UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA
DA COMPUTAÇÃO

UMA ABORDAGEM PARA DETECÇÃO DE PADRÕES EMERGENTES

RICARDO DE SOUSA JOB

CAMPINA GRANDE – PB
2014

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma abordagem para detecção de padrões emergentes

Ricardo de Sousa Job

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Tiago Lima Massoni e Rohit Gheyi
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

J62a Job, Ricardo de Sousa.
Uma abordagem para detecção de padrões emergentes / Ricardo de Sousa Job. – Campina Grande, 2014.
78 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

"Orientação: Prof. Dr. Tiago Lima Massoni, Prof. Dr. Rohit Gheyi".
Referências.

1. Padrão de Projeto. 2. Detecção. 3. Padrões Emergentes. I. Massoni, Tiago Lima. II. Gheyi, Rohit. III. Título.

CDU 004.4 (043)

"UMA ABORDAGEM PARA DETECÇÃO DE PADRÕES DE PROJETO EMERGENTES"

RICARDO DE SOUSA JOB

DISSERTAÇÃO APROVADA EM 05/12/2014

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

ROHIT GHEYI, Dr., UFCG
Orientador(a)

DALTON DARIO SEREY GUERRERO, D.Sc, UFCG
Examinador(a)

SERGIO CASTELO BRANCO SOARES, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Padrões de Projeto são soluções consolidadas para problemas de projeto de software recorrentes. São utilizados amplamente em projetos de software orientados a objetos, tornando-se um artifício de comunicação de soluções conhecidas dentro das equipes de desenvolvimento. É importante que o projetista consiga detectar e identificar os padrões de projetos numa base de código, para compreender as relações entre as classes, como fornecer sugestões úteis para a compreensão e evolução do sistema. Para detecção automática de padrões existem basicamente duas técnicas: análise estática e dinâmica. No primeiro passo, as relações e colaborações estruturais são extraídas. Já no segundo passo monitora-se a execução do programa, rastreando o conjunto de padrões selecionados no passo anterior para identificar quais padrões comportam-se como o esperado. As técnicas de detecção atuais, no entanto, limitam-se a análises estruturais restritivas, omitindo casos em que o comportamento de um padrão está presente, mesmo que não siga a organização estrutural prevista na literatura. Chamamos de *padrões emergentes* estes casos em que o comportamento de um determinado padrão está presente, mesmo que certa região do código apresente estruturação livre. Por exemplo, a essência do padrão de projeto Singleton está presente em uma classe qualquer quando esta possui apenas uma única instância durante as execuções de um programa, mesmo que não haja restrição sintática para que isso seja possível; ou seja, o padrão Singleton *emerge* deste comportamento de um determinado elemento do programa. Ao auxiliar o desenvolvedor na detecção de situações de projeto como esta, pode-se enriquecer o seu conhecimento sobre as consequências de suas decisões, além de propiciar a estruturação explícita do padrão como conhecida, facilitando assim a documentação e comunicação do projeto. Este trabalho explora o conceito de padrões emergentes através das seguintes contribuições: (i) uma revisão sistemática sobre abordagens automáticas de detecção de padrões de projeto, (ii) conceitos de padrões emergentes para vários padrões de projeto bem conhecidos, (iii) uma proposta de abordagem semi-automática de detecção de padrões emergentes e (iv) sua utilização para uma análise de ferramentas de detecção existente acerca de sua capacidade de identificação de padrões emergentes em alguns projetos de código aberto Java.

Abstract

Design Patterns are consolidated solutions to recurring software design problems. They are widely used in object-oriented software design, as communication device of well known solutions within development teams. It is important that the software designer detects and identifies design patterns in a code base, to understand the relationships between classes, provide useful suggestions for the understanding and evolution of the system. For automatic detection of patterns there are basically two techniques: static and dynamic analysis. On the first step, relations and structural collaborations are extracted. In the second step, the program execution is monitored, tracking the selected set of patterns in the first step to identify which patterns behave as expected. However, the current detection techniques are limited to restrictive structural analysis, omitting cases where the behavior of a pattern is presente, even if not follow the structural organization provided in the literature. We call emerging patterns when the behavior of a given pattern is present, even if some code's region presents a free structure. For example, the essence of the Singleton design pattern is present in any given class when it has only a single instance during the execution of a program, even without syntactic restriction for this to be possible; that is, the Singleton pattern emerged from this program element behavior. When developers are assisted in detecting design situations like this, they can enhance their knowledge about the consequences of their decisions, as well as providing the explicit structure of the pattern, facilitating the documentation and communication of the project. This paper explores the concept of emerging patterns through the following contributions: (i) a systematic review of automatic detection approaches of design patterns, (ii) concepts of emerging patterns for several well-known design patterns, (iii) a proposal for semi-automatic detection approach of emerging patterns and (iv) its use for an analysis of existing detection tools about their ability to identify emerging patterns in an open-source Java project

A meu avô e avó, por sempre incentivarem minha educação.

A meu Pai pelo incentivo, mesmo sem sua presença física.

A minha Mãe pela confiança e amor eterno.

Agradecimentos

Primeiramente a Deus, criador do homem e detentor do conhecimento e força, sempre me auxiliando e dando condições de concluir esta jornada.

Aos meus familiares, por todo o apoio dado durante essa difícil caminhada e em especial minha mãe Tozinha, que não mediu esforços para que tudo isso se consolidasse e outro sonho se tornasse realidade.

A minha irmã e família, pelo apoio incondicional nos momentos mais difíceis.

A meu irmão, cabeça dura e de um coração enorme. Que mesmo com a distância nunca mediu esforços para me auxiliar nesta jornada.

A minha namorada Carla por todo amor, carinho e paciência nestes anos distantes.

Aos meus grandes e eternos amigos, em especial a Gastão e Jardel por todo apoio e ajuda nas constantes e incontáveis viagens.

Aos meu amigos com quem dividi apartamento, Diego e Reudismam.

Ao meus amigos do SPLab por todas as conversas discussões produtivas. E claro, pelas horas compartilhadas tomando um bom café.

Aos meu amigos professores do IFPB pelo apoio e sucessivas correções deste trabalho.

Aos professores que sempre disponibilizaram seu tempo e dedicação. Em especial a Tiago Massoni pelo exemplo tanto pessoal quanto profissional. Se hoje sou professor, devo muito ao aprendizado neste período.

As pessoas especiais e amigos, não mencionados, mas que moram em meu coração.

A CAPES por fomentar essa pesquisa e possibilitar minha ida a Campina Grande.

A todos, muito obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Os padrões de projeto, segundo referência seminal da área [18], são descrições de objetos comunicantes e classes customizadas para resolver problemas que repetem-se comumente nos mais variados tipos de projeto de software orientado a objetos. Portanto, os padrões são soluções consolidadas, recorrentes e que são aplicadas a um contexto específico para solucionar um determinado problema de projeto de software [28]. O uso de padrões de projeto no software pode contribuir para melhoria da qualidade. Por exemplo, reusabilidade é normalmente o foco da aplicação das soluções, os padrões por serem soluções recorrentes e já testadas em outros projetos, aumentam a confiança em sua aplicação.

Desde 1994, padrões têm sido utilizados amplamente em projetos de software orientados a objetos, tornando-se um artifício de comunicação de soluções conhecidas dentro das equipes de software. Por exemplo, um problema recorrente em projetos de software orientado a objetos é a necessidade de determinadas classes possuírem apenas um objeto ativo num dado momento, tal como uma variável global. O padrão *Singleton* [18], descrito na Figura 1.1, incorpora este comportamento esperado.

Como visto na Figura 1.1, o padrão *Singleton* possui, em sua estrutura, basicamente: (*i*) uma variável privada e estática do mesmo tipo da classe; (*ii*) um construtor privado, para não ser instanciado por outras classes e; (*iii*) um método público e estático, para dar acesso a esta variável, sendo ele responsável por controlar uma única instância do atributo para as demais classes. Neste exemplo, quando outras classes necessitarem de uma referência, todas irão possuir uma referência única a instância do objeto *Singleton*.

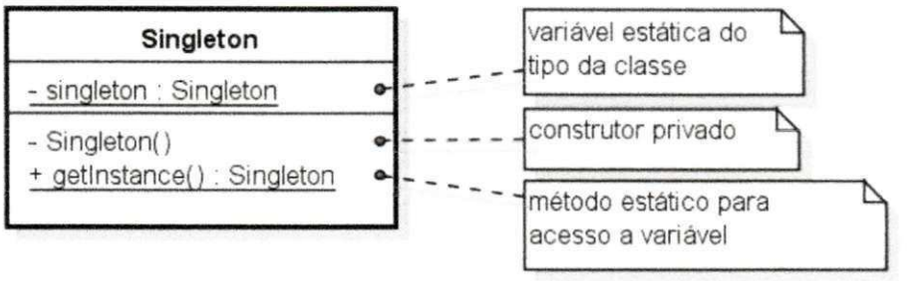

Figura 1.1: Padrão *Singleton*

1.1 Problema

Identificar padrões de projeto representa um significativo passo para compreender as relações entre as classes, prover uma forma de documentação e ajudar nas futuras modificações no sistema [10]. Além disso, detectar padrões é importante para: (*i*) fornecer sugestões úteis para a compreensão e evolução do sistema; (*ii*) prover informações sobre a lógica adotada durante o desenvolvimento do projeto e; (*iii*) auxiliar durante o processo de re-documentação, em particular quando a documentação é pobre, incompleta ou desatualizada [2].

É importante que o projetista consiga detectar e identificar os padrões de projeto em uma determinada base de código. Contudo, detectar os padrões numa base de código é uma atividade complexa, principalmente se essa atividade for realizada manualmente, pois é necessário identificar no código as relações e estruturas correspondentes aos padrões e relacionar essas estruturas a um modelo pré-definido. Por outro lado, para a detecção automática de padrões existem basicamente dois passos: análise estática e dinâmica.

Quando as técnicas de análise estática e dinâmica são aplicadas isoladamente apresentam desvantagens. Utilizando apenas a análise estática seria necessário identificar os estados e tipos dos objetos, mas estas informações são conhecidas apenas em tempo de execução. Por exemplo, um método com um parâmetro do tipo `Object`, só pode ter o tipo real conhecido durante a execução do método, com isso algumas instâncias de padrões podem ser classificadas equivocadamente. Na detecção de padrões de projeto, quando analisada estaticamente a classe `Singleton`, as informações de sua estrutura são extraídas: construtor privado, método estático e um atributo do mesmo tipo da classe. Essas informações extraídas são comparadas com as estruturas que descrevem cada padrão. Se as restrições estruturais forem atendidas, a classe é classificada e incluída no conjunto de padrões candidatos. Entretanto,

aplicando somente a análise dinâmica, apenas uma parte do comportamento do sistema é representado e analisado [11]. Desta forma, quando as técnicas são aplicadas de forma individual, elas não conseguem detectar com eficácia todos os casos em que os padrões estão sendo aplicados.

Por outro lado, quando as técnicas são aplicadas de forma sequencial e complementar elas conseguem identificar com maior eficácia os padrões. No primeiro passo para detecção automática, a análise estática é executada e as relações e colaborações estruturais são extraídas. Esta técnica de análise não requer a execução ou modelo executável para ser conduzida [45; 48]. Os analisadores estáticos de código podem trabalhar diretamente sobre o código fonte do programa ou sobre o código objeto, no caso da linguagem Java, sobre o *bytecode* [48]. Como resultado deste primeiro passo, um conjunto de padrões candidatos é construído, comparando-se as colaborações estruturais com o modelo estrutural já definido pelo padrão, conforme visto na Figura 1.2.

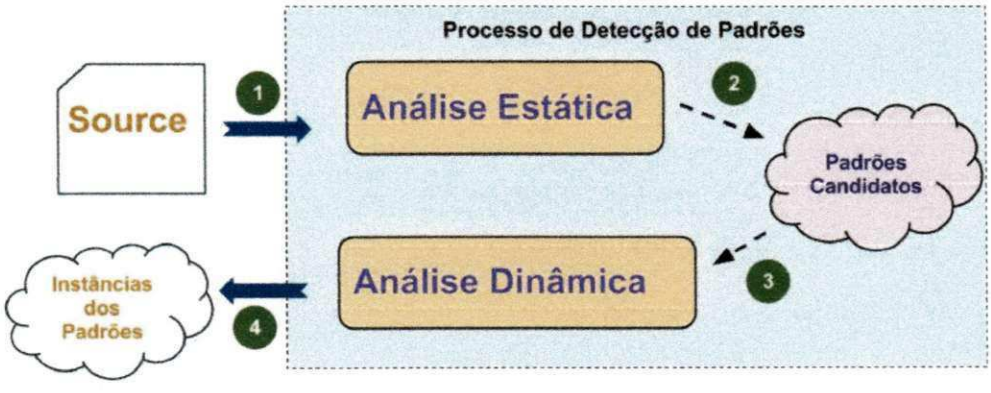

Figura 1.2: Processo de detecção utilizando as técnicas de análise estática e dinâmica

Porém, dentro deste conjunto de padrões pode existir uma grande quantidade de falso-positivos, visto que algumas informações comportamentais são conhecidas apenas em tempo de execução e somente com essas informações é possível classificar o padrão corretamente. Por exemplo, no padrão *Observer* as estruturas podem corresponder às características estruturais do padrão. Porém, apenas em tempo de execução seu comportamento é conhecido e verificado se há uma notificação correta a todos os objetos ouvintes. Visando diminuir os falso positivos, no segundo passo, é executada uma análise dinâmica, que é responsável

por monitorar a execução, rastrear ou instrumentar este conjunto de padrões selecionados no passo anterior e identificar quais padrões comportam-se como o esperado, diminuindo assim os falso-positivos.

Além das limitações quando aplicadas de forma individual, predominam nas técnicas para detecção de padrões, uma filtragem estrutural durante a análise estática, eliminando possíveis instâncias de padrões que não possuem as restrições estruturais descritas pelo modelo estrutural do padrão. Nestes casos quando há o comportamento e as estruturas descritas pelo padrão não estão presentes, estão caracterizados os padrões emergentes. O termo padrão emergente é a nomeclatura proposta para caracterizar o comportamento idêntico entre o padrão e os objetos, sendo que esse comportamento surge durante a evolução do sistema.

1.2 Exemplo de Motivação

Como visto, para identificar os padrões é importante analisar as estruturas do código, porém uma grande quantidade de falso-positivos pode existir, se não for analisado o comportamento do código. Por exemplo, analisando apenas as estruturas não seria possível diferenciar os padrões *Strategy* e *State* [18], apresentados nas Figuras 1.3 e 1.4, respectivamente.

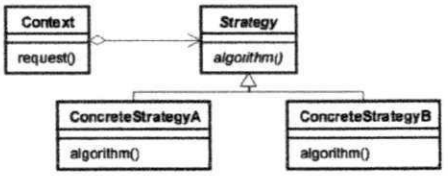

Figura 1.3: Padrão *Strategy*

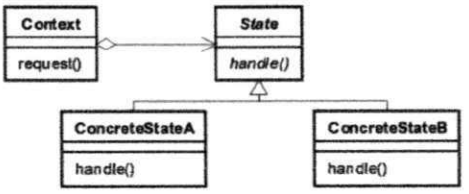

Figura 1.4: Padrão *State*

Portanto, é crucial observar o comportamento para caracterizar corretamente um padrão. No entanto, as abordagens atuais utilizam à análise estrutural anteriormente a análise comportamental, desta forma, são eliminadas algumas instâncias que não possuem a estrutura básica do padrão.

Este problema pode ser identificado em um projeto opensource, o XStream[1]. Na versão 5 do projeto, a classe `XStream` possuía o atributo `converterLoookup` do tipo

[1] http://xstream.codehaus.org/

`DefaultConverterLookup`, conforme pode ser observado no Código 1.1. Se aplicadas as abordagens atuais [49], esta instância não seria detectada, embora possua o comportamento inerente ao padrão *Strategy*. A intenção do padrão Strategy é encapsular estratégias alternativas, ou abordagens, em classes separadas, cada uma das quais implementando uma operação comum. As classes que implementam as várias abordagens implementam a mesma operação e são, assim, intercambiáveis, apresentando a mesma interface aos cliente [37].

```
// Os imports, atributos e demais metodos foram omitidos
public class XStream {
    private DefaultClassMapper classMapper = new DefaultClassMapper
        ();
    private DefaultConverterLookup converterLookup = new
        DefaultConverterLookup(classMapper);
    public XStream() {}
    // Metodo que delega a chamada ao converterLookup
    public void registerConverter(Converter converter) {
        converterLookup.registerConverter(converter);
    }
}
```

Código Fonte 1.1: A classe `XStream` na versão 5

No entanto, quando aplicadas as abordagens atuais na mesma classe, mas na versão 6, é detectada uma instância do padrão *Strategy*. A modificação na classe pode ser vista no Código 1.2, onde a alteração do tipo do atributo para `ConverterLookup`. Com esta modificação, o comportamento permaneceu igual, porém uma nova instância do padrão *Strategy* é identificada. Neste momento a classe `DefaultConverterLookup` implementa a interface `ConverterLookup` e atende aos requisitos estruturais do padrão *Strategy*. O fato das técnicas não detectarem o padrão na versão 5, ocorre porque existe a filtragem estrutural, pois mesmo possuindo o comportamento do padrão *Strategy* a classe `XStream` era descartada do conjunto de padrões candidatos por não possuir a estrutura do padrão.

```
// Os imports, atributos e demais metodos foram omitidos
public class XStream {
    private ConverterLookup converterLookup = new
        DefaultConverterLookup();
    public XStream() {}
    // Metodo que delega a chamada ao converterLookup
    public void registerConverter(Converter converter) {
        converterLookup.registerConverter(converter);
    }
}
```

Código Fonte 1.2: A classe `XStream` na versão 6

1.3 Solução

Além da estrutura, para caracterizar um padrão, deve-se observar o comportamento, dado que as técnicas atuais primam por uma análise estrutural restritiva, eliminando assim as instâncias que não atendem às restrições descritas pelo padrão. Por exemplo, analisada estaticamente uma classe `Singleton`, as informações de sua estrutura são extraídas: construtor privado, método estático e um atributo do mesmo tipo da classe. Se as restrições estruturais forem atendidas, a classe é classificada e incluída no conjunto de padrões candidatos. Porém, caso a classe não possua alguma das estruturas, por exemplo, o construtor privado, essa classe não será classificada, mesmo se a classe possuindo o comportamento do padrão *Singleton*.

Por outro lado, detectar padrões de projeto verificando apenas o comportamento ainda é um dos principais desafios encontrados na área [11]. Esta dificuldade é proveniente da grande quantidade de interações e colaboração entre os objetos, o que torna difícil isolar e tratar todos os possíveis comportamentos do código, além de ser uma atividade custosa quando aplicada a sistemas mais complexos. Para diminuir este problema com a quantidade de estados possíveis dos objetos, De Lucia *et al.* [11] utilizou uma técnica para verificar o estado e comportamento das propriedades no decorrer do tempo, a fim de verificar o comportamento dos objetos.

Portanto, o comportamento do código deve ser observado para detectar a essência com-

portamental do padrão, incluindo os casos em que as estruturas e relações dos objetos não atendam as restrições estruturais impostas pelo padrão. Essa essência caracteriza individualmente cada padrão. Por exemplo, o padrão *Singleton* é identificado quando durante a execução do programa existe apenas uma única referência à instância dessa classe. Se a essência já está presente no código, este caso foi nomeado de padrões emergentes, pois *emerge* do código, como consequência de decisões de projeto ou demandas de negócio. Os padrões emergentes são caracterizados pelo comportamento idêntico entre o padrão e os objetos, mas não possuem as restrições estruturais descritas pelo padrão e este comportamento surge durante a evolução do software. O Código 1.1 mostra um caso que há um padrão emergente. Desta forma, é importante ao projetista identificar esta essência comportamental do padrão, que emerge durante a evolução do software. Por exemplo, se for necessário aplicar modificações no código ou reestruturá-lo, identificar esta essência é primordial para as escolhas do projetista.

Conforme apresentado, se faz necessário um estudo sobre a aplicabilidade das técnicas existentes para detecção dos padrões emergentes. Devido a filtragem estrutural realizada em um primeiro momento, as instâncias que não possuem as estruturas dos padrões ainda que possua comportamento idêntico ao padrão não são incluídas no conjunto de padrões candidatos.

