Uma abordagem para apoio à decisão de refatoração em sistemas de software

João Paulo Lemes Machado

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

Uberlândia
2017

João Paulo Lemes Machado

Uma abordagem para apoio à decisão de refatoração em sistemas de software

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

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Marcelo de Almeida Maia

Uberlândia
2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

681.3
M149a
2017

Machado, João Paulo Lemes, 1986-
Uma abordagem para apoio à decisão de refatoração em sistemas de software / João Paulo Lemes Machado. - 2017.
73 f. : il.

Orientador: Marcelo de Almeida Maia.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Disponível em: http://dx.doi.org/10.14393/ufu.di.2017.85
Inclui bibliografia.

1. Computação - Teses. 2. Refatoração de software - Teses. 3. Coesão - Teses. I. Maia, Marcelo de Almeida. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Uma abordagem para apoio à decisão de refatoração em sistemas de software**" por **João Paulo Lemes Machado** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação.**

Uberlândia, 28 de agosto de 2017

Orientador: Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Banca Examinadora: Prof. Dr. Autran Macedo
Universidade Federal de Uberlândia

Prof. Dr. Eduardo Figueiredo
Universidade Federal de Minas Gerais

Este trabalho é dedicado à minha querida mãe Luzia Machado Lemes, e a minha namorada Thais Gonçalves

Agradecimentos

Agradeço primeiramente à minha família por todo apoio e investimento na minha educação ao longo dos anos.

À minha mãe por todo o suporte.

Ao meu orientador Professor Marcelo de Almeida Maia, por toda compreensão, paciência e motivação.

Aos meus colegas do grupo LASCAM que de forma direta ou indireta contribuíram para a realização deste trabalho.

À minha colega Liliane Nascimento pela imensa ajuda com as análises durante o desenvolvimento deste projeto.

"Saber muito não lhe torna inteligente. A inteligência se traduz na forma que você recolhe, julga, maneja e, sobretudo, onde e como aplica esta informação."
(Carl Sagan)

Resumo

A falta de modularização é um dos principais problemas encontrados em sistemas de software. Diversos estudos abordam esse problema apresentando soluções que visam aumentar a qualidade da modularização. Contudo, ainda não existe uma solução definitiva que possa ser aplicada em qualquer situação na qual esse problema ocorre. Uma situação específica diz respeito à falta de coesão entre os métodos de uma classe em sistemas orientados a objetos. Para essa situação, uma solução apropriada seria aplicar a refatoração de classes. O processo de refatoração de classes tem como objetivo melhorar a modularização de um sistema sem altear suas funcionalidades. Contudo, esse processo pode ser extremamente complexo e difícil de ser executado, pois existem efeitos colaterais que podem ser provocados por alterações indevidas. Nesse contexto, ainda existe uma lacuna por melhores sistemas de apoio à refatoração visando o aumento de coesão entre métodos das classes e uma melhor modularização do sistema. Assim, este trabalho tem como objetivo propor uma abordagem para fornecer informações de apoio à refatoração. Tais informações foram obtidas a partir de uma análise dos impactos gerados pelas refatorações aplicadas em 8 sistemas desenvolvidos na linguagem Java. O impacto dessas refatorações foram identificados e medidos através da análise da evolução das métricas de coesão. Foi realizada uma análise qualitativa sobre as refatorações identificadas com o objetivo de determinar qual tipo de estratégia foi adotada em cada situação. A partir dessa análise foi possível propor guias de refatoração que foram apresentadas aos desenvolvedores dos 8 sistemas através de um *survey*. O estudo obteve resultados positivos onde a partir de discussões realizadas com os desenvolvedores foram observadas situações nas quais as guias propostas claramente ajudariam no processo de refatoração. Também foram constatadas situações nas quais as recomendações podem ser adotadas como novas práticas com o objetivo de evitar perdas de modularização.

Palavras-chave: Modularização. Refatoração. Coesão.

Abstract

The lack of modularization is one of the main problems encountered in software systems. Several studies address this problem by presenting solutions that aim to increase the modularization quality. However, there is still no definitive solution that can be applied in any situation in which this problem occurs. A specific situation concerns the lack of cohesion among methods of a class in object-oriented systems. In this situation, an appropriate solution would be to apply a class refactoring. The class refactoring process aims at improving the modularization of a system without changing its functionalities. However, this process can be complex and difficult to execute because of the side effects that can be caused by improper changes. In this context, better systems for supporting refactoring are still lacking, so the cohesion between class methods and a better modularization of the system could be achieved. This work proposes an approach to provide information to support of refactoring. This information was obtained from an analysis of the impacts generated by the refactorings applied in 8 systems developed in Java language. The impact of these refactorings was identified and measured by analyzing the evolution of cohesion metrics. A qualitative analysis was performed on the refactorings identified with the objective of determining what type of strategy was adopted in each situation. From this analysis it was possible to propose refactoring guides that were presented to the developers of the 8 systems through a survey. The study found positive results in which discussions with the developers led to situations in which the proposed guidelines would clearly help the refactoring process. It was also observed situations in which the recommendations can be adopted as new practices in order to avoid losses of modularization.

Keywords: Modularization. Refactoring. Cohesion.

Lista de ilustrações

Lista de tabelas

Sumário

Capítulo 1

Introdução

Ao longo do tempo, sistemas de software são constantemente alterados pelos seus desenvolvedores. Essas alterações tem como objetivo inserir novas caraterísticas, novas requisições dos usuários, corrigir *bugs* ou otimizar a sua modularização (KANNANGARA; WIJAYANAYAKE, 2015). Para realizar uma alteração, o desenvolvedor precisa primeiramente conhecer a arquitetura do sistema. Esse conhecimento permite que o desenvolvedor possa identificar onde e como alterar o código fonte.

Nesse contexto, para compreender melhor sua arquitetura, os desenvolvedores recorrem às informações contidas na documentação do sistema. Entretanto, em muitos casos essa informação não existe, está incompleta ou desatualizada. Para suprir essa falta de informação, os desenvolvedores recorrem ao código fonte que se torna a sua única fonte de informação confiável sobre a arquitetura do sistema.

Atualmente, as ferramentas disponíveis para auxiliar o desenvolvedor durante o processo de manutenção não são capazes de preencher todas as lacunas deixadas pela falta de informações sobre a arquitetura (GARCIA; IVKOVIC; MEDVIDOVIC, 2013). Desprovido dessas informações, o desenvolvedor pode não conseguir identificar o local correto para realizar uma alteração no sistema. Caso essa alteração seja realizada no local incorreto ela poderá violar a relação de dependência entre os componentes da arquitetura, gerando inconsistências e perdas de modularização.

Em sistemas desenvolvidos sobre o paradigma orientado a objeto, o processo de refatoração de software tem como objetivo melhorar a estrutura e a modularização do sistema sem alterar suas funcionalidades (FOWLER; BECK, 1999). Para determinar quando, como e onde realizar essa alteração, o desenvolvedor precisa de informações que apoiem a sua tomada de decisão. Essas informações podem ser usadas como guias para identificar o local correto, bem como as alterações que devem ser feitas para aplicar a refatoração. Caso o desenvolvedor não tenha acesso a essas informações, ele pode acidentalmente inserir erros no código fonte ou até mesmo alterar alguma funcionalidade.

Com base nesse cenário, este trabalho aborda o seguinte problema: Falta de informações de apoio ao processo de tomada de decisão, de quando, como e onde o desenvolvedor

deve aplicar uma refatoração de classe.

Para suprir essa necessidade, foi proposta uma abordagem para identificação e avaliação de oportunidades de refatoração. Também foram recomendadas guias para auxiliar o desenvolvedor em como deve ser feita a refatoração. Foi realizada uma análise para determinar o impacto gerado pelas refatorações sobre a qualidade do software. A avaliação desse impacto foi medida através das métricas de qualidade que são apresentadas em detalhes no **Capítulo 2 - Seção 2.1**. Essas métricas foram utilizadas por serem reconhecidas como indicadores confiáveis de qualidade e manutenibilidade de software (BOIS; DEMEYER; VERELST, 2004).

1.1 Problema

Como exemplo motivador, é apresentado o caso do sistema Lucene. Lucene é uma ferramenta de código livre para busca em texto desenvolvida sobre o paradigma orientado a objeto em linguagem Java. Ao realizar uma pesquisa no sistema de repositório de software do Lucene onde são reportados problemas referentes ao sistema, foram identificadas 40 *issues*[1] abertas com o tema refatoração. Foram selecionadas somente *issues* do tipo melhoria e com status marcado como aberta. Uma *issue* em particular (**Lucene-2026**) apresentou o seguinte título: "*Refactoring of IndexWriter*". A classe IndexWriter é responsável pela criação de índices que são utilizados para realizar a busca em textos. Analisando a discussão sobre essa *issue* pode ser constatado que os desenvolvedores não possuem informações suficientes para decidir como aplicar a refatoração.

Ao avaliar a evolução da IndexWriter para identificar em qual momento a classe começou a apresentar os problemas que motivariam os desenvolvedores a aplicar a refatoração, foi identificada uma informação importante e que não foi apresentada na discussão. Entre as *releases* 3.6.2 e 4.0.0, a classe IndexWriter havia sofrido uma refatoração. Essa informação foi obtida a partir da análise da evolução das métricas de qualidades dessa classe, tendo como foco a métrica LCOM (*Lack of Cohesion of Methods*). Essa métrica mede o nível de coesão da classe através da contagem dos métodos que não estão relacionados entre si através do compartilhamento de atributos (CHIDAMBER; KEMERER, 1994).

A Figura 1 apresenta a evolução da métrica LCOM para a classe IndexWriter. Analisando o valor assumido por essa métrica em cada release foi identificado que houve uma queda brusca e significativa no valor da métrica na release 4.0.0. Essa queda representa uma aumento na coesão entre os métodos da classe e uma melhoria na sua estrutura.

Para determinar o que pode ter causado essa queda, foi realizada a comparação do código fonte da classe na *release* 3.6.2 com a *release* 4.0.0. Foi identificado que vários métodos do tipo *get()* e *set()* que são responsáveis por ler e atribuir valores aos atributos

[1] Uma *issue* é um problema, um defeito ou uma sugestão de melhoria reportada para uma classe do sistema.

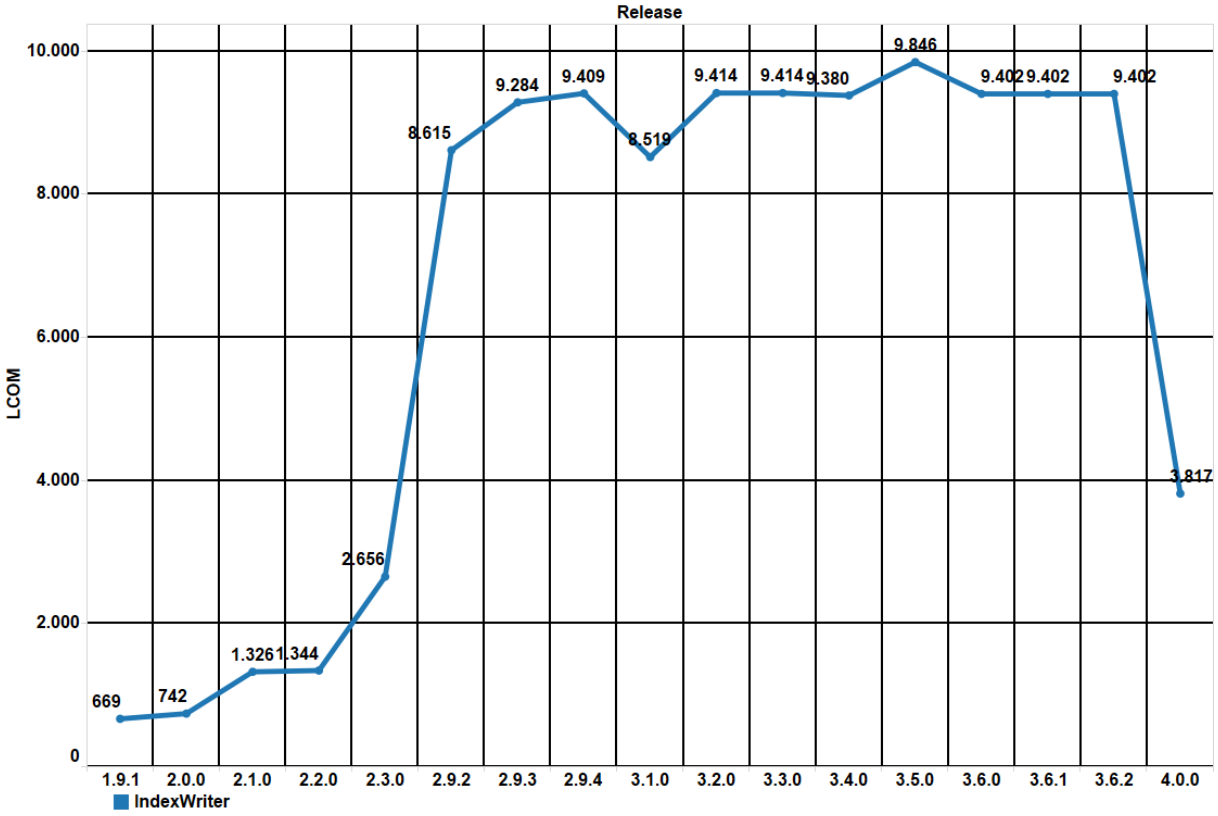

Figura 1 – Evolução LCOM da classe IndexWriter

da classe IndexWriter, foram movidos para uma classe chamada IndexWriterConfig. O Código 1.1 apresenta um exemplo de método que foi movido.

Código 1.1 – Método *get()* movido para *IndexWriterConfig*

```
/*Expert: returns the current MergePolicy in use by this writer.
@see #setMergePolicy
@deprecated use {@link IndexWriterConfig#getMergePolicy()} instead */
@deprecated
public MergePolicy getMergePolicy(){
  ensureOpen();
  return mergePolicy;
}
```

Um fato interessante é que esses métodos haviam sido marcados como *deprecated* em *releases* anteriores. Um método é marcado como *deprecated* quando o desenvolvedor quer indicar que o método poderá ser removido posteriormente ou que existe outro que pode ser usado em seu lugar. A figura 2 apresenta uma análise da evolução do LCOM na classe de destino IndexWriterConfig. Foi identificado que a classe foi criada na mesma *release* na qual os métodos da IndexWriter foram marcados como *deprecated*. Dessa forma a classe IndexWriterConfig foi criada com uma cópia dos métodos *get()* e *set()* marcados na classe IndexWriter.

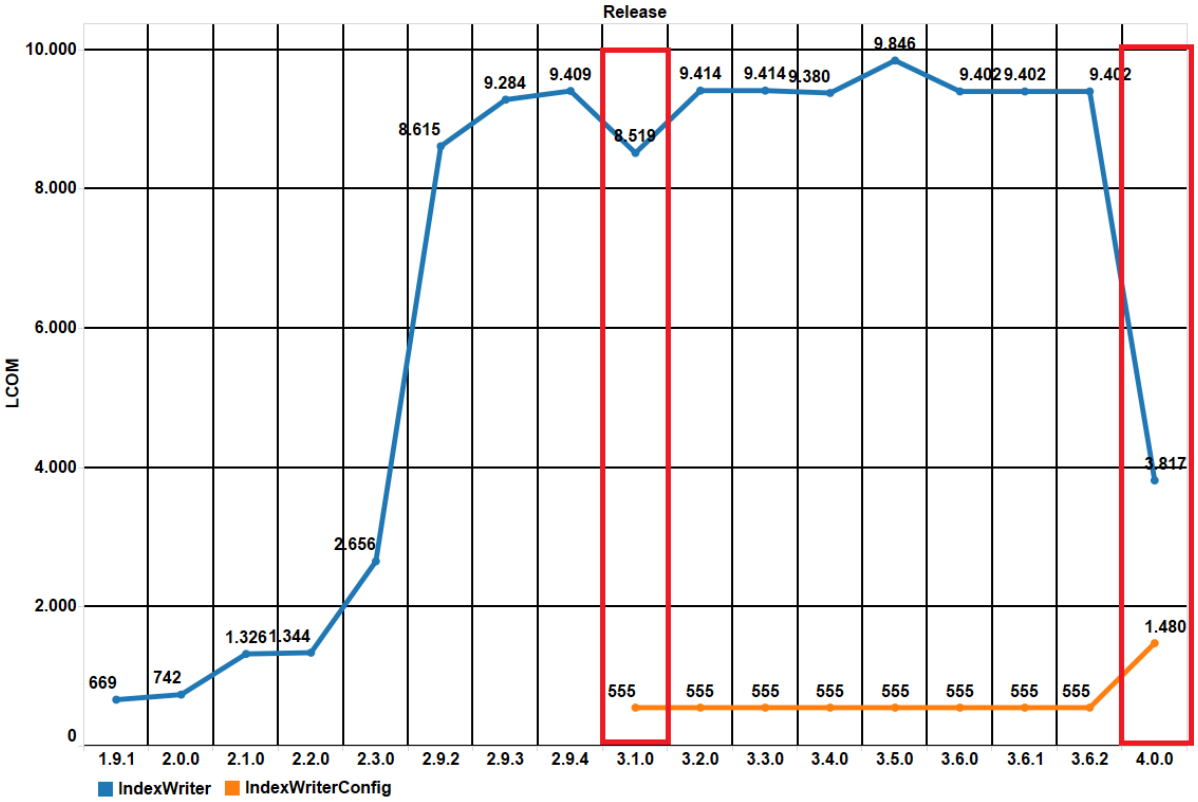

Figura 2 – Refatoração da classe IndexWriter

Foi constatado que a partir da *release* 3.1.0 alguns métodos já começaram a ser removidos gradativamente até a *release* 4.0.0, na qual o restante dos métodos foram completamente removidos. A partir da *release* 4.0.0, os novos métodos *get()* e *set()* passaram a ser criados diretamente na IndexWriterConfig. Como consequência dessas acões, a IndexWriterConfig começa a sofrer alterações em sua estrutura.

