Pós-Graduação em Ciência da Computação

Everton Rennê Barros de Oliveira

RELAÇÃO ENTRE REFATORAÇÕES E CODE SMELLS NA EVOLUÇÃO DE PROJETOS DE SOFTWARE E SEU REFLEXO EM MEDIDAS DE SOFTWARE

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
WWW.CIN.UFPE.BR/~POSGRADUACAO

Recife
2020

Everton Rennê Barros de Oliveira

RELAÇÃO ENTRE REFATORAÇÕES E CODE SMELLS NA EVOLUÇÃO DE PROJETOS DE SOFTWARE E SEU REFLEXO EM MEDIDAS DE SOFTWARE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação, área de concentração em Engenharia de Software, do Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.

Área de Concentração: *Engenharia de Software*

Orientador(a): *Márcio Lopes Cornélio.*

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O48r Oliveira, Everton Rennê Barros de
Relação entre refatorações e code smells na evolução de projetos de software e seu reflexo em medidas de software / Everton Rennê Barros de Oliveira. – 2020.
102 f.: il., fig., tab.

Orientador: Márcio Lopes Cornélio.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências e apêndices.

1. Engenharia de software. 2. Medidas de software. I. Cornélio, Márcio Lopes (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 80

Everton Renne Barros de Oliveira

“Relação entre refatorações e code smells na evolução de projetos de software e seu reflexo em medidas de software”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 14/02/2020.

BANCA EXAMINADORA

_____

Prof. Dr. Henrique Emanuel Mostaert Rebêlo
Centro de Informática / UFPE

_____

Prof. Dr. Everton Leandro Galdino Alves
Centro de Engenharia Elétrica e Informática/UFCG

_____

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática / UFPE
(**Orientador**)

Dedico este trabalho a minha esposa Ana e aos nossos pais.

AGRADECIMENTOS

Agradeço a minha esposa Ana, aos nossos pais e a todos da minha família, por todo o apoio, pois a ajuda de cada um foi essencial para trilhar esse caminho.

Ao meu orientador, professor Márcio Lopes Cornélio pela paciência, oportunidade, ajuda e apoio que permitiram o desenvolvimento e conclusão desse mestrado.

Aos meus amigos que colaboraram com seu tempo para ouvir e opinar sobre o meu trabalho, sempre apresentando sugestões construtivas para enriquecê-lo.

Agradeço a todos os professores do mestrado que contribuíram com conhecimento e compartilharam experiências durante as aulas do mestrado.

Obrigado a todos.

RESUMO

A refatoração ou refactoring, é a atividade de melhorar o código fonte sem afetar o comportamento observável do software, com vistas a reuso, legibilidade e facilidade de manutenção. A refatoração é útil para transformação de trechos de código com baixa qualidade e que se enxerga uma oportunidade de melhoria. Estes trechos também são conhecidos como *code smells* (código com “cheiro”) ou *bad smells* (código que “cheira mal”). Observamos como projetos de software evoluíram a partir da relação entre as refatorações e *code smells*, sob a perspectiva das medidas de software ao longo do ciclo de evolução do software. Desenvolvemos a ferramenta RefactoringLink a fim de identificar *code smells* que foram introduzidos ao longo do desenvolvimento, bem como aqueles que desapareceram. Outra funcionalidade da ferramenta é a de capturar as refatorações que ocorreram entre as versões de um software, com a finalidade de identificar quais foram os recursos afetados por essas transformações e assim criar uma associação entre a refatoração e o *code smell*. Investigamos 11.593 versões em 427 projetos Java/Android, hospedados em repositórios de software público. Em nossos achados, foi possível detectar 4.555.351 *code smells* e um total de 197.525 refatorações. Identificamos que apenas 2.3% (4.508) das refatorações resultaram em correção direta de 0.13% (5.354) dos *code smells*. Coletamos 37 medidas de software para cada classe de cada versão dos projetos analisados. Analisamos cinco medidas de software específicas de orientação a objetos em três cenários para observar o reflexo nas medidas quando na existência ou não de *refatoração ou code smell*. Os achados indicam que as medidas de software podem sofrer variações em razão da inclusão de novas funcionalidades ou manutenção do código que é realizada ao longo da evolução do software, indo além de mudanças advindas de refatorações. Apresentamos um comparativo das medidas em 15 classes de projetos distintos que tiveram a presença de refatoração, *code smell* e da correção de *code smell* a partir de uma refatoração.

Palavras-chave: Refactoring. Code smells. Medidas de software. Engenharia de software. Repositórios de software.

ABSTRACT

Refactoring is the activity of improving the source code without affecting the observable behavior of the software, with a view to reuse, readability, and maintenance. Refactoring is useful to change low-quality code snippets that there is an opportunity for improvement. This code is also known as code smells or bad smells (code that looks strange and almost smells bad). We observed how software projects evolved from the relationship between refactorings and code smells, from the perspective of software metric throughout the software evolution history. We developed the RefactoringLink tool in order to identify code smells that were introduced during development, as well as those that disappeared. Another feature of the tool is to capture the refactorings that occurred between versions in order to identify which resources were affected by these transformations and thus create a link between refactoring and code smell. We investigated 11,593 versions in 427 Java/Android projects, hosted on public software repositories. In our findings, it was possible to detect 4,555,351 code smells and a total of 197,525 refactorings. We found that only 2.3% (4,508) of refactorings resulted in a direct correction of code smells, this means that we only detected that 0.13% (5,354) of code smells were fixed by a refactoring action, intentional or not. We collected 37 software metrics for each class of each version of the analyzed projects. We analyzed five specific object-oriented software metrics in three scenarios to observe the reflection in the measures when refactoring or code smell exists or not. The findings indicate that the software metrics may suffer variations due to the addition of new features or maintenance of the code. It's natural for software to be evolved beyond changes resulting from refactorings. We present a comparison of the measures in 15 classes of different projects that had the presence of refactoring, code smell and the correction of code smell from refactoring.

Keywords: *Refactoring*. *Code smells*. Software metric. Software engineering. Software repositories.

LISTA DE FIGURAS

Figura 1 - Refatorações e suas classificações 21
Figura 2 - Uso de refactoring 23
Figura 3 - Duplicação de código 30
Figura 4 - Extração de método 31
Figura 5 - Exemplo de classe com e sem Feature Envy 32
Figura 6 - Número de ferramentas de detecção de code smells lançadas por ano 34
Figura 7 - Mapeamento das medidas por Chidamber e Kemerer em 1991 e 1994 38
Figura 8 - Contagem DIT 40
Figura 9 - Equação da medida LCOM-HS 42
Figura 10 - Método proposto para construção do framework 50
Figura 11 - Exemplo da Etapa 3 com versões do PhotoEditor 52
Figura 12 - Etapa 4 - Code Smells 54
Figura 13 - Etapa 6 - Processamento 55
Figura 14 - GitHub API REST v3 - listagem de releases 57
Figura 15 - Fluxo de processamento RefLink 65
Figura 16 - Modelo de dados RefLink 69
Figura 17 - Medidas do Extract Method - Cenário 1 74
Figura 18 - Medidas do Long Method 75
Figura 19 - Long Method em CameraRenderer.run() 76
Figura 20 - Medidas após correção do Long Method com Extract Method 77
Figura 21 - Solução para correlacionar refatoração a code smells 82