Neste trabalho é realizado um estudo exploratório em quatro sistemas de código livre, com o objetivo de identificar evidências de padrões emergentes e uma abordagem semi-automática para detecção de padrões emergentes é proposta. A abordagem é executada em duas etapas: extração e visualização. Na primeira, as informações estruturais são extraídas de uma base de código disponível em um repositório de versões. Além de identificar os padrões presentes em cada versão. Na segunda etapa, é realizada uma análise manual para identificar evidências dos padrões emergentes, comparando as versões aos pares e observando se há alguma modificação estrutural no código. Como resultado, após analisados quatro projetos totalizando 809 versões, foram detectadas 22 evidências de padrões emergentes e, de fato, 17 padrões emergentes. Portanto, os resultados apontam para a existência dos padrões emergentes, porém ainda não é possível detectá-los com as técnicas encontradas na literatura.

1.4 Contribuições

Em resumo, as principais contribuições deste trabalho são:

Uma revisão sistemática sobre abordagens automáticas de detecção de padrões de projeto: No intuito de realizar um levantamento bibliográfico sobre a área de detecção de padrões de projeto, foi realizada uma revisão sistemática que reuniu inicialmente um total de 927 estudos, finalizando com um total 41 trabalhos classificados para leitura.

Conceitos de padrões emergentes: Por ser uma nomenclatura nova, foi preciso desenvolver uma formalização para identificar e diferenciar os padrões emergentes. Foram formalizados nove padrões descritos pelo catálogo do GoF (*Gang of Four*) [18].

Uma proposta de abordagem semi-automática de detecção de padrões emergentes: Para detectar os padrões emergentes foi proposta uma abordagem semi-automática. Basicamente, é necessário um histórico de versões, uma formalização dos padrões e uma ferramenta que detecte os padrões. Uma ferramenta foi implementada seguindo essa abordagem e seguindo duas etapas, extração e visualização.

Um estudo exploratório sobre as técnicas de detecção existentes: A avaliação utilizou o protótipo desenvolvido e analisou quatro projetos de código aberto. Foram verificadas um total de 780 instâncias analisadas, 22 instâncias foram selecionadas e 17 padrões emergentes foram identificados.

1.5 Organização do Trabalho

Este trabalho está organizado como se segue. No próximo capítulo é apresentada a fundamentação teórica, necessária para o entendimento deste trabalho. No Capítulo 3, é apresentada uma revisão sistemática sobre detecção de padrões de projeto, onde é feito um levantamento sobre as técnicas, abordagens e ferramental para detecção, dentre outras questões. No Capítulo 4 são apresentados os padrões emergentes, sua formalização e uma abordagem proposta para detectá-los. No Capítulo 5 a avaliação sobre a abordagem proposta é detalhada e seus resultados discutidos. Por fim, no Capítulo 6 são feitas as considerações finais sobre o trabalho e as dificuldades encontradas no decorrer da pesquisa.

Capítulo 2

Fundamentação Teórica

Este capítulo tem a finalidade de apresentar os conceitos necessários para um melhor entendimento do presente trabalho. Inicialmente, será apresentado o conceito de Padrão de Projeto. Em seguida, será abordada a detecção de padrões focando nas técnicas utilizadas. Por fim, serão apresentadas alguns dos trabalhos relacionados à detecção de padrões de projeto.

2.1 Padrões de projeto

Nas linguagens orientadas a objetos um mesmo comportamento pode ser implementado de diversas formas. Por exemplo, na instanciação de um objeto, em ambos os códigos (Código 2.1 e Código 2.2) o atributo `p` possui o mesmo estado. A diferença é que no primeiro momento com o método de criação `novoProduto` e no segundo momento a classe `Cliente` instancia o objeto diretamente com o operador `new`. Mas em ambos os casos o atributo `p` possui o mesmo estado.

```
//Primeiro Momento
public class Cliente{
public void salvar(){
    Produto produto = Produto.novoProduto("Livro",2);
}
class Produto{
    private String descricao;
    private int id;
```

```
    public Produto(String descricao, int id) {
        this.descricao = descricao;
        this.id = id;
    }
    public static Produto novoProduto(String descricao, int id) {
        return new Produto(descricao,id);
    }
   }
}
```

Código Fonte 2.1: Instânciação com um método de criação `novoProduto`

```
//Segundo Momento
public class Cliente{
public void salvar(){
    Produto produto = new Produto("Livro",2);
}
class Produto{
    private String descricao;
    private int id;
    public Produto(String descricao, int id) {
        this.descricao = descricao;
        this.id = id;
    }
   }
}
```

Código Fonte 2.2: Instânciação com `new`

Essa variedade de implementações para solucionar um mesmo problema, motivam o surgimento dos padrões de projeto. 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. Os padrões de projetos são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto

num contexto particular, tornando mais fácil reutilizar projetos e arquiteturas bem sucedidas. Além de ajudar a escolhas no projeto afim de torná-lo reutilizável e auxiliar na documentação e comunicação do software [18]. A Figura 2.1 exibe o padrão Proxy. O objetivo deste padrão é fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto.

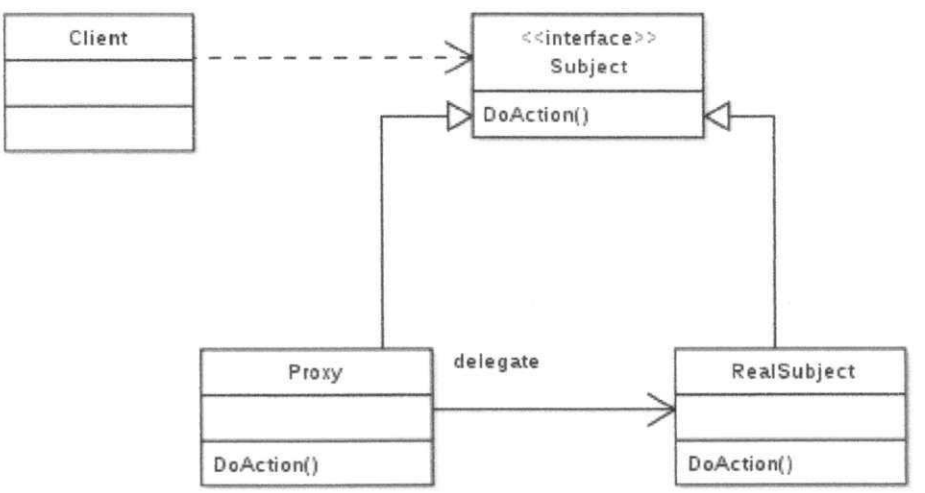

Figura 2.1: Padrão Proxy

Segundo Kerievsky [28], os padrões são soluções consolidadas, recorrentes e que são aplicadas a um contexto específico para solucionar um determinado problema de projeto de software. Auxiliando na comunicação entre os desenvolvedores, os padrões de projeto nomeia, abstraí 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 [18].

2.2 Detecção de padrões de projeto

No processo de detecção algumas técnicas são utilizadas, nesta seção são abordadas as técnicas de análise estática e análise dinâmica.

2.2.1 Analise estática

Esta técnica de análise não requer a execução ou modelo executável para ser conduzida [45; 48]. Nessa análise as relações e colaborações estruturais são extraídas e representadas por um modelo intermediário, por exemplo, CFG (Control Flow Graph), AST (Abstract Syntax Tree) e Call Graphs. O CFG é um grafo dirigido, onde os nós correspondem as declarações, e as arestas representam o possível fluxo de controle. O CFG tem sempre um único ponto de entrada e um único ponto de saída. O CFG é um grafo direcionado em que os nós representam as regiões do código-fonte e as arestas direcionadas representam a possibilidade de execução do programa prosseguir a partir do final de uma região diretamente para o começo de outra [54, cap. 12]. O Call Graph é um grafo que representa a chamada dirigida relações entre sub-rotinas de um programa. Especificamente, cada nó representa um procedimento e cada aresta indica as chamadas deste procedimento.

A análise estática pode ocorrer diretamente sobre o código fonte do programa ou sobre o código objeto, no caso da linguagem Java, sobre o *bytecode* [48]. A abordagem proposta por Bernardi *et al.* [6] utiliza análise estática e é capaz de detectar variações do padrão. Para a identificação dos padrões, primeiro são extraídas as informações estruturais do código, depois é construído um metamodelo do sistema, e por fim, o modelo gerado do sistema e o modelo de cada padrão são pesquisados, armazenados em um repositório e são detectados por um algoritmo de similaridade. Outra abordagem, descrita por Washizaki *et al.*[52] detecta variações de padrões fazendo uso apenas das informações estruturais, que são extraídas pelas ferramentas JavaML e XPath. A abordagem é descrita por dois passos, *Forward method* e *Backward method.*

2.2.2 Analise dinâmica

Um problema ao se observar o comportamento do sistema é a quantidade de possíveis estados que o sistema pode assumir. Esta dificuldade é proveniente da grande quantidade de interações e colaboração entre os objetos, o que torna difícil isolar e tratar todos os possíveis comportamentos do código, além de ser uma atividade custosa quando aplicada a sistemas mais complexos [49]. Para diminuir esse espaço de busca algumas técnicas podem ser utilizadas, por exemplo, O *Model checking* [9].

Outra solução descrita por Ng *et al.* [39] identifica instâncias de padrões comportamentais e criacionais em código fonte, usando somente análise dinâmica. Os padrões são descritos em Diagramas de sequências. Durante a análise dinâmica, um Diagrama de sequência é instanciado a partir da trilha de execução (*execution trace*) do sistema e, então, são analisados para encontrar o comportamento do padrão. Desta forma o problema para identificar o padrão é reduzido a uma checagem de restrições - CSP(*constraint satisfaction problem*)- entre o diagrama de sequência do padrão e do sistema.

2.2.3 Analise estática e dinâmica

Mesmo sendo possível a execução de uma técnica individualmente, é comum encontrar trabalhos [6; 10; 11; 32; 34; 51] que utilizam as duas técnicas de análise, estática e dinâmica, para detectarem com maior precisão os padrões de projetos.

Durante a análise estática são extraídas as informações estruturais dos padrões e um conjunto de padrões são classificados [10; 11; 25; 51]. Porém, dentro deste conjunto de padrões pode existir uma grande quantidade de falso-positivos, visto que algumas informações comportamentais são conhecidas apenas em tempo de execução e somente com essas informações é possível classificar o padrão corretamente. Por isso, no segundo passo, é executada uma análise dinâmica, que é responsável por monitorar a execução, rastrear ou instrumentar este conjunto de padrões selecionados no passo anterior e identificar quais padrões comportam-se como o esperado, diminuindo assim os falso-positivos.

Durante o processo de detecção os padrões podem ser representados e definidos de várias formas, por exemplo *Prolog queries* [47], ASG (Abstract Semantic Graph) [34] e metamodelos [6]. A abordagem proposta por Bernardi *et al.* [6] é independente da estrutura do padrão para realizar a busca, e o processo de detecção é baseado em especificações. Por isso, é possível escrever novas especificações de padrão, ou derivados dos já existentes (para detectar variantes) ou escrevê-los a partir do zero (para detectar novos padrões), sem impacto sobre o algoritmo de detecção. A abordagem define um metamodelo para representar um padrão de projeto através de um conjunto de propriedades relacionadas com os elementos estruturais. Para a identificação dos padrões: *(i)* são extraídas as informações estruturais do código; *(ii)* é construído um metamodelo do sistema; *(iii)* o modelo gerado do sistema e o modelo de cada padrão são pesquisados, armazenados em um repositório e são detectados

por um algoritmos de similaridade. Já a proposta para detecção apresentada por Stoianov *et al.* [47], define (modela) os padrões em *Prolog queries* e visa detectar padrões e anti-padrões utilizando uma abordagem baseada em *logic-based*. O protótipo desenvolvido utiliza a ferramenta JTransformer, que transforma o programa em *Prolog fact base*.

No intuito de diminuir a quantidade de falso-positivos algumas abordagens [10; 12; 11; 36] utilizam técnicas como *Model checking*. *Model checking* é um método para automatizar a verificação dos estados finitos. Para isso, utiliza lógica temporal para especificar o comportamento do sistema. Um programa é um objeto matemático bem definido, embora possivelmente complexo e intuitivamente com um comportamento complexo. A lógica matemática pode ser usada para descrever precisamente o que constitui um comportamento correto, isso faz com que seja possível contemplar matematicamente, que estabelece que o comportamento do programa está em conformidade com a especificação [9].

Na etapa da análise estática os padrões candidatos são extraídos. O *Model checking* é executado posterior a análise estática. E, na análise dinâmica, o comportamento padrão é analisado através de uma instrumentação de código que monitora as instâncias dos padrões candidatos. Lebon *et al.* [32] utiliza a análise dinâmica para diminuir os falso-positivos através de uma filtragem de baixa granularidade, identificada no trabalho como FiG.

2.3 Considerações sobre o capítulo

Neste capítulo foram discutidas algumas técnicas e abordagens para detecção de padrões de projeto. Há algumas técnicas para detecção de padrões de projeto. No entanto, quando aplicadas às abordagens existentes, excluem dos resultados os casos em que o padrão não atende às restrições estruturais que o definem. Desta forma, não é possível detectar possíveis situações onde os padrões podem ser aplicados.

A abordagem proposta neste trabalho busca resolver este problema. Embora não seja implementada uma ferramenta para detecção, fazendo uso da abordagem é possível, por exemplo, de identificar situações onde um determinado padrão pode ser implementado. Desta forma, este trabalho contribui com a área tanto com a revisão sistemática quanto com a abordagem proposta.

Capítulo 3

Revisão Sistemática

Detectar padrões de projeto em uma base de código não é uma atividade simples. Esta dificuldade existe devido às informações individuais que caracterizam cada padrão. Basicamente, os padrões podem ser descritos quanto a sua estrutura e/ou comportamento. Os elementos estruturais de um padrão estão relacionados aos atributos, métodos e classes que compõem o padrão. Já o comportamento está relacionado com a forma que os elementos estruturais se portam durante a execução do software. Em um alto nível de abstração, essas informações possuem as características necessárias para que as abordagens ou técnicas consigam detectar os padrões com uma alta acurácia. No entanto, neste processo, para verificar tanto as relações e colaborações estruturais quanto comportamentais ainda surge um alto número de falso-positivo e falso-negativo.

Desta forma, uma revisão na literatura sobre as abordagens, técnicas e ferramentas para detecção de padrões de projeto pode auxiliar em: *(i)* compreender quais os principais desafios; *(ii)* identificar divergências entre resultados, caso existam; e, *(iii)* mapear a área e estado da arte sobre detecção de padrões. Para este fim, o método de pesquisa adotado para este estudo foi uma revisão sistemática. Uma revisão sistemática é uma abordagem bem definida para identificar, avaliar e interpretar os estudos relevantes a respeito de uma investigação particular, pergunta, área de tópico ou fenômeno de interesse [29].

3.1 Objetivo

Este estudo tem como objetivo fornecer uma visão geral sobre a detecção de padrões de projeto. Em particular, pretende-se observar como os padrões são detectados e identificados, se e como os padrões são formalizados, e em quais projetos foram aplicadas as técnicas para detecção de padrões.

3.2 Metodologia da Revisão

Para realizar uma revisão sistemática conforme processo descrito por Kitchenham [29], é necessário seguir as seguintes etapas:*(i)* planejamento, *(ii)* condução e *(iii)* a escrita do relatório da revisão. Na primeira fase da revisão*(i)*, é levantada a necessidade de realizar uma revisão sistemática sobre detecção de padrões em código orientados a objetos. Além de elaborar o protocolo de revisão, que contempla desde as questões de pesquisas a ameaças à validade. Na etapa de condução*(ii)*, a revisão é executada e envolve tarefas como execução da seleção, extração das informações dos estudos selecionados e análise dos dados. E na última etapa*(iii)*, o relatório da revisão é escrito condensando os resultados, dificuldades e análise provenientes da revisão. A Figura 3.1 mostra a metodologia adotada para este estudo.

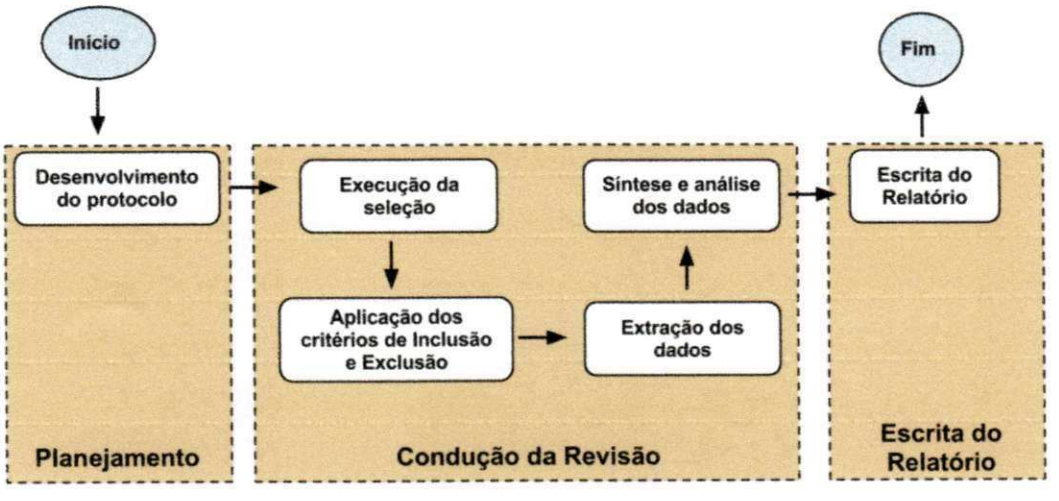

Figura 3.1: Metodologia adotada para revisão sistemática

A revisão foi realizada por dois pesquisadores. Ambos os colaboradores definiram o protocolo para execução da revisão. O processo de execução da seleção foi realizado por

um avaliador. A aplicação dos critérios de inclusão e exclusão passou por uma avaliação dos dois pesquisadores para identificar quais estudos seriam selecionados. Um revisor extraiu os dados a partir dos estudos selecionados. Finalmente, o relatório foi escrito e revisado.

3.3 Protocolo para Execução

Como citado anteriormente, o objetivo desta revisão sistemática é fornecer uma visão geral sobre a detecção de padrões de projeto. Contudo, esta questão é genérica para uma completa avaliação. Assim, foram propostas algumas sub-questões (SQ) que permitem focar nos aspectos específicos para a avaliação. Portanto, as seguintes sub-questões de pesquisa foram definidas:

SQ 1: Qual abordagem para detecção de padrões foi adotada?

SQ 2: Qual a natureza da técnica?

SQ 3: Qual o delineamento experimental?

SQ 4: Qual o grau de automatização da ferramenta, caso exista?

SQ 5: Quais as variações sintáticas a técnica detecta?

Visto que existem várias abordagens para detecção de padrões, a motivação para SQ1 é levantar as principais abordagens de detecção e analisar as vantagens e limitações. Com uma resposta para SQ2 é possível caracterizar os pontos fortes de cada técnica e se é possível utilizá-las em conjunto. Encontrar evidências experimentais na detecção pode auxiliar e validar os resultados encontrados, inclusive quando existem comparativos entre ferramentas, técnicas ou abordagens, por isso é proposta a SQ3. Detectar e identificar os padrões no código não é uma atividade trivial e ferramentas automatizadas, semi-automatizadas ou manuais podem facilitar a detecção ao projetista ou desenvolvedor, a SQ4 analisará esta característica. Além das estruturas descritas nas referências[28; 18] sobre padrões de projeto, existem variações de implementações de padrões e isso pode gerar conflitos e discrepâncias nas avaliações seja dos experimentos ou ferramentas, por esta razão foi elaborada a SQ5.

3.4 Critérios de Inclusão e Exclusão

Durante a execução da revisão sistemática, os trabalhos precisam ser selecionados e filtrados. Os critérios de exclusão são utilizados para excluir os trabalhos que não estão alinhados com o objetivo da revisão sistemática. Já os critérios de inclusão delimitam os trabalhos que serão analisados. Os seguintes critérios de inclusão foram utilizados:

CI1: Artigos que propõem a detecção ou identificação dos padrões de projeto.

CI2: Trabalhos que formalizam os padrões de projeto.

CI3: Trabalhos sobre novas abordagens, técnicas ou ferramentas para detecção ou identificação de padrões de projeto.

CI4: Trabalhos sobre métricas para identificar padrões de projeto.