Com base nessa análise pode ser constatado que apesar de ser necessário criar uma nova classe, ao aplicar a refatoração as classes resultantes possuem mais qualidade em termos de coesão do que a classe original. Portanto, nesse caso observa-se uma melhoria na modularização do sistema. As informações referentes a essa refatoração não são apresentadas na discussão. É possível que durante a evolução do sistema, uma classe possa sofrer mais de uma refatoração. Ao analisar o histórico de refatorações e o impacto gerado sobre as métricas de qualidade é possível obter informações para auxiliar os desenvolvedores em refatorações futuras.

1.2 Objetivos

O principal objetivo deste trabalho é fornecer informações para apoiar o processo de tomada de decisão, de quando, como e onde o desenvolvedor deve aplicar uma refatoração. Tais informações serão obtidas a partir da análise histórica das alterações nas métricas

de qualidade.

1.3 Hipóteses e Perguntas de Pesquisa

- **Hipótese 1:** É possível, através da análise histórica da evolução de métricas de qualidade das classes do sistema identificar uma associação entre mudanças das métricas com respectivas refatorações. Além disso, caso esta associação exista, este será considerado um momento adequado para refatoração uma vez que existiria um padrão de comportamento entre os desenvolvedores, e este padrão será considerado adequado.

- **Pergunta de Pesquisa 1:** Existe um momento típico, em termos da evolução de métricas de qualidade, que os desenvolvedores elegem para realizar o refatoração de uma classe? A justificativa para essa pergunta se dá pelo fato de que ao descobrir o momento no qual se tornaria adequado refatorar uma classe, podemos sugerir refatorações para usuários menos experientes com base no histórico de modificações de outros usuários. Considerando que vamos identificar oportunidades de refatoração, precisamos determinar em qual ponto da evolução das classes, a refatoração deve ser aplicada.

- **Hipótese 2:** É possível, identificar as melhores estratégias para refatoração de classes, por meio da análise do impacto de diferentes alternativas de refatoração sobre as métricas de qualidade.

- **Pergunta de Pesquisa 2:** É possível detectar uma estratégia mais adequada para refatorar uma classe? A justificativa para essa pergunta se dá pelo fato, de que precisamos avaliar quais estratégias de refatoração trazem melhores resultados em termos de melhoria das métricas de *design* estrutural. Esta avaliação requer a medição do impacto da refatoração nas métricas de qualidade das classes envolvidas.

1.4 Contribuições

As principais contribuições deste trabalho são:

- Uma base de dados estruturada, contendo informações detalhadas sobre refatorações em classes de sistemas orientados a objetos. Esta será uma importante contribuição para as linhas de pesquisa em engenharia de software, pois, atualmente existe pouca informação sobre refatorações em classes de projetos orientados a objetos.

- Uma abordagem para identificação de oportunidades de refatoração em classes com baixa coesão. Atualmente, as abordagens para identificação de refatorações, não

levam em consideração a importância e qualidade das classes para o projeto. Ao fazer essa separação, é possível identificar e avaliar as refatorações que geram maior impacto estrutural. Posteriormente, podem ser criadas guias para realizar tais refatorações, minimizando seu impacto sobre o comportamento do sistema.

- Uma análise da evolução de coesão avaliando os impactos, positivos e negativos, das diferentes estratégias de refatoração. Essa será uma contribuição de extrema relevância, pois, existe pouca informação disponível sobre a evolução das métricas de qualidade, e a relação com os impactos de refatoração. Tal informação pode ser usada para identificar o momento no qual as classes tendem a ser refatoradas, permitindo assim, agir preventivamente e minimizar o esforço do processo de refatoração.

1.5 Estrutura da Dissertação

O restante deste deste trabalho está organizado da seguinte forma:

- O Capítulo 2 irá apresentar os conceitos necessários para a compreensão da abordagem e dos resultados obtidos.
- O Capítulo 3 irá apresentar o método de pesquisa e a proposta do modelo preditivo para refatoração.
- O Capítulo 4 irá apresentar o método utilizado para realizar o *survey* com os desenvolvedores
- O Capítulo 5 irá apresentar os resultados obtidos com a aplicação do modelo e do *survey*
- O Capítulo 6 irá apresentar os principais trabalhos relacionados e as diferenças entre eles e este trabalho.
- O Capítulo 7 irá apresentar as conclusões, lições aprendidas e trabalhos futuros.

Capítulo 2

Referencial Teórico

Nesse capítulo serão apresentados os conceitos necessários para a compreensão do método utilizado na abordagem e para a interpretação dos resultados obtidos.

2.1 Métricas de Software

Para realizar a análise de evolução da qualidade de um software é necessária a utilização de métricas que permitam mensurar suas características. Tais métricas consistem em medidas que usam classificações numéricas para quantificar as características e os atributos de uma entidade de software (CHIDAMBER; KEMERER, 1994).

Em sistemas orientados a objetos, a coesão é a característica que indica o grau de especialização de uma classe no sistema (PRESSMAN, 2005). Uma das principais métricas utilizadas para mensurar essa característica é a LCOM (*Lack of Cohesion of Methods*). Essa métrica conta os conjuntos de métodos em uma classe que não estão relacionados entre si através do compartilhamento de atributos acessados internamente (CHIDAMBER; KEMERER, 1994). Ao calcular o LCOM de uma classe quanto maior for o valor obtido, mais baixa será a sua coesão. Outra métrica que também é utilizada para realizar a medição dessa característica é a CAM (*Cohesion Among Methods of Class*). Essa métrica calcula a relação entre os métodos de uma classe com base na lista de parâmetros passadas para os métodos (BANSIYA et al., 1999).

Do ponto de vista arquitetural, classes com baixa coesão tendem a ser mais complexas e a possuir mais de uma responsabilidade. A baixa coesão ocorre quando as tarefas de uma classe têm pouca ou nenhuma relação entre si. Quando as tarefas de uma classe são relacionadas entre si e precisam ser executadas em determinada ordem, esta é uma situação de coesão moderada. Uma classe que realiza exclusivamente uma única tarefa possui um nível de coesão alta (PRESSMAN, 2005).

A complexidade de uma classe pode ser medida analisando a quantidade de métodos, suas respectivas complexidades e a sua quantidade de linhas de código. Nesse contexto a métrica WMC (*Weighted Methods Per Class*) Consiste na soma das complexidades dos

métodos de uma classe (CHIDAMBER; KEMERER, 1994). A definição de complexidade de um método varia de acordo com a implementação da métrica, mas geralmente está associada à complexidade ciclomática (MCCABE, 1976). A complexidade ciclomática de um método de uma classe é calculada através da analise das instruções que ele executa internamente. Métodos que fazem chamadas a outros métodos tendem a possuir complexidade mais elevada. Em classes onde os métodos possuem complexidade unitária, isto é, executam apenas uma instrução simples como a atribuição de um valor a um atributo, a métrica WMC representa a quantidade total de métodos da classe.

Outra métrica utilizada para medir a complexidade de uma classe é a NPM (*Number of Public Methods*), que basicamente faz a contagem de métodos públicos declarados na classe. Em conjunto com a métrica LOC (*Lines of Code*) que representa a quantidade de linhas de código, as métricas WMC e NPM permitem determinar o grau de complexidade de uma classe. Classes que possuem elevado nível de complexidade podem ser quebradas em classes menores dividindo suas responsabilidades (CHIDAMBER; KEMERER, 1994).

No que diz respeito à arquitetura, herança é uma das principais características que diferenciam sistemas orientados a objetos de sistemas convencionas. Com essa característica Uma sub-classe Y herda todos os atributos e métodos associados à uma super-classe X. Dessa forma toda a estrutura de dados e algoritmos desenvolvidos e implementados para X são disponibilizados diretamente em Y (PRESSMAN, 2005). Nesse sentido a métrica DIT (*Depth of Inheritance Tree*) determina a quantidade classes que estão em níveis acima na hierarquia e que podem afetar diretamente a classe. Quanto mais baixo a classe está na hierarquia, mais complexo será o comportamento de seus métodos (CHIDAMBER; KEMERER, 1994).

Outra métrica importante para essa característica é a NOC (*Number of Children*) que calcula o número de sub-classes subordinadas a uma classe na hierarquia. Essa métrica faz uma estimativa da quantidade de sub-classes que vão herdar os métodos da super-classe (CHIDAMBER; KEMERER, 1994). Com base nesse cálculo, é possível ter uma ideia da influência que a classe exerce sobre a arquitetura do sistema. Classes que exercem grande influência na arquitetura tendem a apresentar dificuldades em serem divididas em classes menores (CHIDAMBER; KEMERER, 1994).

2.2 Refatoração de Classe

A refatoração é o processo de mudança de um sistema de software de tal forma que não altere o comportamento externo do código, mas que melhore a sua estrutura interna (FOWLER; BECK, 1999). Refatorar consiste em aplicar uma reestruturação ou remodularização no sistema sem alterar suas funcionalidades. O principal objetivo da refatoração é tornar o código mais fácil de ser modificado e compreendido (FOWLER; BECK, 1999). Existem diferentes estratégias de refatoração as quais possuem características específicas

que podem gerar impactos sobre a qualidade o sistema. *Extract Class* é uma técnica bastante utilizada de refatoração que permite que uma classe seja divida em duas, onde as classes resultantes da divisão terão as mesmas responsabilidades da classe original (FOWLER; BECK, 1999).

Para aplicar a estratégia de *Extract Class* é necessário identificar um subconjunto de métodos e atributos que possuam alguma relação entre si. Após identificar esse conjunto são realizadas operações de *Move Field* e *Move Method* para mover os atributos e os métodos do conjunto para uma nova classe. Também é necessário estabelecer uma ligação entre a nova classe e a classe de origem, para que os métodos movidos possam ser acessados. Dependendo da forma como essa relação for estabelecida, o *Extract Class* pode ser classificado como *Extract Super-Class*, situação na qual a classe extraída se torna pai da classe de origem, ou *Extract Sub-Class* onde a classe extraída se torna filha da classe original.

Neste trabalho as análises terão como foco a estratégia de refatoração do tipo *Extract Class* nas suas duas formas: *Extract Super-Class* e *Extract Sub-Class*.

2.3 Árvore de Decisão

Em mineração de dados, a árvore de decisão é um modelo preditivo que pode ser usado para representar ambos modelos de classificação e de regressão, que são algoritmos utilizados para determinar se uma amostra pertence à uma determinada classe (ROKACH; MAIMON, 2014). Uma árvore de decisão representa um modelo hierárquico de decisões e suas consequências. Quando uma árvore é empregada para tarefas de classificações ela é chamada de árvore de classificação, enquanto que para tarefas de regressão é chamada de árvore de regressão. Árvores de classificação são usadas para classificar objetos em um conjunto de classes previamente definidas com base nos valores de seus atributos. Árvores de regressão são utilizadas para estimar o valor de uma variável numérica do objeto, com base no resultado de uma função linear que usa seus atributos como variáveis independentes.

Uma árvore de decisão é um classificador expresso como uma partição recursiva da instância espaço. A Figura 3 apresenta um exemplo de árvore de decisão cujo o funcionamento é detalhado a seguir. A árvore de decisão consiste em nós que formam uma árvore enraizada, o que significa que é uma árvore direcionada com um nó chamado de raiz que não tem entrada de arestas. Todos os outros nós têm exatamente uma aresta de entrada. Um nó com arestas de saída é referido como um nó interno ou nó de teste. Todos os outros nós são chamados de folhas (também conhecidos como nós terminais ou de decisão). Na árvore de decisão, cada nó interno divide o espaço da instância em dois ou mais sub-espaços de acordo com uma determinada função discreta do atributo de entrada Valores. No caso mais simples e mais frequente, cada teste considera um único

atributo, de modo que o espaço da instância seja particionado de acordo com o valor dos atributos. No caso de atributos numéricos, a condição se refere a um intervalo entre valores (ROKACH; MAIMON, 2014).

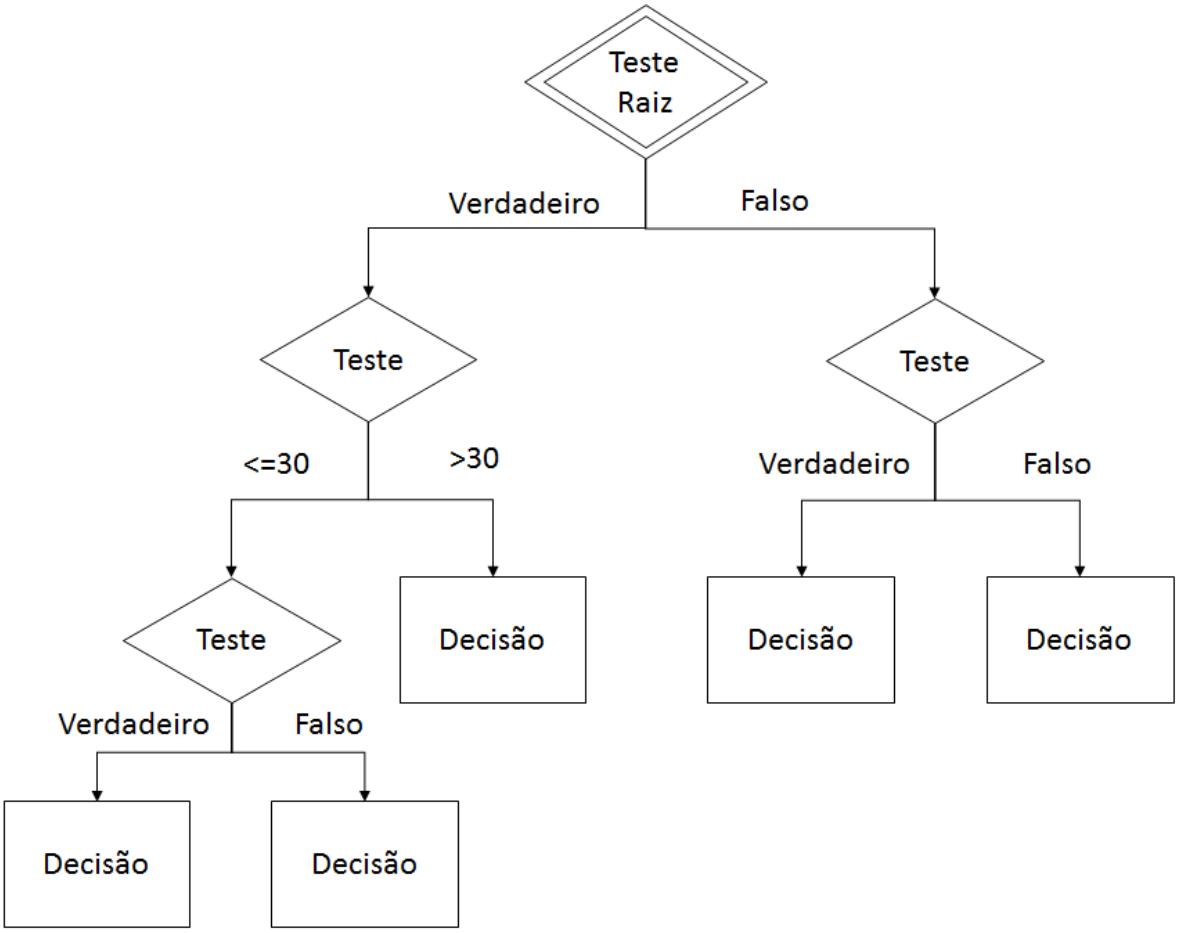

Figura 3 – Exemplo de Árvore de Decisão

A árvore de decisão por se tratar de um modelo de fácil interpretação, e por conseguir lidar tanto com atributos numéricos inteiros quanto reais, será utilizada como algoritmo para o modelo preditivo de refatoração proposto neste trabalho.