LISTA DE TABELAS

Tabela 1 - Exemplo de refactorings por nível de abstração 22
Tabela 2 - Atividades necessárias no processo de refatoração segundo Mens e Tourwé 23
Tabela 3 - Refatorações segundo critério de Fowler 26
Tabela 4 - Versões do PhotoEditor 52
Tabela 5 - Jquality Code Smell List 58
Tabela 6 - aDoctor Code Smell List 59
Tabela 7 - Refactoringcrawler refactoring 60
Tabela 8 - Medidas coletadas pela CK Tool 61
Tabela 9 - Quantitativo de projetos 70
Tabela 10 - Quantitativo de versões e classes 70
Tabela 11 - Quantitativo de refatorações e code smells 71
Tabela 12 - Refatorações e Code Smells mais comuns e corrigidos 71
Tabela 13 - Medidas de Software para Extract Method e Long Method 73
Tabela 14 - Ranking de code smells 78

SUMÁRIO

1 INTRODUÇÃO 12
1.1 JUSTIFICATIVA 14
1.2 QUESTÃO DE PESQUISA 14
1.3 OBJETIVOS 15
1.4 ESTRUTURA DA DISSERTAÇÃO 15
2 REFERENCIAL TEÓRICO 17
2.1 REFACTORING 17
2.1.1 Importância da Refatoração 18
2.1.2 Classificação e Métodos de Refatoração 21
2.1.3 Como aplicar a Refatoração 23
2.1.4 Catálogo de refatorações 25
2.2 CODE SMELLS 27
2.2.1 Considerações iniciais 27
2.2.2 Code Smells e suas diretrizes 27
2.2.3 Principais code smells 30
2.2.3.1 Duplicated Code 30
2.2.3.2 Long Method 30
2.2.3.3 Large Class 31
2.2.3.4 Long Parameter List 31
2.2.3.5 Feature Envy 32
2.2.3.6 Data Class 33
2.2.3.7 Outros smells 33
2.2.4 Ferramentas para detecção de code smell 33
2.3 MEDIDAS DE SOFTWARE 35
2.3.1 Métricas de Software Orientado a Objetos 37
2.3.2 Métricas C&K 38
2.3.3 Utilização de métricas para avaliação de software 45
2.4 CONSIDERAÇÕES FINAIS 46
3 MATERIAL E MÉTODO 47
3.1 SELEÇÃO DE REPOSITÓRIO DE SOFTWARE 47
3.2 SELEÇÃO DE FERRAMENTA DE DETECÇÃO DE CODE SMELLS 48
3.3 SELEÇÃO DA FERRAMENTA DE DETECÇÃO DE REFATORAÇÃO 49
3.4 MÉTODO 50
3.4.1 Etapa 1 - Construção 50

3.4.2 Etapa 2 - Inicialização 51
3.4.3 Etapa 3 - Refactoring 51
3.4.4 Etapa 4 – Code Smells 53
3.4.5 Etapa 5 – Medidas de Software 54
3.4.6 Etapa 6 – Processamento 54
3.4.7 Etapa 7 – Sumarização 56
3.5 FERRAMENTAS SELECIONADAS 56
3.5.1 JQuality/SmartSmell 57
3.5.2 aDoctor 58
3.5.3 Ferramenta de Identificação de Refatoração 59
3.5.4 CK 61
3.6 CONSIDERAÇÕES FINAIS 62
4 RESULTADOS E DISCUSSÃO 63
4.1 REFACTORING LINK (REFLINK TOOL) 64
4.1.1 Integração de ferramentas 64
4.2 DATASET REFLINK 68
4.3 DADOS COLETADOS 69
4.4 AMEAÇAS À VALIDADE 79
5 CONSIDERAÇÕES FINAIS 81
5.1 TRABALHOS RELACIONADOS 83
5.1.1 Refatoração e repositórios de software 83
5.1.2 Code Smells e Medidas de Software 85
5.2 LIMITAÇÕES 86
5.3 TRABALHOS FUTUROS 87
REFERÊNCIAS 89
APÊNDICE A - CATÁLOGO DE CODE SMELLS DE FOWLER 94
APÊNDICE B - LISTAGEM DE PROJETOS SELECIONADOS DO GITHUB 95
APÊNDICE C - REFATORAÇÕES SUPORTADAS PELO REFACTORINGMINER 102

1 INTRODUÇÃO

Este Capítulo descreve as principais motivações para realização desta dissertação, introduzindo conceitos acerca de refatoração de software e *code smells*, apresentando nas Seções seguintes sua justificativa, objetivos de pesquisa almejados e, finalmente, apresenta como está estruturada a dissertação.

As mudanças ocorridas nos últimos anos, sobretudo os avanços tecnológicos, têm importância nas empresas públicas e privadas, bem como nos contextos social, político e econômico. A sociedade passou a ser também caracterizada pela uso crescente de aparatos tecnológicos e pelo avanço das Tecnologias de Informação e Comunicação (TICs).

O desenvolvimento de código eficiente e o uso de metodologias de desenvolvimento tornou-se um valioso ativo das organizações, visto que os bons sistemas de informação desenvolvidos podem, por exemplo, agregar valor a serviços, fazendo com que os times de desenvolvimento desempenhem cada vez mais um papel estratégico (GHAPANCHI; WOHLIN; AURUM, 2014). Com o avanço da tecnologia, cada vez mais o conhecimento acerca de processos de desenvolvimento de software – como práticas ágeis, gerenciamento de time, gerenciamento de escopo e risco – se torna essencial.

Nesta nova situação, percebe-se a ampliação dos papéis do time de desenvolvimento, que é responsável pelo gerenciamento de sistemas que entreguem valor. Consequentemente, a atividade de construção dos sistemas passou a ser crítica para algumas organizações, devendo ter seus riscos mitigados e contemplados dentro de um planejamento.

Os sistemas são frequentemente objetos de atividades de evolução e de manutenção para adicionar novas funções, corrigir problemas e ajustar a novas necessidades do ambiente. Diante deste cenário, tais atividades são normalmente executadas de maneira adhoc devido as pressões de tempo e com pouco ou nenhum

planejamento. Isto torna o contexto de desenvolvimento de software complexo e de alto risco.

Ciente de que o desenvolvimento de software tem foco na organização e traz como princípios básicos a garantia da entrega de um produto de qualidade com baixo custo de manutenção e atendendo a todos os requisitos da aplicação, ações que afetem tais princípios, sejam intencionais ou não, podem comprometer o desempenho da organização, chegando, em casos mais críticos, a causar grandes impactos financeiros (SOMMERVILLE, 2011).

Mesmo diante da criticidade na área de desenvolvimento de software, as organizações não contam com metodologia adequadas e, nem sempre, têm equipes de desenvolvimento maduras e que seguem processos bem definidos (FOWLER; BECK, 1999).

O processo de desenvolvimento de software no ambiente atual carece de melhoramentos e pesquisas contínuas devido ao aumento crescente no número de sistemas que exigem constantes atividades de manutenção e correção de bugs. Segundo (BERTRAND, 1994), a manutenção é a fase que mais demanda esforço na atividade de desenvolvimento de um sistema, salientando ser a manutenção penalizada pela dificuldade de realizar mudanças no sistema. Ainda segundo (BERTRAND, 1994), a manutenção representa uma custo total de 70% de um sistema.

Mesmo com a utilização de tantas metodologias de desenvolvimento de sistemas, existe a carência de artefatos que sinalizem se um sistema possui ou não a necessidade de manutenção, em função da presença, por exemplo, um caso de teste que falha, um bug ou código com problemas estruturais, de lentidão, dificuldade de entendimento e modificação (MOHAN; GREER; MCMULLAN, 2016). Sendo necessário, também olhar para medidas de software como ferramenta de análise para indicar se a manutenção do código está sendo eficaz. Em outras palavras, se a manutenção do código está resultando em uma melhoria (LIU, 1999).