Além dos seguintes critérios de exclusão:

CE1: Estudos sobre detecção de padrões não relacionados aos padrões de projetos.

CE2: Trabalhos que contêm somente uma revisão sobre o estado da arte ou uso de padrões de projetos, que não envolvem a detecção.

CE3: Artigos que relacionam padrões de projetos a qualidade de software.

CE4: Estudos duplicados ou inacessíveis (publicação não disponível).

3.5 Processo e Estratégia de Seleção

Com a finalidade de responder as questões foram definidos os mecanismos de buscas e palavras-chave. Como mecanismos de busca, foram utilizados os portais do IEEE, ACM e SpringerLink, conforme visto na Tabela 3.1.

Tabela 3.1: Mecanismos de busca

Fonte de Dados	**Referência**
ACM Digital Library	http://portal.acm.org/
IEEE Xplore	http://ieeexplore.ieee.org/
SpringerLink	http://link.springer.com/

Utilizando os operadores lógicos é possível refinar a busca por palavras-chaves. Os seguintes termos foram utilizados nas seleções automáticas:

Design **AND** Pattern **AND** (Detection **OR** Identification)

Design **AND** Pattern **AND** (Tool **OR** Techniques)

Design **AND** Pattern **AND** (Manual Analysis **OR** Automatic Analysis)

3.5.1 Coleta de Dados

Dos trabalhos selecionados algumas informações foram coletadas e armazenas em uma planilha eletrônica. Todos os dados coletados podem ser acessados no site[1] construído para disponibilizar os dados, configurações e protocolo desta revisão sistemática. Os seguintes dados foram coletados:

- Titulo;
- Ano de publicação;
- Autores;
- Contexto;
- Técnica utilizada;
- Ferramenta;
- Problema;
- e, Solução.

3.6 Execução da seleção

Para seleção dos estudos, as strings de busca foram executadas e elaborada uma lista inicial com os estudos primários. Na Tabela 3.2 pode ser visto um resumo dos resultados das buscas.

A etapa seguinte foi a análise dos títulos dos estudos. O principal objetivo desta etapa é analisar se os artigos estão de acordo com os critérios de inclusão e exclusão. Muitos dos resultados encontrados eram provenientes da área inteligência artificial, onde trabalham com detecção de padrões, mas não relacionados a padrões de projetos. Uma filtragem inicial foi

[1] https://sites.google.com/site/detectacaodepadrao/

Tabela 3.2: Seleção inicial com todos os resultados, agrupados por mecanismos de busca

Fonte de Dados	**Resultados**
ACM Digital Library	162
IEEE Xplore	469 (de todas as áreas)
SpringerLink	296 (de todas as áreas)
Total	927

executada para eliminar essas ocorrências e as possíveis duplicatas. Após esta filtragem a lista de artigos foi reduzida de 927 para 128. A Tabela 3.3 sumariza estes resultados.

Tabela 3.3: Resultado da seleção após a filtragem dos critérios de inclusão e exclusão

Fonte de Dados	**Resultados**
ACM Digital Library	75
IEEE Xplore	17
SpringerLink	36
Total	128

Após a execução da etapa de leitura dos títulos, o número de artigos foi reduzido. A Tabela 3.4 sumariza estes resultados. Apenas 58 artigos prosseguiram para a fase de leitura de *abstract*.

Tabela 3.4: Resultado da seleção após leitura dos títulos

Fonte de Dados	**Resultados**
ACM Digital Library	38
IEEE Xplore	6
SpringerLink	14
Total	58

Na etapa de leitura de *abstract* a maioria dos estudos primários prosseguiram para próxima etapa. A Tabela 3.5 exibe o resultado encontrado.

Na Tabela 3.6 são listados os 41 estudos selecionados para leitura e análise completa.

Tabela 3.5: Resultado da seleção após leitura dos *abstracts*

Fonte de Dados	**Resultados**
ACM Digital Library	34
IEEE Xplore	5
SpringerLink	2
Total	41

Tabela 3.6: Trabalhos relacionados para leitura e análise completa

#	Source	Título do Artigo
1	IEEE	Precise specification and automatic application of design patterns [15]
2	ACM	Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects [19]
3	ACM	Automatic Design Pattern Detection [25]
4	ACM	Language-independent detection of object-oriented design patterns [16]
5	ACM	DPVK - An Eclipse Plug-in to Detect Design Patterns in Eiffel Systems [50]
6	IEEE	Elemental Design Patterns Recognition In Java [1]
7	ACM	SPQR: formal foundations and practical support for the automated detection of design patterns from source code [44]
8	ACM	Design Pattern Detection in Eiffel Systems [51]
9	ACM	A Formal Description of Design Patterns Using OWL [13]
10	ACM	Efficient Identification of Design Patterns with Bit-vector Algorithm [26]
11	SpringerLink	Detection of design patterns by combining static and dynamic analyses [34]
12	ACM	Automatic Detection of Design Pattern for Reverse Engineering [33]

13	ACM	PTIDEJ and DECOR: identification of design patterns and design defects [38]
14	ACM	Detection of Diverse Design Pattern Variants [46]
15	ACM	Design Pattern Detection by Using Meta Patterns [23]
16	ACM	Design pattern detection by template matching [14]
17	ACM	Can design pattern detection be useful for legacy systemmigration towards SOA? [3]
18	ACM	Identification and Extraction of Design Pattern Information in Java Program [24]
19	ACM	DeMIMA: A Multilayered Approach for Design Pattern Identification [20]
20	ACM	Evaluating Quality of Software Systems by Design Patterns Detection [43]
21	ACM	Detecting Design Patterns Using Source Code of Before Applying Design Patterns [52]
22	ACM	Design improvement through dynamic and Structural pattern identification [8]
23	ACM	Design pattern recovery through visual language parsing and source code analysis [36]
24	IEEE	Standing on the shoulders of giants - A data fusion approach to design pattern detection [30]
25	SpringerLink	Improving design-pattern identification: a new approach and an exploratory study [22]
26	ACM	Improving Behavioral Design Pattern Detection through Model Checking [11]
27	ACM	Evaluation of Accuracy in Design Pattern Occurrence Detection [41]
28	ACM	Identification of behavioural and creational design motifs through dynamic analysis [39]

29	ACM	Identification of design motifs with pattern matching algorithms [27]
30	IEEE	Detecting patterns and antipatterns in software using Prolog rules [47]
31	ACM	An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis [10]
32	ACM	DPDX–Towards a Common Result Exchange Format for Design Pattern Detection Tools [31]
33	ACM	Model-driven detection of Design Patterns [6]
34	ACM	Design patterns detection using SOP expressions for graphs [21]
35	ACM	Flexible design pattern detection based on feature types [42]
36	ACM	Understanding the relevance of micro-structures for design patterns detection [4]
37	ACM	A tool for design pattern detection and software architecture reconstruction [5]
38	ACM	Fine-Grained Design Pattern Detection [32]
39	ACM	DPJF - Design Pattern Detection with High Accuracy [7]
40	ACM	DPB: A Benchmark for Design Pattern Detection Tools [17]
41	IEEE	Design Pattern Detection Using Similarity Scoring [49]

3.7 Análise dos Resultados

Os resultados da Revisão Sistemática foram tabulados e os artigos organizados por ano. Além das ferramentas de detecção de padrões que foram abordadas nos artigos analisados. Nesta seção serão apresentados os artigos tabulados por ano e as ferramentas levantadas.

3.7.1 Ano de publicação

Quanto ao ano de publicação, o gráfico da Figura 3.2 apresenta a divisão desses estudos proporcionalmente e organizados por ano.

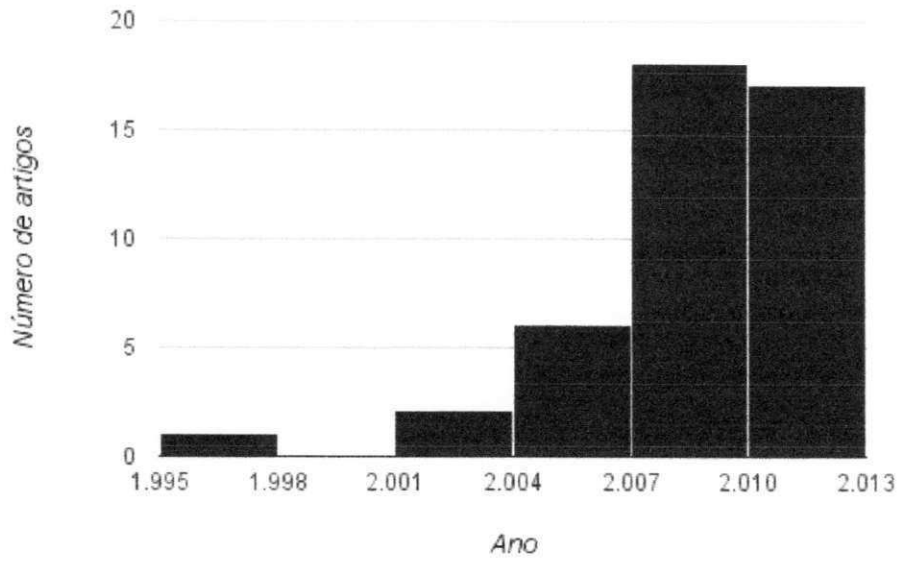

Figura 3.2: Artigos organizados por ano

3.7.2 Ferramentas

Alguns trabalhos propõem *benchmarks* [17] para ferramentas de detecção, discutem sobre métricas [43], ou buscam unificar um formato comum às ferramentas de detecção de padrões de projetos [30; 31]. Outros trabalhos selecionados [8; 13; 15; 22; 26; 27] apresentam novas abordagens para detecção de padrão e não possuem um suporte ferramental para avaliação da abordagem. Por outro lado, algumas ferramentas são propostas. Na Tabela 3.7 são exibidas as ferramentas identificadas nos trabalhos selecionados. Não foram selecionados os protótipos de ferramentas que são utilizados apenas para validar uma abordagem.

Tabela 3.7: Ferramentas analisadas

Ferramenta	**Referência**	**Ano**	**Natureza da técnica**
PTIDEJ	[19] [38]	2001/2007	Análise estática
DPVK	[51]	2004/2005	Análise estática e dinâmica
SPQR	[44]	2005	Análise estática
SSA	[49]	2006	Análise estática

D^3	[46]	2008	Análise estática
DeMIMA	[20]	2008	Análise estática e dinâmica
DPRE	[36] [11]	2009/2010	Análise estática e dinâmica
ePad	[10]	2010	Análise estática e dinâmica
DPF	[6]	2010	Análise estática e dinâmica
MARPLE	[5]	2011	Análise Estática
DPJF	[7]	2012	Análise Estática + (Control Flow e Data Flow)

Na Tabela 3.7, alguns trabalhos são identificados em mais de um estudo e para estes casos a coluna *Ano* identifica quais os anos em que os trabalhos foram publicados e na coluna *Referência* estão as referências para os referidos trabalhos. Por exemplo, a ferramenta PTIDEJ foi utilizada por Guéhéneuc *et al.* [19] e Moha *et al.* [38], nos anos 2001 e 2007, respectivamente. Além deste caso, a ferramenta DPRE, que foi utilizada em [36] e [11], foi modificada e evoluiu para outra ferramenta, atualmente conhecida por ePad [10].

Para a avaliação deste trabalho, foi selecionada a ferramenta SSA[2] de detecção de padrões, que manipula os *bytecodes* e detecta os padrões através de uma análise de similaridade [49]. Essa abordagem detecta variações dos padrões, utilizando um algoritmo de similaridade entre os vértices de um grafo. Essa ferramenta foi selecionada para a avaliação, pois além de detectar variações de padrões, possui uma boa performance.

3.8 Ameaças à validade

A principal ameaça encontrada está relacionada a quantidade de pessoas envolvidas para realizar os processo de seleção dos trabalhos. Este processo de seleção foi realizado por um pesquisador e revisado por outro. Na existência de conflitos quanto aos artigos selecionados, eram levantados argumentos para inclusão ou exclusão do artigo e uma discussão decidia se o artigo prosseguia, ou não.

Contudo, todo o processo está catalogado e a listagem com os artigos incluídos e excluídos, está disponível no site de divulgação. Uma provável validação da revisão sistemática seria adicionar outra pessoa para validar o protocolo, processo e execução da revisão. Além

[2]http://java.uom.gr/ nikos/pattern-detection.html

desta ameaça, um total de onze trabalhos não estavam disponíveis para leitura, por questões de restrição de acesso.

3.9 Considerações sobre o capítulo

Neste capítulo, foi apresentada uma revisão sistemática sobre abordagens automáticas de detecção de padrões de projeto. A revisão sistemática é organizada, basicamente, em três etapas: planejamento, execução e escrita do relatório. Inicialmente, foi desenvolvido e validado um protocolo para execução desta revisão. Durante a fase de execução, o estudo reuniu um total de 927 estudos, finalizando com um total 41 trabalhos classificados para leitura.

O intuito com esta revisão foi realizar um levantamento bibliográfico sobre a área de detecção de padrões de projeto e fornecer uma visão geral sobre a detecção de padrões de projeto. Em particular, observando como os padrões são detectados e identificados, se e como a padrões são formalizados, e em quais projetos foram aplicadas as técnicas para detecção de padrões.

Capítulo 4

Padrões Emergentes

Neste capítulo é apresentado o conceito de padrões emergentes e posteriormente uma abordagem semi-automática para detectá-los. Na seção que trata sobre os padrões emergentes há alguns exemplos que foram encontrados durante a avaliação. Outros, possuem apenas pseudocódigos que descrevem sua estrutura e comportamento desejado. A abordagem aqui descrita visa detectar evidências dos padrões emergentes através de um processo semi-automático. Na etapa automática, as informações são extraídas e, na outra, essas informações são verificadas manualmente.

4.1 Definição

Cada padrão de projeto é descrito basicamente por seu comportamento - estado esperado que o objeto esteja em determinado momento-, e sua estrutura: classes, métodos e tipos de relação entre as classes (herança, agregação, composição). No entanto, como visto na Seção 1.2, os padrões podem possuir estruturas semelhantes, mas diferenciar-se um dos outros por seu comportamento.

Por outro lado, há situações onde existe o comportamento de um padrão, mas não sua estrutura. Este fato pode estar relacionado a: *(i)* falta de conhecimento da implementação do padrão por parte do desenvolvedor; *(ii)* sucessivas mudanças em outras classes e refletidas nessa parte do código; *(iii)* falta de refatoração no código; ou, *(iv)* caracteristica do problema e não necessita implementar o padrão.

Os padrões emegentes podem ser derivados de algum desses fatores. Os padrões emer-

gentes são caracterizados quando há o comportamento inerente ao padrão, mas as estruturas que o descrevem não estão presentes. Os padrões emergentes capturam a essência comportamental de cada padrão. Por exemplo, o padrão *Singleton* é caracterizado por as classes compartilharem uma única instância durante toda a execução do sistema. Essa nomenclatura de padrões emergentes está sendo proposta neste trabalho para caracterizar a essência comportamental dos padrões.

Para exemplificar os padrões emergentes, as subseções que se seguem apresentam uma descrição e, quando possível, um exemplo encontrado no código dos projetos analisados na avaliação.

4.1.1 Singleton

Definição: Existe uma classe C que, em qualquer momento de execução, possui zero ou uma instância; quando existir a instância I, futuramente, só pode haver esta mesma instância I criada para todos os possíveis caminhos do programa. Por exemplo, no Código 4.1, a classe `Licences` possui as restrições estruturais do padrão *Singleton* exceto por seu construtor `default`. Essa classe não atende às condições do padrão *Singleton*, mas é classificado como um padrão emergente.

```
// Os imports, atributos e demais metodos foram omitidos
public class Licences {
    public static final String GPL = "GNU LICENSE";
    public static final String LGPL = "GNU LESSER PUBLIC LICENSE";
    /** Um atributo privado e estatico. */
    private static Licences singleton;
    /** Returna uma referencia desta classe. */
    public static Licences getInstance() {
        if (singleton == null) {
            singleton = new Licences();
        }
        return singleton;
    }
    public String getGPL() {
```

```
        return GPL;
    }
    public String getLGPL() {
        return LGPL;
    }
}
```

Código Fonte 4.1: O Padrão Singleton na classe `Licences`

4.1.2 Factory Method

Definição: Existe um método m que, retorna um objeto do tipo T, e, em qualquer momento, instâncias do tipo T são criadas apenas no contexto do método m. Por exemplo, no Código 4.2, a classe `StackTraceElementFactory` não possui as restrições estruturais do padrão *Factory Method*, pois não há subclasses para escolher entre as possíveis implementações de como instanciar objetos do tipo `StackTraceElement`, mas é classificada com um padrão emergente.

```
// Os imports, atributos e demais metodos foram omitidos
public class StackTraceElementFactory {

    public StackTraceElement nativeMethodElement(String
       declaringClass, String methodName) {
        return create(declaringClass, methodName, "Native", -2);
    }

    public StackTraceElement unknownSourceElement(String
       declaringClass, String methodName) {
        return create(declaringClass, methodName, "Source", -1);
    }
    private StackTraceElement create(String declaringClass, String
       methodName, String fileName, int lineNumber) {
        StackTraceElement result = new Throwable().getStackTrace()
           [0];
```

```
            setField(result, "declaringClass", declaringClass);
            setField(result, "methodName", methodName);
            setField(result, "fileName", fileName);
            setField(result, "lineNumber", new Integer(lineNumber));
            return result;
        }
}
```

Código Fonte 4.2: O Padrão Factory Method na classe `StackTraceElementFactory`

4.1.3 Adapter

Definição: Existe um método $m1$, na classe $C1$, que em qualquer momento de execução, invoca o método $m2$, da classe $C2$, as chamadas ao método $m2$ são realizadas apenas no contexto do método $m1$ e as chamadas são realizadas apenas em variável local ou atributos da classe $C1$. Por exemplo, na classe `DoctypeSupport` delega a execução do método `read()` a uma instância da classe `DoctypeInputStream`.

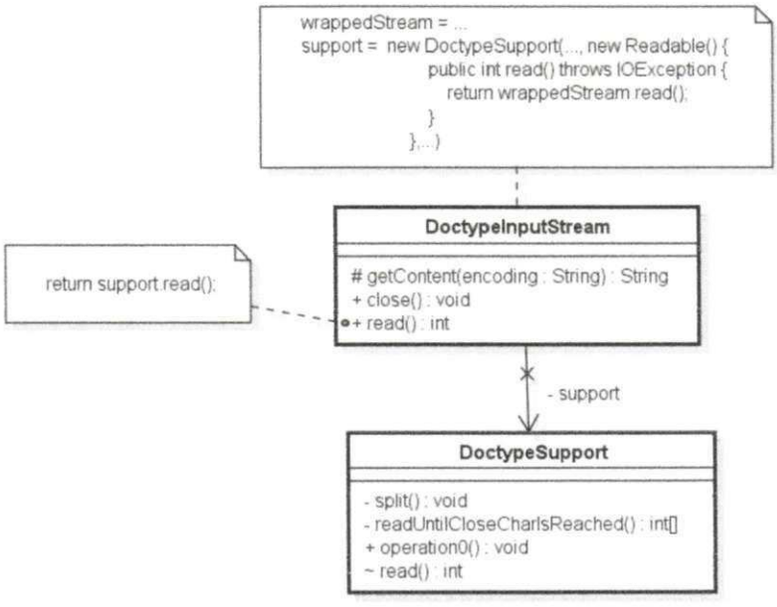

Figura 4.1: Padrão Adapter

4.1.4 Strategy

Definição: Existe um atributo a que, em qualquer momento de execução, pode ter seu valor alterado e modificar, futuramente, o algoritmo da classe C, a qual o atributo a pertence. Por exemplo, na classe `XStream` possui uma instância da classe `DefaultConverterLookup`, conforme visto no Código 4.3 e na Figura 4.2. Essa instância da classe `DefaultConverterLookup` pode alterar o comportamento da classe `XStream` quando o método `fromXML` for invocado.