2.4 Ferramental Utilizado

As ferramentas utilizadas durante as análise e para a construção da abordagem de recomendação de refatoração são descritas a seguir:

- Ferramentas de repositórios de software e *issue tracker*: Para apresentar as recomendações de refatoração aos desenvolvedores foram criadas *issues* nos repositórios dos sistema avaliados. O Jira[1] é o *issue tracker* onde são reportados problemas

[1] https://issues.apache.org/jira/secure/Dashboard.jspa

referentes aos sistemas Lucene[2], Xerces[3] e Log4j[4]. O sistema Bugzilla[5] é o sistema onde são reportados os bugs referentes as sistema Tomcat[6], JMeter[7] e Ant[8]. O GitHub[9] é o sistema no qual ficam armazenados os códigos fonte e também onde são reportados problemas para os sistemas: Jetty[10] e Javacc[11].

- A ferramenta utilizada para a plotagem dos gráficos e para as análises visuais durante a abordagem foi o Tableau[12]. Trata-se de uma ferramenta de visualização de dados, utilizada para criação de *dashboards* e análises visuais.

- Para construir o modelo de árvore de decisão foi utilizada a ferramenta RapidMiner[13]. O RapidMiner é uma ferramenta para criação de modelos estatísticos de forma visual através do uso de componentes e operadores pré-programados.

- Para realizar a coleta das métricas de qualidade foi utilizada a ferramenta CKJM extended[14]. Essa ferramenta implementa as métricas de qualidade de software propostas em: (CHIDAMBER; KEMERER, 1994), apresentadas no **Capítulo 2 - Seção 2.1**.

2.5 Considerações Finais do Capítulo

Neste capítulo foram apresentados os principais conceitos referentes às métricas de software, refatorações e modelos de árvore de decisão. A aplicação desses conceitos será apresentada em detalhes nos próximos capítulos.

[2] https://lucene.apache.org/
[3] http://xerces.apache.org/
[4] https://logging.apache.org/log4j
[5] https://bz.apache.org/bugzilla/
[6] http://tomcat.apache.org/
[7] http://jmeter.apache.org/
[8] https://ant.apache.org/
[9] https://github.com
[10] http://www.eclipse.org/jetty/
[11] https://javacc.org/
[12] https://www.tableau.com
[13] https://rapidminer.com/
[14] https://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Capítulo 3

Estudo exploratório sobre análise temporal de coesão e refatoração

Neste capítulo, é apresentada a análise temporal para identificar e medir mudanças na coesão da classe determinando se houve de fato uma refatoração. O principal objetivo desta análise é determinar qual tipo de estratégia(tipo e processo de refatoração) foi adotada quando uma refatoração ocorreu.

Neste contexto, o códio fonte das classes nas quais ocorreram mudanças significativa na métrica LCOM foram comparadas entre as duas *releases*. Esta comparação tem como objetivo determinar se os métodos foram movidos desta classe para uma nova classe. Neste caso, uma refatoração pode ter ocorrido.

Os principais pontos avaliados durante as análises foram:

- ❏ Quais tipos de métodos foram movidos?
- ❏ Para quais classes os métodos foram movidos?
- ❏ Como a conexão foi feita entre as classe de origem e a classe de destino?
- ❏ Em qual momento a classe de destino foi criada?
- ❏ Qual foi o impacto gerado na métrica LCOM na classe de destino?

3.1 Construção da Base de Dados

Para realizar a análise foi necessário criar uma base de dados contendo informações sobre refatorações e os valores das métricas de qualidade das classes antes das refatorações. Nesse contexto, 8 sistemas *open source* desenvolvidos sobre o paradigma orientado a objetos em linguagem Java foram selecionados. A ferramenta CKJM *extended* foi utilizada para coletar as métricas de qualidade para todas as classes dos sistemas em todas as suas *releases* disponíveis para download. A Tabela 1 apresenta a sumarização da base

de dados na qual são apresentadas respectivamente: a quantidade de *releases* disponíveis para download de cada sistema e a quantidade total de classes considerando todas as *releases* dos sistemas.

Sistema	**Releases**	**Classes**
Log4j	48	1721
Xerces	36	1921
Lucene	58	3576
Ant	30	1557
Javacc	12	180
Jetty	167	2430
Tomcat	207	4554
JMeter	36	1089

Tabela 1 – Sumarização da Base de Dados.

A contagem de classes apresentada não é distinta, pois, uma classe pode aparecer em mais de uma *release*, tendo suas métricas calculadas em cada uma delas. Com base nos dados coletados é possível analisar a evolução das caraterísticas de uma classe através dos valores assumidos por suas métricas ao longo da evolução do sistema em diferentes *releases*.

3.2 Análise de Limiares de LCOM

A evolução de uma classe pode ser analisada comparando os valores assumidos por suas métricas de uma *release* para outra. A métrica LCOM (CHIDAMBER; KEMERER, 1994) foi escolhida por ser empiricamente associada a questões como baixa produtividade, maior esforço de manutenção e maior retrabalho (CHIDAMBER; DARCY; KEMERER, 1998). Apesar do fato de a métrica LCOM não assumir uma distribuição normal (GUPTA, 1997), neste trabalho a métrica LCOM foi utilizada como um indicador de alteração estrutural da classe. Neste contexto, foi analisada a variação do valor dessa métrica para uma mesma classe definindo-se um limiar a partir do qual essa variação foi significativa e avaliando se nesse momento ocorreu uma refatoração.

Para analisar a evolução da coesão de uma classe é necessário calcular a diferença entre os valores da métrica LCOM de uma *release* para outra. Os casos nos quais as classes foram renomeadas foram tratados ao verificar se na *release* anterior já existia uma classe de mesmo nome em outro pacote, ou se existia uma classe de nome diferente mas com o mesmo conjunto de métodos. Nesse contexto a evolução da métrica LCOM foi calculada da seguinte forma: Seja **C** uma classe e **R** uma *release* do sistema na qual a classe está contida, a fórmula (1) apresenta o cálculo da evolução da métrica LCOM. Onde **R** varia de 1 até **N**, sendo **N** a quantidade de *releases* nas quais a classe aparece.

$$\Delta LCOM = LCOM(C_{R+1}) - LCOM(C_R) \quad (1)$$

Ao realizar o cálculo do **ΔLCOM** existem 3 possíveis resultados:

1. **ΔLCOM** = 0 Representa uma situação de estabilidade da classe na qual não houve alterações no nível de coesão entre os seus métodos.

2. **ΔLCOM** > 0 Representa uma situação de perda de coesão entre os métodos da classe, pois, neste caso a métrica LCOM aumentou de uma *release* para a outra.

3. **ΔLCOM** < 0 Representa uma situação de aumento de coesão entre os métodos da classe, pois, neste caso a métrica LCOM diminuiu de uma *release* para a outra.

A tabela 2 apresenta o resultado da aplicação do cálculo sobre a base descrita na etapa anterior. A **Quantidade de ΔLCOM** corresponde ao conjunto dos valores obtidos pelo cálculo do ΔLCOM para cada classe em cada *release* de cada sistema. A coluna **ΔLCOM = 0** apresenta o subconjunto nos quais os do ΔLCOM foram igual a 0, isto é, não houve alteração no valor do LCOM. A coluna **ΔLCOM <> 0** apresentas o subconjunto de valores nos quais os do ΔLCOM foram diferentes de 0, isto é, houve alteração no valor LCOM. A coluna **ΔLCOM > 0** apresenta o subconjunto dos valores nos quais houve aumento no valor do LCOM, e a coluna **ΔLCOM < 0** corresponde ao conjunto dos valores nos quais houve diminuição do LCOM.

Quantidade de ΔLCOM: 471784		
ΔLCOM = 0	**ΔLCOM <> 0**	
455903 = 96%	15881 = 4%	
	ΔLCOM > 0	**ΔLCOM < 0**
	9805 = 62%	6076 = 38%

Tabela 2 – Aplicação dos Deltas

A situação na qual ΔLCOM < 0 será avaliada de forma mais criteriosa para verificar se de fato ocorreu uma refatoração nesse momento. Entretanto, para determinar se essa alteração foi significativa é necessário definir um limiar para o valor ΔLCOM. O limiar é um limite a partir do qual a alteração no valor da métrica de uma *release* para outra pode ser considerado significativo.

Para definir o valor desse limiar foi realizado o cálculo do desvio absoluto da mediana para o conjunto dos valores nos quais o ΔLCOM < 0. A fórmula 2 apresenta o cálculo do **DAM(Desvio Absoluto da Mediana)** (HUBER, 1996) que é realizado da seguinte forma: Seja **D** o conjunto dos valores de ΔLCOM < 0, e seja **di** um elemento desse conjunto. **DAM(D)** é a mediana dos desvios de cada **di** em relação à mediana do conjunto **D**. O desvio é calculado subtraindo-se o valor da mediana do conjunto **D** do valor de cada elemento **di** e aplicando a função módulo ao resultado da subtração.

$$DAM(D) = mediana(\{|d_i - mediana(D)|\}) \quad (2)$$

O motivo pelo qual o cálculo do **DAM(Desvio Absoluto da Mediana)** foi escolhido ao invés do desvio absoluto da média ou desvio padrão, se dá pelo fato que o **DAM** é uma medidas mais robustas para determinar a presença de *outliers*(Valores Atípicos) (LEYS et al., 2013). A Tabela 3 apresenta o resultado da aplicação do cálculo do **DAM(D)**, onde o **tamanho(D)** representa a cardinalidade do conjunto D, a **mediana(D)** é o valor da mediana desse conjunto, **DAM(D)** é o resultado de aplicação do cálculo do desvio absoluto da mediana, e **Limiar(D)** é o resultado da soma da **mediana(D)** com **DAM(D)**. A desigualdade **ΔLCOM>Limiar(D)** representa o subconjunto formado pelos elementos do conjunto D que possuem valores acima do Limiar(D). Com base no resultado desse cálculo, o valor considerado para o limiar de ΔLCOM foi a mediana(D) + DAM(D). Dessa forma os valores de ΔLCOM que estiverem acima do limiar serão analisados na próxima etapa para verificar se de fato ocorreu uma refatoração.

tamanho(D)	**6076**
mediana(D)	**10**
DAM(D)	**8**
Limiar(D)	**18**
ΔLCOM>Limiar(D)	**2137**

Tabela 3 – Resultado Limiar

Para verificar se os valores acima do limiar são frequentes durante a evolução das classes, foi realizada uma análise de distribuição dos valores pela quantidade de classes nas quais esse valores ocorreram. O gráfico da Figura 4 apresenta a distribuição da frequência de valores ΔLCOM acima do limiar, isto é, $\Delta LCOM > 18$, pela quantidade de classes nas quais eles ocorreram. O eixo X representa a frequência de ocorrência dos $\Delta LCOM > 18$ em ordem crescente, enquanto o eixo Y apresenta a quantidade de classes distintas para cada valor de frequência. Com o objetivo de facilitar as comparações dos valores do conjunto **D**($\Delta LCOM < 0$) foi aplicada a função módulo. Portanto, ao realizar a comparação $\Delta LCOM > 18$ é equivalente à comparação $\Delta LCOM <$ -18, uma vez que todos os elementos do conjunto **D** são negativos. Ao dizer que ocorreu um $\Delta LCOM > 18$ em uma classe, significa que de uma *release* para outra houve um decréscimo maior que 18 na métrica LCOM.

A Tabela 4 apresenta a distribuição e a frequência de ocorrência de valores ΔLCOM acima do limiar. Também é apresentado o percentual acumulado da quantidade de classes para cada valor de frequência. Observa-se que em mais de 75% das classes nas quais ocorrem essa variação ela ocorre no máximo duas vezes, enquanto que nos 25% restantes das classes ocorre três vezes ou mais.

Com base nessa informação pode-se constatar que o valor de $\Delta LCOM > 18$ não ocorre com frequência na maior parte das classes. Caso o esse valor fosse frequente, poderia ser considerado o valor mediana(D) + 2DAM(D) para localizar somente as variações mais significativas durante a evolução da classe. Com base nos dados da tabela 4 pode ser

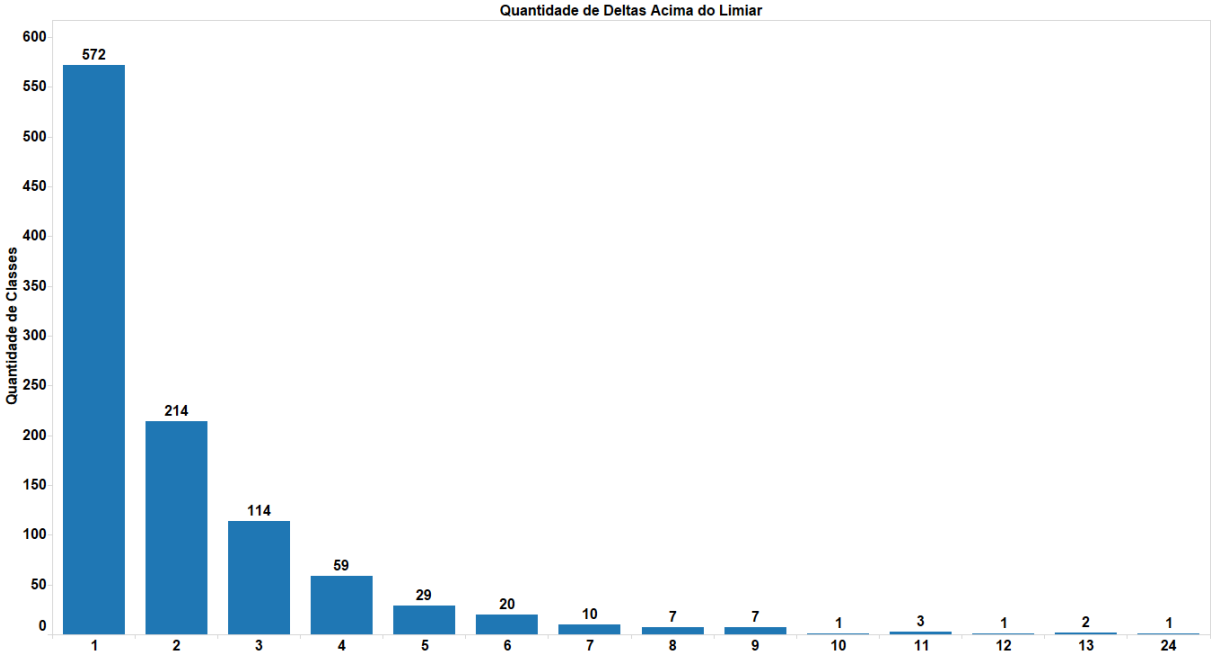

Figura 4 – Distribuição dos deltas acima do limiar

classes	**%classes acumulado**	**frequência deltas**	**total deltas**
572	55,0%	1	572
214	75,6%	2	428
114	86,5%	3	342
59	92,2%	4	236
29	95,0%	5	145
20	96,9%	6	120
10	97,9%	7	70
7	98,6%	8	56
7	99,2%	9	63
1	99,3%	10	10
3	99,6%	11	33
1	99,7%	12	12
2	99,9%	13	26
1	100,0%	24	24
Total: 1040	100,0%		2137

Tabela 4 – Distribuição de Deltas por Classe.

constatado que os ΔLCOM acima do limiar estão distribuídos em 1040 classes distintas. Também observa-se algumas classes nas quais esse valor é recorrente. Para as classes onde o valor ocorre mais de uma vez será considerado o maior valor de ΔLCOM. Dessa forma para classes como a IndexWriter que possuem valores elevados de LCOM, será analisado a sua variação mais significativa. Portanto para as próximas etapas será considerada a quantidade de 1040 ΔLCOM acima do limiar, que corresponde a exatamente a quantidade de classes distintas.

3.3 Análise Qualitativa da Evolução do LCOM

Para avaliar quais estratégias de refatoração foram aplicadas nas classes onde ocorreram essas variações, foi realizada uma análise qualitativa similar à realizada para a classe IndexWriter. Devido a grande quantidade de classes um total de 1040, fica inviável realizar a análise para cada uma delas. Nesse contexto, foi realizada uma análise por amostragem. O tamanho da amostra foi definida com base nos seguintes critérios:

1. Para cada sistema foi considerado 10% da quantidade de classes nas quais ocorreram ΔLCOM acima do limiar. Esse valor foi estimado para obter uma amostra representativa e estatisticamente significante para análise.