O presente Capítulo está dividido em seis seções. Na primeira Seção será apresentada a justificativa do trabalho. Na segunda Seção, o problema de pesquisa será abordado. Na Seção 4 será explicitada uma possível proposta de solução. Na quinta Seção os objetivos geral e específicos serão explanados. E, por fim, na última Seção, a organização do trabalho será demonstrada.

1.1 JUSTIFICATIVA

No desenvolvimento de software, muitos programadores se deparam com problemas profundos que afetam diretamente a qualidade e o projeto dos sistemas. Tais problemas representam indicações de variações estruturais, também denominados *code smells* (código com cheiro em tradução literal). Estes problemas pode ser resolvidos através de transformações nos código fonte, sendo comum um tipo de transformação conhecida como refatoração.

A refatoração de código é uma maneira de reestruturar o código existente sem alterar seu comportamento. É uma maneira de melhorar a qualidade do código. De acordo com a definição de Fowler (FOWLER; BECK, 1999);

> "É uma técnica disciplinada para reestruturar um corpo de código existente, alterando sua estrutura interna sem alterar seu comportamento externo".

Ainda sobre a concepção de Fowler, a refatoração de código não deve ser feita a qualquer momento, principalmente se outros fatores como prazos de entrega, escopo e custo do projeto estiverem envolvidos. Estes fatores são mais para se buscar eliminar erros em vez de melhorar a qualidade do código.

1.2 QUESTÃO DE PESQUISA

A dificuldade em avaliar os problemas estruturais (*code smells*) de um sistema, bem como verificar a efetividade das transformações aplicadas ao código (refatoração), impõe um desafio significativo em razão da ausência de ferramentas que possam auxiliar o processo de identificação e verificação de maneira automatizada.

Desta forma, apresenta-se um desafio que é entender se as refatorações eliminam *code smells* e perceber como isso reflete em medidas de software, buscando entender se as refatorações estão melhorando o código a partir de aspectos de qualidade, utilizando medidas de software como instrumento (FONTANA; SPINELLI, 2011).

A nossa questão de pesquisa concentra-se na pergunta: é possível estabelecer uma relação entre *code smells* e refatorações, observando o reflexo em medidas de software? A partir disso, estabelecemos a seguinte hipótese: refatorações eliminam *code smells* alterando medidas de software correlacionadas.

Para verificar a hipótese, desenvolvemos um sistema que inspeciona os projetos de software ao longo do desenvolvimento, buscando identificar problemas estruturais,

medidas e correlacionar se as refatorações no código impactaram diretamente os problemas estruturais e medidas de software.

1.3 OBJETIVOS

O objetivo geral deste trabalho é propor um framework para identificação de refatoração realizada que resulta em eliminação de *code smells* e aferição de medidas de software. Para alcançar tal objetivo, temos os seguintes objetivos específicos:

- Compreender as soluções existentes naquilo que concerne a ferramentas de identificação de refatorações, *code smells* e métricas de software;
- Definir um fluxo de trabalho para correlacionar refatorações e *code smells*;
- Propor uma solução automatizada para inspecionar repositórios de software, identificar refatorações e *code smells,* buscando correlaciona-los durante toda a vida do software;
- Coletar medidas de software para auxiliar o entendimento da correlação entre refatorações e *code smells;*
- Construir um *dataset* com refatorações, *code smells* e medidas de software a partir de repositórios de software.

1.4 ESTRUTURA DA DISSERTAÇÃO

No Capítulo 2 apresentamos os principais conceitos dos assuntos relacionados ao tema desta dissertação. Abordamos sobre refatorações e suas características, passando sob a perspectiva de como aplicar e o catálogo de refatorações. Além disso vamos apresentar os *code smells*, como eles afetam o design dos softwares e suas ferramentas de detecção. Também abordamos sob medidas de software e suas particularidades em orientação a objetos.

No Capítulo 3 descrevemos o que utilizamos para o desenvolvimento do trabalho, desde o repositório de software, ferramentas e o processo de desenvolvimento. Primeiramente é explicado o processo de seleção do material, passando pelas etapas de escolha do repositório de software, seleção da ferramenta de detecção de *code smells* e refatoração. Neste Capítulo ainda abordamos o método para construção de uma solução para identificação de refatoração que resulta em eliminação de *code smells* e obtenção de medidas de software. Por fim, fazemos as considerações finais acerca dos critérios para obtenção do material e o método proposto.

Apresentaremos os resultados no Capítulo 4, obtidos a partir da construção de uma solução chamada de Refactoring Link (Reflink) e um *dataset* com dados coletados de 1000 projetos.

Finalmente, no Capítulo 5 apresentamos nossas considerações finais, as principais contribuições desta dissertação, as limitações do trabalho e as possibilidades de evolução em trabalhos futuros.

2 REFERENCIAL TEÓRICO

O presente capítulo explana os principais conceitos dos assuntos relacionados ao tema desta dissertação. Este capítulo está dividido em três sub Seções. A Seção 2.1 aborda sobre Refactoring e suas características, passando sobre a perspectiva de como aplicar e o catálogo de refatorações. A Seção 2.2 discorre sobre os *Code Smells*, como eles afetam o design dos softwares e suas ferramentas de detecção. A Seção 2.3 aborda sobre Medidas de Software e suas particularidades em orientação a objetos. Por fim, a Seção 2.4 faz as considerações finais acerca do referencial teórico e os assuntos discutidos.

2.1 REFACTORING

De acordo com Opdyke (OPDYKE; B.S., 1992), *refactoring* ou refatoração é a reestruturação do código de modo a preservar seu comportamento e a permitir que mudanças sejam realizadas facilmente. Para Fowler (FOWLER; BECK, 1999), *refactoring* é uma técnica que altera a estrutura interna do código sem afetar o comportamento externo. Fowler também destaca que *refactoring* é uma atividade arriscada, pois ao se ajustar algo que está funcionando, podem surgir novos problemas.

Em engenharia de software, o termo *refactoring* remete ao processo ou técnica de intervir no código com o objetivo de melhorar a qualidade, reuso, legibilidade e coesão sem que o comportamento externo ou passível de observação seja afetado.

Existem diversas abordagens sobre o conceito de comportamento de um código, alguns autores exploram o comportamento do software sob a ótica de contexto, ou seja, observa-se os dados que são gerados e mantidos ao final de uma execução deste código desde que mantida a mesma entrada. No paradigma orientado a objeto, podemos dizer que o comportamento de um código está associado ao conjunto de valores de seus atributos e a troca de mensagens através da invocação de métodos com os seus argumentos, ou seja, contexto.

Ainda é possível olhar para comportamento do código sob a ótica de performance, consumo de energia e segurança. É possível encontrar trabalhos que

apontam, por exemplo, que o uso de *refactoring* pode contribuir para redução no consumo de energia (PINTO, 2015).

Segundo Fowler (FOWLER; BECK, 1999), *refactoring* é um conjunto de pequenas transformações no código que preservam seu comportamento original. Cada transformação aplicada irá promover uma restruturação para um código mais reutilizável, legível e mais coeso. Um código possui o mesmo comportamento se após a sua execução, os dados gerados e mantidos são os mesmo. Esta é a visão original quando se fala em comportamento de um código. A ideia principal é que um código refatorado continue funcionando normalmente após cada transformação, ou seja, um refactoring deverá preservar a semântica do código transformado e seu funcionamento sob a perspectiva de um observador externo.