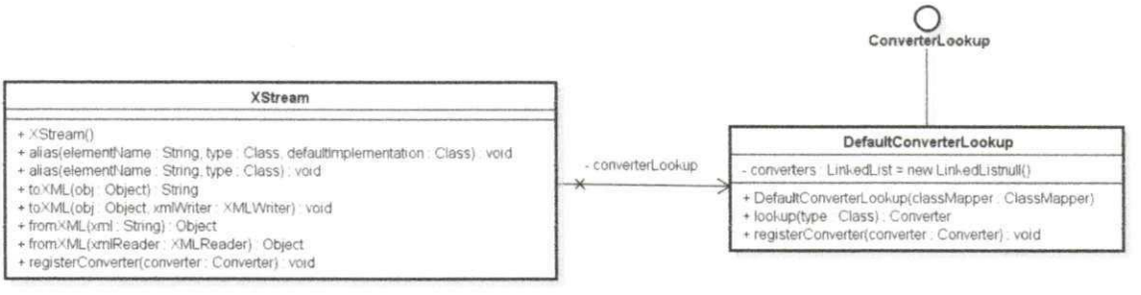

Figura 4.2: A classe XStream na versão 5

```
// Os imports, atributos e demais metodos foram omitidos
public class XStream {
    private DefaultClassMapper classMapper = new DefaultClassMapper
        ();
    private DefaultConverterLookup converterLookup = new
       DefaultConverterLookup(classMapper);
    public XStream() {}
    public Object fromXML(XMLReader xmlReader) {
        Class type = classMapper.lookupType(xmlReader.name());
        Converter rootConverter = converterLookup.lookup(type);
        rootConverter.fromXML(objectGraph, xmlReader,
           converterLookup, type);
        return objectGraph.get();
    }
    public void registerConverter(Converter converter) {
        converterLookup.registerConverter(converter);
    }
```

```
}
```

Código Fonte 4.3: O Padrão Strategy na classe `XStream`

4.1.5 Decorator

Definição: Existe uma classe C que possui um atributo a, do mesmo tipo da classe C, um método m e, em qualquer momento de execução e qualquer possível execução do método m, o método m do atributo a, será invocado. Por exemplo, no Código 4.4, a classe `ThrowableConverter` possui um atributo do tipo `Converter` e na execução do método `marshal`, há uma execução anterior a invocação do método `marshal` do atributo.

```
// Os imports, atributos e demais metodos foram omitidos
public class ThrowableConverter implements Converter {
    private Converter defaultConverter;
    public ThrowableConverter(Converter defaultConverter) {
        this.defaultConverter = defaultConverter;
    }
    public void marshal(Object source, HierarchicalStreamWriter
       writer, MarshallingContext context) {
        Throwable throwable = (Throwable) source;
        throwable.getStackTrace();
        defaultConverter.marshal(throwable, writer, context);
    }
}
```

Código Fonte 4.4: O Padrão Decorator na classe `ThrowableConverter`

4.1.6 Builder

Definição: Existe um método m que, retorna uma instância I, da classe C, e, antes de sua execução, outros métodos da mesma classe foram invocados; e, em qualquer momento, instâncias da classe C são criadas apenas no contexto do método$m$. Por exemplo, no pseucódigo do Código 4.5, alguns métodos (`menuLateral` e `menuCentral`) são invocados

antes da chamada ao método `build` para recuperar uma instância da classe `Site`. Dessa forma algumas operações são executadas antes na instanciação do objeto.

```
class Site{
    void menuLateral(); //chamada 1
    void menuCentral(); //chamada 2
    Site build(){    //chamada 3
        return new C();
    }
}
```

Código Fonte 4.5: Pseucodigo do padrão Builder

4.1.7 State

Definição: Existe um atributo a que, em qualquer momento de execução, pode ter seu valor alterado e modificar, futuramente, o comportamento da classe C, a qual o atributo a pertence; e, antes da alteração do valor do atributo a, existe uma checagem sobre seu atual valor. Por exemplo, no pseucódigo do Código 4.6, na execução do método `m` o atributo é `state` é alterado, modificando o estado da classe `C`.

```
class C{
 A state;
 void m(){
  IF (state == "StateA"){//executa algo e muda o estado
    state ="StateB";
  }ELSE IF (state == "StateB"){//executa algo e muda o estado
    state ="StateC";
  }ELSE IF (state == "StateC"){//executa algo e muda o estado
    state ="StateA";
  }
 }
}
```

Código Fonte 4.6: Pseucodigo do padrão State

4.1.8 Composite

Definição: Existe uma classe C que possui um método m e um atributo a, de um conjunto de objetos do mesmo tipo da classe C, e que, futuramente, o método m invocará, para todo elemento do conjunto presente no atributo a, o método m deste elemento. Por exemplo, no pseucódigo do Código 4.7 durante a execução do método `desenhar` todos os elementos presentes no `atributo` têm seu método `desenhar` invocado.

```
class Figura{
    Figura[] atributo;// Deve ser um conjunto de elementos do tipo
       Figura
        void desenhar(){
            for (atributo.lenght()){
                atributo.desenhar();// Para cada elemento do
                   conjunto
            }
        }
}
```

Código Fonte 4.7: Pseucodigo do padrão Composite

4.1.9 Observer

Definição: Existe uma classe $C1$ que possui um método m e um atributo a, de um conjunto de objetos do tipo da classe $C2$, e que, futuramente e qualquer possível execução do método m, o método m invocará, para todo elemento do conjunto presente no atributo a, o método $m2$ deste elemento. Por exemplo, no pseucódigo do Código 4.8 durante a execução do método `notificaTodos` todos os elementos presentes no `atributo` têm seu método `notifica` invocado.

```
class Modelo{
    Ouvinte[] atributo; // Deve ser um conjunto de elementos do
       tipo C
        void notificaTodos(){
            for (atributo.lenght()){
```

```
                atributo.notifica(); // Para cada elemento do
                    conjunto
            }
        }
}
```

Código Fonte 4.8: Pseucodigo do padrão Observer

4.2 Abordagem para detecção de Padrões Emergentes

Nesta seção é proposta uma abordagem para detecção dos padrões emergentes. Esta abordagem é similar à utilizada por Palomba *et al.* [40], onde o histórico de mudanças foi utilizado para detecção de *Bad Smells*. A abordagem proposta neste trabalho, necessita de: *(i)* um conjunto de versões de um sistema; *(ii)* uma especificação ou formalização para identificar e diferenciar os padrões; e, *(iii)* uma ferramenta que detecte os padrões em cada versão analisada.

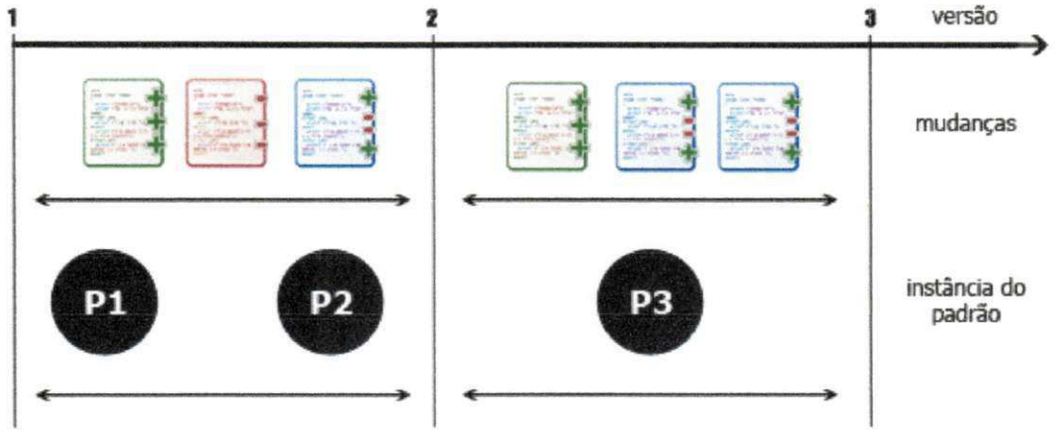

Figura 4.3: Overview dos elementos presentes na abordagem

Conforme visto na Figura 4.3, a cada nova versão, ocorrem mudanças e novas instâncias de padrões podem ser adicionadas, ou removidas, do código. Primariamente, a abordagem proposta visa identificar casos onde novos padrões surgem, são decorrentes de uma modificação no código e seu comportamento já existia antes da modificação.

Para identificar quando uma nova instância do padrão surge no código é necessário analisar as versões par-a-par e observar em qual versão a nova instância surgiu. O critério para que uma versão seja analisada é ser compilável e que tenha ocorrido alguma modificação em arquivos java, esse critério deve ser atendido se o código for implementado na linguagem Java. Uma vez identificadas as versões, o conjunto de mudanças, entre a versão atual e a versão anterior, é analisado. São identificadas quais classes foram modificadas. As mudanças podem ser de três tipos: *(i) classe adicionada*; *(ii) classe modificada*; e, *(iii) classe deletada.* Se a mudança estiver relacionada a uma classe que foi removida, mudança do tipo *classe deletada*, essa mudança não é considerada para análise. Por outro lado, se a mudança for decorrente de uma adição ou modificação da classe, essa mudança será analisada. Neste momento, para cada nova instância é preciso confrontar as classes e elementos participantes da instância do padrão com o conjunto de mudanças. O objetivo dessa etapa é saber se a nova instância do padrão surgiu devido a alguma mudança. O próximo passo é observar qual o tipo da mudança ocorreu para o surgimento da instância. Quanto ao tipo da mudança, podem ocorrer duas situações:*classe adicionada* ou *classe modificada.*

Mudança do tipo *classe modificada*, é preciso observar quais estruturas foram modificadas na classe. Para isso, as versões da classe são comparadas, as estruturas modificadas são identificadas e checado junto a formalização do padrão se o comportamento referente ao padrão já existia anterior a mudança. Em caso positivo, a nova instância é classificada.

Mudança do tipo *classe adicionada*, é preciso analisar todo o conjunto de mudanças e verificar se a nova classe criada é decorrente de uma refatoração. Se a classe adicionada mantiver o comportamento já existente, a nova instância é classificada.

A abordagem proposta pode ser descrita formalmente considerando um conjunto sequencial de versões $V = \{v_1, .., v_n\}$ e um conjunto de padrões formalizados $J = \{j_1, ..., j_n\}$, além das seguintes funções:

- $instancia(V_i)$, retorna um conjunto de instâncias do padrão j na versão V_i.
- $mudanca(V_i)$, retorna o número de mudanças que ocorreram na versão V_i.

O processo de detecção descrito anteriormente, pode ser formalizado por:

$$DeteccaoP_j(V_{i+1}) = \begin{cases} 0, & \text{se } \forall y |\ y \in instancia(V_{i+1}) \ e \ y \in instancia(V_i); \\ x, & \text{se } \forall y |\ y \in instancia(V_{i+1}) \ e \ y \notin instancia(V_i). \end{cases}$$

Onde:

$$\forall x|\ x = \begin{cases} 0, & \text{se } mudanca(V_i) = 0; \\ 1, & \text{se } mudanca(V_i) > 0. \end{cases}$$

Desta forma é realizada uma análise aos pares e sequencial de um conjunto de versões. Contudo, é necessário um *Oracle* que conheça a definição dos padrões para que na função $instancia(V_i)$ seja retornado um conjunto correto de padrões. Além disso, é preciso um mecanismo para identificar todas as mudanças que ocorreram entre as versões V_i e V_{i-1}, na função $mudanca(V_i)$. A ferramenta que implementar essa abordagem, deve prover uma forma automática para encontrar o conjunto de mudanças entre as versões V_i e V_{i-1}, além de uma formalização que será utilizada para verificar se as instâncias atendem as requisistos mínimos dos padrões.

Seguindo esta definição, quando um novo padrão é adicionado e não há qualquer modificação nas classes já existentes, apenas acréscimos de novas classes, ele não será considerado como padrão emergente. Isso pode ocorrer quando um padrão é planejado e inserido no código, diferentemente do que acontece quando um padrão emergente surge. O surgimento de um padrão emergente está diretamente relacionado a preservação do comportamento do código na versão anterior. Quando há alguma mudança no comportamento no código que originou o padrão emergente, isso pode significar que houve alguma modificação na essência do padrão, o que descaracterizaria o padrão emergente.

4.3 Exemplo da abordagem

Para exemplificar a abordagem proposta, inicialmente é necessário uma ferramenta para detectar as instâncias dos padrões. Na Tabela 4.1 são apresentadas as instâncias detectadas na versão 164 do projeto XMLUnit. Neste exemplo, será utilizada uma instância do padrão Adapter.

Na Tabela 4.1, a coluna *Tipo*, *Versão*, *Identificador* e *Padrão*, exibem respectivamente, se o padrão foi adicionado ou removido, qual a versão que a instância surgiu (ou foi deletada), um identificador único para cada instância dos padrões, e o nome do padrão. Na tabela pode ser visto que existem duas instâncias do padrão Adapter. A instância (ver diagrama na Figura 4.4) que será discutida neste exemplo possui o identificador 911909115, conforme visto na

Tabela 4.1: Instâncias detectadas na versão 164 do projeto XMLUnit

Tipo	Versão	Identificador	Padrão
ADD	164	-1601860227	Strategy
ADD	164	441251862	Adapter
ADD	164	911909115	Adapter

Tabela 4.1.

O próximo passo é comparar a versão 164 com a versão anterior. Após aplicado o critério de seleção, a versão anterior que é compilável e possui alguma mudança em arquivos java, foi a versão 161. Neste momento é necessário observar quais classes compõem a instância do padrão detectado. As classes podem ser vistas na Figura 4.4.

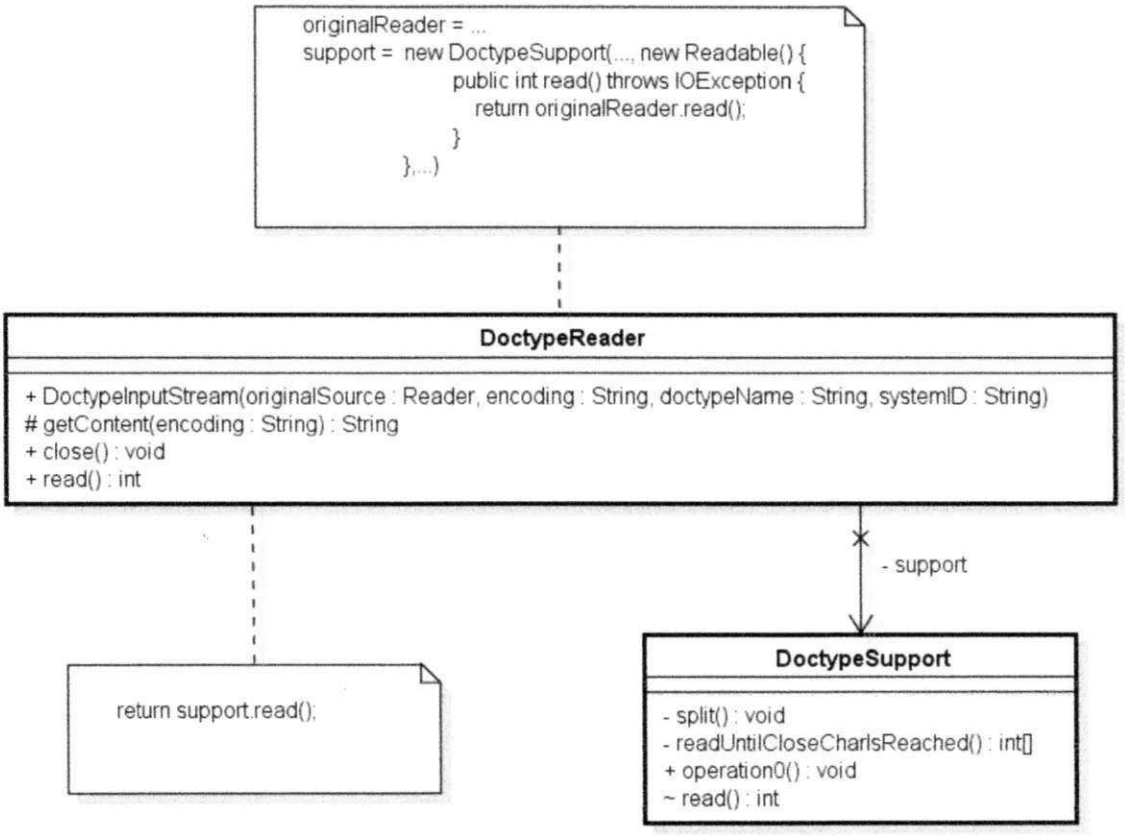

Figura 4.4: Instância do padrão Adapter na versão 164 do projeto XMLUnit

Identificadas as classes, o próximo passo é checar se alguma dessas classes pertence ao conjunto de mudanças. A Tabela 4.2, exibe as mudanças ocorridas na versão 164 do

projeto XMLUnit. As colunas *Tipo da mudança* e *Classe*, representam respectivamente, o tipo da mudança:*classe adicionada (A)*, *classe modificada (M)* e *classe deletada (D)*; e qual classe que foi modificada. Houve uma mudança na classe `DoctypeReader` e a execução do método `read()` passou a retornar o processamento do método `read()` da classe `DoctypeSupport`.

Tabela 4.2: Mudanças na versão 164, do projeto XMLUnit

Tipo da mudança	**Classe**
D	`org.custommonkey.xmlunit.DoctypeConstants`
A	`org.custommonkey.xmlunit.util.test.IntegerBuffer`
A	`org.custommonkey.xmlunit.AbstractDoctypeTests`
A	`org.custommonkey.xmlunit.util.IntegerBuffer`
M	`org.custommonkey.xmlunit.DoctypeSupport`
M	`org.custommonkey.xmlunit.test.DoctypeReader`
M	`org.custommonkey.xmlunit.DoctypeInputStream`
M	`org.custommonkey.xmlunit.DoctypeReader`
M	`org.custommonkey.xmlunit.Validator`
M	`org.custommonkey.xmlunit.test.DoctypeInputStream`

Como pode ser observado na Figura 4.4 e na Tabela 4.2, as classes `DoctypeReader` e `DoctypeSupport` foram modificadas e pertencem à instância do padrão Adapter. Nesse momento, é preciso analisar as mudanças e se na versão 161 as classes já possuíam o comportamento presente na versão 164. Esta etapa é realizada manualmente, confrontando as classes na versão 161 e 164. Observado que a mudança ocorrida é do tipo *classe modificada* e que o código foi extraído da classe `DoctypeReader` para `DoctypeSupport`.

Por fim, é necessário observar o porquê da instância não ter sido detectada na versão 161. Neste exemplo, a classe `DoctypeReader` possuia o comportamento, que foi migrado à classe `DoctypeSupport` (na versão 164), mas não possuía os elementos estruturais básicos definidos para o padrão, logo não foi identificada. Este é um exemplo de detecção de padrões emergente, pois o comportamento do padrão Adapter já existia, mas não a sua estrutura descrita pelo padrão.

4.4 Considerações sobre o capítulo

Neste capítulo, foram apresentados os padrões emergentes, além de formalizados utilizando lógica de predicados. Os padrões emergentes são caracterizados nas situações em que há o comportamento e as estruturas descritas pelo padrão não estão presentes. Também é apresentada uma abordagem para detecção dos padrões emergentes, e para que a abordagem seja executada é preciso: *(i)* um conjunto de versões de um sistema; *(ii)* uma especificação ou formalização para identificar e diferenciar os padrões; e, *(iii)* uma ferramenta que detecte os padrões em cada versão analisada. Um exemplo de padrão emergente foi detectado seguindo, passo a passo, a abordagem proposta.

Capítulo 5

Avaliação

Neste capítulo é descrito um estudo de caso exploratório utilizado para avaliar a abordagem proposta (Seção 4.2) para detecção dos padrões emergentes. Primeiramente, será apresentado o planejamento do estudo, caracterizando os projetos de código livre utilizados como objetos de estudos, os objetivos e questões de pesquisa. Posteriormente, a abordagem é executada e seus resultados discutidos.

Seguindo a abordagem proposta para detectar essas evidências, um protótipo de ferramenta foi implementado. Este protótipo realizou uma análise no histórico de quatro projetos de código aberto. Com este protótipo é possível visualizar, para cada versão do software, as novas instâncias dos padrões de projeto e quais modificações ocorreram nessa versão. Foram elaboradas três perguntas de pesquisa, todas relacionadas às evidências dos padrões emergentes. A primeira questão tem por objetivo identificar evidências dos padrões emergentes nos projetos analisados. Já a segunda questão de pesquisa busca analisar se em projetos com uma maior quantidade de classes, há mais padrões emergentes. Por fim, a terceira questão de pesquisa estuda se existe uma correlação entre o número de mudanças entre versões e a quantidade de novas instâncias de padrões emergentes. Após realizada a avaliação foram encontrados 17 padrões emergentes.

