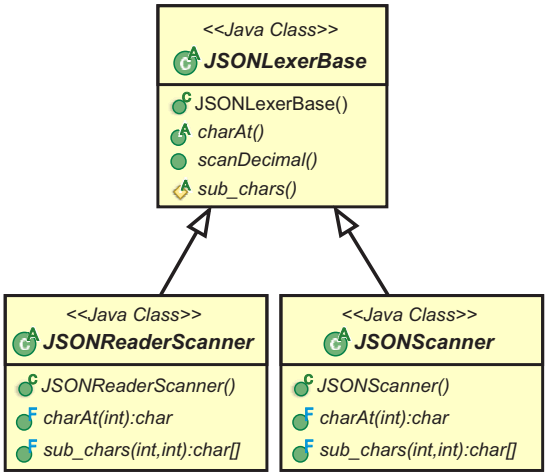


Figura 5.2 Fragmento do diagrama de classe do *Template Method* do sistema FastJSON

A Figura 5.2 mostra um fragmento do diagrama do padrão identificado no sistema *FastJSON*. O diagrama contém a classe *JSONLexerBase*, que encapsula o método *scanDecimal()*. Ele foi identificado como sendo o método *template* do padrão *Template Method*. A classe também define os métodos abstratos *charAt()* e *sub_chars()*. As subclasses *JSONReaderScanner* e *JSONScanner* implementam os métodos abstratos, que são chamados no método *scanDecimal()* da superclasse *JSONLexerBase*.

No trecho exibido em referência ao código-fonte na Figura 5.3, pode-se perceber que o método *scanDecimal* apresenta muitas linhas de código. Possivelmente esse foi um dos motivos para que fosse detectado o *code smell Long Method*. Ao analisar a estrutura do padrão e a formação do método *template* com o *smell Long Method* não foi possível

```
2971 public BigDecimal scanDecimal(char seperator) {
2972         matchStat = UNKNOWN;
2973 
2974         int offset = 0;
2975         char chLocal = charAt(bp + (offset++));
2976         final boolean quote = chLocal == '"';
2977         if (quote) {
2978             chLocal = charAt(bp + (offset++));
2979         }
2980 
2981         boolean negative = chLocal == '-';
2982         if (negative) {
2983             chLocal = charAt(bp + (offset++));
2984         }
2985 
2986         BigDecimal value;
2987         if (chLocal >= '0' && chLocal <= '9') {
2988             for (;;) {
2989                 chLocal = charAt(bp + (offset++));
2990                 if (chLocal >= '0' && chLocal <= '9') {
2991                     continue;
2992                 } else {
2993                     break;
2994                 }
2995             }
2996 
2997             boolean small = (chLocal == '.');
2998             if (small) {
2999                 chLocal = charAt(bp + (offset++));
3000                 if (chLocal >= '0' && chLocal <= '9') {
3001                     for (;;) {
3002                         chLocal = charAt(bp + (offset++));
3003                         if (chLocal >= '0' && chLocal <= '9') {
3004                             continue;
3005                         } else {
3006                             break;
3007                         }
3008                     }
3009                 } else {
3010                     matchStat = NOT_MATCH;
3011                     return null;
3012                 }
3013             }
```

Figura 5.3 Fragmento do código do método *scanDecimal* do sistema FastJSON

apontar uma relação causal entre os dois aspectos, contudo, outros fatores podem estar envolvidos na formação de tal coocorrência.

Ao investigar a lista de desenvolvedores que contribuíram com adições e alterações na classe, percebe-se que o trabalho é centralizado em uma única pessoa. Embora ocorram algumas contribuições de demais desenvolvedores, essas modificações são esporádicas e diluídas ao longo da evolução do projeto.

Outro ponto a ser considerado pode ser a frequência de modificações realizadas sobre os métodos. Considerando a classe *alibaba.fastjson.parser.JSONLexerBase*, as atividades do desenvolvedor sobre ela e seus métodos se concentram em um período específico no tempo. A Tabela 5.5 contém um resumo do histórico de atualizações, segundo o *GitHub*.