Embora esteja focada no código, a refatoração tem um grande impacto na estrutura do sistema (design). É vital que os desenvolvedores e arquitetos compreendam os princípios da refatoração e os usem em seus projetos de maneira adequada. A refatoração é melhor introduzida por um desenvolvedor já experiente na prática de aperfeiçoar o código, ou no termo utilizado comumente, refatorar o código. Esse desenvolvedor pode entender melhor os princípios por trás da refatoração e adaptar esses princípios ao código específico, explorando todos os benefícios no design e manutenção do sistema (CARNEIRO, 2003).

2.1.1 Importância da Refatoração

Com o avanço na demanda pelo uso e evolução de sistemas, as empresas estão cada vez mais afunilando seus prazos e custo, ajustando suas perspectivas através do uso de soluções e metodologias que permitam o desenvolvimento de maneira ágil e competitiva dos sistemas (CARNEIRO, 2003). Sabe-se que a refatoração de código quando feita de forma correta, ajuda não só a diminuição de futuros erros, mas evita o sistema de acumular códigos mal escritos que podem deturpar o design e o comportamento do sistema como um todo.

Apesar de existirem trabalhos que levantem dúvidas sobre a melhoria no código a partir da prática de refatoração (DIG; JOHNSON, 2005; KIM; CAI; KIM, 2011). A proposta da prática de refatoração é que se busca melhorar a estrutura interna de um sistema, a partir da melhoria do código. Esta melhoria não deve alterar o comportamento observável do sistema. A refatoração é vista como uma abordagem disciplinada que permite que o código seja aperfeiçoado sem que novos erros sejam

introduzidos ou que o comportamento seja alterado. A refatoração produz como consequência um aprimoramento no design do sistema (CARNEIRO, 2003).

Quando os desenvolvedores recebem a tarefa de escrever novas funcionalidades, cabe a eles verificar se essa alteração seria mais fácil de implementar se o código existente fosse reestruturado ou se é mais viável um desenvolvimento totalmente novo. Caso optem por um código novo, isto é um indicativo que o sistema pode possuir problemas de design que dificultam sua manutenção ou a inclusão de novos recursos. Por este motivo, a prática de refatoração é uma atividade já presente em metodologias de desenvolvimento modernas. A ideia principal é permitir que os desenvolvedores sempre se perguntem se é possível tornar um programa mais fácil de manter e reutilizar.

A refatoração frequente dificilmente é possível sem uma boa cobertura de testes automatizados. Os testes devem fazer parte do processo de refatoração porque o risco é muito grande de interferir nos componentes em funcionamento em uma extremidade do sistema ao refatorar trechos de códigos com anomalias (*code smells*). Os testes representam um recurso importante para garantir que haverá alerta se um determinado código ou componente teve seu comportamento alterado.

Esse medo de efeitos colaterais “indesejados” ao se alterar um código, é uma das principais razões pelas quais muitos desenvolvedores parecem ignorar a prática de refatoração de componentes funcionais, mas que podem ter problemas de design e que, uma vez restruturado, pode levar o código a ser mais fácil e coeso, ajudando na manutenção e em seu processo evolutivo. A longo prazo, isso leva a sistemas complexos e com custo elevado de manutenção, onde constantemente os desenvolvedores deparam-se com situações em que se opta por construir um novo código e deixar o legado inalterado (evitando a quebra de comportamento). Desta forma, não é surpreendente que a necessidade de refatoração disciplinada para controlar a evolução de um software seja reconhecida como uma prática fundamental (BARROS, 2015).

Barros (BARROS, 2015), dimensiona a importância de se refatorar um código, compreendendo-se o seguinte:

- **Para manter o código limpo:** a refatoração mantém-se longe do que é conhecido como noção de problema de código. Isso pode significar algumas coisas, como código duplicado, muitos parâmetros, métodos

mais longos etc. Por sua vez, afeta a legibilidade do código e pode levar a um tempo de depuração mais longo.

- **Para melhorar o desempenho de um produto:** Performance é um dos motivadores que leva as equipes a desempenharem refatoração. O que significa que apesar de funcionar favoravelmente na visão dos desenvolvedores e/ou clientes. Existe a preocupação que no momento da adição de novos recursos ou dados, a experiência do usuário ficará comprometida.
- **Para economizar tempo e dinheiro:** é sempre melhor manter o código limpo e claro. Isso significa que, quando quiser adicionar novos recursos no futuro, seus desenvolvedores não precisarão começar a desembaraçar o código-fonte complexo e bagunçado, consumindo mais tempo e um orçamento maior. A literatura também mostra que apesar da refatoração não ser prática amplamente adotada nas equipes, não existem estudos empíricos que suportem a premissa de redução de custos a longo prazo (KADAR et al., 2016).
- **Para reduzir débito técnico:** o custo de um projeto de software não é finalizado depois que se lança sua primeira versão. A refatoração do código contribui para redução do débito técnico (MOHAN; GREER; MCMULLAN, 2016).
- **Código desatualizado:** se já faz um tempo desde que alguém o examinou, é possível que possa existir uma forma mais legível e fácil de manter. É preciso verificar sempre a existência destas oportunidades de evolução no código, essa é a importância de se refatorar o código.

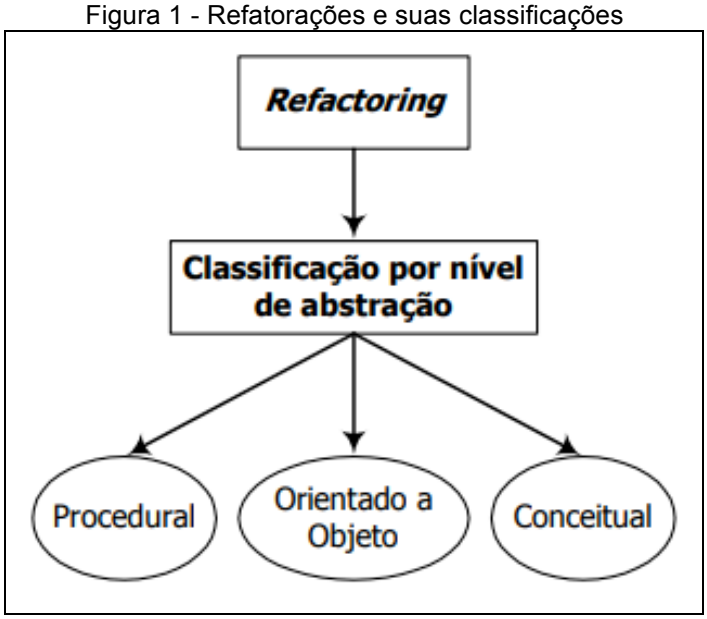
A refatoração ocorre em etapas muito pequenas. Em princípio, cada refatoração individual (por exemplo, renomeando uma classe, deslocando uma operação para uma superclasse) pode ser realizada em alguns minutos. Atividades complexas de refatoração, sempre devem ser quebradas em atividades menores para permitir que versões intermediárias operacionais sejam integradas periodicamente (CARNEIRO, 2003).

Desta forma, a refatoração tende a ser demorada por natureza. As pressões de tempo em um projeto geralmente levam a decisões de deixar um sistema com código de baixa qualidade inalterado. Não obstante, a refatoração sempre deve ser considerada se um sistema tiver uma vida útil longa, tendo em visto o ganho com a facilidade de manutenção e caso necessite permanecer capaz de estar apto a desenvolvimento adicional a qualquer momento.