2. Em sistemas com poucas classes esse percentual é aumentado para atingir o mínimo de 5 classes por sistema. Dessa forma a representatividade de cada sistema foi garantida.

Com base nesses critérios a representatividade de cada sistema é garantida na amostra. A tabela 5 apresenta a quantidade de classes selecionadas para cada sistema. Observa-se que o único sistema que não atingiu a cota mínima de classes foi o Javacc, portanto o percentual de classes selecionadas para esse sistema foi maior.

Sistema	**Total de Classes**	**10% do Total**	**Amostra**
jmeter	53	5,3	5
lucene	139	13,9	14
ant	66	6,6	7
log4j	45	4,5	5
jetty	219	21,9	21
xerces	122	12,2	12
javacc	10	1	5
tomcat	386	38,6	38
Total:	1040	104	107

Tabela 5 – Amostragem de Classes por Sistema.

Para determinar qual tipo de estratégia e quais operações foram realizadas, os códigos fontes das classes foram comparados entre as duas *releases* nas quais ocorreram o ΔLCOM. Os principais pontos avaliados durante a análise são:

- ❏ Quais os tipos de métodos foram movidos
- ❏ Para qual classe os métodos foram movidos
- ❏ Como foi realizada a ligação entre a classe de origem e a classe de destino
- ❏ Em qual momento a classe de destino foi criada no sistema
- ❏ O impacto gerado na métrica LCOM da classe de destino

3.4 Resultado da Análise Qualitativa das Refatorações

Para determinar quais tipos de estratégias foram adotadas ao aplicar as refatorações, foi realizada uma análise qualitativa comparando os códigos das classes entre as *releases* nas quais ocorreram as refatorações. Foi selecionada uma amostra contendo 10% das classes que sofreram refatorações em cada sistema. Para sistemas cuja amostra não atingiu o mínimo de 5 classes, esse percentual foi aumentado. Os pontos avaliados durante a análise foram:

1. Percentual de métodos get()/set() movidos: Para determinar se exite uma priorização na escolha dos métodos, foram analisados os percentuais de métodos get()/set() movidos da classe de origem para a classe de destino.

2. Tipo de ligação entre as classes: Foram analisados os tipos de ligação e a relação de dependência entre a classe de origem e classe de destino. Através dessa análise foi possível identificar se existe uma tendência no uso de variável de instância para acessar os métodos movidos.

3. Momento na qual a classe de destino foi criada e os métodos movidos: Para determinar se a estratégia de mover gradativamente os métodos para a classe de destino ocorre com frequência, foi avaliado se a classe de destino já existia antes de ocorrer a refatoração na classe de origem.

A Tabela 6 apresenta o resultado da análise feita sobre a amostra da base de refatorações. Na coluna Refatoração são apresentados os tipos de estratégia em termos de momento na qual a classe foi criada e os métodos movidos. Uma refatoração do tipo gradativa indica que a classe de destino já havia sido criada antes da remoção dos métodos da classe original. Uma refatoração do tipo imediata indica que a classe de destino foi criada na mesma *release* que os métodos foram movidos. Com relação à escolha dos métodos a serem movidos, pode ser observado uma tendência à escolher métodos públicos do tipo get()/set(). Com relação as demais métodos movidos foi observado que apesar de não possuírem nomes get() e set(), eles assumem papéis semelhantes em modificar os valores dos atributos, como por exemplo add(), read(), update(). Esses métodos apesar de não terem os nomes get() e set() possuíam funções de adicionar, ler, e atualizar os valores dos atributos da classe.

Com relação ao tipo de ligação entre as classes foi identificado uma predominância do uso de variável de instância para referenciar os métodos movidos. A ligação entre as classes através de herança ocorreu com menos frequência, independente da forma como os métodos foram movidos.

Refatoração	Ocorrência	get()/set() Movidos	Ligação entre Classes
Gradativa	30,77%	58%	Variável de Instância
Imediata	46,15%	61%	Variável de Instância
Gradativa	7,69%	42%	Herança
Imediata	15,38%	50%	Herança

Tabela 6 – Resultado Analise Qualitativa.

A seguir são discutidas as vantagens e desvantagem sobre cada tipo de estratégia, bem como o impacto gerado sobre as métricas de qualidade.

3.4.1 Refatoração imediata

A estratégia de refatoração imediata identificada na análise qualitativa apresenta as seguintes vantagens:

- Impacto imediato sobre as métricas de qualidade da classe de origem: Ao aplicar esse tipo de estratégia o cojunto de métodos é removido ao mesmo tempo em que a classe de destino é criada. Dessa forma o impacto sobre a classe de origem é imediato em termos de redução da quantidade de métodos e aumento da coesão.

- Não gera duplicação de código: Ao realizar esse tipo de refatoração os métodos movidos não são duplicados, dessa forma, as alterações futuras realizadas na classe de origem não precisam ser replicadas na classe de destino.

Em contrapartida a estratégia de refatoração imediata apresenta a seguinte desvantagem:

- Impacto direto e indireto sobre as sub-classes do sistema: Ao realizar esse tipo de refatoração um impacto pode ser gerado sobre as classes que de alguma forma dependem de recursos provenientes da classe a ser refatorada. Por exemplo se um método da super-classe que é utilizado pela sub-classe é movido para nova classe durante a refatoração, deverá ser criada uma ligação entre a sub-classe e a nova classe para que o método possa ser acessado. Essa relação de dependência pode ser tanto em termos de ligação via variável de instância, herança, ou até mesmo por duplicação de código.

A figura 5 apresenta um exemplo de refatoração imediata realizada sobre a classe **BaseResourceCollectionWrapper** do sistema **Ant**

Pode ser observado que o impacto gerado sobre a classe de origem é imediato e ocorre na mesma release na qual a classe de destinto **AbstractResourceCollectionWrapper** é criada. Entretanto, observa-se que no momento da criação a classe de destino apresenta uma perda de LCOM maior do que a classe de origem. Isso ocorre em parte devido ao fato que a partir daquele momento a nova classe já começa a receber os novos métodos que seriam inseridos na classe de origem.

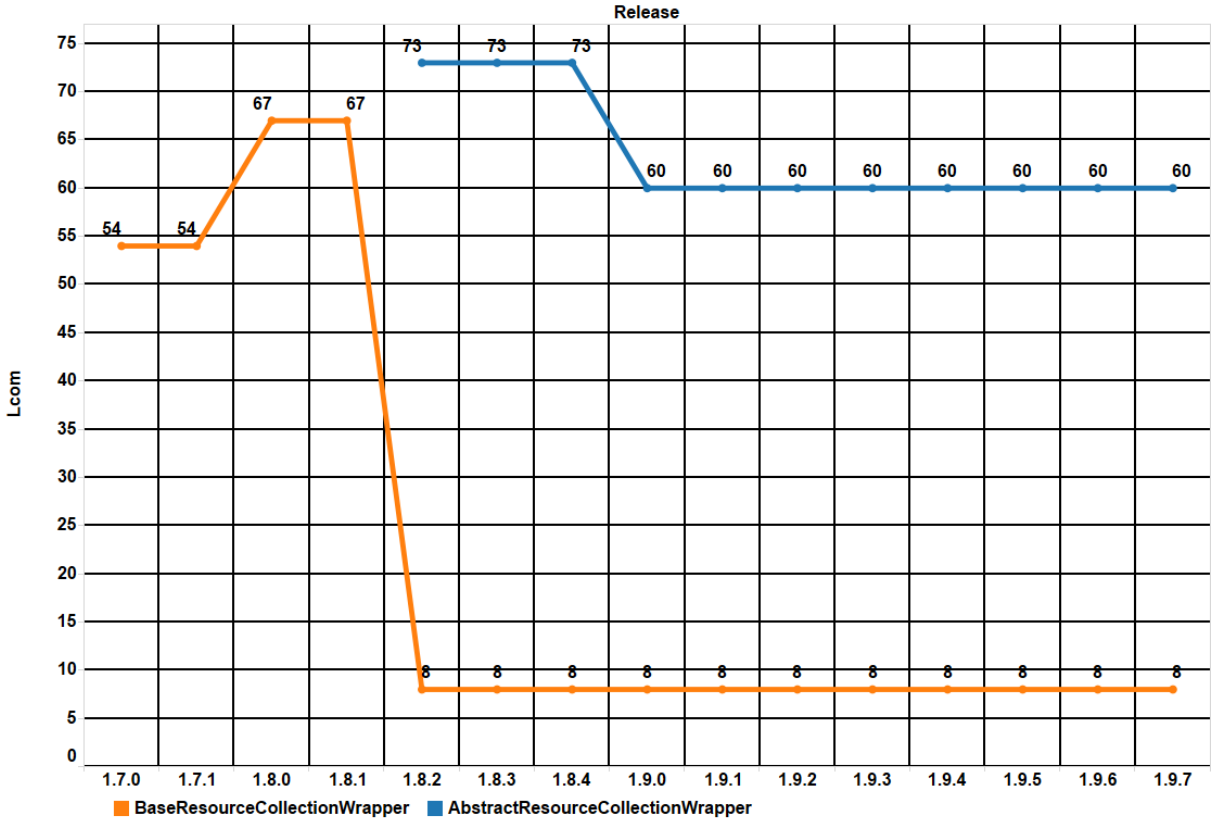

Figura 5 – Refatoração imediata da classe BaseResourceCollectionWrapper

3.4.2 Refatoração gradativa

A estratégia de refatoração gradativa identificada na análise qualitativa apresenta as seguintes vantagens:

- Tempo de adaptação: Ao remover gradativamente os métodos da classe de origem os desenvolvedores que consomem recursos dessa classe podem adaptar o código para usar os métodos da classe de destino. A adaptação do código refere-se a alteração nas classes para que elas passem a utilizar os recursos movidos para a classe de destino e deixem de usar os métodos marcados como *deprecated* da classe de origem. Essa estratégia minimiza o impacto sobre as classes que dependem direta ou indiretamente da classe a ser refatorada.

Em contrapartida a estratégia de refatoração gradativa apresenta as seguintes desvantagens:

- Geração de código duplicado temporariamente: Ao optar por remover gradativamente os métodos selecionados serão temporariamente duplicados na classe de destino. Durante esse período qualquer manutenção envolvendo a classe de origem pode se tornar mais complexa.

A figura 6 apresenta refatoração gradativa da classe **CompressedResource** do sistema **Ant**.

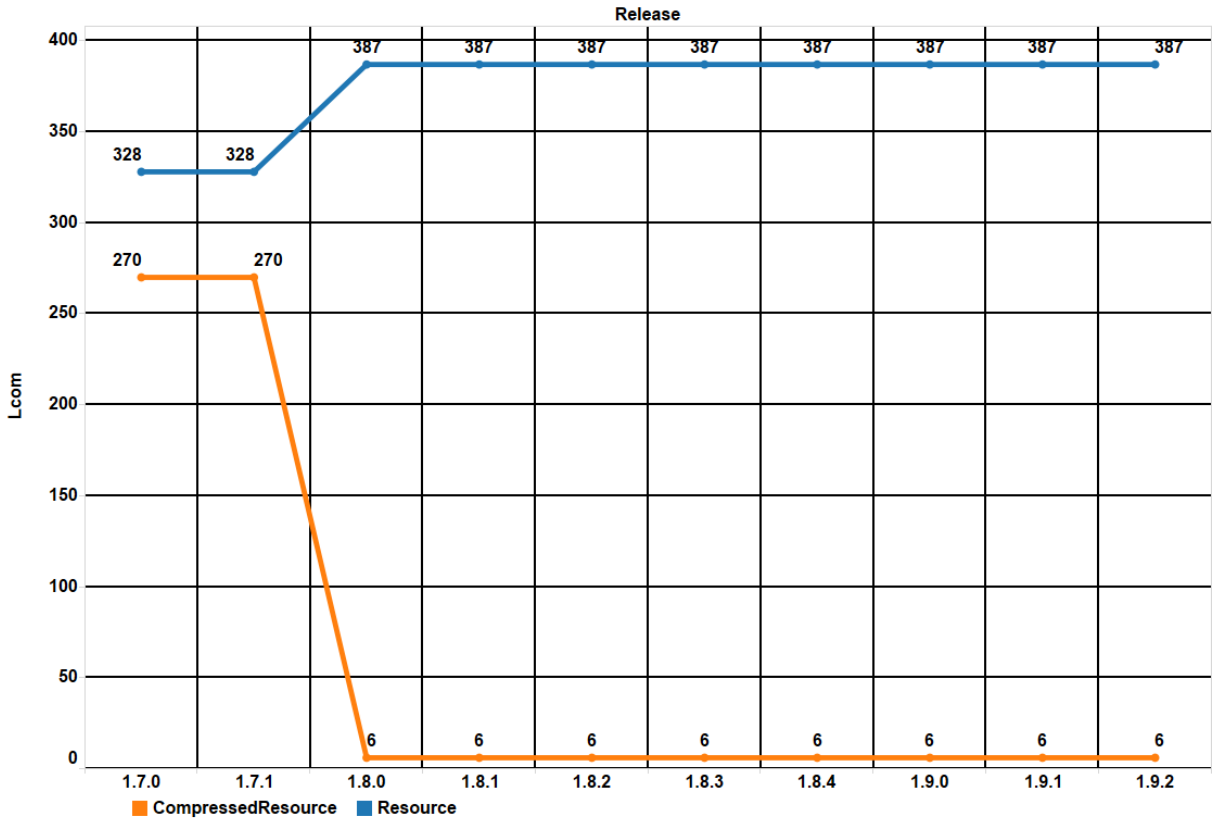

Figura 6 – Refatoração gradativa da classe CompressedResource

Pode ser observado que após a refatoração a classe de destino começa a perder a coesão. Com relação ao período que os métodos permanecem duplicados, foi constatado que a duração é geralmente o mínimo de duas *releases* . Entretanto, ao realizar esse tipo de recomendação o tempo que os métodos devem permanecer na classe de origem ficou a critério do desenvolvedor. Neste contexto o desenvolvedor pode escolher o melhor momento para remover os métodos marcados como **deprecated** da classe do origem.

Neste capítulo foram apresentados o método e os resultados da análise temporal. Estes resultados foram usados para criar um modelo preditivo de refatoração que será descrito em detalhes no próximo capítulo.

Capítulo 4

Modelo Preditivo para Refatoração

A análise temporal apresentada no capítulo anterior identificou as classes que passaram por importantes reestruturações. A identificação dessas classes foi possível devido à comparação da métrica LCOM da classe entre duas *releases* consecutivas.

Entretanto, para prever se uma classe será reestruturada não possível analisar as suas versões consecutivas da classe com LCOM alto e baixo, é somente possível analisar as informações da *release* atual. Portanto foi proposto um modelo preditivo de refatoração com base nas 1040 classes selecionadas na análise anterior que passaram por refatorações. A hipótese a ser testada é de que métricas estruturais e de tamanho podem produzir um modelo preditivo adequado para classes com oportunidades de refatoração.

4.1 Construção do Modelo Preditivo

Para determinar se existe algum padrão nas demais métricas que permita prever as situações identificadas nas análises anteriores, foi aplicado um modelo preditivo de árvore de decisão. O objetivo do modelo é analisar os valores assumidos pelas métricas na *release* anterior à queda da métrica LCOM. Para isso é necessário ter uma base na qual as classes não sofreram alterações no nível de LCOM para comparar com a base de 1040 classes. A Tabela 7 apresenta os filtros aplicados para obter ambas as bases necessárias para o modelo.

Para obter a base de comparação utilizada no modelo, uma amostra de tamanho equivalente à da base de refatoração foi selecionada aleatoriamente do conjunto ΔLCOM = 0. Foi criada uma marcação para identificar se a classe pertente à base de refatoração ou de comparação. O conjunto de dados composto pelas 2080 amostras proveniente dessas bases será utilizado para treinar e avaliar a precisão e acurácia do modelo. Uma terceira base de dados será necessária para realizar as recomendações de refatorações para os desenvolvedores. Essa base será composta pelos dados das métricas da classes coletados na última *release* dos sistema. Dessa forma as recomendações de refatoração serão feitas

Base de Refatorações	
Filtro	**ΔLCOM: 471784**
ΔLCOM <> 0	15881
ΔLCOM < 0	6076
ΔLCOM > Limiar	2137
Maior ΔLCOM	1040

Base de Comparação	
Filtro	**ΔLCOM: 471784**
ΔLCOM = 0	455903
Amostra ΔLCOM	1040

Tabela 7 – Filtros aplicados para a base do modelo

para as classes nas versões mais recentes do sistema. A Tabela 8 apresenta a quantidade de classes para cada sistema em sua última *release*.

sistema	**Classes última Release**
lucene	1595
ant	1313
tomcat	793
javacc	179
JMeter	634
xerces	850
jetty	740
log4j	913

Tabela 8 – Base Recomendações

Após realizar a separação das bases de dados necessárias, o modelo de árvore de decisão foi aplicado. A Figura 7 apresenta o processo de aplicação do modelo.