É perceptível que grande parte das atualizações sobre a classe e seus métodos ocorreram entre 2016 e 2019. Durante os demais anos a atividade foi menor. Observou-se que a coocorrência foi introduzida em uma atualização realizada sobre a classe em 19 de agosto de 2017. Nessa atualização, um total de 499 linhas foram modificadas ao longo dos seus métodos e outros 9 artefatos de código foram afetados pelas modificações. Provavelmente as coocorrências (como as da referida classe) podem ser ocasionadas pelo único desenvolvedor trabalhando sobre ela não ter como empregar boas estratégias de codificação durante períodos de muita atividade. Ou seja, existe uma possibilidade de o desenvolvedor não ter tido tempo suficiente para implementar padrões e, ao mesmo tempo, fazer uso de boas

Tabela 5.5 Histórico da classe *alibaba.fastjson.parser.JSONLexerBase*

Ano	Número de *Commits*
2020	4
2019	15
2018	10
2017	21
2016	24
2015	2
2014	10
2013	3

práticas de codificação que pudessem evitar *code smells*.

De acordo com o histórico de modificações da classe exibida pelo *GitHub*, durante a maior parte de 2014 (quando a classe foi criada) e anos posteriores, um dos desenvolvedores do projeto foi o único a contribuir com o histórico de evolução regularmente. É possível que, ao longo do tempo, o desenvolvedor tenha se tornado um especialista nas funcionalidades fornecidas pelos métodos da classe. Provavelmente, isso acabou ocultando dos demais envolvidos no projeto qualquer tipo de problema ocasionado pela presença do *smell Long Method*. Também pode ser apontado que, talvez, a possibilidade de um desenvolvedor se concentrar por tanto tempo em uma parte do código fez com que ele se tornasse bastante eficaz em evoluir o código sem que seja afetado ou perceba os efeitos ruins do *smell*. Talvez os engenheiros de software tenham que levar em conta o quanto que possíveis problemas em código são isolados de um grupo de desenvolvedores porque um deles se tornou especialista em sua manutenção e evolução.

Para o contexto estudado, não foi possível determinar que a implementação do padrão *Template Method* tenha ocasionado o surgimento do *smell Long Method*, ou seja, não foi possível apontar uma relação de causa-efeito entre as duas variáveis, e esse parece ser o caso dos diversos métodos presentes na amostra. A causa mais provável observada é por falta de modularização do código, dividindo-o em partes menores, tais como, por exemplo, em outros métodos.

Com relação à refatoração, um possível ponto de partida para remoção dos *smells* seria considerar a divisão lógica do método em blocos mediante a inserção de espaços em branco. Por exemplo, o trecho entre as linhas 2974 e 2979 da Figura 5.3 parece ser uma unidade lógica independente do outro trecho entre as linhas 2981 e 2984. De acordo com Wang, Pollock e Vijay-Shanker (2011), a inserção de linhas em branco é um recurso comum utilizado por desenvolvedores para dividir o código-fonte de sistemas em segmentos relacionados logicamente. Isso pode promover a leitura facilitada das linhas de um programa (SCHACH, 2007) e pode trazer benefícios à sua compreensão tanto quanto comentários (BUSE; WEIMER, 2008).

Assim, a refatoração pode se basear na ocorrência das linhas em branco. Deste modo, é possível propor a seguinte recomendação a ser seguida pelos engenheiros de software: (i) identificar métodos de padrões, tais como o *Template Method*; (ii) verificar se neles coocorrem *smells* associados aos métodos (e.g., *Brain Method, Conditional Complexity* e *Long Method*); (iii) verificar se os métodos, por encapsularem os *smells*, sofrem de

problemas durante a evolução (muitos *bugs* e número excessivo de modificações); e (iv) avaliar se é possível particionar os métodos mediante o critérios de divisão, e.g., espaços em branco.

Os resultados enumerados na Tabela 5.4 evidenciam, de uma forma geral, baixa ocorrência dos padrões de projeto GoF que surgiram simultaneamente com os *code smells* presentes na amostra. Com exceção de 1,43% dos métodos de padrões de projeto que foram inseridos simultaneamente com *smells*. A relação de contingência apontada entre o padrão *Template Method* e os *smells*, embora significativa, pode não representar um resultado prático. Como consequência, talvez não seja interessante para os desenvolvedores e engenheiros de software dedicarem tempo e esforço na busca de soluções para identificar *code smells* na implementação inicial de padrões de projeto. Todavia, cabem mais estudos futuros, de caráter qualitativo, para esclarecer as suposições levantadas.