2.1.2 Classificação e Métodos de Refatoração

Segundo Carneiro (CARNEIRO, 2003), podemos classificar as refatorações em três níveis de abstração, são eles: procedural, orientado a objeto e conceitual (conforme apresentado na Figura 2-1). As refatorações procedurais representam o código propriamente dito, onde de fato as transformações acontecem. A refatoração orientada a objeto atua diretamente no design do sistema, afetando a estrutura do programa. Por fim, de acordo com Carneiro (CARNEIRO, 2003), a refatoração conceitual foca na modificação de conceitos do sistema. A Tabela 2-1 apresenta exemplos de refatorações para cada nível de abstração.

Figura 1 - Refatorações e suas classificações



Fonte: Carneiro (CARNEIRO, 2003), modelo adaptado.

Tabela 1 - Exemplo de refactorings por nível de abstração

Nível Procedural	Nível Orientado a Objetos	Nível Conceitual
Extract Method	Move Method	Introdução de um padrão

A vantagem da refatoração é que ela evita a deterioração da estrutura de um sistema e possui uma série de vantagens (FOWLER; BECK, 1999; OPDYKE; B.S., 1992). Dentre elas podemos citar:

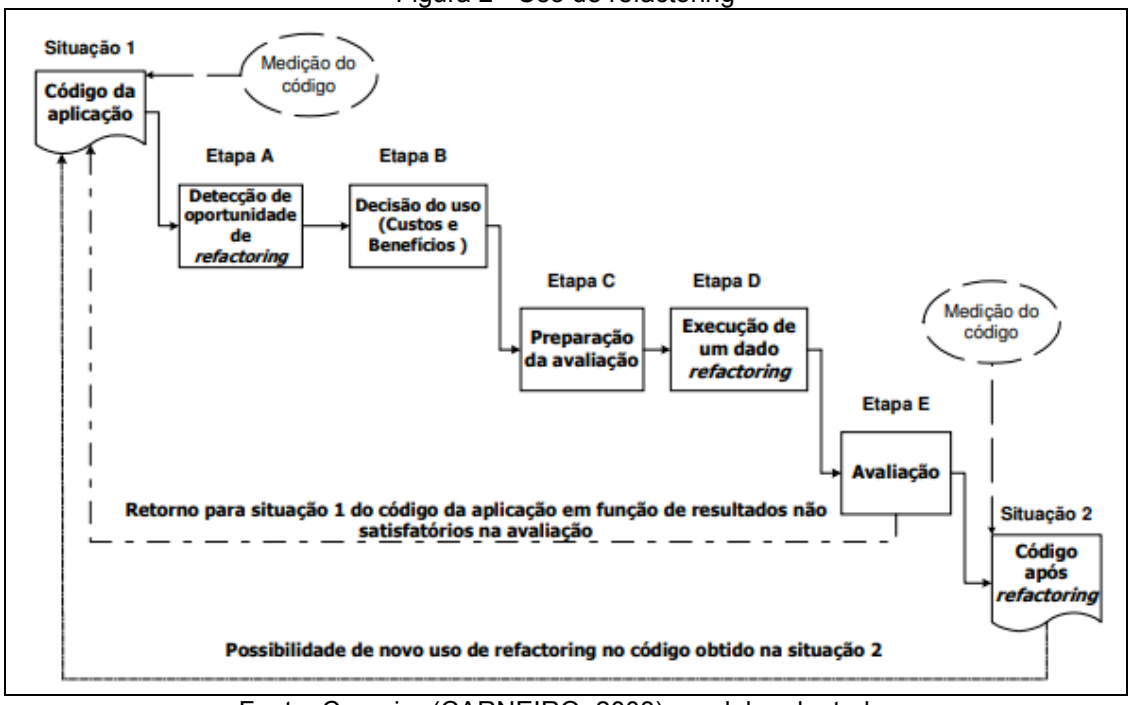
- Design aprimorado;
- Software compreensível;
- Coesão de código
- Localização de erro mais fácil;
- Tempos de desenvolvimento mais curto;
- Diminuição no custo de manutenção.

À primeira vista, os dois últimos pontos podem causar alguma surpresa. No entanto, se pensar bem, um sistema com design aprimorado, ajuda a acelerar o processo de desenvolvimento de software porque facilita o entendimento do código, permite a integração mais fácil de novos desenvolvedores, e por consequência ocasiona um custo menor de manutenção.

Afinal, sem um bom design, o desenvolvimento pode prosseguir sem maiores problemas por um tempo, mas mais cedo ou mais tarde um design ruim irá impactar diretamente a estrutura do sistema e seus componentes, acarretando no oposto as vantagens apontadas na prática de refatoração. Um código pode nascer com uma estrutura adequada naquele momento e após modificações evolutivas ou corretivas, poderá conter problemas (*code smells*) (LEHMAN, 1996).

A refatoração, portanto, ajuda a garantir que o software possa continuar a ser desenvolvido evitando a deterioração da arquitetura de um sistema. Através da refatoração, o design deve de fato melhorar com o tempo. A Figura 2-2 reflete como Carneiro (CARNEIRO, 2003) propôs o uso de refatorações associado a medições no código e avaliação do resultado e do comportamento observável após a transformação.

Figura 2 - Uso de refactoring



Fonte: Carneiro (CARNEIRO, 2003), modelo adaptado.

2.1.3 Como aplicar a Refatoração

Além da compreensão do conceito de refatoração e de sua importância quanto ao processo de aplicação (BECK, 1999; FANTA; RAJLICH, 1998). Fowler (FOWLER; BECK, 1999) também define um processo de como aplicar a refatoração. Don Roberts (ROBERTS, 1999) também sugere, tendo em vista pré e pós condições na aplicação do processo de refatoração. Mens e Tourwé (TOURWE; MENS, 2003) identificaram três passos necessários em 2003 para o processo de refatoração e depois evoluíram o método no trabalho (MENS; TOURWE, 2004) publicado em 2004 para contemplar um contexto de qualidade e preservação de comportamento conforme Tabela 2-2:

Tabela 1 - Atividades necessárias no processo de refatoração segundo Mens e Tourwé

2003	2004
Identificar quando um código deve ser refatorado	Identificar quando um código deve ser refatorado
Identificar quais transformações deverão ser utilizadas no código	Identificar quais transformações deverão ser utilizadas no código
Executar as transformações, se possível de maneira automatizada e que possa ser verificada	Garantir que a refatoração preserva o comportamento do código
	Aplicar a refatoração
	Avaliar o impacto da refatoração na qualidade do software (complexidade, compreensibilidade, manutenabilidade)
	Manter a consistência entre o código refatorado e os artefatos de software (documentação, requisitos, testes, etc.)