As etapas de aplicação do modelo sobre as bases de dados são descritas a seguir:

1. Leitura da base de dados de contendo as métricas e a marcação de refatoração. A marcação será FALSE para as amostras da base de comparação, e será TRUE para as da base de refatorações.

2. Seleção de amostra balanceada. Nesta etapa os filtros da base de comparação descritos na Tabela 7 são aplicados para garantir que tenham 1040 amostras de cada base(comparação e refatoração).

3. Separação em conjunto de dados de treinamento e de teste. Nesta etapa a base composta pelas 2080 amostras é dividida em dois conjuntos. O primeiro é o conjunto de treino composto por 70% das amostras, e o segundo é o conjunto de teste utilizado para avaliar a precisão do modelo composto por 30%. A separação dos conjuntos é feita de forma estratificada, garantindo que a mesma proporção de amostras de refatoração e de comparação estejam em ambos os conjuntos.

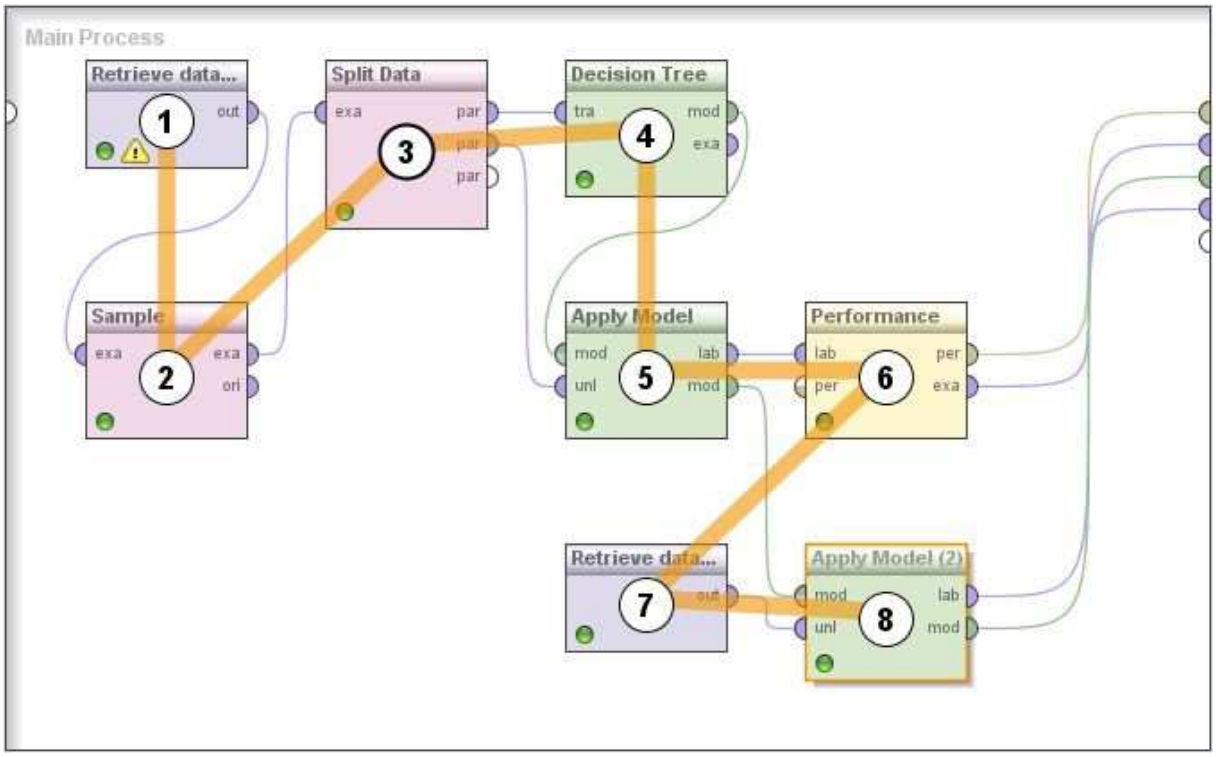

Figura 7 – Processo do Modelo de Árvore de Decisão

4. Parametrização e treinamento do algoritmo de árvore de decisão. Nesta etapa, são configurados os parâmetros da árvore como: quantidade mínima de amostras para realizar o split dos nós, quantidade mínima para criar uma folha e nível máximo de profundidade.

5. Aplicação do algoritmo sobre o conjunto de dados de teste. Nesta etapa o algoritmo treinado é aplicado sobre a base de teste.

6. Avaliação da performance do algoritmo criação da matriz de confusão. Nesta etapa a cada nova configuração de parâmetros para o algoritmo foi executado e sua precisão e acurácia foram medidos.

7. Leitura da base de validação. Nesta etapa, é feita a leitura da base contendo as métricas das classes dos sistema na última *release*.

8. Aplicação do algoritmo sobre o conjunto de dados de validação. Nesta etapa o algoritmo é aplicado sobre a base que será usada para gerar as recomendação de refatoração para os desenvolvedores.

4.2 Resultado do Modelo Preditivo de Refatoração

4.2.1 Árvore de decisão de refatoração

A Figura 8 apresenta a árvore de decisão gerada pelo modelo preditivo de refatoração. Na árvore de decisão os nós localizados em níveis mais altos são as variáveis que possuem maior peso para a classificação. Os rótulos das arestas representam os intervalos de valores assumidos pelas variáveis. Por exemplo um rótulo com a marcação **>9.5** indica que ao assumir valores assumir valores que 9.5 a variável leva a uma determinada classificação apresentada na fola. Ao analisar os testes realizados em cada nó da árvore que levam às folhas cuja marcação é **TRUE**, é possível identificar os padrões nos valores das métricas de qualidade que levam uma classe a ser refatorada. Os casos nos quais as folhas possuem uma barra azul e vermelha indicam quem não é sempre que as variáveis assumem a valor apontado na aresta que a classe é refatorada. Um detalhe importante é que dentre as métricas apontadas pelo modelo, somente a métrica CAM(*Cohesion Among Methods*) possui valores fracionários, com as demais métricas apresentando valores inteiros. O motivo pelo qual essas métricas aprecem com valores fracionários é devido à forma como o algoritmo da árvore realizou o *split* e separou os intervalos.

O primeiro padrão identificado diz respeito à métrica WMC(*Weighted Methods Per Class*). De acordo com (CHIDAMBER; KEMERER, 1994) o número de métodos é um indicador de quanto tempo e esforço é necessário para realizar manutenções na classe. Outra informação importante com relação à aplicação dessa métrica é que os dados empíricos obtidos por (CHIDAMBER; KEMERER, 1994) apontam que a maior parte das classe apresentam um valor entre 0 e 10 para essa métrica. Analisando o teste realizado para essa métrica na árvore pode ser constatado que classes que apresentam valores > 9.5 (arredondado para 10 por ser uma métrica de valores inteiros) tendem a sofrer refatorações. De fato analisando a base de refatorações em 78% das classes nas quais as métricas assumiam esse valor, a classe havia sido refatorada.

O segundo padrão identificado diz respeito à métrica LOC(*Lines of Code*), que ao assumir um valor maior que 597, juntamente com um valor de WMC menor ou igual 9,5 e maior que 8,5 apresenta uma tendência à refatoração. Observa-se uma correlação entre os valores assumidos pelas duas métricas, pois quanto maior a quantidade de métodos maior será a quantidade de linhas de código em uma classe. Em 98% das vezes que essa condição foi satisfeita ocorreu a refatoração.

O terceiro padrão identificado foi WMC <= 9,5 e LOC <= 597, NPM(*Number of Public Methods*) > 7,5 e CAM(Coheson Among Methods) <= 0,3. A variável NPM conta a quantidade de métodos públicos declarados na classe, ao realizar análise qualitativa das refatorações foi identificado uma tendência em mover métodos do tipo get() e set() que são geralmente declarados como *public*. Com relação à métrica CAM, quanto menor o seu valor, menor será a coesão entre os métodos da classe.

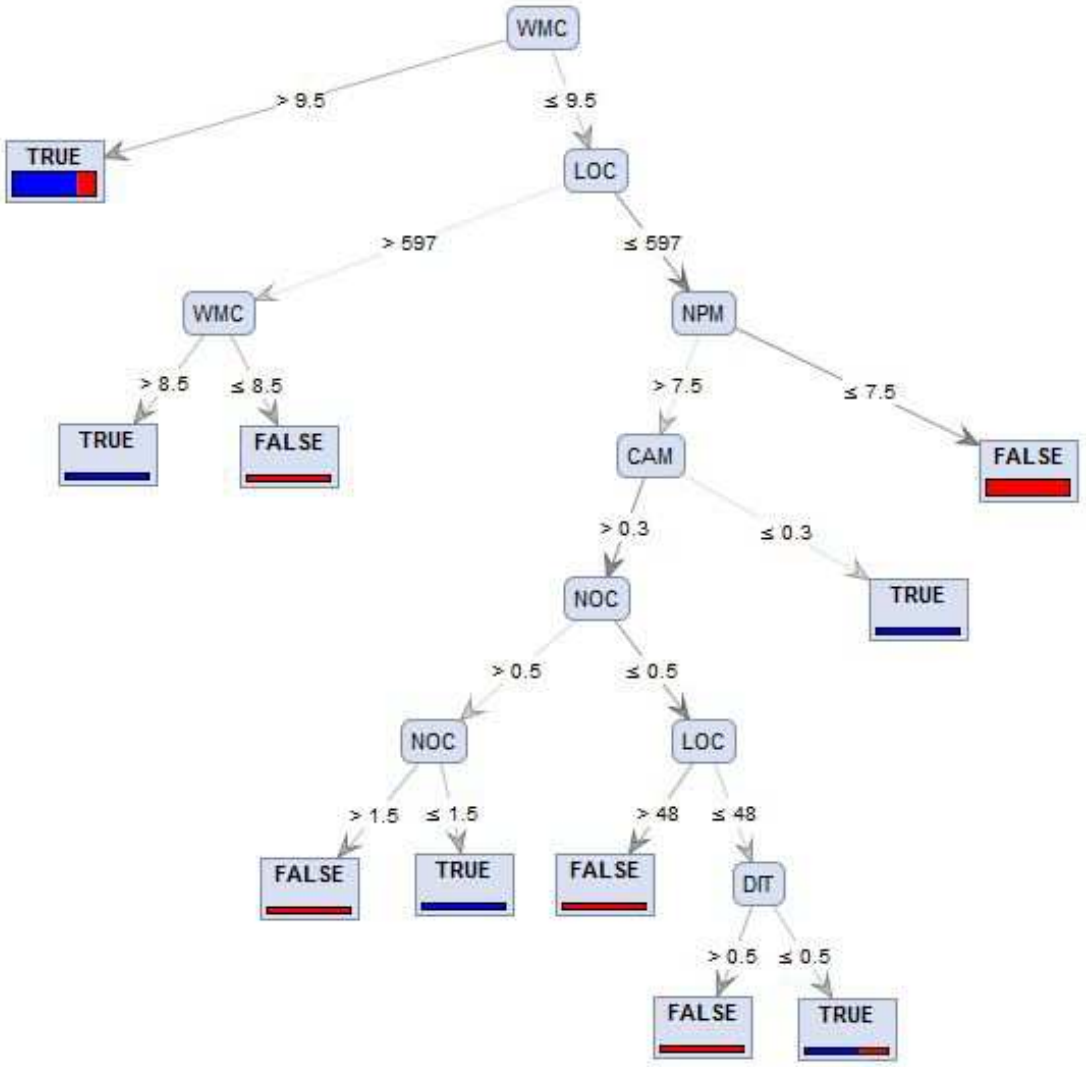

Figura 8 – Árvore de Decisão de Refatoração

O quarto padrão identificado diz respeito à métrica NOC(Number of Children), quando a métrica CAM assume um valor maior que 0,3 e a métrica NOC um valor entre 0,5 e 1,5. A métrica NOC indica a quantidade de sub-classes ligadas diretamente à classe. Classes com grande quantidade de sub-classes tendem a exercer maior influência sobre o sistema e a serem mais difíceis de refatorar (CHIDAMBER; KEMERER, 1994). Na árvore quando essa métrica assume valor maior do que 1,5 a classe tende a não ser refatorada, isto é somente classes com no máximo uma sub-classe foram refatoradas. Esse padrão ocorreu 97,05% das vezes que a condição foi satisfeita.

O quinto padrão diz respeito à métrica DIT(*Depth of Inheritance Tree*), essa métrica mede em que nível da árvore de herança a classe se encontra. Classes em níveis muito baixos na hierarquia apresenta maior complexidade devido à influência de classes nos níveis superiores. Na árvore de decisão do modelo quando a métrica DIT assume valores

menor do que 0,5 e a métrica LOC <= 48 a classe tende a ser refatorada. Esse padrão ocorreu em 65% das vezes nas quais a condição foi satisfeita.

4.2.2 Avaliação da precisão do modelo

A tabela 9 apresenta os resultados da aplicação do modelo sobre a base de teste. A base composta por 604 amostras foi utilizada para avaliar a precisão, revocação e acurácia do modelo. O cálculo da precisão é feito dividindo os verdadeiros positivos pela soma dos verdadeiros positivos com os falso positivos, o mesmo é feito para os verdadeiros negativos. Precisão de verdadeiros positivos = 294/(294+78) = 79,03%. Precisão de verdadeiros negativos = 224/(8+224) = 96,55%. O cálculo da revocação é feito dividindo os verdadeiros positivos pela soma dos verdadeiros positivos com os falsos negativos, o mesmo é feito para os falso positivos. Revocação verdadeiros positivos = 294/(294+8) = 97,35%. Revocação falsos positivos = 224/(294+8) = 74,17%. Pode ser observado que o modelo obteve uma precisão de 79,03% em prever que a classe seria refatorada. Isso indica uma baixa incidência de falso positivos aproximadamente 21%.

		Real		
		TRUE	FALSE	Precisão
Previsto	TRUE	294	78	79,03%
	FALSE	8	224	96,55%
	Revocação	97,35%	74,17%	

Tabela 9 – Matriz de Confusão do Modelo

Também pode ser observado um alto nível de precisão em identificar as classes que não tem propensão à refatoração 96,55%. O que indica um baixo nível de falsos negativos, aproximadamente 5%. De uma forma geral, considerando ambas as situações o modelo apresenta uma acurácia de 85,76%. O cálculo da acurácia é feito somando a quantidade de verdadeiros positivos e verdadeiros negativos e dividindo pela soma das quatro situações possíveis: Acurácia = ((294+224)/(294+78+8+224)) = 85,76% Portanto o modelo possuí acurácia o suficiente para ser utilizado para gerar recomendações de refatorações.

No próximo capítulo serão apresentadas as percepções dos desenvolvedores sobre as recomendações geradas.

Capítulo 5

Survey com Desenvolvedores

Após desenvolver um modelo que permite identificar oportunidades de refatoração nas classes dos sistemas avaliados, um *survey* com os desenvolvedores foi realizado para discutir sobre as recomendações propostas. Nesse contexto, o *survey* possibilitará obter uma visão dos desenvolvedores sobre o tema, medindo o seu nível de interesse e engajamento em aplicar refatorações que melhorem a modularização do sistema.

5.1 Objetivos

Os principais objetivos do *survey* são:

- Realizar a validação externa da precisão do modelo em localizar oportunidades de refatoração. Com base nas respostas dos desenvolvedores será avaliado se as classes para as quais as recomendações foram geradas são locais apropriados para refatoração.

- Complementar a análise qualitativa realizada sobre a base de refatoração. A partir da discussão em torno da guia de recomendação, as vantagens e desvantagens apontadas pelos desenvolvedores serão analisadas para obter estratégias mais adequadas.

- Ajustar o formato das guias de recomendações para uma melhor compreensão da abordagem. Caso os desenvolvedores tenham dificuldades em compreender a recomendação, o formato da guia poderá ser ajustado.

- Identificar outras características nas classes e no sistema que podem dificultar o processo de refatoração. De acordo com as considerações dos desenvolvedores, poderão ser identificadas outras variáveis que podem dificultar a aplicação de determinada estratégia.