5.6 AMEAÇAS À VALIDADE

Ameaças à validade externa dizem respeito à generalização dos resultados. Neste trabalho, foi conduzido um estudo experimental em que selecionamos cinco sistemas *open source* desenvolvidos na linguagem Java. Embora os sistemas sejam de diferentes domínios, tamanhos e desenvolvedores, é possível que os resultados não possam ser extrapolados para outros projetos, pois é necessário um aprofundamento no campo de pesquisa e analisar outros cenários que podem influenciar diretamente na interpretação dos resultados, como sistemas comerciais, sistemas desenvolvidos em outras linguagens de programação como também em outros paradigmas. Assim, é necessário replicar o estudo ou aplicá-lo em outros contextos com as mesmas características do estudo de caso original.

Uma ameaça à validade de construto deste trabalho diz respeito às imprecisões e erros nas medições realizadas. A principal ameaça reside na detecção automática de padrões de projeto e *code smells* no *design* dos sistemas de software, em razão de que os resultados podem ser afetados pelos valores limiares empregados, como também, a presença de falsos-positivos e falsos-negativos. Para minimizar esse efeito, procurou-se selecionar ferramentas que já foram aplicadas e avaliadas em outros estudos sobre padrões de projeto ou *code smells*. Dessa forma, é possível que os resultados da detecção sejam mais confiáveis e assertivos, permitindo reduzir os casos de falsos-positivos e falsos-negativos.

Em termos de ameaças interna, a ferramenta de detecção de padrões de projeto, utilizada no estudo, emprega uma estrutura de manipulação de *bytecode Java*, que permite a obtenção de informações que ajudam a compreender a estrutura estática do sistema. No entanto, houve *commits* que não foram compilados e consequentemente foram excluídos do processo de análise. Isso poderia afetar os resultados, pois se muitos *commits* forem excluídos, a diferença entre dois *commits* compiláveis adjacentes pode ser considerável, o que torna mais provável encontrar associações padrão-*smell*, ainda que eles não tenham sido introduzidos simultaneamente. Essa ameaça foi eliminada comparando apenas os *commits* adjacentes nos quais ambos puderam ser compilados. Todavia, é possível que, com a exclusão de alguns *commits*, tenhamos deixado de identificar possíveis associações. No entanto, isso limita o possível impacto dessa ameaça.

Outra ameaça interna se refere ao conjunto de dados analisados decorrentes do estado

inicial em que os sistemas foram incorporados no Git. Por esse motivo, a análise se deu a partir do segundo *commit* de cada projeto, pois o primeiro *commits* são importações dos sistemas já iniciados em outras plataformas, o que torna provável, como observado em uma inspeção manual, a existência de associações padrão-*smell*. Portanto, para mitigar possíveis inconsistências, não foram consideradas na análise as associações identificadas no primeiro *commit*. Além disso, o processo de análise se expande por todo o histórico do sistema de controle de versão, ou seja, por todas as ramificações dos projetos, uma vez que pretende-se identificar o método em que o *smell* aparece no *commit* que introduz o padrão. Essa estratégia tem como base o estudo de Kovalenko, Palomba e Bacchelli (2018), que mostraram que a inserção das *branches* (ramificações) na análise do histórico de versões é indispensável para um rastreamento preciso das contribuições no Git. Portanto, levou-se em consideração as ramificações para capturar corretamente as alterações que são incorporadas na *branch* principal dos sistemas.

Outra ameaça interna importante está relacionada ao fato de que a introdução de *code smells* em métodos de padrões de projeto pode ou não ser influenciado pela aplicação de padrões de projeto em si, ou seja, outros fatores poderiam ter motivado a introdução dos *smells*; por exemplo, desenvolvedores sob pressão de cronogramas de entrega, sua experiência e carga de trabalho. Portanto, os resultados deste estudo não permitem confirmar uma relação direta de causa-efeito entre a implementação de padrões de projeto e a introdução de *code smells*. A investigação desses fatores é proposta como trabalho futuro.