Portanto, a refatoração deve ser encarada como uma série de pequenas etapas direcionadas a transformar o código em algo mais fácil de reutilizar e manter. Diversos trabalhos mencionam passos para aplicar refatoração (BECK, 1999; DIG et al., 2006a; FOWLER; BECK, 1999; KIM et al., 2010; SINGH; CHOPRA, 2013; WEISSGERBER; DIEHL, 2006), tais como:

- **Vá devagar, passo a passo:** não tente consertar tudo de uma vez, mas divida o processo em partes consideráveis e siga-o com testes antes de prosseguir. Opdyke (OPDYKE; B.S., 1992) recomenda se faça entrada de dados e se colete a saída antes e após a refatoração como mecanismo básico de teste.
- **Ir para um desenvolvimento orientado a teste:** muitas vezes acontece que a refatoração leva a novos erros. Para evitá-los, se deve sempre testar durante a refatoração. Portanto, antes de iniciar qualquer projeto de refatoração, sempre verifique se há casos de teste automatizados suficientes.
- **Sempre refatorar antes de implementar novos recursos:** um dos principais motivos da refatoração é quando se deseja atualizar ou adicionar novas funcionalidades, pode parecer uma etapa desnecessária que impacta no tempo de desenvolvido, contudo, a longo prazo, é realmente uma economia de tempo e dinheiro.
- **Escolha a automação de refatoração:** como em praticamente qualquer coisa, algumas refatorações podem ser realizadas de forma rápida e eficiente quando feitas automaticamente. Felizmente, o número de IDEs com suporte integrado à refatoração automatizada só cresce através do uso de plug-ins ou add-ons. As mais populares são: o *Eclipse, IntelliJ IDEA, VSCode e Visual Studio*. Vale mencionar também ferramentas que avaliam e detectam a mudança de comportamento, permitindo que se crie um mecanismo de teste e validação automática para o processo de refatoração automática (SOARES; GHEYI; MASSONI, 2013).
- **Refatorar não significa adicionar novos recursos:** não se pode confundir a refatoração de código com a atualização e a adição de novos

> recursos. Não tem nada a ver com isso. Nenhuma nova funcionalidade deve ser criada durante a refatoração. Não é sobre isso. Trata-se de manter o código limpo e fácil de manter para quando chegar a hora de implementar novos recursos.

O objetivo da refatoração de código é bastante claro: manter o código legível, de fácil manutenção e reuso, ou seja, limpo e em ordem. Nem sempre se vê um benefício imediato, mas, a longo prazo, é um investimento valioso em qualquer produto software.

2.1.4 Catálogo de refatorações

Em seu livro "*Refactoring: Improving The Design of Existing Code*", Fowler (FOWLER; BECK, 1999) mostra como o uso de refatoração pode tornar um código orientado a objetos mais simples e fácil de manter. Na obra de 1999, são demonstradas dezenas de refatorações que são agrupadas conforme a Tabela 2-3. Este catálogo foi a origem para o repositório de refatorações (catálogo online (FOWLER, 2013)) que é atualmente mantido em um domínio público e conta com demonstrações na linguagem Java.

Tabela 2 - Refatorações segundo critério de Fowler

Agrupamento	Refatorações
Composing Methods	Extract Method Inline Method Inline Temp Replace Temp with Query Introduce Explaining Variable Split Temporary Variable Remove Assignments to Parameters Replace Method with Method Object Substitute Algorithm
Moving Features Between Objects	Move Method Move Field Extract Class Inline Class Hide Delegate Remove Middle Man Introduce Foreign Method Introduce Local Extension
Organizing Data	Self Encapsulate Field Replace Data Value with Object Change Value to Reference Change Reference to Value Replace Array with Object Duplicate Observed Data Change Unidirectional Association to Bidirectional Change Bidirectional Association to

	Unidirectional Replace Magic Number with Symbolic Constant Encapsulate Field Encapsulate Collection Replace Record with Data Class Replace Type Code with Class Replace Type Code with Subclasses Replace Type Code with State/Strategy Replace Subclass with Fields
Simplifying Conditional Expressions	Decompose Conditional Consolidate Conditional Expression Consolidate Duplicate Conditional Fragments Remove Control Flag Replace Nested Conditional with Guard Clauses Replace Conditional with Polymorphism Introduce Null Object Introduce Assertion
Making Method Calls Simpler	Rename Method Add Parameter Remove Parameter Separate Query from Modifier Parameterize Method Replace Parameter with Explicit Methods Preserve Whole Object Replace Parameter with Method Introduce Parameter Object Remove Setting Method Hide Method Replace Constructor with Factory Method Encapsulate Downcast Replace Error Code with Exception Replace Exception with Test
Dealing with Generalization	Pull Up Field Pull Up Method Pull Up Constructor Body Push Down Method Push Down Field Extract Subclass Extract Superclass Extract Interface Collapse Hierarchy Form Template Method Replace Inheritance with Delegation Replace Delegation with Inheritance

Big Refactorings	Tease Apart Inheritance Convert Procedural Design to Objects Separate Domain from Presentation Extract Hierarchy

Fonte: (FOWLER, 2013; FOWLER; BECK, 1999)

Algumas refatorações tornaram-se bastante populares entre os desenvolvedores e a comunidade, seja pela simplicidade que a sua transformação requer no código, seja pela facilidade em identificar problemas que são elegíveis para resolver com as respectivas refatorações. Dentre estas, podemos citar como exemplo as refatorações que envolvem operações com algum tipo de extração ou movimentação de atributos, métodos ou classes.

2.2 CODE SMELLS

2.2.1 Considerações iniciais

Durante o ciclo de vida de um software, o mesmo passa por inúmeras mudanças, sendo comum o objetivo de aperfeiçoar, aprimorar e buscar sempre incorporar funcionalidades que, respondam aos anseios dos clientes. Contudo, tais mudanças promovidas pelos desenvolvedores, normalmente são realizadas dentro de prazos curtos que são determinados pela necessidade de clientes ou a própria demanda de mercado. Desta forma, pode-se dizer que a qualidade do software muitas vezes é prejudicada, por vezes negligenciada.

Percebe-se que a curto prazo, tais medidas podem até trazer algum tipo de benefício pelas demandas atendidas, mas, por outro lado, acarreta em manutenções de estruturação de código futura, onde nem sempre o programador que escreveu o código em sua origem, será o mesmo que irá realizar tais manutenções, começando a evidenciar os problemas de código (LAVALLÉE; ROBILLARD, 2015).

2.2.2 Code Smells e suas diretrizes

Fowler (FOWLER; BECK, 1999) listou um conjunto de situações onde o código é passível de reestruturação e as chamou de *code smells*. O conceito de *code smells* é proveniente da percepção de um código “cheirar mal”. Um *code smell* é uma violação dos princípios de design de software que pode impactar diretamente a qualidade do

código, manutenibilidade e boas práticas, tais como: KISS (Keep It Simple), YAGNI (You Aren't Gonna Need It), DRY (Don't Repeat Yourself).

Os *code smells* foram definidos por Kent Beck no livro de Fowler (FOWLER; BECK, 1999) como um meio para diagnosticar sintomas que podem ser indicativos de algo errado no código do sistema. Eles se referem a qualquer sintoma no código fonte de um programa que possivelmente indica um problema mais profundo.

Desta forma, os *code smells* são fragmentos de código que sugerem a possibilidade de refatoração. Originalmente, 22 (vinte e dois) *smells* foram descritos e catalogados por Fowler (FOWLER, 2013; FOWLER; BECK, 1999), juntamente com as refatorações sugeridas na obra que é tida como marco inicial na associação entre problemas na estrutura no código e a forma de resolvê-los através de uma transformação.

Outros problemas também foram propostos na literatura, como o *Spaghetti Code* (BROWN et al., 1998) e o *Swiss Army Knife* (BROWN et al., 1998; MOHA et al., 2010). O *Spaghetti Code* é quando um código não segue as regras da programação estruturada e abusa de desvios, condicionais ou não, tornando a leitura e compreensão do código difícil. A expressão é uma crítica a códigos mal organizados, difíceis de analisar, corrigir e modificar. Já o *Swiss Army Knife* é a tentativa do programador em fornecer todas as possibilidades possíveis através de uma interface, tornando o uso complexo e excessivo de um mesmo recurso.

A necessidade de refatoração é indicada pela presença de *code smells*. Os problemas podem degradar aspectos de design e qualidade do sistema, como estrutura, manutenção e compreensão, dificultando o processo de desenvolvimento, tornando o código difícil de entender e, consequentemente, manter (FOWLER; BECK, 1999; LANZA; MARINESCU, 2006).