5.1 Contexto

O principal objetivo deste estudo é analisar evidências dos padrões emergentes, com a proposta de caracterizar evidências de padrões emergentes em sistemas de software. O enfoque

da qualidade está na precisão de detecção das ferramentas, em novas instâncias dos padrões. Enquanto que a perspectiva é de pesquisadores, que querem avaliar a eficácia das informações históricas para identificar evidências dos padrões emergentes. Para a execução deste estudo exploratório foi desenvolvido um protótipo de ferramenta, que possibilitou gerar a base de dados utilizada para avaliação, além de dispor de uma interface gráfica para auxiliar no processo manual de deteção de padrões, exibindo uma lista de padrões adicionados a uma versão, uma descrição individual de cada padrão, e uma lista de mudanças que ocorreram na versão. Essa ferramenta foi implementada com base na abordagem proposta.

Foi feita uma análise em quatro projetos de código aberto disponíveis em repositórios públicos. Fontes de projetos de código aberto foram exploradas, tais como Ohloh[1], SourceForge[2] e Open Source Projects[3]. No entanto, uma das etapas necessária para execução da ferramenta é o processo de compilação. Desta forma, os projetos deveriam ser compiláveis. Após essa busca, foram utilizados os seguintes projetos: DbUnit[4], XMLUnit[5], JFreechart[6] e XStream[7]. Na Tabela 5.1 são descritas algumas informações sobre os projetos analisados.

Tabela 5.1: Projetos utilizados

Projeto	**Versão** (inicial-final)	**Classes** (inicial-final)	**Data** (inicial-final)	**KLOC** (inicial-final)
XmlUnit	3 - 543	6 - 102	20/03/2001 - 12/09/2013	0.4 - 16.8
XStream	2- 579	55 - 326	26/09/2003 - 25/04/2005	2.1 - 22.3
DbUnit	1089 - 1259	296 - 610	03/11/2009 - 19/10/2012	44.5 - 91.8
JFreechart	1 - 99	158 - 711	19/06/2007 - 03/07/2007	51.0 - 238.8

O DbUnit é uma biblioteca que estende o JUnit e é destinada a projetos que utilizam banco de dados. O objetivo dessa biblioteca é garantir que o banco de dados permaneça em um estado conhecido para execução dos testes. O XMLUnit é uma biblioteca que estende o JUnit e permitir o teste de unidade de arquivos no formato XML. O JFreechart é uma

[1]https://www.openhub.net/

[2]http://sourceforge.net/

[3]https://opensourceprojects.eu/

[4]http://dbunit.sourceforge.net/

[5]http://xmlunit.sourceforge.net/

[6]http://www.jfree.org/jfreechart/

[7]http://xstream.codehaus.org/

biblioteca gráfica, em Java, que permite os desenvolvedores exibirem gráficos de qualidade profissional em suas aplicações. O XStream é uma biblioteca para serializar e deserializar objetos Java para XML.

5.2 Configurações do ambiente

A ferramenta desenvolvida foi executada em um computador com processador Intel Core i5 2.27GHz com 4GB de RAM, sob sistema operacional Windows 7 Home Premium. Foram utilizadas a JDK versão 1.7.0 e a IDE Eclipse Indigo. Para coleta das métricas sobre os projetos foi utilizado o Metrics, na versão 1.3.8. Metrics é um *plugin* para a IDE Eclipse que calcula várias métricas [35].

5.3 Metodologia

Com o intuito de identificar as evidências dos padrões emergentes, uma ferramenta (ainda em protótipo) foi desenvolvida, seguindo os passos descritos na abordagem proposta para detecção dos padrões emergentes, conforme descrita na Seção 4.2. A ferramenta executa dois processamentos básicos: extração e visualização. No diagrama de sequência da Figura 5.1 podem ser vistos, resumidamente, os passos necessários para execução da etapa de extração.

Primeiramente, a ferramenta extrai as informações das versões individualmente. Para cada versão é extraída uma lista de mudanças do repositório e uma lista de padrões. Cada mudança contém o tipo (Modificado, Deletado ou Adicionado), a data e o arquivo que foi afetado por esta mudança. Cada padrão possui uma lista de instâncias, que representam as ocorrências dos padrões, por exemplo, duas ocorrências do padrão Decorator são relacionadas a duas instâncias deste padrão. Após a extração destas informações, a ferramenta as exporta para um arquivo no formato XML. Este arquivo no formato XML contém as informações sobre as versões e a qual projeto elas pertencem. Com este arquivo é possível visualizar as informações históricas provenientes do repositório.

A segunda etapa consiste na visualização das instâncias dos padrões que foram extraídos na etapa anterior. O protótipo dispõe de uma interface gráfica para auxiliar a inspeção manual sobre essas instâncias. Como resultado desta inspeção, uma lista de instâncias foi

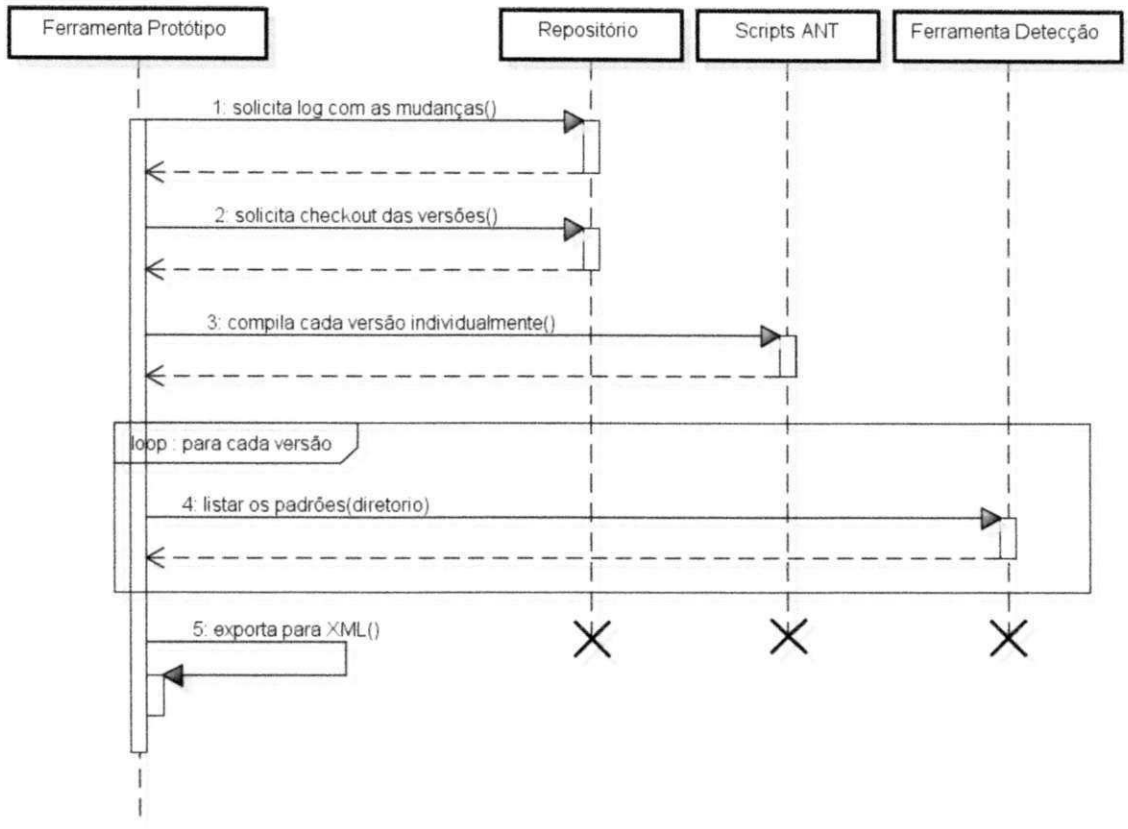

Figura 5.1: Algoritmo para execução do protótipo desenvolvido

selecionada. Cada instância possui características de um padrão emergente.

5.4 Ferramental de suporte

O principal objetivo desde trabalho é identificar evidências dos padrões emergentes. Para isso, serão extraídas informações do histórico de mudanças derivado de um sistema de controle de versão. Seguindo a abordagem proposta, foi desenvolvido um protótipo na linguagem Java e, de uma forma geral, a sua execução é realizada em duas etapas: extração e visualização.

5.4.1 Etapa de extração

A primeira etapa consiste em realizar o procedimento automático desde a extração do código até exportação dos dados em um arquivo no formato XML. Na Figura 5.2 podem ser identi-

ficados os passos adotados para a primeira etapa, onde é executada a extração e detecção dos padrões em um repositório SVN[8].

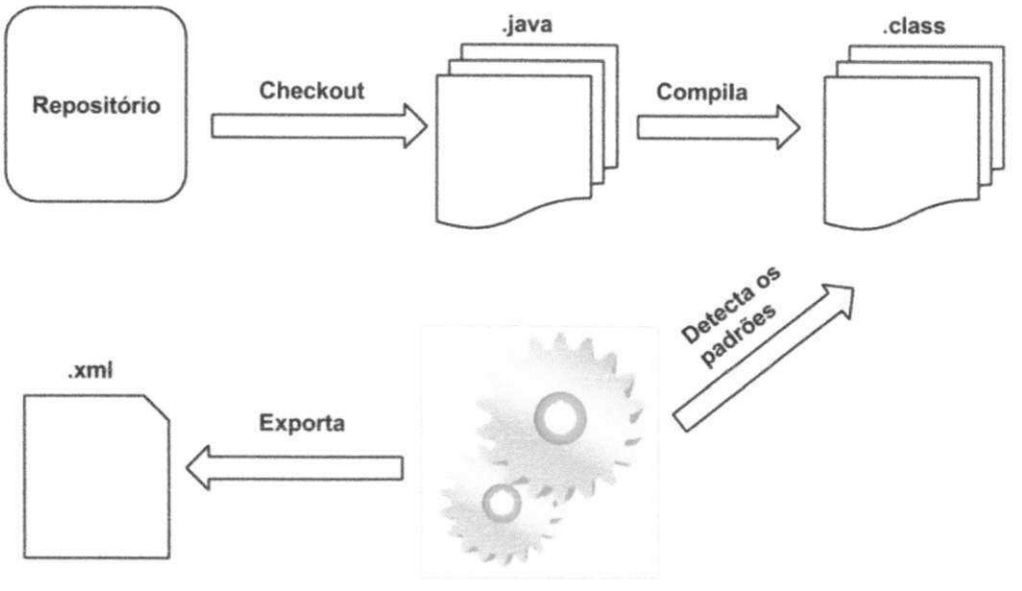

Figura 5.2: Etapas para extração das informações

O primeiro passo a ser realizado é extrair uma lista de versões com suas repectivas mudanças. Com base nessa lista de versões, são filtradas as versões em que não houve qualquer mudança em arquivos Java (com extensão .java). Essa filtragem é útil, pois será realizado o *ckeckout* somente das versões que contenham alguma modificação em arquivos Java. Posteriormente, é realizado o *ckeckout* do código para a máquina local e são extraídas algumas informações referente às mudanças ocorridas no repositório. Neste passo é utilizada a biblioteca SVNKit 1.7.4 [9] para manipular as operações de checkout e log do repositório. O SVNKit é uma biblioteca em Java que implementa os recursos do SVN e fornece uma API para acessar e manipular repositórios SVN de uma aplicação Java.

No segundo passo, os arquivos Java são compilados por *scripts* configurados pelo Apache Ant[10], que é uma biblioteca em Java e uma ferramenta em linha de comando utilizada

[8] https://subversion.apache.org/

[9] http://svnkit.com/

[10] http://ant.apache.org/

principalmente para a construção de aplicações Java. Porém, este processo para compilar os arquivos é complexo, pois *(i)* as dependências das classes podem não estar presentes no *classpath* da aplicação; *(ii)* os arquivos possuem código não-compilável (faltam: ";", ")", "*not a statement*"); *(iii)* os arquivos usam palavras reservadas de um código que foi desenvolvido em uma versão anterior da linguagem Java, por exemplo: *enum* que na versão 1.5 do Java passou a ser uma palavra reservada; e, *(iv)* caso se trate de versões iniciais, elas podem ser utilizadas para configurações de ambiente ou de projeto.

No entanto, o maior problema identificado foram as dependências dos projetos, pois as versões podem possuir dependências diferentes. Para solucionar este problema, as dependências quando conhecidas eram gerenciadas pelo Maven[11] e, caso contrário, era realizada uma busca sobre quais dependências estavam em falta.

Para o terceiro passo, foi utilizada uma ferramenta SSA[12] de detecção de padrões, que manipula os *bytecodes* e detecta os padrões através de uma análise de similaridade [49]. Neste passo, para cada padrão uma lista de instâncias é detectada. Cada instância representa uma ocorrência de um determinado padrão.

No quarto passo, após a extração dos padrões e suas respectivas listas de instâncias, as informações sobre as versões são exportadas para um arquivo no formato XML, que contém uma representação do projeto. Este arquivo contém as informações básicas sobre o projeto como url de acesso ao repositório e nome do projeto, além de uma lista de versões do projeto. Cada versão armazena uma lista de mudanças e uma lista de padrões detectados nessa versão.

5.4.2 Etapa de visualização

Nesta etapa é realizada uma inspeção visual com o intuito de identificar evidências dos padrões emergentes. Para este fim, a ferramenta dispõe de uma interface gráfica para auxiliar neste processo, como mostrado na Figura 5.3. A inspeção é um procedimento necessário pois é preciso comparar as versões das classes afim de verificar se houve mudança em seu comportamento.

Como mostra a Figura 5.3, esta interface está organizada em três áreas. A primeira é responsável por listar informações sobre o padrão: o tipo que determina se ele foi adicionado

[11] http://maven.apache.org/

[12] http://java.uom.gr/ nikos/pattern-detection.html

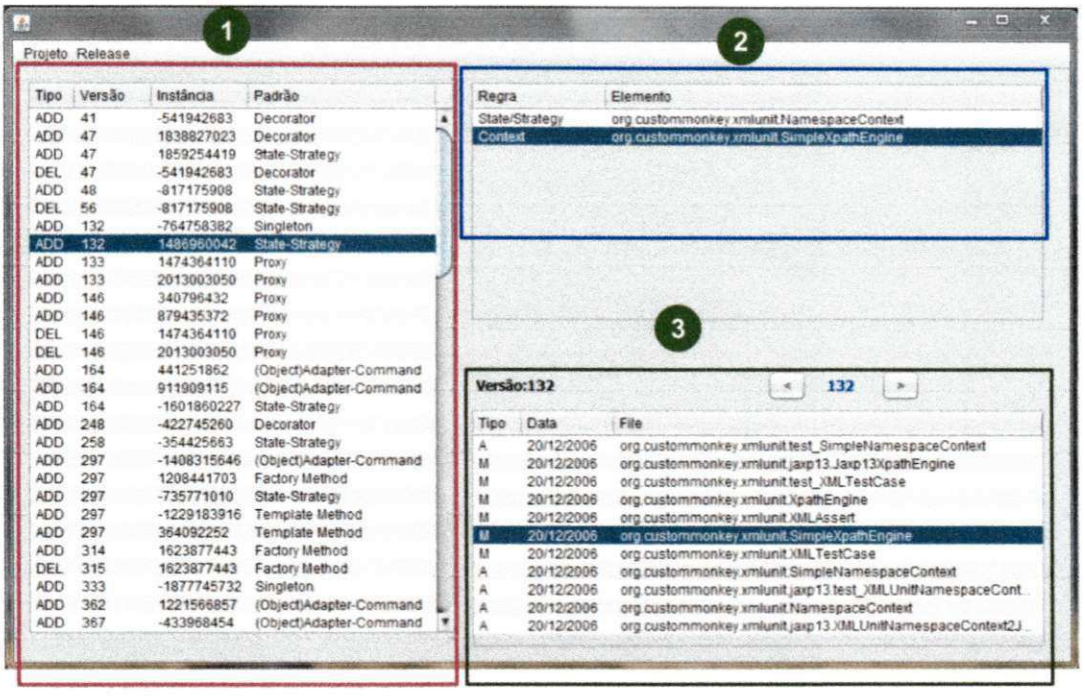

Figura 5.3: Interface do protótipo durante o processo de inspeção

ou removido (coluna Tipo), a versão do arquivo Java que foi analisado (coluna Versão), um identificador único para cada instância do padrão (coluna Instância) e o nome do padrão (coluna Padrão). Com essa área é possível visualizar quando um novo padrão foi adicionado. A segunda área mostra as informações individuais de cada instância, como quais classes e métodos fazem parte do padrão. Já a terceira área apresenta todas as modificações que ocorreram na versão em que o padrão foi adicionado, ou removido. Para cada modificação é exibido o tipo (Merge, Add ou Delete), a data da modificação e o arquivo que foi modificado.

Para um melhor entendimento da interface gráfica, será apresentado um exemplo. Conforme ilustrado na área 1, na versão 132 foi adicionada uma nova instância (1486960042) do padrão Strategy. Quando selecionada essa instância, a área 2 exibe quais classes (`SimpleXpathEngine` e `NamespaceContext`) fazem parte deste novo padrão. O próximo passo é observar na área 3 se ambas as classes foram adicionadas nesta versão ou se houve alguma modificação em uma classe existente. Neste exemplo, a classe `NamespaceContext` foi adicionada nesta versão, mas a classe `SimpleXpathEngine` foi modificada. Neste momento é necessário observar o código desta classe na versão anterior e observar se o comportamento da classe, que foi criada, já estava presente na versão

anterior. Caso o comportamento que antes existia na classe `SimpleXpathEngine` passou a existir na classe `NamespaceContext` e, na versão seguinte, surgiu um novo padrão, este é um padrão emergente.

Essa inspeção se torna mais complexa quando entre as versões analisadas existem uma grande quantidade de versões não compiladas, pois a mudança na classe pode ter ocorrido no intervalo entre as versões compiláveis. Isto aumenta a quantidade de modificações que precisam ser exploradas para caracterizar uma evidência de um padrão emergente.

5.5 Questões de Pesquisa

Nesta seção será descrito sobre as questões de pesquisa levantadas para analisar os padrões emergentes. Para cada das três questões de pesquisa é proposta uma análise. A primeira análise tem o objetivo de identificar evidências dos padrões emergentes e se as instâncias detectadas na abordagem são, de fato, padrões emergentes. Na segunda análise, são analisadas se existe alguma relação entre o número de classes e novas instâncias. Por fim, é realizado um estudo com o objetivo de identificar se existe uma correlação entre a quantidade de mudanças efetuadas em uma versão com a quantidade de novas instâncias de padrões.

A primeira questão de pesquisa *(QP1)* está organizada conforme o formato GQM *(Goal, Question e Metric)*, descrito por Wohlin *et al.* [53].

Objetivo: O principal objetivo é identificar evidências dos padrões emergentes, com base nas informações do histórico de mudanças derivado de um sistema de controle de versão.

Questão de Pesquisa: Quantos casos de padrões emergentes (que foram, em algum momento, adicionados no histórico do projeto) existem nestes projetos open source?

Métrica: Será utilizada a função para detecção formalizada na seção 4.2. Em um conjunto sequencial de versões $V = \{v_1, .., v_n\}$, a função que identifica um padrão formalizado, e pertencente ao conjunto $J = \{j_1, ..., j_n\}$, é definida por:

$$DeteccaoP_j(V_{i+1}) = \begin{cases} 0, & \text{se } \forall y |\ y \in instancia(V_{i+1}) \ e \ y \in instancia(V_i); \\ x, & \text{se } \forall y |\ y \in instancia(V_{i+1}) \ e \ y \notin instancia(V_i). \end{cases}$$

Onde:

$$\forall x |\ x = \begin{cases} 0, & \text{se } mudanca(V_i) = 0; \\ 1, & \text{se } mudanca(V_i) > 0. \end{cases}$$

E as funções $instancia(V_i)$ e $mudancao(V_i)$, retornam respectivamente, um conjunto de instâncias do padrão P_j na versão V_i e o número de mudanças que ocorreram na versão V_i.

O objetivo desta questão de pesquisa visa identificar evidências dos padrões emergentes. Há evidências quando uma classe é modificada e seu comportamento não é alterado entre as versões. Nestes casos, serão observados se os padrões foram planejados (projetados) e inseridos no software, ou são decorrência de um comportamento emergente já existente e modificado para a estrutura do padrão.