5.7 CONSIDERAÇÕES FINAIS

Este capítulo apresentou um estudo sobre o histórico de evolução de cinco sistemas Java de código-fonte aberto. Foi realizado com o objetivo de verificar se métodos específicos de padrões de projeto já nascem com algum *smell* ou se os *smells* são introduzidos à medida que o código evolui. Realizamos inspeções manuais no código-fonte sobre as associações mais significativas, a fim de entender as possíveis causas das coocorrências.

Os resultados encontrados sugerem que o padrão *Template Method* está mais propenso a ser introduzido simultaneamente com *code smells* do que outros padrões de projeto. Uma análise manual mais detalhada revelou que ele surgiu de forma já associada com os *smells Brain Method, Conditional Complexity* e *Long Method*. Esses resultados, mostraram, de maneira geral, baixa ocorrência dos padrões GoF surgindo simultaneamente com os *code smells* presentes na amostra. Apenas de 1,43% dos métodos de padrões de projeto surgiram concorrentemente com *smells*. Isso é pouco, mas demonstra que mesmo na implementação inicial de padrões de projeto, visando a qualidade do software, já pode ocorrer a introdução de *code smells*.

Foi realizada uma inspeção manual no código-fonte a fim de identificar algumas possíveis causas do relacionamento entre o padrão *Template Method* e o *smell Long Method*. No entanto, não foi possível apontar uma relação de causa-efeito entre os objetos estudados. A análise considerou como a causa mais provável a falta de modularização do código-fonte por parte dos desenvolvedores dos sistemas.

De uma forma geral, os resultados descobertos sugerem uma baixa ocorrência dos

padrões de projeto GoF surgirem simultaneamente com os *code smells* analisados no estudo. Aparentemente, considerando os resultados observados, não haveria uma necessidade, por parte dos desenvolvedores e engenheiros de software dos projetos analisados, de se dedicarem a evitar a incidência de *code smells* na aplicação inicial de métodos de padrões de projeto. Contudo, salienta-se ainda que existe a necessidade de mais estudos que busquem replicar os resultados a fim de fortalecer ou validar as afirmações aqui realizadas. Também é importante que essa análise seja replicada com uma massa maior de projetos com o intuito de generalizar os resultados e ampliar o conhecimento sobre os casos de coocorrência.

Capítulo

6

Neste capítulo de conclusão é apresentado de maneira sintética, os principais resultados alcançados neste trabalho, que dizem respeito à relação entre code smells e padrões de projeto.

CONCLUSÃO

Este trabalho apresentou dois estudos experimentais com o objetivo de identificar as relações de coocorrência entre padrões de projeto *GoF* definidos por Gamma et al. (1995) e os *code smells* descritos por Lanza e Marinescu (2007). O propósito foi investigar se métodos que fazem parte da implementação de padrões de projeto coocorrem com *code smells*. A análise foi realizada em métodos individuais de padrões de projeto, pois pretendia-se determinar se a implementação desses métodos está relacionada à incidência de *code smells*.

A metodologia foi aplicada em dois estudos experimentais, que conduziram etapas equivalentes, visto que os objetivos finais são os mesmos para ambos os casos: entender a relação entre as duas variáveis. Utilizou-se ferramentas automatizadas para detecção de padrões de projeto e *code smells* em projetos Java *open source*. O estudo experimental E1 (descrito no capítulo 4) analisou uma versão específica de 25 projetos do *Qualitas.class Corpus*. A meta era identificar as associações mais significativas e realizar uma inspeção manual no código-fonte, a fim de entender algumas possíveis causas da coocorrência. O segundo estudo experimental E2 (discutido no capítulo 5) foi aplicado sobre cada *commit* proveniente de repositórios *open source* do *GitHub* de cinco sistemas de software livre. E2 tinha como objetivo entender a relação temporal entre as duas variáveis (*padrão de projeto* e *code smells*) ao longo da evolução do software.