5.2 Método

As etapas do método utilizadas para apresentar as recomendações, bem como o formato das guias, são apresentados a seguir:

1. Seleção das classes para as quais serão geradas as recomendações. Nessa etapa as classes indicadas pelo modelo com maior propensão à refatoração são selecionadas. O conjunto de amostras é ordenado de forma decrescente pelo nível de confiança que o modelo a classificou. Dessa forma as recomendações são geradas para as classes com maior probabilidade de sofrer uma refatoração. Foram selecionadas as 7 classes com maior probabilidade de refatoração de cada sistema. Essa quantidade de classes foi escolhida para obter o equilíbrio entre uma amostra suficientemente grande para obter uma maior confiabilidade do resultado, mas que ao mesmo tempo seja viável de ser analisada pelos desenvolvedores.

2. Criação da guia de recomendação. Nessa etapa para cada classe selecionada na etapa anterior uma guia de recomendação de refatoração é criada. A guia de refatoração é composta por tês partes:

    a) Local onde a oportunidade foi encontrada. Nessa parte é indicado o nome da classe apontada pelo modelo como propensa à refatoração.

    b) Referência a uma refatoração passada. Nessa parte é apresentada a estratégia usada para refatorar uma classe do sistema que estava em situação similar. Essa informação foi obtida através da análise qualitativa da base de refatorações.

    c) Estratégia a ser aplicada. Nessa etapa são descritas quais operações devem ser realizadas para a refatoração. Como a estratégia envolve a criação de uma nova classe, o nome é sugerido com base no nome da classe original. Os métodos a serem movidos também são listados juntamente com a sugestão de como deve ser feita a ligação entre as classes.

3. Criação das *issues* para gerar as recomendações. Nessa etapa as guias de recomendação são apresentadas através de *issues* criadas nos sistemas de repositório dos sistemas analisados.

5.3 Modelo de guia de refatoração

A seguir é apresentado um exemplo de guia de refatoração, bem como a descrição de cada parte do formato da guia.

5.3.1 Local da oportunidade

Nesta parte é apresentada o nome da classe na qual foi identificada a oportunidade de refatoração. No exemplo a classe selecionada foi a SegmentInfo do sistema Lucene.

- *"Hello everyone. I was analyzing the modularization of some classes, and I identified that the class SegmentInfos has an opportunity for cohesion improvement."*

5.3.2 Referência à uma refatoração passada

Nesta parte é apresentada uma referência à uma refatoração realizada em uma classe que possuía problemas similares à da classe SegmentInfos. Neste caso a referência foi feita à refatoração da classe IndexWriter.

- *"The class IndexWriter was in the same situation and the problem was solved as follows: The IndexWriterConfig class was created, and several get() and set() methods that were used only to configure the class parameters were moved from IndexWriter to IndexWriterConfig. The new class was then accessed through an instance variable in IndexWriter. This strategy has cleaned and improved IndexWriter cohesion."*

5.3.3 Estratégia a ser adotada

Nesta parte da guia é apresentada a estratégia a ser adotada para realizar a refatoração. É recomendada a criação de uma nova classe SegmentInfosConfig para receber os métodos de configuração da classe SegmentInfos. Também é apresentada a forma como a nova classe deverá ser referenciada a partir da classe original, neste a caso a ligação é feita através de variável de instância.

- *"With this in mind, I would recommend creating a new class: SegmentInfosConfig, and moving the following methods: getLastCommitGeneration, getLastCommitSegmentsFileName, getSegmentsFileName, getNextPendingGeneration, getId, getVersion, getGeneration, getLastGeneration, setInfoStream, getInfoStream, setNextWriteGeneration, setUserData, setVersion, getCommitLuceneVersion, getMinSegmentLuceneVersion from the SegmentInfos. Those parameters accessed by an instance variable in the SegmentInfos. Moreover, the orthogonality is the design would be enhanced. What do you think about that?"*

5.4 Resultado do Survey

A Tabela 10 apresenta os resultados obtidos com survey para as recomendações das classes selecionadas. Foram realizadas um total de 56 recomendações das quais 31 obtiveram respostas. Dessas 31 respostas, 13 foram classificadas como neutras. Uma resposta

neutra significa que o desenvolvedor não avaliou a recomendação, mas apresentou questões referentes à dificuldade de aplicação de refatoração e problemas relacionados. Das 18 respostas restantes que efetivamente avaliaram as recomendações, os resultados foram computados considerando se a classe para qual a recomendação foi feita realmente precisava ser refatorada e se a guia de como refatorar estava correta.

					Local		**Recomendação**	
sistema	**Guias**	**Respostas**	**Neutras**	**Avaliações**	**correto**	**incorreto**	**correta**	**incorreta**
ant	7	7	0	7	6	1	6	1
log4j	7	1	0	1	1	0	1	0
Jetty	7	1	0	1	1	0	0	1
Lucene	7	7	5	2	1	1	0	2
Xerces	7	1	1	0	0	0	0	0
Jmetter	7	7	7	0	0	0	0	0
Tomcat	7	7	0	7	7	0	0	7
Javacc	7	0	0	0	0	0	0	0
Total:	56	31	13	18	16	2	7	11
%Total:	100%	55,36%	41,94%	58,06%	88,89%	11,11%	38,89%	61,11%

Tabela 10 – Resultado das recomendações do survey.

Para garantir o anonimato dos desenvolvedores os seus nomes foram substituídos por pseudônimos com as iniciais **dev**, o sistema que ele faz parte e um número de identificação. Exemplo: **dev_ant_1**, **dev_ant_2**, **dev_jetty_1**, **dev_log4j_1**.

5.4.1 Discussão sobre as respostas positivas

Nessa seção serão discutidas as respostas positivas dadas pelos desenvolvedores com relação às recomendações. Também serão discutidas as alternativas adotadas para contornar as dificuldades no processo de refatorção, bem como novas estratégias identificadas durante a interação com os desenvolvedores.

5.4.1.1 Identificação de uma estratégia de refatoração preventiva

Ao realizar a recomendação de refatoração para a classe Project do sistema Ant o desenvolvedor dev_ant_1 respondeu:

"*I'm sure your changes would improve the quality of the Project class but at the same time it scares me.*"

Sobre os motivos que o deixaram com receio em relação a aplicar a refatoração, ele afirma que a principal preocupação está relacionada com os desenvolvedores que consomem recursos dessa classe e que precisariam ser informados sobre a mudança:

"*I guess what I'm trying to say is be extra careful with this class and ensure the refactoring doesn't break any assumption anybody who knows the current code may have about the inner workings of the class.*"

Por fim ele completa que devido à essa restrição nem todas as alterações que melhoram o código poderão ser realizados:

"*In the end this may mean you can't make all the changes that would improve the code.*"

Considerando essas respostas, uma estratégia de refatoração gradativa foi proposta no lugar da refatoração imediada. Dessa forma os métodos seriam marcados como *deprecated* como um aviso para os desenvolvedores de que eles serão removidos e que existem outros para serem usados no lugar. Sobre a proposta dessa nova estratégia ele responde o seguinte:

"*This strategy is a sound strategy if you expect users of your API to follow releases closely. Unfortunately this is not what we see with Ant. Right now we are fielding a bug reported by somebody who is migrating from Ant 1.6.2 to 1.10.1 - 1.6.2 has been superseeded by 1.6.3 in 2005. This is not uncommon.*"

"*This has led to us never removing any methods at all, no matter how long it has been deprecated.*"

Tendo em vista a dificuldade em atualizar os desenvolvedores sobre as alterações realizadas, e considerando que remover os métodos da classe não é uma opção viável, foi proposta uma terceira estratégia de refatoração. A estratégia em criar uma nova classe para receber os métodos get() e set() que possam vir a ser inseridos na classe Project. Dessa forma apesar de não ser possível solucionar o problema atual da classe ao menos será evitada uma maior perda de coesão no futuro. Essa estratégia foi apresentada ao desenvolvedor com o nome de **refatoração preventiva**. Ao perguntar ao desenvolvedor se ele adotaria essa estratégia inclusive como um padrão para criação de novas classes, ele respondeu: "*Absolutely.*"

Com base nessa discussão e na nova estratégia de refatoração, foi realizada uma análise para identificar se a dificuldade para remoção de métodos ao refatorar ocorre também em outros sistemas. A Tabela 11 apresenta a quantidade de métodos movidos pela quantidade de ΔLCOM acima do limiar para cada sistema. Observa-se que devido à dificuldade de

Sistema	**ΔLCOM > Limiar**	**Métodos Movidos**	**Métodos por ΔLCOM**
jmeter	53	264	5
lucene	139	1059	8
ant	66	192	3
log4j	45	393	9
jetty	219	1617	8
xerces	122	835	7
javacc	10	100	10
tomcat	386	1859	5

Tabela 11 – Métodos por ΔLCOM.

remoção de métodos, o sistema Ant possui o pior índice de movimentação de métodos por refatoração, seguido pelo JMeter e o Tomcat.

Para determinar se a estratégia de refatoração preventiva já foi utilizada no Ant, uma amostra de 6 classes as quais tiveram seus códigos duplicados em novas classes, mas que não tiveram os métodos *deprecated* removidos foram analisadas.

A Figura 9 apresenta uma refatoração preventiva feita sobre a classe FTP. Pode ser observado que após a criação da nova classe, a classe de origem não sofreu mais nenhuma degradação em termos de coesão.

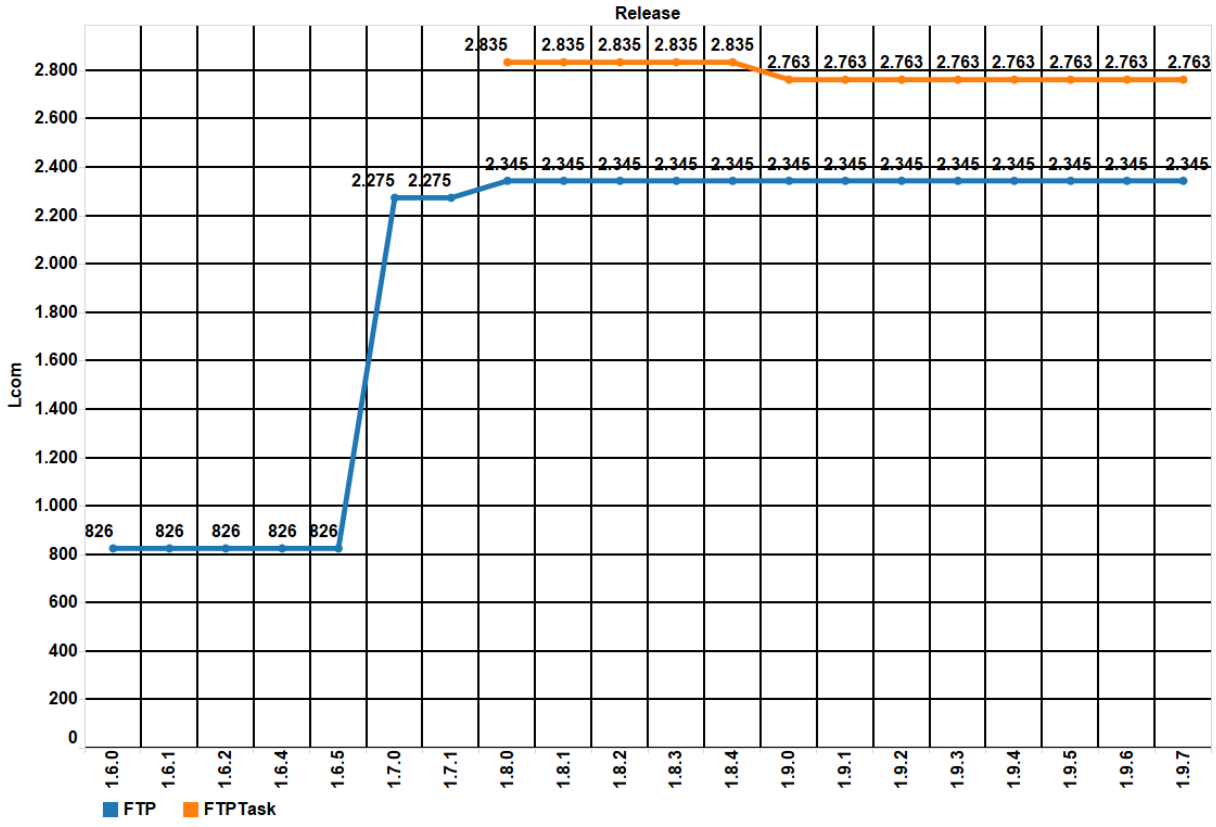

Figura 9 – Refatoração preventiva da classe FTP

Essa estratégia evidencia uma preocupação em não agravar ainda mais o problema já existente na classe. Entretanto, devido à dificuldade para realizar a remoção de métodos, a classe de origem permanece com um nível de LCOM.

5.4.1.2 Localização de oportunidades de refatoração e desenvolvimento de estratégias

Apesar de nem todas as estratégias de refatoração sugeridas tenham sido avaliadas como positivas, a abordagem se demonstrou bastante promissora com relação à localização das oportunidades. Alguns desenvolvedores se dispuseram a ajudar a formular novas estratégias mais adequadas às necessidades e restrições do sistema, ao reconhecer a necessidade de melhoria nas classes apontadas pela abordagem. Ao avaliar uma recomendação e tendo rejeitado a estratégia de refatoração o desenvolvedor afirma:

dev_jetty_1: "*So yes I'd like to see a cleaner class, but it is a much more difficult task than it looks.*"

dev_jetty_1: "*But if you want to have a crack at it, we'd be happy to receive a pull request and would review and work with you to see what could be done to improve the class.*"

Ao receber as recomendações para as demais classes da lista os desenvolvedores fizeram os seguintes comentários:

dev_ant_1: "*Many, many thanks for your interest. Once we've agreed on what would be good candidates to refactor, opening issues for each candidate would be fine. I'd be interested in hearing what kind of changes you'd like to make*"

dev_lucene_2: "*Feel free to create a JIRA and attach a patch with any refactorings you think are beneficial*"

dev_jmeter_2: "*Hello, Thanks for your proposal and analysis. It is good to have Theorical ideas of clean design.*"

Estes comentários mostram um interesse por parte dos desenvolvedores em estratégias que promovam melhorias nos códigos dos seus sistemas. Também apresenta uma oportunidade para trabalhos futuros em parceria com esses desenvolvedores, aplicando conceitos teóricos na prática.

5.4.2 Discussão sobre as respostas neutras

Nesta seção são discutidas as respostas que apesar de não avaliarem a qualidade das recomendações, elas apresentam pontos de vistas interessantes relacionas à dificuldade de refatoração.

5.4.2.1 Reuso e duplicação de código

O reuso de código através da duplicação não é algo incomum em sistemas de software. Ao mesmo tempo que copiar o código pode economizar tempo em curto prazo, essa prática torna o processo de manutenção muito mais complexo. Ao recomendar a refatoração de uma classe para o sistema Log4J o desenvolvedor faz o seguinte comentário:

dev_log4j_4: "*this class derives from Apache Commons Lang. Every now and then we look at the Lang code to see what patches have been applied and update this class with them.*"

O mesmo aconteceu para o sistema Tomcat:

dev_tomcat_2: "*This code originates in Commons DBCP 2 and is copied into Tomcat (with a little re-naming)*"

E na discussão para estratégias alternativas de refatoração do Jetty:

dev_jetty_1: "*it is hard to strike a balance that does not have too much duplication or data copying.*"

Observa-se que os desenvolvedores de alguma forma tentam evitar a duplicação, mas ao mesmo tempo reconhecem a dificuldade em evitar essa prática. Uma classe gerada a partir de duplicação de código ainda pode ser refatorada, porém o esforço necessário será maior. Ao ver a discussão e o comentário apontando para a duplicação de código o desenvolvedor do Log4j respondeu o seguinte:

dev_log4j_1: "*Right, so you might want to start in Commons Lang and then have the changes trickle down to Log4j.*"

Ele não fez objeção sobre a estratégia, desde que essa seja implementada na classe original, e posteriormente na duplicada.