Para responder esta questão de pesquisa, será a utilizado o protótipo desenvolvido para a abordagem proposta. Serão utilizados projetos reais de código aberto. Individualmente, cada versão do projeto foi analisada com sua versão anterior e posterior, respectivamente. Essa análise aos pares foi útil para identificar os cados onde houve alguma modificação e, também, o surgimento de um novo padrão de projeto. Essa avaliação foi feita por um estudante de Mestrado da Universidade Federal de Campina Grande, onde para evitar qualquer tipo de análise subjetiva ou viés, quando encontradas evidências de um padrão emergente, era necessário recorrer a formalização descrita na Seção 4.2, para avaliar a existência do padrão emergente. Por esta razão, deve existir uma especificação dos padrões e ser mantida com o maior formalismo possível.

A segunda questão de pesquisa *(QP2)* visa analisar se o número de novas instâncias está relacionando a quantidade de classes existentes nos projetos. O objetivo dessa questão de pesquisa é observar se projetos que possuem muitas classes possuem, também, muitas instâncias de padrões emergentes. Apesar de intuitivo, não há uma premissa sobre o surgimento e existência dos padrões emergentes. Desta forma, se faz necessário observar se há alguma relação entre o número de classes e o número de novas instâncias de padrões de projeto surgirem no software.

Por fim, o objetivo da terceira questão de pesquisa *(QP3)* é avaliar se a quantidade de mudanças ocorridas em uma versão possui correlação com o número de novas instâncias de padrões. Os dois fatos que podem influenciar na evidência de um padrão emergente é a existência de uma nova instância de um padrão de projeto, e a mudança que resultou essa nova

instância. Essa terceira questão de pesquisa *(QP3)* objetiva avaliar se em projetos que ocorrem muitas mudanças existem muitos padrões emergentes, ou se existe alguma correlação entre esses fatos.

5.6 Análise dos Resultados

Nesta seção são discutidos os principais resultados encontrados. A abordagem proposta encontrou 22 evidências de padrões emergentes, confirmando a existência de 17 instâncias de padrões emergentes.

Seguindo a abordagem proposta, foram realizados 830 *checkout* de versões, 809 versões foram manualmente compiladas com sucesso e um total de 780 instâncias de padrões, adicionados ou deletados, foram analisados, conforme visto na Tabela 5.2. Foram realizadas três análises relacionadas às instâncias dos padrões. Cada análise será discutida nas subseções seguintes.

Tabela 5.2: Informações sobre a execução do experimento

Projeto	**Versão** (inicial-final)	**Classes** (inicial-final)	**Data** (inicial-final)	**KLOC** (inicial-final)
XmlUnit	3 - 543	280	276	93
XStream	2- 579	392	382	531
DbUnit	1089 - 1259	80	79	21
JFreechart	1 - 99	78	72	135
Total		**830**	**809**	**780**

5.7 Primeira análise: Identificar os Padrões Emergentes

Essa primeira análise visa avaliar se as instâncias selecionadas, após a execução da abordagem proposta, podem ser classificadas como um padrão emergente. Cada instância foi selecionada observando se possui o comportamento do padrão e se ocorreu alguma modificação na classe, seja modificada ou adicionada. Foram analisadas 780 instâncias (ver Tabela 5.3) e, após uma filtragem manual, foram selecionadas 22 instâncias(ver Tabela 5.4).

Tabela 5.3: Resultados Gerais.

Projeto	**Diferença no número de padrões**	**Instâncias**
XmlUnit	49	93
XStream	89	531
DbUnit	13	21
JFreechart	41	135

Deste total de instâncias analisadas, no projeto JFreechart 41 novos padrões surgiram, e 135 instâncias foram analisadas, porém nenhuma instância foi selecionada. Neste caso, as classes são adicionadas e já fazem parte do padrão. Por exemplo, a classe `UnitType`, no projeto JFreechart, foi adicionada na versão 44. E, quando foi adicionada já era uma instância do padrão Singleton, pois possui em sua estrutura, basicamente: *(i)* uma variável estática do mesmo tipo da classe; *(ii)* um construtor privado, para não ser instanciado por outras classes; e, *(iii)* uma constante publica e estática, para dar acesso a esta variável, conforme visto na Figura 5.4.

Além dos casos onde as classes adicionadas já fazem parte de um padrão, outra situação que pode ajudar na diminuição de instâncias selecionadas no projeto, é a presença de falso-positivos. E essa situação está relacionada com qual ferramenta de detecção de padrão for utilizada na abordagem proposta. Por exemplo, com a ferramenta de detecção utilizada a classe `Licences` foi classificada como um Singleton (ver Figura 5.5). Porém, a classe possui o construtor público, com isso outras classes podem instanciá-la diretamente, assim a classe não poderia ser classificada como Singleton.

Desta forma, mesmo possuindo um quantitativo de 809 versões compiladas e um número de 780 instâncias analisadas, apenas 22 foram selecionadas. Todas as instâncias selecionadas estão listadas na Tabela 5.4. Dentre essas instâncias, 17 pertencem ao projeto XStream. Isso ocorre devido a grande quantidade de versões compiladas (382) e número de instâncias analisadas (531) presentes no projeto XStream, que possui tanto um numero de versões compiladas quanto de instâncias analisadas maior que a soma de todos os demais projetos juntos.

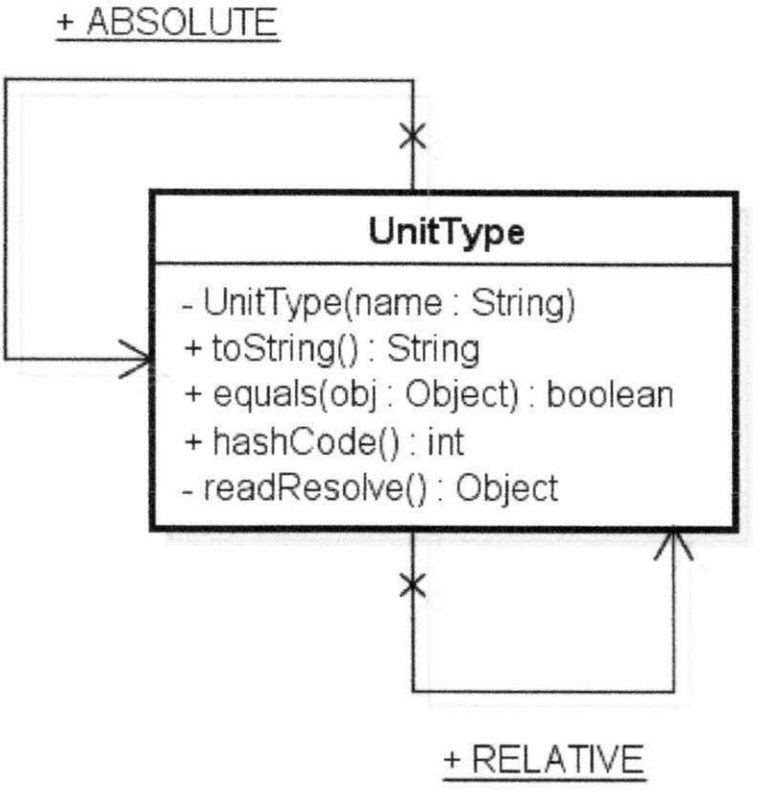

Figura 5.4: Padrão Singleton no Projeto JFreechart

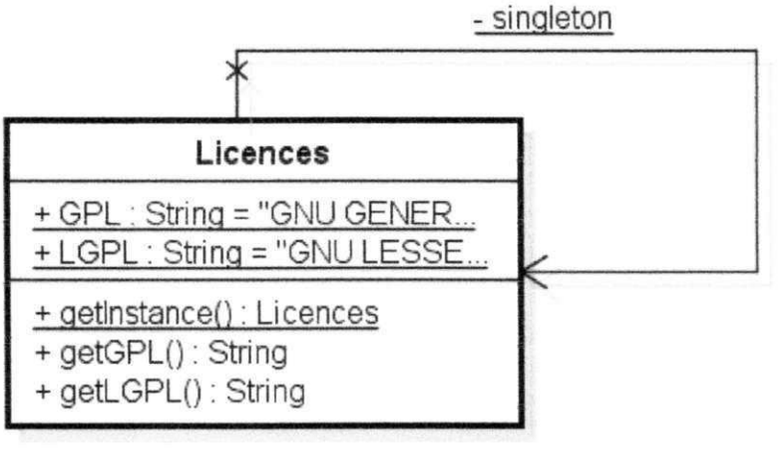

Figura 5.5: Falso-positivo do padrão Singleton

uma implementação de `DomXMLReaderDriver`. Além disso, o processamento deixou de ser executado na classe `XStream`, na versão 2 e a classe `DomXMLReaderDriver` passou a ser responsável por este processamento. Desta forma, surgiu uma nova ocor-

rência do padrão Strategy (ver Figura 5.6) e com as características de um padrão emergente, pois o comportamento antes executado na classe `XStream` foi movido para a classe `DomXMLReaderDriver`.

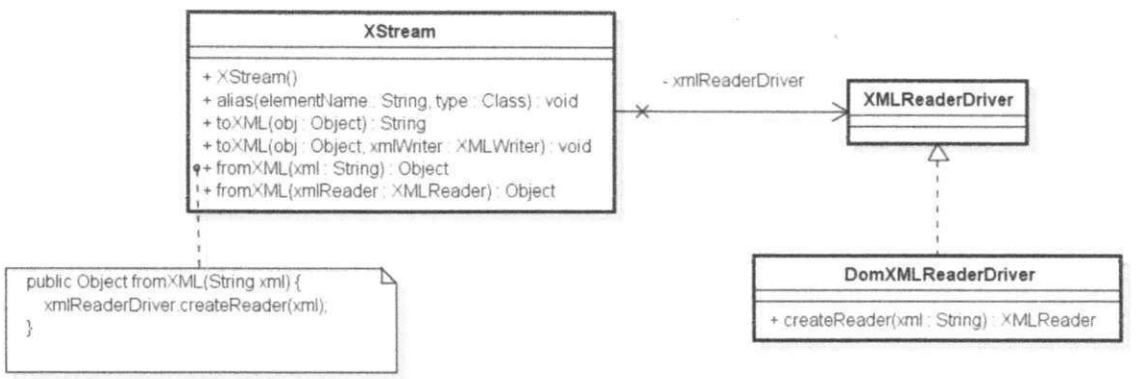

Figura 5.6: Ocorrência do Padrão Strategy

Outro padrão emergente ocorre na versão 164 do projeto XMLUnit com a classe `DoctypeSupport`. Houve uma mudança e o comportamento presente na classe `DoctypeSupport` na versão anterior, foi movido para a classe `IntBufferReadable`, surgindo, assim, uma nova instância do padrão Strategy, ver Figura 5.7.

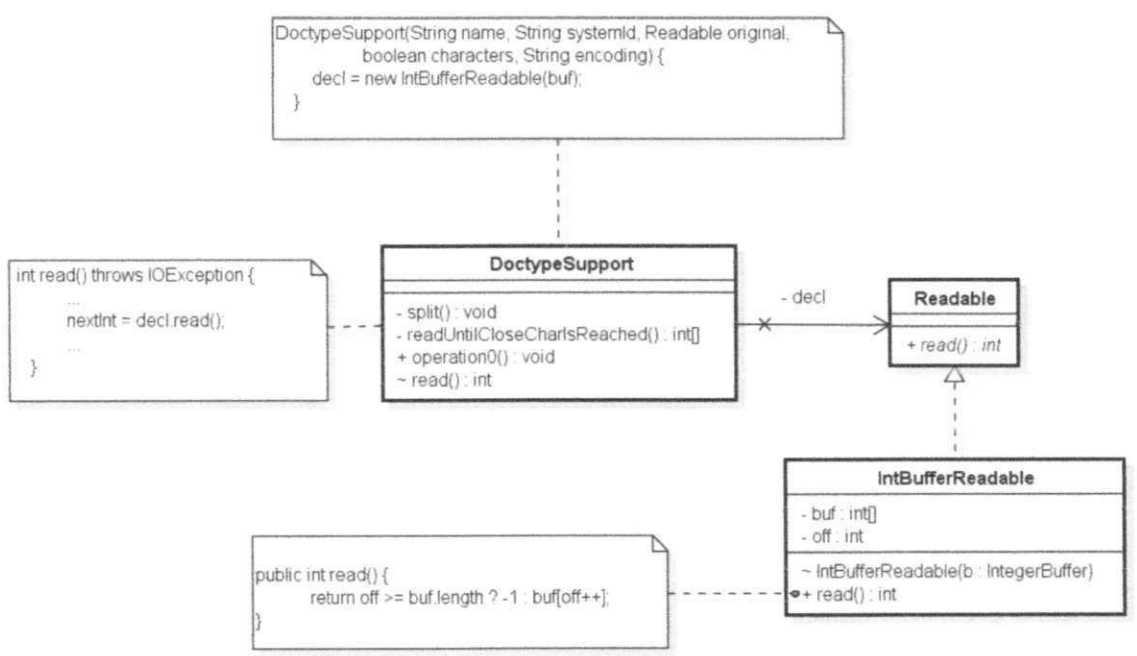

Figura 5.7: Padrão Strategy no Projeto XMUnit na versão 164

Nessa mesma versão no projeto XMLUnit, surgiu uma nova instância do padrão

Adapter. A classe `DoctypeInputStream` foi modificada e a execução do método `read()` passou a retornar, como resultado, o processamento do método `read()` da classe `DoctypeSupport`, conforme visto na Figura 5.8. E o fluxo de chamadas ao método `read()` pode ser visto no diagrama de sequência representado na Figura 5.9.

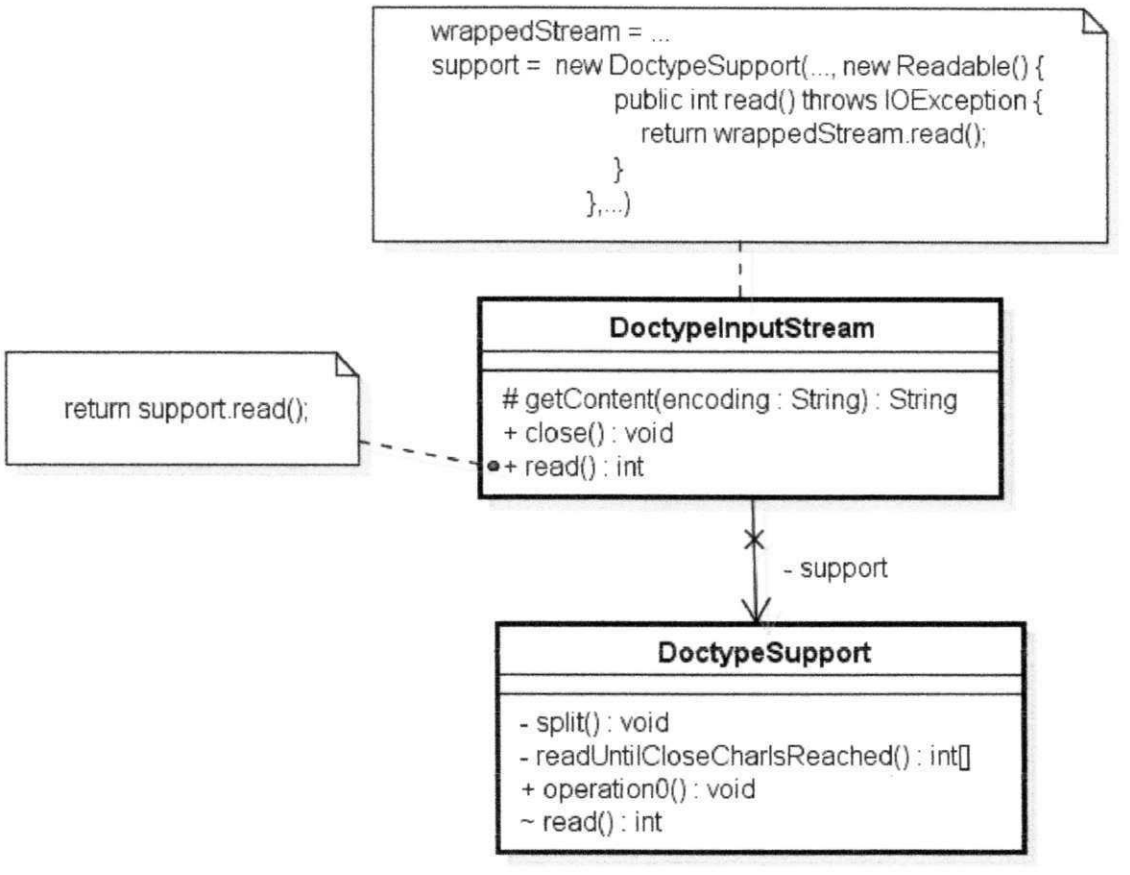

Figura 5.8: Padrão Adapter no Projeto XMUnit na versão 164

5.8 Segunda análise

A primeira análise realizada identificou que a quantidade de instâncias analisadas no projeto XStream é quase quatro vezes maior que o número de instâncias analisadas no projeto JFreechart (ver Tabela 5.1), mesmo este possuindo metade do número de classes, considerando a última versão analisada (ver Tablea 5.1), do projeto JFreechart.

Inicialmente, podem haver dois motivos básicos. O primeiro está relacionado à quantidade de padrões em cada projeto. Porém, quando comparados os número de padrões exis-

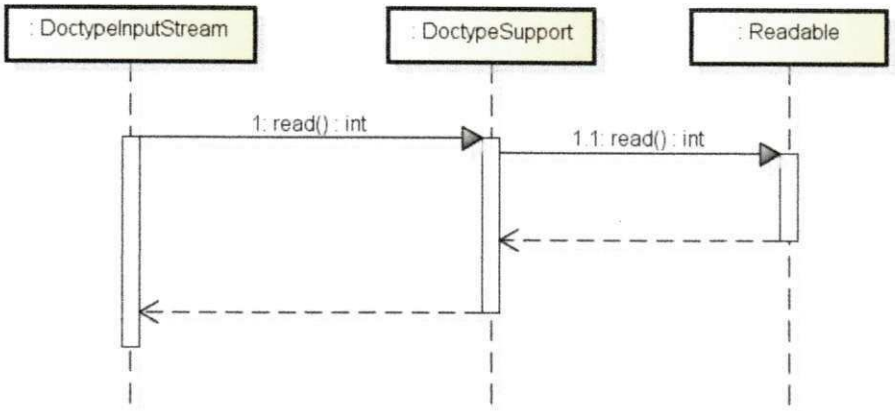

Figura 5.9: Fluxo de chamadas ao método read() - Padrão Adapter no Projeto XMUnit

tentes em cada projeto (ver Figura 5.10), percebe-se que apenas a quantidade de padrões não é um fator crucial para determinar o número de instâncias a serem analisadas.

Projeto	Versão	Factory Method	State-Strategy	(Object)Adapter-Command	Template Method	Prototype	Singleton	Composite	Decorator	Observer	Proxy	Total
XmlUnit	543	3	17	14	4	0	4	0	5	0	2	49
XStream	579	1	39	49	3	0	1	0	5	0	0	98
DbUnit	1259	8	70	41	12	0	9	0	24	0	1	165
JFreechart	99	10	52	42	7	73	36	1	2	4	12	239

Figura 5.10: Número de padrões existentes na última versão

O segundo motivo que explicaria essa divergência no número de instâncias entre os projetos XStream e JFreechart é a variância entre o número de padrões existentes na última e a versão inicial compilável. Desta forma, apenas as novas ocorrências de padrões serão analisados, seja de padrões adicionados ou deletados. Observando a Figura 5.11 e analisando a diferença entre o número de padrões da última e a versão inicial, compreende-se que o projeto vai possuir muitas instâncias a serem avaliadas se houve um considerável acréscimo de novos padrões.