Os resultados encontrados em E1 apontam que métodos que participam da implementação de padrões de projeto estão mais propensos a conter *code smells* do que outros. Todavia, o nível de significância estatística das relações varia entre os padrões de projeto presentes na amostra. Além disso, com a inspeção manual, foi possível identificar situações em que a implementação de padrões de projeto levaram a ferramenta de detecção de *code smells* a identificar falsos-positivos. O estudo tornou possível propor recomendações para os fornecedores de ferramentas de detecção de *code smells*. Por exemplo, reduzir a detecção de falsos-positivos em determinadas estruturas de padrões de projeto. Foi identificado também que o padrão *Adapter* e o *State* são os que mais contribuíram para a

associação. Por meio das estatísticas levantadas, nota-se que os pares de padrões-*smells* mais propensos a acontecerem são: *Template Method-Shotgun Surgery*, *Adapter-Feature Envy*, e *Factory Method-Shotgun Surgery*.

O segundo experimento, E2, foi conduzido sobre o histórico de versões de cinco sistemas. Ele teve como objetivo identificar se a criação de um determinado padrão de projeto leva ao surgimento concorrente com *code smell* ao longo da evolução de sistemas. Os resultados apontam que o padrão *Template Method* apresenta uma propensão maior de ser introduzido concorrentemente com *code smells* do que outros padrões. Foi realizada uma inspeção manual dos casos analisados a fim de compreender algumas possíveis causas para a coocorrência entre o par *Template Method-Long Method*. Contudo, não foi possível apontar uma relação causal entre a relação. Observou-se que, de modo geral, os resultados evidenciam baixa ocorrência dos padrões de projeto GoF coocorrendo simultaneamente com os *code smells*.

Sabe-se muito pouco sobre a formação de *code smells* em padrões de projeto e os resultados apresentados nesta dissertação revelaram novos achados relacionados à incidência de *smells* na aplicação de padrões de projeto. Almeja-se que este trabalho possa inspirar outros pesquisadores a realizarem análises mais detalhadas contribuindo para a expansão do conhecimento acerca do tema. Espera-se que os resultados deste trabalho também possam: (i) contribuir com o catálogo proposto por Fontana et al. (2016), (ii) ajudar os fornecedores de ferramentas de detecção a diminuírem a taxa de casos de falsos-positivos, (iii) apoiar os desenvolvedores a atuarem eficientemente na aplicação de padrões para melhorar a qualidade de seus projetos de software e (iv) contribuir com o desenvolvimento de uma nova geração de ferramentas de detecção de *code smells* que passem a identificar as instâncias desse tipo de ocorrência, levando em consideração a implementação de padrões de projetos.

Com relação a trabalhos futuros, pretende-se: (i) realizar uma investigação de caráter qualitativo sobre os fatores que influenciam a incidência de *code smells* em métodos de padrões de projeto, (ii) reproduzir o estudo com outros sistemas, (iii) ampliar a quantidade de *smells* da amostra e (iv) incluir a avaliação de sistemas de soluções comerciais. Considera-se que um aumento de amostras contribuiria para aumentar a confiança dos resultados e possibilitaria promover a generalização das conclusões.

REFERÊNCIAS BIBLIOGRÁFICAS

ABBES, M.; KHOMH, F.; GUEHENEUC, Y.-G.; ANTONIOL, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: *15th European conference on Software maintenance and reengineering (CSMR)*. [S.l.: s.n.], 2011. p. 181–190.

AL-OBEIDALLAH, M. G.; PETRIDIS, M.; KAPETANAKIS, S. A survey on design pattern detection approaches. *International Journal of Software Engineering (IJSE)*, v. 7, n. 3, p. 41–59, 2016.

ALFADEL, M.; ALJASSER, K.; ALSHAYEB, M. Empirical study of the relationship between design patterns and code smells. *Plos one*, v. 15, n. 4, p. e0231731, 2020.

ASSUNÇÃO, E.; SOUZA, R. Incidence of code smells in the application of design patterns: a method-level analysis. In: *13th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*. [S.l.: s.n.], 2019. p. 73–82.

BROWN, W. H.; MALVEAU, R. C.; MCCORMICK, H. W.; MOWBRAY, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis*. [S.l.]: John Wiley & Sons, Inc., 1998.

BUSE, R. P.; WEIMER, W. R. A metric for software readability. In: *17th International Symposium on Software Testing and Analysis (ISSTA)*. [S.l.: s.n.], 2008. p. 121–130.

CARDOSO, B.; FIGUEIREDO, E. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In: *11th Simpósio Brasileiro de Sistemas de Informação (SBSI)*. [S.l.: s.n.], 2015. p. 347–354.

CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, v. 10, n. 1, p. 3–18, 2014.

DEURSEN, A. V.; MOONEN, L.; BERGH, A. V. D.; KOK, G. Refactoring test code. In: *2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. [S.l.: s.n.], 2001. p. 92–95.

DIAS, R. S.; NETO, P. de Alcântara dos S.; IBIAPINA, I. M. de S.; AVELINO, G. A.; CASTRO, O. C. da C. Effects of visualizing technical debts on a software maintenance project. In: *18th Brazilian Symposium on Software Quality (SBQS)*. [S.l.: s.n.], 2019. p. 39–48.

FONTANA, F. A.; DIETRICH, J.; WALTER, B.; YAMASHITA, A.; ZANONI, M. Antipattern and code smell false positives: Preliminary conceptualization and classification. In: *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. p. 609–613.

FONTANA, F. A.; FERME, V.; MARINO, A.; WALTER, B.; MARTENKA, P. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In: *29th International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2013. p. 260–269.

FOUNDATION, A. S. *Apache Maven Project*. 2002. ⟨https://maven.apache.org/⟩. Acessado em 19/09/2020.

FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design patterns: Elements of reusable object-oriented software addison-wesley. *Reading, MA*, v. 99, p. 1995, 1995.

GITHUB. *Github - Where the world builds software*. 2020. ⟨https://github.com/⟩. Acessado em 25/07/2020.

GOMES, F. G. de S.; MENDES, T. S.; SPÍNOLA, R. O.; MENDONÇA, M.; FARIAS, M. Uma análise da relação entre code smells e dívida técnica auto-admitida. In: *7th Workshop on Software Visualization, Evolution and Maintenance (VEM)*. [S.l.: s.n.], 2019. p. 37–44.

GRADY, R. B. *Practical software metrics for project management and process improvement*. [S.l.]: Prentice-Hall, Inc., 1992.

IBIAPINA, I.; CASTRO, O.; MOURA, V.; DIAS, R.; NETO, P. S. Tdvision: Um módulo computacional para visualização de dı vidas técnicas. In: *6th Escola Regional de Informática do Piauí (ERIPI)*. [S.l.: s.n.], 2018. p. 103–108.

JAAFAR, F.; GUÉHÉNEUC, Y.-G.; HAMEL, S.; KHOMH, F. Mining the relationship between anti-patterns dependencies and fault-proneness. In: *20th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2013. p. 351–360.

JAAFAR, F.; GUÉHÉNEUC, Y.-G.; HAMEL, S. et al. Analysing anti-patterns static relationships with design patterns. *Electronic Communications of the EASST*, v. 59, p. 1–26, 2014.

KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, v. 17, n. 3, p. 243–275, 2012.

KHOMYAKOV, I.; MAKHMUTOV, Z.; MIRGALIMOVA, R.; SILLITTI, A. An analysis of automated technical debt measurement. In: *21st International Conference on Enterprise Information Systems (ICEIS)*. [S.l.: s.n.], 2019. p. 250–273.

KOVALENKO, V.; PALOMBA, F.; BACCHELLI, A. Mining file histories: should we consider branches? In: *33rd International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2018. p. 202–213.

LANZA, M.; MARINESCU, R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. [S.l.]: Springer Science & Business Media, 2007.

MARINESCU, R. Detection strategies: Metrics-based rules for detecting design flaws. In: *20th International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2004. p. 350–359.

MCCONNELL, S. Real quality for real engineers. *IEEE software*, v. 19, n. 2, p. 5–7, 2002.

MENDES, T.; NOVAIS, R.; MENDONCA, M.; CARVALHO, L.; GOMES, F. *RepositoryMiner-uma ferramenta extensível de mineração de repositórios de software para identificação automática de Dívida Técnica*. [S.l.]: CBSoft, 2017.

MENDES, T. S.; GOMES, F. G.; GONÇALVES, D. P.; MENDONÇA, M. G.; NOVAIS, R. L.; SPÍNOLA, R. O. Visminertd: a tool for automatic identification and interactive monitoring of the evolution of technical debt items. *Journal of the Brazilian Computer Society*, v. 25, n. 1, p. 2, 2019.