É importante reforçar que um *code smell* não necessariamente indica um problema real, um bug, uma falha que precisa ser corrigida. É apenas um indicativo de algo que pode ser visto como suscetível a uma refatoração, ou melhor, um *code smell* é um sinal que pode existir uma oportunidade de refatoração.

Geralmente, esses problemas não surgem imediatamente, mas acumulam-se ao longo do tempo à medida que o software evolui (e especialmente quando ninguém se esforça para erradicá-los). Há algumas coisas a considerar ao decidir-se detectar e

identificar estes *code smells*. Alguns podem ser observados diretamente no código e alguns podem ser derivados de fatos extraídos do código. Também é possível observar smells que foram introduzidos ou removidos a partir de informações de repositórios de software com sistema de gerenciamento de versões, onde é possível identificar o estado anterior e posterior do código.

Ferramentas para detecção automática ou semiautomática de *code smells* dão suporte aos desenvolvedores na identificação das oportunidades de refatoração que o código que está sendo desenvolvido possa ter. A implementação de técnicas de detecção permite que as ferramentas realcem as entidades com indicativo de *code smells* (SOUZA, 2017).

As muitas ferramentas de análise de software disponíveis para detectar *code smells* (FERNANDES et al., 2016), evidenciam a importância de controlar a qualidade do produto de software que está sendo desenvolvido e mantido. Por outro lado, traz um novo desafio de como avaliar e comparar estas ferramentas, para permitir que se selecione as mais adequadas de acordo com o contexto ou tipo de desenvolvimento.

De fato, a avaliação da eficácia das ferramentas para detectar *code smells* apresenta alguns problemas (FONTANA; BRAIONE; ZANONI, 2012). Por exemplo, a ambiguidade e, às vezes, a imprecisão nas definições do que pode ser considerado como um problema no código levam a diferentes interpretações para cada *code smell*. Essa falta de definições precisas, implica em ferramentas que implementam diferentes técnicas de detecção para o mesmo tipo de problema de código.

Além disso, essas técnicas geram resultados diferentes, pois geralmente são baseadas no cálculo de um conjunto específico de métricas combinadas, variando de métricas orientadas a objetos a métricas definidas de maneira *ad hoc* para o objetivo de detecção de problemas (LANZA; MARINESCU, 2006).

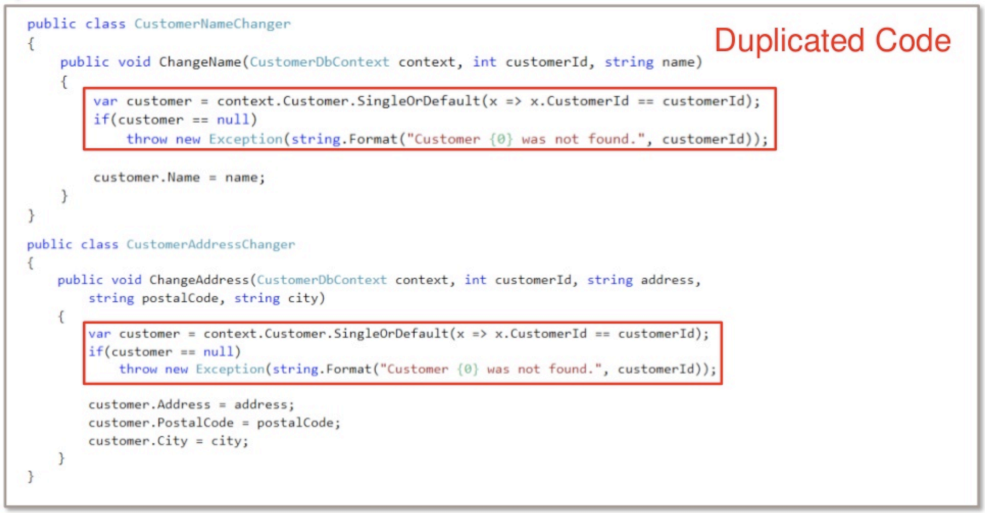
Por fim, mesmo que as mesmas métricas sejam usadas, o valor limite *(threshold)* pode ser diferente porque é definido considerando fatores diferentes, como domínio do sistema e seu tamanho, práticas organizacionais e a experiência dos engenheiros de software que os definem. A alteração dos limites tem um grande impacto no número de problemas de código detectados (VALE et al., 2015).

2.2.3 Principais code smells

2.2.3.1 Duplicated Code

Segundo Fowler (FOWLER; BECK, 1999), a duplicação de código *(Duplicated Code)* é o número um entre os *smells* de seu catálogo. O código duplicado deve ser totalmente evitado no desenvolvimento. Caso encontre uma estrutura de código duplicado, encontre uma maneira de unificar os dois trechos e utilizar apenas uma referência. A Figura 2-3 demonstra um exemplo de duplicação de código que deve ser evitado.

Figura 2 - Duplicação de código



Fonte: https://pt.slideshare.net/kindblad/avoid-code-duplication-principles-patterns/4

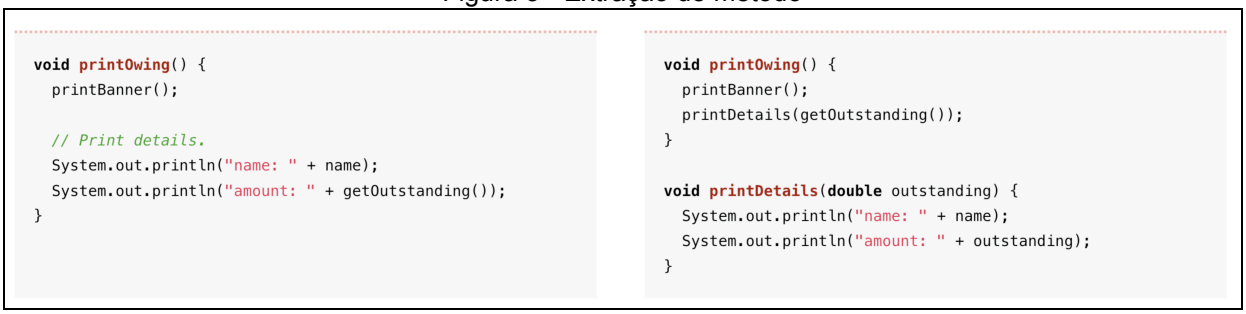
Para o caso demonstrado na Figura 2-3, pode-se utilizar uma refatoração *Extract Method*. O objetivo é criar um novo método com o trecho selecionado, extraindo-o do método original e permitindo que aquele código possa ser invocado e reutilizado em vários locais sem duplicação. A refatoração *Extract Method* também é bastante utilizada para resolver o *code smell Long Method*.

2.2.3.2 Long Method

Para Sobrinho (SOBRINHO, 2019), este *code smell* se relaciona diretamente com *Large Class* e é um indicativo da presença de Spaghetti Code (BROWN et al., 1998). Um método mais curto é mais de ler, mais fácil de entender e mais fácil de manter. Fowler ainda destaca que é o fator chave não é o tamanho do método, é a distância semântica entre o que o método deve ou não fazer. Diretamente, é um conceito ligado a coesão e ao propósito de cada método.

Existem diversas estratégias para resolver um método longo, por exemplo, pode-se lançar mão de uma refatoração denominada extração de método para remover trechos/blocos completos do método que são independentes e possam ser invocados a partir de um novo método (conforme demonstrado na Figura 2-4).