5.4.2.2 Falta de conhecimento sobre a arquitetura

Durante as discussões um tema recorrente foi sobre o papel das classes nos sistema e o nível de influência que elas exercem. Nesse ponto o conhecimento prático dos desenvolvedores demonstrou-se de extremo valor para analisar as recomendações. Ao receber a recomendação de refatoração para uma classe o desenvolvedor respondeu o seguinte:

dev_xerces_1: "*I don't know this code, so every change is a major risk.*"

Ao identificar que a classe era uma duplicação de código o desenvolvedor do Log4j disse:

dev_log4j_3: "*Should we maybe add a comment in this class stating that it is borrowed from Commons Lang?*"

E outro desenvolvedor respondeu:

dev_log4j_4: "*I thought it already does.*"

Ao analisar o nível de influência que a classe Project exerce sobre o sistema o desenvolvedor comenta o seguinte:

dev_ant_1: "*AFAIK the components that integrate Ant into IDEs use subclasses of Project. We don't really know what those subclasses do.*"

Outro desenvolvedor que também não possuía essa informação sugere a utilização de uma ferramenta:

dev_ant_2: "*I would suggest running the refactored Ant through japicmp [2] or revapi [3] and examining the binary incompatibilities.*"

Ao ponderar sobre o nível de dificuldade de refatoração o desenvolvedor faz o seguinte comentário:

dev_jmeter_2: "*You'll learn a lot from starting with small fixes, then medium fixes, then architectural breaking changes.*"

Analisando os comentários dos desenvolvedores contata-se que a falta de conhecimento sobre a arquitetura do sistema aumenta significativamente a dificuldade e o risco ao aplicar uma refatoração de classe. Um fato interessante é que nenhum deles citou o uso da documentação do sistema, o que pode indicar que as documentações atuais não atendem às necessidades dos desenvolvedores. O conhecimento da arquitetura obtido a partir

da experiência em resolução de problemas, bem como a utilização de ferramentas para compreensão do código parecem se tornar mais comuns e com maior aceitação pelos desenvolvedores.

5.4.2.3 Compatibilidade com versões anteriores

Por se tratarem de sistemas que possuem diversas *releases* disponíveis, um ponto que ficou evidente nas discussões foi a necessidade de compatibilidade com versões anteriores. Ao apresentar as características do sistema o desenvolvedor faz o seguinte comentário:

dev_jmeter_2: "*having in mind that there are a lot of existing plugins and that backward compatibility is important within the project*"

Sobre a dificuldade em manter as versões antigas funcionando o desenvolvedor aponta o seguinte:

dev_ant_1: "*I am one of the people who've upheld the "backwards compatible at all cost"mantra.*"

A compatibilidade com versões anteriores apresenta um desafio muito grande e que dificulta realizar alterações no sistema, especialmente aquelas que geram impacto em nível arquitetural. Observa-se também que muito esforço é direcionado para a resolução desse problema evidenciando oportunidades para implementação de soluções automatizadas.

5.4.2.4 Quebra de builds

Em sistemas que possuem uma relação de dependência muito ampla com pacotes, APIs e plugins externos, fica evidente a dificuldade em evitar a quebrar de builds. Uma simples remoção de métodos ou mudança no tipo de atributos pode gerar essa inconsistência. Ao explicar sobre os problemas de integração do sistema o desenvolvedor afirma o seguinte:

dev_ant_1: "*We cannot remove any method used by it at all without breaking build files of people who rely on ant-contrib.*"Para evidenciar o nível de fragilidade do sistema ele completa com um exemplo:

dev_ant_1: "*Sometime last year I made an attribute final and promptly broke Eclipse's Ant integration - see bug 60582*"

Fica evidente a complexidade do ambiente sobre o qual o sistema é construído. Técnicas que visem uma integração contínua minimizando as quebras poderiam ser incorporadas às estratégias de refatoração.

5.4.2.5 Bugs com maior prioridade

Um pontos nas discussões que ficou evidente foi com relação a quais alterações devem ser priorizadas e sobre um melhor aproveitamento dos recursos humanos disponíveis. A refatoração tem como objetivo melhorar a estrutura sem alterar as funcionalidades do sistema (FOWLER; BECK, 1999). Entretanto se as funcionalidades do sistema apresentam

erros é esperado que os desenvolvedores priorizem essas correções. Sobre o estado atual do sistema o desenvolvedor a ponta o seguinte problema:

dev_xerces_1: "*There are dozens of serious complex bugs open and that's where time should be spent*"

E o sobre a disponibilidade de pessoas para avaliar as recomendações ele comenta:

dev_xerces_1: "*There are exactly two people contributing to this code base in any meaningful way right now.*"

Outro desenvolvedor sobre a situação do sistema:

dev_jmeter_2: "*there are already a lot of existing enhancements/bugs to fix*"

Sobre a disponibilidade para avaliar ou aplicar as estratégias de refatoração:

dev_tomcat_2: "*Overall, that isn't something I'm interested in pursuing. That doesn't mean someone else won't be interested. Just that I'm not.*"

dev_lucene_2: "*Sorry, I don't have the spare cycles at present...*"

Com base nessa discussão, fica evidente que as propostas para melhorias na modularização despertaram interesse por parte dos desenvolvedores, apesar de não terem sido devidamente avaliadas. Entretanto, essas propostas de melhorias possuem uma prioridade baixa. Isso fica ainda mais evidente em sistema que possuam poucos colaboradores e muitas correções esperando atendimento.

5.4.2.6 Formato da guia e informações adicionais

Sobre o formato da guia de refatoração, alguns desenvolvedores fizeram sugestões que ajudariam a compreender melhor a estratégia proposta.

dev_lucene_2: "*It would be best for you to make concrete proposals, basically an outline (I'd recommend relatively small chunks) of what kinds of changes you are thinking about. It doesn't even have to be a functioning patch for a first cut, just something concrete that people can comment on.*"

dev_jmeter_2: "*I think that proposal shoud be if possible based on concrete ode (POC or full code).*"

A questão de como deve ser apresentada a proposta varia de acordo com o sistema e o método que os desenvolvedores utilizam para avaliar as sugestões. Para alguns a descrição da estratégia foi suficiente, para outros o código resultante da refatoração deveria ser apresentado. Com base nessa discussão fica evidente a necessidade de encontrar um equilíbrio entre a quantidade de informações fornecidas e o esforço necessários para gerar as guias.

5.4.3 Discussão sobre as respostas negativas

Nesta seção serão apresentadas as respostas que dos desenvolvedores que rejeitaram as estratégias de refatoração propostas. Essas respostas são de vital importância para

uma melhor compreensão do processo de refatoração e os fatores que o impedem de ser executado.

5.4.3.1 Erro na escolha dos métodos

Sobre a recomendação de refatoração para a classe SegmentInfos do Lucene o desenvolvedor respondeu:

dev_lucene_1: "*It feels a bit wrong to me since commit gen, generation, version, etc. are not configuration parameters. I think SegmentInfos is fine as-is.*"

Ao analisar a lista de métodos sugeridos para serem movidos para a nova classe, foi observado que apesar de terem nomes get() e set(), e realizarem operações de atribuição e leitura, essas operações não estavam sendo feitas para atributos da classe. Foi constatado também que alguns desses métodos estavam sobrecarregados pelos tipos de parâmetros que eles recebiam. Em uma declaração o método atribuía um valor à uma variável, em outra declaração o mesmo método chamava outros métodos. Essa resposta ajudou a melhorar a forma como a lista de métodos é apresentada, uma vez que ao apresentar somente o nome dos métodos a sobrecarga não fica evidente.

5.4.3.2 Falta de análise de classes internas

Sobre a recomendação de refatoração para a classe Request do sistema Jetty, o desenvovledor respondeu:

dev_jetty_1: "*Note also that Request already does have an inner instance that is kind of a RequestConfig. It is already passed a MetaData.Request instance which holds method, URI, version, headers and trailers of the request. Many of the methods you list are already handled by referencing this instance*"

Ao avaliar a classe não foi possível identificar que os métodos já estavam organizados dentro uma classe interna, e devido ao tamanho da classe Request essa organização não ficou evidente. Essa resposta ajudou a direcionar a analise qualitativa para a estrutura de organização interna da classe. Essa discussão também aponta para a necessidade de criação de métricas direcionadas a medir essa características, como quantidade de classes internas e como elas são declaradas e utilizadas.

5.4.3.3 Quantidade de métodos insuficientes

Sobre a recomendação de refatoração para classe IndexSearcher do Lucene, o desenvolvedor fez o seguinte comentário:

dev_lucene_3: "*In the case of IndexWriter there were so many getters and setters that we made this change due to the sheer number*"

De fato, a proporção de métodos do tipo get() e set() na classe IndexWriter antes da refatoração era superior ao da classe IndexSearcher. Essa observação indica que é

necessário definir um limar para a quantidade de métodos get() e set() necessários para justificar a refatoração. Outro fator a ser considerado é se a classe IndexSearcher está sendo constantemente modificados e com que frequência métods get() e set() são inseridos. Com essa análise seria possível determinar se existe uma tendência a aumentar essa proporção, justificando a aplicação da refatoração.

5.4.3.4 Idade da classe e modificações constantes

Sobre a idade da classe e as modificações feitas ao longo do tempo os desenvolvedores comentam:

dev_jetty_1: "*well it is true that Request is a very large somewhat ugly class that is carrying lots of history with it. It is the mutability aspect of the class that makes it rather difficult to simplify.*"

dev_ant_1: "*Ant has been around for more than fifteen years and an eco system of xtensions has ground around it.This is something that forces us to be extra careful with refactoring. AbstractFileSet is an extremely dangerous one, as it has certainly seen a lot of extensions outside of our control*"

Com base nesses comentários fica evidente que a idade da classe no sistema apresenta dificuldades para a implementação de estratégias de refatorações. Ao ser constantemente modificada a complexidade da classe aumenta significativamente. Um ponto a ser melhorado na abordagem é identificar a quantidade de alterações sofrida pela classe ao longo do tempo e avaliar a complexidade e o esforço necessário para aplicar a refatoração.

5.4.3.5 Dificuldade de uso e perda de performance ao criar uma nova classe

Sobre o fato de ser necessário criar uma nova classe os desenvolvedores fizeram os seguintes comentários:

dev_lucene_3: "*Its an additional class the user must worry about and more complicated than a POJO: i think it makes something hard to use.*"

Nesse caso o desenvolvedor se refere a um POJO (*Plain Old Java Object*). Trata-se de um objeto java simples que não implementa nenhuma interface e não estende nenhuma classe.

dev_jetty_1: "*I would be concerned about the additional object creation and dereferences required from a performance point of view.*"

dev_tomcat_1: "*So what if the class has 61 properties? If you divide it into a "doer"class and a "configurator"class, you'll have two classes, one of which has 61 properties and the other of which can't do anything without /yet another/ object being available to configure it.*"

De fato, em nenhuma das estratégias de refatorações recomendadas as classes resultantes podem prescindir da ligação e da relação de dependência. Isso se deve principalmente ao conjunto de métodos escolhidos ser responsável pela configuração da classe de origem.

Para recomendar uma refatoração na qual as classes resultantes não dependam uma da outra, um outro conjunto de métodos deve ser identificado.

5.4.3.6 Relação Custo Benefício

Outra questão importante que pode desmotivar a aplicação da refatoração é o esforço necessário para realizar comparado com o resultado. Sobre esse ponto o desenvolvedor fez o seguinte comentário: dev_tomcat_2: "*I don't think the cost is worth the benefit.*"

Quando perguntado sobre quais custos ele se referia, ele respondeu:

dev_tomcat_2: "*The time taken to do it, the duplicated code until the deprecated code can be removed, the cost of downstream users updating their code.*"

Esta resposta indica que para realizar uma refatoração de classe muitas operações ainda são realizadas de forma manual. Sobre o real benefício obtido com a refatoração os outros desenvolvedores comentaram:

dev_jetty_1: "*I do not see how creating a RequestConfig class helps all that much*"

dev_jmeter_1: "*I wouldn't refactor just for synthetic reasons.*"

dev_jmeter_2: "*I share opinion of "not refactoring just for synthetic reasons"*"

A questão de ter uma classe mais limpa do ponto de vista dos desenvolvedores parece ainda não estar relacionada com a facilidade de compreensão e manutenção. Com base no resultado dessa discussão fica evidente que uma maior automatização da abordagem ajudaria a diminuir os custos, já que os benefícios dependem em parte da visão do desenvolvedor. Esta relação custo benefício também pode estar associada à experiências passadas nas quais o desenvolvedor aplicou refatorações e não obteve resultados satisfatórios.

Capítulo 6

Lições Aprendidas

Neste capítulo são apresentadas as respostas às perguntas de pesquisa bem como as lições aprendidas com base no resultados das análises e do survey.

6.1 Respostas às perguntas de pesquisa

6.1.1 Hipóteses e Perguntas de Pesquisa

- **Hipótese 1:** É possível, através da análise histórica da evolução de métricas de qualidade das classes do sistema identificar uma associação entre mudanças das métricas com respectivas refatorações. Além disso, caso esta associação exista, este será considerado um momento adequado para refatoração uma vez que existiria um padrão de comportamento entre os desenvolvedores, e este padrão será considerado adequado.

- **Pergunta de Pesquisa 1:** Existe um momento típico, em termos da evolução de métricas de qualidade, que os desenvolvedores elegem para realizar o refatoração de uma classe?

  Com base nas análises de evolução da métrica LCOM na qual foram identificadas refatorações nos pontos de variação significativa no valor da métrica. E com base no resultado do modelo preditivo que identificou cinco padrões envolvendo as demais métricas de qualidade amplamente discutido na seção de resultados. Conclui-se que existe uma associação entre os valores assumidos pelas métricas e a aplicação de refatoração de classe. De acordo com os padrões identificados, o momento típico escolhido para se refatorar está relacionado principalmente com a quantidade e complexidade dos métodos e com o nível de influência que a classe exerce sobre o sistema. Para os casos nos quais as classes não possuem grande quantidade de métodos, o nível no qual ela se encontra na hierarquia da arquitetura, bem como a quantidade de sub-classes que ela possui determinam se é possível aplicar a refato-

ração. Dada a precisão do modelo ao localizar essas classes, e o reconhecimento por parte dos desenvolvedores pode-se afirmar que ao analisar a evolução das métricas é possível determinar se a classe atende os requisitos para ser refatorada.

- **Hipótese 2:** É possível, identificar as melhores estratégias para refatoração de classes, por meio da análise do impacto de diferentes alternativas de refatoração sobre as métricas de qualidade.

- **Pergunta de Pesquisa 2:** É possível detectar uma estratégia mais adequada para refatorar uma classe?

  Com base no resultado da análise qualitativa das refatorações identificadas, e juntamente com os resultados do *survey* foram detectadas 3 estratégias de refatoração. As duas primeiras: Refatoração Gradativa e Refatoração Imediata foram identificadas comparando o código fonte das classes que sofreram alterações significativas na métrica LCOM. Essas estratégias estão relacionadas com a movimentação de métodos do tipo get() e set(), para classes que exercem a função de configuração. Tais estratégias ajudam na modularização movendo a tarefa de configuração para uma classe separada, deixando na classe de origem somente os métodos relacionados à sua funcionalidade. A forma como os métodos são removidos das classes de origem apresentam vantagens e desvantagens, demostrando que a aplicação dessas estratégias depende da situação do sistema. A terceira estratégia foi identificada através dos resultados *survey* realizado com os desenvolvedores e trata-se de uma forma de refatoração preventiva que tem como objetivo evitar uma maior degradação da classe. Portanto, conclui-se que é possível identificar estratégias de refatorações, entretanto para determinar se a mesma é adequada para a classe é necessário um profundo conhecimento sobre a arquitetura do sistema.

As análises e o método da abordagem ajudaram a compreender melhor as dificuldades envolvidas no processo de refatoração. Nesse contexto algumas lições foram aprendidas e são descritas a seguir:

6.2 Lições Aprendidas

- Melhor direcionamento das análises qualitativas: Durante a pesquisa as análises qualitativas foram realizadas como uma forma de explorar os dados para determinar a qualidade dos resultados das análises quantitativas. Um melhor direcionamento da analise qualitativa ajudaria a identificar características da estrutura de organização interna da classe. Dessa forma, com um conhecimento mais aprofundado sobre as classes, as estratégias de refatoração poderiam ser discutidas com mais detalhes.

- Definição de um limiar para determinar padrões na escolha dos tipos de métodos: Algumas das recomendação foram rejeitadas devido à má escolha da lista dos métodos a serem movidos. Com base nessas respostas fica evidente que é necessário definir um limiar para a quantidade de métodos get() e set() para justificar a refatoração.

- Relação entre quantidade de alteração e complexidade da classe: Um ponto a ser melhorado na abordagem é identificar a quantidade de alterações sofrida pela classe ao longo do tempo e avaliar a complexidade e o esforço necessário para aplicar a refatoração. Em alguns casos as estratégias de refatoração não foram aplicadas devido à alta complexidade da classe e seu histórico de alterações.

6.3 Ameaças à Validade

Neste trabalho foram identificadas as seguintes ameaças à validade:

- Valor limiar de variação do LCOM: Para identificar se houve uma refatoração em uma determinada classe, foi necessário estabelecer um limiar a partir do qual a variação de LCOM poderia ser considerada significativa. Para minimizar o risco de o valor do limiar não ser influenciado por *outliers*(Valores Atípicos), foi realizado o cálculo da DAM(Desvio Absoluto da Mediana)(HUBER, 1996). Esse cálculo apresenta um resultado mais confiável do que o desvio padrão tradicional sendo menos sensível à presença de *outliers* (LEYS et al., 2013).