MENDES, T. S.; GONÇALVES, D. P.; GOMES, F. G.; NOVAIS, R.; SPINOLA, R. O.; MENDONÇA, M.; SALVADOR, B. Visminertd: Uma ferramenta para identificação automática e monitoramento interativo de dıvida técnica. 2015.

OIZUMI, W.; GARCIA, A.; SOUSA, L. da S.; CAFEO, B.; ZHAO, Y. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: *38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 440–451.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. Detecting bad smells in source code using change history information. In: *28th International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2013. p. 268–278.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, v. 41, n. 5, p. 462–489, 2014.

PRECHELT, L.; UNGER, B.; TICHY, W. F.; BROSSLER, P.; VOTTA, L. G. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE transactions on Software Engineering*, v. 27, n. 12, p. 1134–1144, 2001.

SCHACH, S. R. *Object-oriented and classical software engineering*. [S.l.]: McGraw-Hill New York, 2007.

SOMMERVILLE, I. *Engenharia de software 9. ed.* [S.l.]: Pearson Education Companion, 2011.

SOUSA, B. L.; BIGONHA, M. A.; FERREIRA, K. Evaluating co-occurrence of gof design patterns with god class and long method bad smells. In: *13th Simpósio Brasileiro de Sistemas de Informação (SBSI)*. [S.l.: s.n.], 2017. p. 396–403.

SOUSA, B. L.; BIGONHA, M. A.; FERREIRA, K. A. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. *Software: Practice and Experience*, v. 49, n. 7, p. 1079–1113, 2019.

SOUSA, B. L.; BIGONHA, M. A. S.; FERREIRA, K. A. M. A systematic literature mapping on the relationship between design patterns and bad smells. In: *33rd Symposium on Applied Computing (SAC)*. [S.l.: s.n.], 2018. p. 1528–1535.

SUBBURAJ, R.; JEKESE, G.; HWATA, C. Impact of object oriented design patterns on software development. *International Journal of Scientific & Engineering Research*, v. 6, n. 2, p. 961–967, 2015.

TEMPERO, E.; ANSLOW, C.; DIETRICH, J.; HAN, T.; LI, J.; LUMPE, M.; MELTON, H.; NOBLE, J. The qualitas corpus: A curated collection of Java code for empirical studies. In: *17th Asia Pacific Software Engineering Conference (APSEC)*. [S.l.: s.n.], 2010. p. 336–345.

TERRA, R.; MIRANDA, L. F.; VALENTE, M. T.; BIGONHA, R. S. Qualitas. class corpus: A compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, v. 38, n. 5, p. 1–4, 2013.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of type-checking bad smells. In: *12th European Conference on Software Maintenance and Reengineering (CSMR)*. [S.l.: s.n.], 2008. p. 329–331.

TSANTALIS, N.; CHATZIGEORGIOU, A.; STEPHANIDES, G.; HALKIDIS, S. T. Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, v. 32, n. 11, p. 896–909, 2006.

TSANTALIS, N.; CHATZIGEORGIOU, A.; STEPHANIDES, G.; HALKIDIS, S. T. *Design Pattern detection using Similarity Scoring*. 2018. ⟨https://users.encs.concordia.ca/~nikolaos/pattern_detection.html⟩. Acessado em 13/02/2021.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; PENTA, M. D.; LUCIA, A. D.; POSHYVANYK, D. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, v. 43, n. 11, p. 1063–1088, 2017.

VIDAL, S.; VAZQUEZ, H.; DIAZ-PACE, J. A.; MARCOS, C.; GARCIA, A.; OIZUMI, W. Jspirit: a flexible tool for the analysis of code smells. In: *34th International Conference of the Chilean Computer Science Society (SCCC)*. [S.l.: s.n.], 2015. p. 1–6.

WALTER, B.; ALKHAEIR, T. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, v. 74, p. 127–142, 2016.

WANG, X.; POLLOCK, L.; VIJAY-SHANKER, K. Automatic segmentation of method code into meaningful blocks to improve readability. In: *18th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2011. p. 35–44.

WANG, Y.; ZHANG, C.; WANG, F. What do we know about the tools of detecting design patterns? In: *6th International Conference on Progress in Informatics and Computing (PIC)*. [S.l.: s.n.], 2018. p. 379–387.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: *35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 682–691.