Projeto	Factory Method	State-Strategy	(Object)Adapter-Command	Template Method	Prototype	Singleton	Composite	Decorator	Observer	Proxy	Total
XmlUnit	3	17	14	4	0	4	0	5	0	2	49
XStream	0	33	48	2	0	1	0	5	0	0	89
DbUnit	2	3	3	1	0	4	0	0	0	0	13
JFreechart	2	11	10	0	0	15	0	0	0	3	41

Figura 5.11: Variação entre o número de padrões existentes entre última e a versão inicial

Quando observado o número de padrões já existentes na versão inicial dos projetos (ver Figura 5.12) verifica-se que é preciso escolher tanto os projetos quanto o intervalo de versões que serão analisadas. Pois, como citado anteriormente, apenas a quantidade de padrões não determina o número de instâncias que serão analisadas. Quanto ao intervalo das versões, se a versão inicial for alta, provavelmente, já vai existir um número considerável de padrões, diminuindo a quantidade de instâncias a serem avaliadas. Por exemplo, o projeto DbUnit possui apenas 21 instâncias a serem avaliadas e sua versão inicial é a 1089. Por outro lado, os demais projetos contemplam o intervalo inicial dos projetos, possuindo as primeiras versões e consequentemente um número maior de instâncias a serem observadas.

Projeto	Versão	Factory Method	State-Strategy	(Object)Adapter-Command	Template Method	Prototype	Singleton	Composite	Decorator	Observer	Proxy	Total
XmlUnit	3	0	0	0	0	0	0	0	0	0	0	0
XStream	2	1	6	1	1	0	0	0	0	0	0	9
DbUnit	1089	6	67	38	11	0	5	0	24	0	1	152
JFreechart	1	8	41	32	7	73	21	1	2	4	9	198

Figura 5.12: Número de padrões existentes na versão inicial

5.9 Terceira Análise: Correlação entre Mudanças e Instâncias

A terceira análise a ser realizada é avaliar se a quantidade de mudanças ocorridas em uma versão, possuem correlação com o número de novas instâncias de padrões. Intuitivamente, quanto maior o número de mudanças em um código mais provável existir uma nova instância de um padrão.

Inicialmente, foi analisada a dispersão das mudanças e instâncias. Para exemplificar, são descritos os dados do projeto JFreechart, porém os quatro projetos avaliados possuem resultados similares. Na Figura 5.13 pode-se notar que há uma dispersão entre as mudanças, ou seja, os dados não estão concentrados em uma região do gráfico. Por outro lado, a quantidade de instâncias permanece praticamente sem qualquer alteração.

Posteriormente, foram sobrepostos os gráficos referentes a quantidade de mudanças e padrões, representados por linhas em vermelho e azul, respectivamente, na Figura 5.14. Com exceção do intervalo entre as versões 26 e 48, as mudanças aparentemente não possuem uma

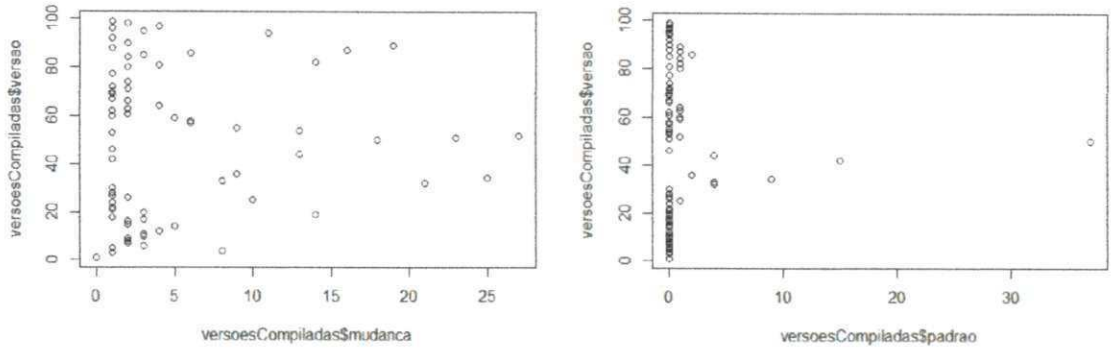

Figura 5.13: Mudanças x versão - Projeto JFreechart

correlação com o número de instâncias.

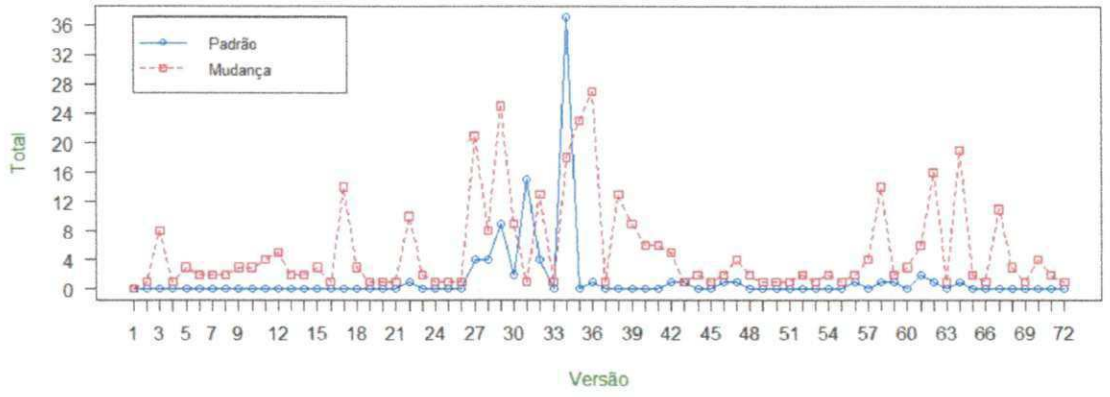

Figura 5.14: Novas instâncias e quantidade de mudanças

No entanto, sempre que ocorre uma nova instância há uma mudança relacionada. E isso se deve ao fato de uma nova instância sempre ocorrer devido a mudança em alguma classe do projeto, quando visualizada a Figura 5.15, isso é evidenciado.

Após essa análise, foi verificado se existe uma correlação entre a quantidade de mudanças e o número de instâncias. Cada projeto foi analisado individualmente e os dados referentes a essa análise estão sumarizado e podem ser conferidos na Tabela 5.5.

Se observados os dados da função cor, que calcula a correlação entre as variáveis, pode-ser notar que a porcentagem é pequena, entre 0,08 a 0,44, de correlação entre a quantidade de mudanças e o número de instâncias. Essa porcentagem pequena se deve ao fato da di-

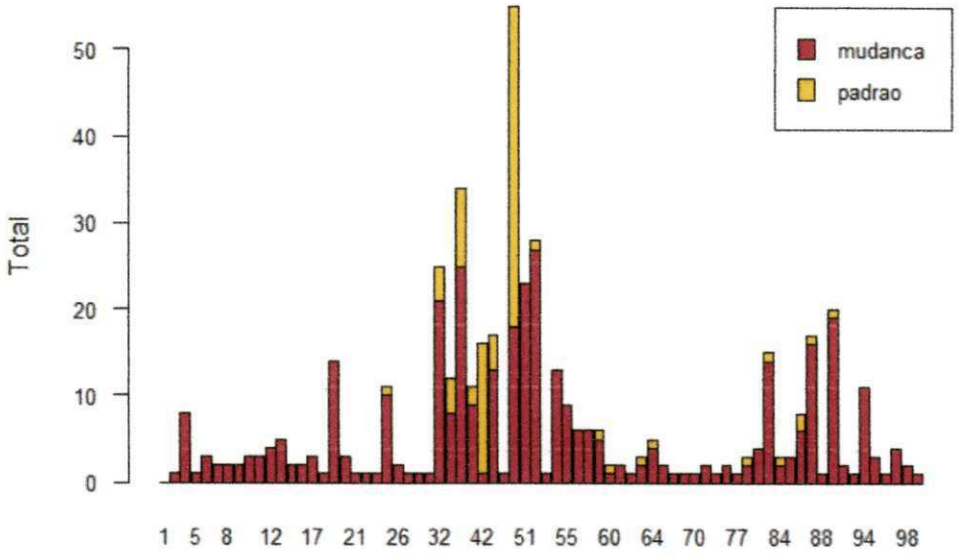

Figura 5.15: Relação entre as novas instâncias e as mudanças

Tabela 5.5: Correlação entre o número de mudanças e padrões

Projeto	**cor**	**Coeficientes**	**R-squared**	**mudanças**
XmlUnit	8544766	y = 0.012456x + 0.210403	37	1038
XStream	4489353	y = 0.18693x + 0.00752	1994	1643
DbUnit	1110681	y = 0.008641x + 0.183034	-5	294
JFreechart	3422416	y = 0.25647x -0.11713	1045	376

ficuldade em predizer o número de instâncias analisando apenas um único fator, no caso a quantidade de mudanças. E, quando analisada o modelo de regressão de cada projeto, percebe-se que os pontos se afastam da reta, indicando não existir uma correlação entre as variáveis, por exemplo, o modelo para o projeto JFreechart visto na Figura 5.16. A regressão linear tenta predizer uma função que determine o número de instâncias a partir da quantidade de mudança.

Analisando apenas o modelo de regressão do projeto JFreechart, a correlação positiva de 0.34 é pequena, e o modelo possui apenas 10% de confiança. Ou seja, com 10% de confiança

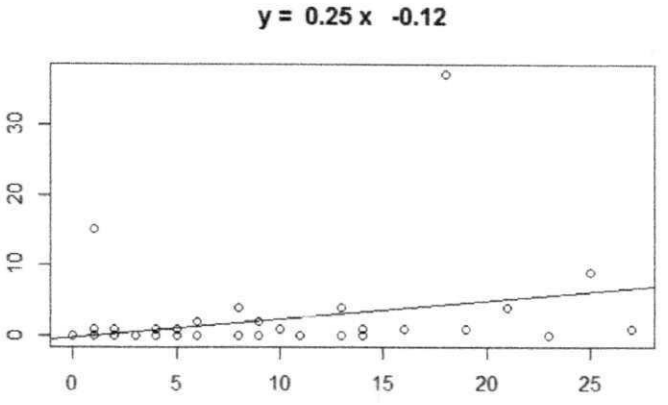

Figura 5.16: Modelo de regressão linear do projeto JFreechart

o modelo pode predizer o número de instâncias a partir do número de mudanças. Por tanto, devido a pouca correlação entre a quantidade de mudanças e o número de instâncias, há indícios de não haver uma correlação forte entre essas variáveis.

5.10 Ameaças à validade

Não houve um Experimento para comparar as ferramentas, pois não existe: *(i)* uma ferramenta que detecte os padrões emergente; *(ii)* uma base de dados com padrões emergentes; *(iii)* Não existia uma especificação para integrar com as ferramentas para a detecção. Como o processo é semi-automático, a verificação das instâncias com a formalização pode ser influenciada pela experiência de quem a estiver realizando. Durante a análise manual, a avaliação é subjetiva e isso pode influenciar os resultados da avaliação. Além disso, a ferramenta escolhida para a detecção de padrões pode não capturar corretamente instâncias verdadeiras e classificar instâncias que não atendem aos requisitos dos padrões. Outra ameaça refere-se a generalização dos resultados, pois a amostrar é pequena e a quantidade de projetos analisados não permite uma generalização a outros projetos.

5.11 Considerações sobre o capítulo

Neste capítulo, foi apresentada a avaliação da abordagem proposta. A abordagem proposta visa identificar evidências de padrões emergentes em código fonte. O processo de avaliação se deu por meio de um estudo exploratório, avaliando a abordagem em quatro projetos de

código aberto. Na avaliação, foi utilizado um protótipo implementado seguindo a abordagem descrita neste trabalho.

Uma ferramenta (ainda em protótipo) foi implementada e executada no histórico de quatro projetos de código aberto. Para observar se há evidência de padrões emergentes no código, é necessário analisar aos pares as versões do histórico de cada projeto. Desta forma, é possível comparar as versões e identificar quais versões possuem novas instâncias de padrões de projeto (para essa detecção foi utilizada a ferramenta SSA), além de um conjunto de mudanças que ocorreram em cada versão. Com o protótipo é possível visualizar este casos, onde há uma nova instância de um padrão de projeto e houve alguma modificação no código. A análise desta evidência é realizada comparando se em ambas as versões já existia o comportamento pertencente ao padrão.

Nesta avaliação não foi possível comparar as ferramentas levantadas na revisão sistemática. O principal motivo para isso é o fato das ferramentas não utilizarem uma padronização de entrada ou saída. Desta forma, simplesmente comparar valores entre as ferramentas seria imprudente. Essa falta de padronização dificultou o planejamento e execução da avaliação, pois a depender da técnica ou ferramenta escolhida seria preciso uma readaptação ao processo de avaliação. Neste sentido, outo fato que dificultou a avaliação foi a falta de uma métrica para mensurar os padrões detectados. Inicialmente, as métricas de precisão e cobertura foram levantadas para realizar este estudo. Porém, como o número de instâncias dos padrões emergentes era desconhecido, não foi viável aplicar a métrica de cobertura.

Foram planejadas três perguntas de pesquisa, todas relacionadas às evidências dos padrões emergentes. A primeira questão tem por objetivo identificar evidências dos padrões emergentes nos projetos analisados. Após execução da avaliação foram detectadas 17 instâncias de padrões emergentes. A detecção destes padrões emergentes auxilia os desenvolvedores na compreensão do software. Seja por falta de conhecimento, ou desatenção, alguns desenvolvedores podem não implementar o padrão conforme é descrito estruturalmente. Esse padrão não implementado estruturalmente, dificulta tanto a compreensão quanto a comunicação.

A segunda questão de pesquisa busca analisar se em projetos com uma maior quantidade de classes, há mais padrões emergentes. Após analisar essa questão de pesquisa não foi possível estabelecer alguma relação entre o número de classe de um projeto com a quantidade

de padrões emergentes. Este fato mostra que mesmo possuindo muitas classes, a inserção de um padrão de projeto pode ser planejada. Embora não analisada, outra possibilidade seria o fato de haver poucas modificações nas classes existentes, diminuindo assim a probabilidade de surgir um padrão emergente.

Por fim, a terceira questão de pesquisa estuda se existe uma correlação entre o número de mudanças entre versões e a quantidade de novas instâncias de padrões emergentes. Enquanto a segunda questão analisava as classes pertencentes ao projeto, esta questão estuda a quantidade de mudanças que ocorrem em cada versão do projeto a ser analisado. Após realizado o estudo de correlação entre as variáveis, não foi constatado existir qualquer causa entre as mudanças e as novas instâncias dos padrões emergentes. Durante o planejamento era esperado que quanto mais a quantidade de mudanças maior seria o número de padrões emergentes, mas com base nos resultados, essa hipótese não foi aceita.

Capítulo 6

Considerações Finais

Nesta dissertação foi apresentado o conceito de padrões emergentes através das seguintes contribuições: *(i)* uma revisão sistemática sobre abordagens automáticas de detecção de padrões de projeto, *(ii)* conceitos de padrões emergentes para vários padrões de projeto bem conhecidos, *(iii)* uma proposta de abordagem semi-automática de detecção de padrões emergentes e *(iv)* sua utilização para uma análise de ferramentas de detecção existente acerca de sua capacidade de identificação de padrões emergentes em alguns projetos de código aberto Java. A ideia inicial da abordagem é detectar padrões em um histórico de versões de um sistema, com isso é possível identificar em qual versão uma nova instância do padrão surgiu. O próximo passo é checar se a nova instância atende as características básicas do padrão, que são descritas por alguma formalização. Resumidamente, a abordagem necessita que existam: *(i)* um conjunto de versões de um sistema; *(ii)* uma especificação ou formalização para identificar e diferenciar os padrões; e, *(iii)* uma ferramenta que detecte os padrões em cada versão analisada. Além da abordagem, também, são apresentados os padrões emergentes. Padrão emergente é uma nomenclatura atribuída quando um conjunto de classes possui o comportamento de um padrão de projeto, mas não sua estrutura, e esse comportamento emerge durante a evolução do código.

Além disso, foi realizada uma revisão sistemática sobre detecção de padrões de projeto, com o objetivo de fornecer uma visão geral sobre a detecção de padrões de projeto. Em particular, observando como os padrões são detectados e identificados, se e como a padrões são formalizados, e em quais projetos foram aplicadas as técnicas para detecção de padrões. Após concluída a revisão, onze ferramentas foram identificadas e uma escolhida para a rea-

lização da avaliação.

O processo de avaliação se deu por meio de um estudo exploratório, avaliando a abordagem em quatro projetos de código aberto. Na avaliação, foi utilizado um protótipo implementado seguindo a abordagem descrita neste trabalho. O processo é executado em duas etapas, extração e visualização. Na etapa de extração, o protótipo extrai as informações do histórico do projeto, executa uma ferramenta de detecção de padrões e organiza as informações para cada versão, como as mudanças ocorridas e quais novas instâncias de padrões foram adicionados. A segunda etapa, é realizada uma checagem para classificar as instâncias selecionadas. Para essa checagem é preciso confrontar, manualmente, par-a-par as versões e observar se a nova instância é decorrente de algumas adição ou modificação nas classes que compõem a instância.

Os resultados apontam para a existência dos padrões emergentes. Na avaliação, foram identificadas 17 instâncias de padrões emergentes. Contudo, intuitivamente, era esperado um número maior de padrões emergentes, pois acreditava-se que seria comum encontrar situações onde já existia o comportamento dos padrões. Esse baixo número pode ser reflexo de como são identificados os padrões emergentes, pois as versões são observadas apenas quando a ferramenta de detecção identifica um novo padrão, excluindo assim, situações onde o comportamento do padrão existe, porém não foram introduzidas as estruturas que definem o padrão.

6.1 Contribuições

Resumidamente, as principais contribuições deste trabalho são:

A formalização do padrões emergentes: Por ser uma nomenclatura nova, foi preciso desenvolver uma formalização para identificar e diferenciar os padrões emergentes. Foram formalizados 9 padrões descritos pelo catálogo do GoF [18].

Uma abordagem para detecção dos padrões emergentes: Para detectar os padrões emergentes foi proposta uma abordagem semi-automática. Basicamente, é necessário um histórico de versões, uma formalização dos padrões e uma ferramenta que detecte os padrões. Uma ferramenta foi implementa seguindo essa abordagem e seguindo duas etapas, extração e visualização.

Uma revisão sistemática sobre detecção de padrões de projeto: No intuito de realizar um levantamento bibliográfico sobre a área de detecção de padrões de projeto, foi realizada uma revisão sistemática sobre abordagens automáticas de detecção de padrões de projeto, que reuniu inicialmente um total de 927 estudos, finalizando com um total 41 trabalhos classificados para leitura.

Um estudo exploratório sobre as técnicas de detecção existentes: A avaliação utilizou o protótipo desenvolvido e analisou quatro projetos opensource. Foram verificadas um total de 780 instâncias analisadas, 22 instâncias foram selecionadas e 17 padrões emergentes foram identificados.

6.2 Dificuldades encontradas

Inicialmente quando planejada a Revisão sistemática esperava-se uma quantidade menor de artigos selecionados para leitura. Ao fim do processo de seleção dos trabalhos, restaram 41 artigos, um número relativamente alto se considerado que a leitura dos artigos foi realizada por um único pesquisador, demandando um tempo bem elevado. Um problema comum aos trabalhos é a falta de uma padronização, tanto a formalização que descreve os padrões quanto das ferramentas de detecção. Cada trabalho formaliza os padrões de uma maneira diferente e isso pode acarretar em uma divergência nos resultados das avaliações. Outro problema é o fato das ferramentas não utilizarem uma padronização nem de entrada nem de saída. Desta forma, simplesmente comparar valores entre as ferramentas seria imprudente. Essa falta de padronização dificultou o planejamento e execução da avaliação, pois a depender da técnica ou ferramenta escolhida seria preciso uma readaptação ao processo de avaliação. Além disso, algumas ferramentas não estavam disponíveis para realizar qualquer tipo de avaliação.

Para implementação do protótipo, inicialmente, foi preciso definir qual ferramenta seria utilizada para detecção dos padrões. Após essa escolha, foi preciso adaptar as entradas aceitas pela ferramenta e compreender com extrair as informações da saída da ferramenta. Este procedimento demandou muito tempo, principalmente porque a ferramenta escolhida, a SSA, realiza instrumentação de *bytecode*, ou seja, é preciso que o código a ser analisado seja compilável.

O protótipo necessita extrair informações de um histórico do projeto. Foi implementado

um algoritmo capaz de recuperar essas informações de um repositório SVN. Essa atividade também foi custosa por ser preciso integrar ao protótipo um cliente SVN, pois além de recuperar as informações sobre cada versão é necessário realizar o *checkout* automático das versões, para realizar o procedimento de análise de forma offline.

Após realizado o *checkout* as versões são compiladas individualmente. O problema desta etapa é que conforme o código evolui, as dependências das versões podem ser modificadas. Para todos os projetos analisados foi necessário encontrar as dependências de cada versão.