- Vícios nas amostras coletadas. Como algumas das análises foram feitas por amostragem, existe o risco de ocorrer algum vício amostral no qual a amostra não representa a população. Para minimizar esse risco foram selecionadas amostras suficientemente grandes e balanceadas. Dessa forma a representatividade dos sistemas foi garantida em todas as etapas da análise

- Valores falsos positivos do modelo preditivo. Como todo modelo preditivo corre o risco de errar a predição, é necessária a validação dos resultados. Para validar o modelo foi realizada validação interna com amostras de treino obtendo resultados satisfatórios, e uma validação externa através da avaliação dos desenvolvedores. Com o resultado do *survey* sobre a localização das oportunidades ficou claro que o modelo consegue identificar o local das oportunidades de refatoração com precisão satisfatória.

- Variações nas métricas que não representam refatorações. Durante a análise de evolução das métricas existe o risco de que a variação ocorrida não represente uma refatoração. Para minimizar esse risco uma análise de distribuição dessas variações foi realizada. Para as classes nas quais a variação ocorreu mais de uma vez, foi

considera a maior variação. Com esse resultado foi realizada uma análise qualitativa para determinar se a variação representa de fato uma refatoração.

Capítulo 7

Trabalhos Relacionados

A seguir são apresentados os trabalhos relacionados e as diferenças entre eles e a abordagem proposta neste trabalho:

RIPE(*Refactoring Impact PrEdiction*) é uma técnica que estima o impacto das operações de refatoração sobre as métricas de qualidade de software, cujo objetivo é ajudar o desenvolvedor a escolher a estratégia de refatoração mais adequada (CHAPARRO et al., 2014). Como resultado a técnica RIPE obteve uma acurácia de 38% na estimativa do impacto das estratégias de refatorações avaliadas. Na abordagem deste trabalho, nós não apenas tentamos estimar os impactos das estratégias de refatoração, como também determinar outras condições que a classe precisa satisfazer para que seja aplicada uma refatoração.

O problema de identificação de mudanças na modularidade durante a evolução do software também foi investigado por (ANTONIOL; PENTA; MERLO, 2004). Eles propuseram uma abordagem automática para identificar casos de possíveis refatorações baseado em espaço vetorial e na distância de cosseno. ARCADE(*Architecture Recovery, Change, And Decay Evaluator*) é uma técnica para identificar e avaliar as mudanças arquiteturais em sistemas de software (LE et al., 2015). O principal objetivo do ARCADE é auxiliar o desenvolvedor a identificar os pontos nos quais ocorreram alterações na arquitetura do sistema. A avaliação das mudanças arquiteturais foi feita utilizando métricas que comparam as diferenças nas arquiteturais. Tais como: a2a (*Architecture-to-architecture*) que mede a distância entre duas arquiteturas. Como resultado o estudo demonstrou que o sistema de numeração de versões não é um bom indicador de mudanças arquiteturais. Também foi constatado que as mudanças arquiteturais tendem a ocorrer mais nas primeiras versões dos sistemas. Em contraste com a abordagem proposta neste trabalho, estas ferramentas não realizam uma análise qualitativa das refatorações investigando o impacto das estratégias adotas em cada caso.

A relação entre as estratégias de refatoração e as mudanças nas métricas de qualidade de software tem sido amplamente discutida (SIMON; STEINBRUCKNER; LEWERENTZ, 2001), (STEIDL; EDER, 2014), (SZŐKE et al., 2015), (SIMONS; SINGER;

WHITE, 2015). Ao comparar os valores das métricas antes e após as refatoração é possível medir os efeitos na qualidade do software (KATAOKA et al., 2002).

Métricas de software também tem sido utilizadas para recomendar refatorações com o objetivo de ajudar os desenvolvedores a melhorar a modularidade (DEMEYER; DUCASSE; NIERSTRASZ, 2000), (OUNI et al., 2016), (VIDAL; MARCOS; DÍAZ-PACE, 2016), (BAVOTA et al., 2015), (OUNIA et al., 2015). (RATZINGER et al., 2007) apresentam um estudo empírico para determinar quais características podem ser utilizadas para prever a a necessidade de refatoração nos próximos dois meses de desenvolvimento. Como resultado, o estudo identificou que métricas de software tais como LOC(Lines of Code) e métricas que indicam modificações d código tais como a Quantidade de Linhas adicionadas ou removidas podem ser utilizadas para construir modelos preditivos de refatoração. Estes estudos focam nas métricas quantitativas, enquanto que nosso estudo apresenta uma análise qualitativa das mudanças realizadas para determinar se a refatoração foi realizada, e qual estratégia foi adotada.

(BOIS; DEMEYER; VERELST, 2004) apresentam um estudo empírico cujo objetivo é identificar em quais condições, a refatoração traz melhorias de coesão. Para isso foram propostas guias para aplicação de diferentes estratégias de refatorações. Para validar a aplicação das guias foi escolhido o sistema Apache Tomcat. Foram identificadas no total 20 oportunidades de refatoração distribuídas em 12 classes. A identificação das oportunidades foi feita escolhendo classes com problemas de coesão, e dentre essas classes, foram escolhidas aquelas que atendem os requisitos para aplicação da guia.

(BAVOTA; LUCIA; OLIVETO, 2011) propuseram um método para identificar oportunidades de *Extract Class* baseada em teoria dos grafos e na combinação de medidas semânticas e estruturais. Na abordagem as arestas do grafo representam o valor da métrica SSM(*Structural Similarity between Methods*). Outros estudos também propuseram identificar e aplicar *Extract Class* e *Move Methods* usando clusterização e teoria dos jogos (BAVOTA et al., 2010), (FOKAEFS et al., 2012), (BAVOTA et al., 2014), (MURPHY-HILL; BLACK, 2008), (SILVA; TERRA; VALENTE, 2014). Em contraste com a nossa abordagem eles não consideram uma análise temporal da coesão para determinar quando a classe apresenta a oportunidade para refatoração.

Um dos grandes desafios ao recomendar estratégias de refatorações é entender a percepção dos desenvolvedores, e o que os leva a refatorar. (SILVA; TSANTALIS; VALENTE, 2016) estudaram as motivações para desenvolvedores aplicar refatoração. Um *survey* foi conduzdido com os desenvolvedores e descobriu que em 75% casos a motivação para refatorar uma classe está mais relacionada com a inserção de novas funcionalidades e correção de bugs do que problemas estruturais no código. Outro estudo (PALOMBA et al., 2017) investigou a relação entre refatoração e mudanças, e observou que que desenvolvedores tendem a refatorar para melhorar a coesão quando eles implementam novas características enquanto que quando eles corrigem bugs, a refatoração tem o objetivo de aumentar a

manutenibilidade e a compreensão do código.

(KIM; ZIMMERMANN; NAGAPPAN, 2012) apresentaram um estudo sobre os desafios e benefícios da refatoração. O objetivo desse estudo é entender melhor a definição de refatoração sobre a perspectiva dos desenvolvedores. O estudo constatou que na prática a refatoração nem sempre preserva o comportamento do sistema. Também foi observado que os desenvolvedores entendem que as refatorações envolvem custos e riscos, e que nem sempre é possível preservar o comportamento do sistema ao modificar o código fonte. A análise quantitativa demonstrou que o processo de refatoração traz benefícios, tais como: a redução de dependências entre classes, e redução de defeitos. Nós observamos percepções similares em nosso estudo.

(TEMPERO; GORSCHEK; ANGELIS, 2017) apresentou as barreiras e rasões que levam desenvolvedores a não refatorar. Os resultados apresentam algumas similaridade com a nossa abordagem e os resultados do *survey*. A falta de conhecimento sobre a arquitetura e o risco de introduzir novos defeitos no sistema foram destacados pelos desenvolvedores, isto é, recomendações de refatorações devem prover conhecimento sobre as classes involvidas, para que os desenvolvedores tenham um melhor entendimento dos impactos gerados pela refatoração.

Capítulo 8

Conclusão

Neste trabalho foi apresentada uma abordagem para identificação de oportunidade de refatoração, bem como estratégias de como realizar essas operações. Foram identificadas três estratégias, sendo duas identificadas através da análise da evolução das métricas de qualidade e a terceira através de um *survey* com os desenvolvedores. Com base nos resultados obtidos com as análises tanto quantitativas quanto qualitativas, e com base nas discussões sobre as considerações dos desenvolvedores, pode-se constatar que a tarefa de refatorar uma classe é extramente complexa.

8.1 Trabalhos Futuros

Durante a discussão com os desenvolvedores alguma ideias para trabalhos futuros foram apresentadas:

- Automatização de operações de refatoração: Com base no resultado do *survey* fica evidente que uma maior automatização da abordagem ajudaria a diminuir os custos de refatoração. Nesse contexto, seria necessário mapear quais operações de refatoração são sempre realizadas independente da estratégia adotada.
- Monitoramento de correção de bugs envolvendo classes com propensão à refatoração: Um dos pontos que desmotivou os desenvolvedores a aplicar refatoração é a grande quantidade de bugs abertos esperando para serem resolvidos. Ao monitorar as discussões referentes a esses bugs seria possível identificar se alguma classes com propensão à refatoração está envolvida. Caso esteja, a recomendação de refatoração pode ser feita durante o processo de manutenção no qual o bug é resolvido.
- Ambiente para simulação de refatorações: Um dos pontos importantes que deve ser considerado ao aplicar uma refatoração é o impacto gerado sobre o sistema. Para garantir uma maior segurança ao refatorar a classe, um fork do projeto original poderia ser cria para testar as refatorações antes de recomendá-las. Dessa forma somente as refatorações que geram menor impacto seriam relizadas.

- Assistente de refatoração e aprendizado por recompensa: Uma tendência tecnológica que está sendo difundida é o uso de assistentes com inteligência artificial. Esses assistente auxiliam os usuários em tarefas específicas gerando recomendações e fornecendo informações. A cada recomendação o usuário avalia a qualidade e utilidade da mesma. Com base nessa avaliação o assistente atualiza sua base de conhecimento para gerar novas recomendações. A ideia seria usar essa estratégia para recomendar refatorações.

- Identificações de padrões nas guias de refatoração fornecidas pelo próprios usuários: Foi observado que em sistema diferentes as formas como as informações são apresentas e discutidas pelos desenvolvedores é diferente. Nesse contexto poderia ser avaliado se existe um padrão na forma como os desenvolvedores recomendam refatoração. Com isso as guias ficariam mais personalizadas e atenderiam melhor as necessidades de cada sistema.

- Criação de métricas de software direcionadas para operações de refatoração: Um fator que gerou dificuldades para identificar se a classe era realmente uma candidata à refatoração, foi a falta de métricas específicas para auxiliar esse processo. A quantidade de classes internas, a quantidades de métodos de atribuição e leitura de valores: get(), set(), add(), read(), quantidade de métodos sobrecarregados são características que poderiam ser mapeadas através de novas métricas de qualidade. Com essas métricas ficaria mais fácil priorizas as classes com maior propensão à refatoração.

Referências

ANTONIOL, G.; PENTA, M. D.; MERLO, E. An automatic approach to identify class evolution discontinuities. In: IEEE. **Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of**. [S.l.], 2004. p. 31–40.

BANSIYA, J. et al. A class cohesion metric for object-oriented designs. **Journal of Object-Oriented Programming**, SIGS PUBLICATIONS INC 71 WEST 23RD ST, 3RD FLOOR, NEW YORK, NY 10010 USA, v. 11, n. 8, p. 47–52, 1999.

BAVOTA, G.; LUCIA, A. D.; OLIVETO, R. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. **Journal of Systems and Software**, Elsevier, v. 84, n. 3, p. 397–414, 2011.

BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, Elsevier, v. 107, p. 1–14, 2015.

_____. Methodbook: Recommending move method refactorings via relational topic models. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 7, p. 671–694, 2014.

_____. Playing with refactoring: Identifying extract class opportunities through game theory. In: IEEE. **Software Maintenance (ICSM), 2010 IEEE International Conference on**. [S.l.], 2010. p. 1–5.

BOIS, B. D.; DEMEYER, S.; VERELST, J. Refactoring-improving coupling and cohesion of existing code. In: IEEE. **Reverse Engineering, 2004. Proceedings. 11th Working Conference on**. [S.l.], 2004. p. 144–151.

CHAPARRO, O. et al. On the impact of refactoring operations on code quality metrics. In: IEEE. **Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on**. [S.l.], 2014. p. 456–460.

CHIDAMBER, S. R.; DARCY, D. P.; KEMERER, C. F. Managerial use of metrics for object-oriented software: An exploratory analysis. **IEEE Transactions on software Engineering**, IEEE, v. 24, n. 8, p. 629–639, 1998.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. Finding refactorings via change metrics. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2000. v. 35, n. 10, p. 166–177.

FOKAEFS, M. et al. Identification and application of extract class refactorings in object-oriented systems. **Journal of Systems and Software**, Elsevier, v. 85, n. 10, p. 2241–2260, 2012.

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

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIC, N. A comparative analysis of software architecture recovery techniques. In: IEEE PRESS. **Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.], 2013. p. 486–496.

GUPTA, B. S. **A critique of cohesion measures in the object-oriented paradigm**. Dissertação (Mestrado) — Michigan Technological University, 1997.

HUBER, P. J. **Robust statistical procedures**. [S.l.]: SIAM, 1996.

KANNANGARA, S.; WIJAYANAYAKE, W. An empirical evaluation of impact of refactoring on internal and external measures of code quality. **arXiv preprint arXiv:1502.03526**, 2015.

KATAOKA, Y. et al. A quantitative evaluation of maintainability enhancement by refactoring. In: IEEE. **Software Maintenance, 2002. Proceedings. International Conference on**. [S.l.], 2002. p. 576–585.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. In: ACM. **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**. [S.l.], 2012. p. 50.

LE, D. M. et al. An empirical study of architectural change in open-source software systems. In: IEEE PRESS. **Proceedings of the 12th Working Conference on Mining Software Repositories**. [S.l.], 2015. p. 235–245.

LEYS, C. et al. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. **Journal of Experimental Social Psychology**, Elsevier, v. 49, n. 4, p. 764–766, 2013.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MURPHY-HILL, E.; BLACK, A. P. Breaking the barriers to successful refactoring: observations and tools for extract method. In: ACM. **Proceedings of the 30th international conference on Software engineering**. [S.l.], 2008. p. 421–430.

OUNI, A. et al. Multi-criteria code refactoring using search-based software engineering: An industrial case study. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 25, n. 3, p. 23, 2016.

OUNIA, A. et al. A multi-objective refactoring approach to introduce design patterns and fix anti-patterns. In: **First North American Search Based Software Engineering Symposium, NASBASE**. [S.l.: s.n.], 2015.

PALOMBA, F. et al. An exploratory study on the relationship between changes and refactoring. In: **Proceedings of the 25th International Conference on Program Comprehension**. Piscataway, NJ, USA: IEEE Press, 2017. (ICPC '17), p. 176–185.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: Palgrave Macmillan, 2005.

RATZINGER, J. et al. Mining software evolution to predict refactoring. In: IEEE. **Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on**. [S.l.], 2007. p. 354–363.

ROKACH, L.; MAIMON, O. **Data mining with decision trees: theory and applications**. [S.l.]: World scientific, 2014.

SILVA, D.; TERRA, R.; VALENTE, M. T. Recommending automated extract method refactorings. In: ACM. **Proceedings of the 22nd International Conference on Program Comprehension**. [S.l.], 2014. p. 146–156.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: ACM. **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.], 2016. p. 858–870.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics based refactoring. In: IEEE. **Software Maintenance and Reengineering, 2001. Fifth European Conference on**. [S.l.], 2001. p. 30–38.

SIMONS, C.; SINGER, J.; WHITE, D. R. Search-based refactoring: Metrics are not enough. In: SPRINGER. **International Symposium on Search Based Software Engineering**. [S.l.], 2015. p. 47–61.

STEIDL, D.; EDER, S. Prioritizing maintainability defects based on refactoring recommendations. In: ACM. **Proceedings of the 22nd International Conference on Program Comprehension**. [S.l.], 2014. p. 168–176.

SZŐKE, G. et al. Faultbuster: An automatic code smell refactoring toolset. In: IEEE. **Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on**. [S.l.], 2015. p. 253–258.

TEMPERO, E.; GORSCHEK, T.; ANGELIS, L. Barriers to refactoring. **Communications of the ACM**, ACM, v. 60, n. 10, p. 54–61, 2017.

VIDAL, S. A.; MARCOS, C.; DÍAZ-PACE, J. A. An approach to prioritize code smells for refactoring. **Automated Software Engineering**, Springer, v. 23, n. 3, p. 501–532, 2016.