No entanto, talvez a maior dificuldade encontrada na avaliação foi a falta de um *benchmarks* ou catálogos com o histórico de um projeto. O projeto Markos european project[1] foi utilizado por Palomba *et al.* [40] para detectar bad smells em um histórico de um sistema. Esse projeto possui um histórico com as mudanças ocorridas a cada versão. Contudo, por se tratar de uma ferramenta em desenvolvimento não foi possível utilizá-lo.

6.3 Limitações

As principais limitações deste trabalho referem-se a formalização dos padrões emergentes e a ferramenta de detecção de padrões. A formalização dos padrões emergentes é complexa, pois devem ser modeladas as características que definem os padrões, além de ser flexível o suficiente para caracterizar as classes que não atendam à restrição estrutural dos padrões. Desta forma, a formalização apresentada, neste trabalho, necessita evoluir para abranger as situações que não foram analisadas.

Quanto a ferramenta de detecção, a limitação está restrita a própria ferramenta. A depender da quantidade de padrões detectados, a ferramenta pode incluir ou excluir uma grande quantidade de instâncias de padrões emergentes. Como a ferramenta foi escolhida dentre as ferramentas selecionadas na revisão sistemática, excluem-se outras ferramentas que poderiam detectar um número maior de instâncias de padrões.

Outra limitação deste trabalho refere-se à quantidade de ferramentas de detecção utilizadas. Como a abordagem propõe o uso de uma ferramenta, não foram analisadas informações da abordagem sendo aplicada com outras ferramentas.

[1] http://www.markosproject.berlios.de

6.4 Trabalhos Relacionados

Nessa seção serão apresentadas alguns do trabalhos selecionados na Revisão Sistemática (ver seção 3).

A abordagem proposta por Stencel *et al.* [46] utiliza a análise estática para detecção dos padrões de projeto. Uma ferramenta de linha de comando foi implementada para validar a metodologia definida na abordagem. Inicialmente, é estabelecido um metamodelo do programa. O metamodelo do programa consiste em um conjunto de elementos básicos e um conjunto de relações entre esses elementos. Os elementos básicos representam os tipos, métodos, atributos ou instâncias, enquanto as relações descrevem as características estruturais e comportamentais do programa. As relações estruturais descrevem a hierarquia de tipo de um programa, incluindo um método de override implícito pela hierarquia. As relações comportamentais em sua maioria referem-se a um fluxo de dados e um fluxo de chamadas.

O processo de detecção consiste em *parsing*, análise e detecção. Na etapa de parsing, a ferramenta Recoder é utilizada para criar uma AST (Abstract Syntax Tree) a partir do código, posteriormente os elementos básicos são extraídos da AST. A etapa de análise envolve a execução de um conjunto de análises (análise estrutural, análise de fluxo de chamadas e análise de fluxo de dados) para descobrir as relações entre os elementos.

Durante a fase de análise estrutural, são identificadas as relações estruturais do programa. Em seguida, os elementos básicos e as relações descobertas são armazenados em um banco de dados relacional. Análise de fluxo de chamadas identifica as relações definidas pela estrutura de invocações, por exemplo, invocação e instanciação. E, a análise de fluxo de dados realiza uma análise estática do fluxo de dados em um contexto top-down e flow-sensitive.

Na etapa de detecção, as consultas SQL são executadas para descobrir instâncias do padrão. As consultas seguem as definições dos padrões de projeto expressos em lógica de primeira ordem.

Resumidamente, a ferramenta constrói um metamodelo do programa usando a biblioteca Recoder e análise estática. Então, armazena as relações extraídas em uma base de dados. Como as consultas de lógica de primeira ordem pode ser implementadas em SQL, as fórmulas lógicas que descrevem os padrões foram traduzidos manualmente para consultas SQL e essas consultas são usadas para procurar instâncias do padrão. No trabalho, após analisar

algumas variações dos padrões, foram definidos formalmente, em lógica de primeira ordem, 4 dos padrões criacionais definidos pelo *GoF* [18].

Outra abordagem é proposta por De Lucia *et al.* [11; 36]. Como citado anteriormente, durante a análise estática são extraídas as informações estruturais dos padrões e um conjunto de padrões são classificados. Posteriormente, é executada uma análise dinâmica, que é responsável por observar a execução do conjunto de padrões selecionado na etapa anterior. A Ferramenta DPRE [11; 36] foi implementa seguindo este mesmo procedimento. O diferencial desta ferramenta é a utilização de *Model checking* (ver Figura 6.1) após a análise estática, para diminuir a quantidade de falso-positivos. A abordagem de recuperação padrão de projeto é totalmente automatizada e analisa o comportamento de instâncias de padrão tanto estática quanto dinamicamente.

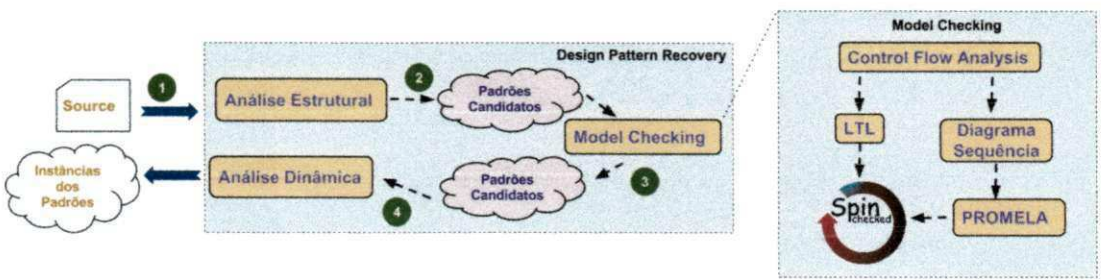

Figura 6.1: Processo de detecção utilizando Model Checking

A abordagem utiliza *Model checking* para verificar estaticamente os aspectos comportamentais de instâncias do padrão. O comportamento dos padrões são descritos em lógica temporal. O SPIN é utilizado para checar se os padrões candidatos atendem as restrições escritas em lógica temporal. Para realizar a análise dinâmica o código é instrumentado e os padrões candidatos são monitorados. Atualmente, a ferramenta DPRE foi incorporada a um plugin da IDE Eclipse, o ePAD [10]. O ePAD realiza a detecção de padrão utilizando a mesma abordagem descrita para a ferramenta DPRE. Um diferencial deste plugin é a possibilidade de inserir estruturas e comportamentos personalizáveis na ferramenta e realizar o processo de detecção com essa configuração.

Outra abordagem que utiliza análise estática foi proposta por Tsantalis *et al.* [49]. Essa abordagem detecta variações dos padrões, utilizando um algoritmo de similaridade entre vértices de um grafo. Embora no referido artigo não tenha sido atribuído um nome a ferramenta

desenvolvida, nesta dissertação a ferramenta será identificada por SSA.

Nessa abordagem, tanto o sistema quanto o padrão são representados por matrizes. A ideia é que um diagrama de classe é essencialmente um grafo direcionado, que pode ser mapeado em uma matriz quadrada (nxn). Inicialmente, são extraídas as características que representam o sistema, por exemplo, associação e generalização. Essas características são representadas por uma matriz adjacente nxn, onde n é o número de classes. O segundo passo é detectar as hierarquias de herança presente no sistema e representá-las em uma árvore. A Figura 6.2 exibe como são representadas as hierarquias de classes.

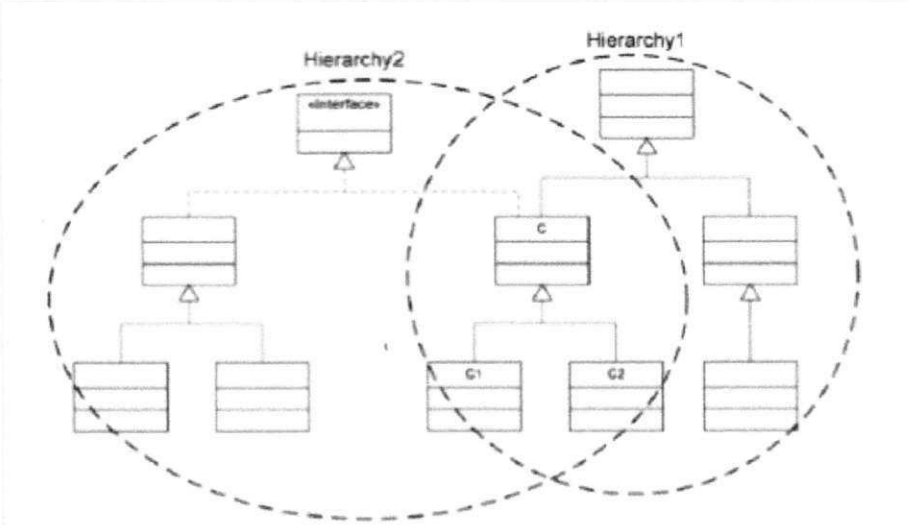

Figura 6.2: Hierarquias de classes descritas por Tsantalis *et al.* [49]

No terceiro passo, as matrizes de cada subsistema são construídas. Cada subsistema é definido pelas classes pertencentes a uma ou mais hierarquias. O conjunto de matrizes que representam um subsistema é construído para preservar as informações relativas às classes pertencentes às hierarquias. No quarto passo é executado o algoritmo de similaridade entre as matrizes que representam o subsistema e as matrizes que representam o padrão. Por fim, no quinto passo é a extração dos padrões de cada subsistema. Neste passo é calculado um score, e individualmente os subsistemas são ordenados em ordem decrescente do valor do seu score, então para cada padrão é criada uma lista. As classes do subsistema que tiverem um score maior que o padrão, são adicionadas à respectiva lista.

Um problema recorrente à detecção de padrões é a falta de um *benchmark* comum aos experimentos realizados na área. Assim, não é possível comparar resultados entre as várias

técnicas e abordagem existentes. Na tentativa de oferecer uma forma de comparação dos resultados das ferramentas de detecção de padrão, com o benefício de ter um modelo único, flexível e colaborativo é proposto um *benchmark* por Fontana *et al.* [17]. Essa abordagem é caracterizada por um meta-modelo genérico para representar os padrões de projeto; a possibilidade de avaliar as instância e discutir sobre a exatidão na detecção; e, uma forma de comparar os resultados provenientes de diferentes ferramentas.

Em 2009, Kniesel *et al.* [30] discutiu sobre a diversidade nas especificações dos padrões de projeto, além de propor uma abordagem *(data fusion)* que utiliza várias ferramentas e técnicas para aumentar a acurácia na detecção. No ano seguinte, os mesmos autores se reuniram com outros pesquisadores da área e foi apresentada uma abordagem [31], que tem como objetivo promover um formato para suprir as limitações provenientes da existência de diferentes tipos de formatos de saída das ferramentas. DPDX fornece a base para uma forma aberta de ferramentas que realizam a comparação, visualização, ou validação dos resultados das ferramentas de detecção de padrões. São descritos meta-modelos para especificar os padrões, identificar os padrões e um meta-modelo para os resultados das ferramentas.

Não é trivial caracterizar um padrão, Rasool *at al.* [42] apresenta as *feature types*, onde cada *feature* representa uma *regra*, algo que caracteriza o padrão. Com esta solução é possível detectar variações dos padrões. A abordagem escolhida constrói um modelo a partir do código e utiliza SQL Queries para identificar os padrões. Cada padrão é catalogado como um conjunto de *feature* e no momento de classificação são analisadas, caso atenda a todos, o padrão é classificado corretamente. Na avaliação, foi criado um baseline com instâncias de padrões presentes em três projetos: Junit, JHotDraw e JRefactory. Os resultados mostraram que a customização dos padrões nos catálogos aumentam a acurácia.

6.5 Trabalhos futuros

Com a conclusão deste trabalho, vislumbra-se alguns trabalhos futuros para continuação ao que já foi desenvolvido. Dentre eles é possível destacar:

Refinar a revisão sistemática: A revisão sistemática realizada abordou a área de detecção. Porém, seria interessante realizar um levantamento geral da área de padrões de projeto, não apenas de detecção. Assim, algumas questões, técnicas e ferramentas de identificação

podem ser conhecidas e avaliadas. Além de explorar outras áreas como qualidade, micro-arquiteturas e violações de restrições, temas já abordados juntamente com os padrões de projeto.

Concluir o protótipo da ferramenta: O processo utilizado na ferramenta é semi-automático, ainda sendo preciso uma checagem manual do código. Essa verificação pode ser facilitada se a ferramenta provesse um ambiente onde fosse possível comparar automaticamente as mudanças comportamentais entre as versões. Mas, principalmente construir uma ferramenta que implemente, de fato, a abordagem proposta. O que foi apresentado é apenas uma prova de conceito que visa auxiliar no processo de validação da abordagem.

Realizar outros experimentos: O estudo exploratório realizado não abordou outras ferramentas de detecção, assim como, não utilizou outros projetos de código livre para avaliar a abordagem proposta. Outros experimentos podem avaliar: a aplicabilidade dos padrões emergentes; a relação entre padrões emergentes e qualidade de software; e, o uso de outras ferramentas para detecção.

Avaliar os padrões emergentes junto aos desenvolvedores: A relevância dos padrões emergentes podem ser avaliados juntos aos desenvolvedores. O objetivo deste estudo é observar se os padrões emergentes podem ser identificados por outros desenvolvedores. Uma provável avaliação envolveria um ambiente controlado onde os desenvolvedores irão ter acesso a um conjunto de código e selecionaria os casos em que o comportamento é similar ao descrito pelo padrão.

Bibliografia

[1] F. Arcelli, S. Masiero, and C. Raibulet. Elemental design patterns recognition in java. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 196–205, 2005.

[2] Francesca Arcelli, Andrea Caracciolo, and Marco Zanoni. A benchmark for design pattern detection tools: a community driven approach. *ERCIM News*, 2012(88), 2012.

[3] Francesca Arcelli, Christian Tosi, and Marco Zanoni. Can design pattern detection be useful for legacy systemmigration towards soa? In *Proceedings of the 2Nd International Workshop on Systems Development in SOA Environments*, SDSOA '08, pages 63–68, New York, NY, USA, 2008. ACM.

[4] F. Arcelli Fontana, S. Maggioni, and C. Raibulet. Understanding the relevance of micro-structures for design patterns detection. *J. Syst. Softw.*, 84(12):2334–2347, December 2011.

[5] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Inf. Sci.*, 181(7):1306–1324, April 2011.

[6] Mario Luca Bernardi and Giuseppe Antonio Di Lucca. Model-driven detection of design patterns. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Alexander Binun and Gunter Kniesel. Dpjf - design pattern detection with high accuracy. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 245–254, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Nadia Bouassida and Hanene Ben-Abdallah. Design improvement through dynamic and structural pattern identification. In *Proceedings of the Third International Conference on Innovation and Information and Communication Technology*, ISIICT'09, Swinton, UK, UK, 2009. British Computer Society.

[9] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009.

[10] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–6, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Improving behavioral design pattern detection through model checking. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 176–185, Washington, DC, USA, 2010. IEEE Computer Society.

[12] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Improving behavioral design pattern detection through model checking. pages 176–185, 2010.

[13] Jens Dietrich and Chris Elgar. A formal description of design patterns using owl. In *Proceedings of the 2005 Australian Conference on Software Engineering*, ASWEC '05, pages 243–250, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Jing Dong, Yongtao Sun, and Yajing Zhao. Design pattern detection by template matching. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 765–769, New York, NY, USA, 2008. ACM.

[15] A.H. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 143–152, Nov 1997.

[16] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. *Comput. Lang. Syst. Struct.*, 30(1-2):21–33, April 2004.

[17] F.A. Fontana, A. Caracciolo, and M. Zanoni. Dpb: A benchmark for design pattern detection tools. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 235–244, March.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[19] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[20] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*, 34(5):667–684, September 2008.

[21] Manjari Gupta, Akshara Pande, and A. K. Tripathi. Design patterns detection using sop expressions for graphs. *SIGSOFT Softw. Eng. Notes*, 36(1):1–5, January 2011.

[22] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, and Houari Sahraoui. Improving design-pattern identification: a new approach and an exploratory study. *Software Quality Journal*, 18(1):145–174, 2010.

[23] Shinpei Hayashi, Junya Katada, Ryota Sakamoto, Takashi Kobayashi, and Motoshi Saeki. Design pattern detection by using meta patterns. *IEICE - Trans. Inf. Syst.*, E91-D(4):933–944, April 2008.

[24] Chengwan He, Zheng Li, and Keqing He. Identification and extraction of design pattern information in java program. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, SNPD '08, pages 828–834, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program*

Comprehension, IWPC '03, pages 94–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Olivier Kaczor, Yann-Gael Gueheneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Identification of design motifs with pattern matching algorithms. *Inf. Softw. Technol.*, 52(2):152–168, February 2010.

[28] J. Kerievsky. *Refatoração para Padrões*. Bookman, 1995.

[29] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33:2004, 2004.

[30] G. Kniesel and A. Binun. Standing on the shoulders of giants - a data fusion approach to design pattern detection. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 208–217, May 2009.

[31] Gunter Kniesel, Alexander Binun, Peter Hegedus, Lajos Jeno Fulop, Alexander Chatzigeorgiou, Yann-Gael Gueheneuc, and Nikolaos Tsantalis. Dpdx–towards a common result exchange format for design pattern detection tools. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 232–235, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Maurice Lebon and Vassilios Tzerpos. Fine-grained design pattern detection. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*, COMPSAC '12, pages 267–272, Washington, DC, USA, 2012. IEEE Computer Society.

[33] Hakjin Lee, Hyunsang Youn, and Eunseok Lee. Automatic detection of design pattern for reverse engineering. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, SERA '07, pages 577–583, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Fan Li, Qing-shan Li, Yang Su, and Ping Chen. Detection of design patterns by combining static and dynamic analyses. *Journal of Shanghai University (English Edition)*, 11(2):156–162, 2007.

[35] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.

[36] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *J. Syst. Softw.*, 82(7):1177–1193, July 2009.

[37] Steven John Metsker. *Padrões de Projeto em Java*. Bookman, 2004.

[38] Naouel Moha and Yann-Gaël Guéhéneuc. Ptidej and decor: Identification of design patterns and design defects. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 868–869, New York, NY, USA, 2007. ACM.

[39] Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Identification of behavioural and creational design motifs through dynamic analysis. *J. Softw. Maint. Evol.*, 22(8):597–627, December 2010.

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

[41] Niklas Pettersson, Welf Lowe, and Joakim Nivre. Evaluation of accuracy in design pattern occurrence detection. *IEEE Trans. Softw. Eng.*, 36(4):575–590, July 2010.

[42] Ghulam Rasool and Patrick Mader. Flexible design pattern detection based on feature types. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 243–252, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Parvinder Singh Sandhu, Parwinder Pal Singh, and Anil Kumar Verma. Evaluating quality of software systems by design patterns detection. In *Proceedings of the 2008 International Conference on Advanced Computer Theory and Engineering*, ICACTE '08, pages 3–7, Washington, DC, USA, 2008. IEEE Computer Society.

[44] J.M. Smith and D. Stotts. Spqr: flexible automated design pattern extraction from source code. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 215–224, Oct 2003.

[45] I. Sommerville. *Engenharia de Software*, chapter 22–23, pages 341–373. Addison-Wesley, São Paulo, SP, 8 edition, 2007.

[46] Krzysztof Stencel and Patrycja Wegrzynowicz. Detection of diverse design pattern variants. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 25–32, Washington, DC, USA, 2008. IEEE Computer Society.

[47] A. Stoianov and I. Şora. Detecting patterns and antipatterns in software using prolog rules. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 253–258, May 2010.

[48] R. Terra and R.S. Bigonha. Ferramentas para análise estática de códigos java. *Encontro Brasileiro de Teste de Software*, 1(3):1–5, 2008.

[49] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, Nov 2006.

[50] Wei Wang and Vassilios Tzerpos. Dpvk - an eclipse plug-in to detect design patterns in eiffel systems. *Electron. Notes Theor. Comput. Sci.*, 107:71–86, December 2004.

[51] Wei Wang and Vassilios Tzerpos. Design pattern detection in eiffel systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 165–174, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Hironori Washizaki, Kazuhiro Fukaya, Atsuto Kubo, and Yoshiaki Fukazawa. Detecting design patterns using source code of before applying design patterns. In *Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and*

Information Science, ICIS '09, pages 933–938, Washington, DC, USA, 2009. IEEE Computer Society.

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[54] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 8 edition, 2008.