Figura 3 - Extração de método



```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

Fonte: https://refactoring.guru/extract-method

2.2.3.3 Large Class

As classes grandes, assim como os métodos longos, são difíceis de ler, entender e solucionar problemas. Se a classe contém muitas responsabilidades, pode ser reestruturada ou dividida em classes menores.

Fowler também coloca que uma classe que tenta fazer muito, frequentemente apresenta várias instâncias de variáveis e por consequência aumenta-se o risco de duplicação de código. Da mesma forma que no *Long Method*, uma refatoração de extração pode ser utilizada para agrupar as variáveis e métodos em um único componente que seja consistente. As refatorações aplicadas a uma classe geram novas classes ou subclasses.

2.2.3.4 Long Parameter List

Quanto mais parâmetros um método possui, mais complexo é. Limitar o número de parâmetros necessários em um determinado método ou usar um objeto para combinar os parâmetros é importante neste processo.

Em linguagens orientadas a objetos, sempre há a possibilidade de troca de mensagens entre os objetos através da invocação de métodos, permitindo que valores sejam recuperados a qualquer momento. Diante disto, passar todos os parâmetros necessários no método é uma prática desnecessária e oriunda das linguagens imperativas baseadas em rotinas e funções.

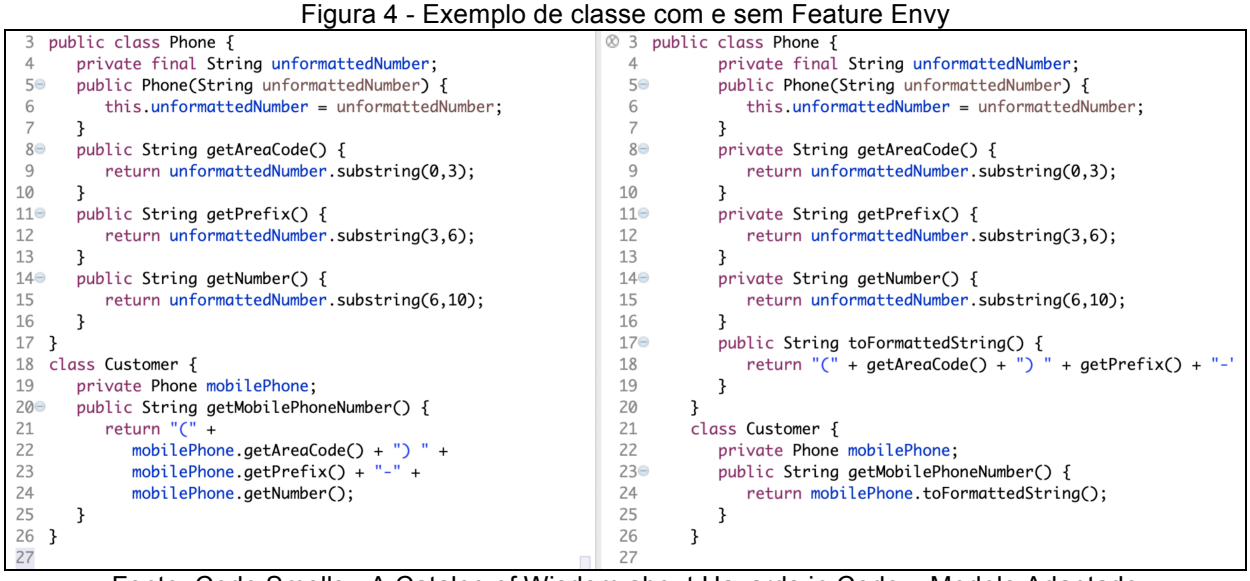
2.2.3.5 Feature Envy

Este *code smell* ocorre quando uma classe se apropria das características de outra classe de maneira indevida. É quando uma classe acessa diretamente as propriedades de outra classe mais do que suas próprias. O conceito de "*envy*" (inveja) decorre exatamente desta característica da classe assumir o comportamento de outra.

Segundo Sobrinho (SOBRINHO, 2019), este *code smell* ocorre porque existe um método interessado pelas propriedades de outra classe, afetando a coesão e o acoplamento da classe. De fato, métodos com este *code smell*, reduzem a coesão da classe porque provavelmente implementa diferentes responsabilidades. A Figura 2-5 demonstra um exemplo de *Feature Envy* na classe *Customer* (imagem do lado direito) e a classe sem o *smell* (do lado esquerdo). O método *getMobilePhoneNumber()* utiliza apenas recursos da classe Phone para desempenhar seu comportamento. Por este motivo a entende-se que a classe está assumindo um comportamento que não lhe pertence. Na implementação do lado direito da Figura 2-5, podemos observar que a classe *Customer* não utiliza mais os atributos da classe Phone. Na solução sem o *FeatureEnvy*, podemos notar a utilização do método *toFormattedString*() encapsulando o comportamento desejado pela classe *Customer*.

Fowler (FOWLER; BECK, 1999) menciona que a solução para este *smell* é bem óbvia e você deve utilizar *Move Method* ou *Extract Method* para colocar o método na devida classe. Outras soluções também são aceitas, como encapsular o acesso as propriedades que são utilizadas através de métodos de acesso público.

Figura 4 - Exemplo de classe com e sem Feature Envy



```
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    public String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6,10);
    }
}
class Customer {
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return "(" +
           mobilePhone.getAreaCode() + ") " +
           mobilePhone.getPrefix() + "-" +
           mobilePhone.getNumber();
    }
}
```

```
public class Phone {
        private final String unformattedNumber;
        public Phone(String unformattedNumber) {
            this.unformattedNumber = unformattedNumber;
        }
        private String getAreaCode() {
            return unformattedNumber.substring(0,3);
        }
        private String getPrefix() {
            return unformattedNumber.substring(3,6);
        }
        private String getNumber() {
            return unformattedNumber.substring(6,10);
        }
        public String toFormattedString() {
            return "(" + getAreaCode() + ") " + getPrefix() + "-"
        }
    }
    class Customer {
        private Phone mobilePhone;
        public String getMobilePhoneNumber() {
            return mobilePhone.toFormattedString();
        }
    }
```

Fonte: Code Smells - A Catalog of Wisdom about Hazards in Code – Modelo Adaptado

2.2.3.6 Data Class

Este *smell* é caracterizado por classes que têm apenas atributos e métodos de recuperação e atribuição (*get/set)* e nada mais (FOWLER; BECK, 1999). As classe de dados devem ser evitadas porque armazenam dados passivamente. As classes devem conter dados e métodos para operar com esses dados também. Segundo Fowler, as classes de dados são como crianças, você poderá utilizá-las no início, mas a partir de algum momento deverão possuir responsabilidade bem definida e uma razão para existir.

2.2.3.7 Outros smells

Apresentamos apenas alguns exemplos de *code smells*, o catálogo de Fowler apresenta uma lista com 22 *smells* de código. A relação completa pode ser consultada no Apêndice A.

2.2.4 Ferramentas para detecção de code smell

As ferramentas de detecção de *code smells* podem ajudar os desenvolvedores a manter a qualidade do software empregando diferentes técnicas para detectar e identificar *code smells* (DIG et al., 2006b; FONTANA et al., 2013; MOHA et al., 2010; NONGPONG; BOYLAND, 2012; SZOKE et al., 2015; TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2008). Outros trabalhos como (BOSCH, 2018; EMDEN; MOONEN, 2002; HAMID et al., 2013; PALOMBA et al., 2017; ROPERIA, 2009; TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2018; VAN EMDEN; MOONEN, 2002) também exploraram e propuseram suas respectivas ferramentas.

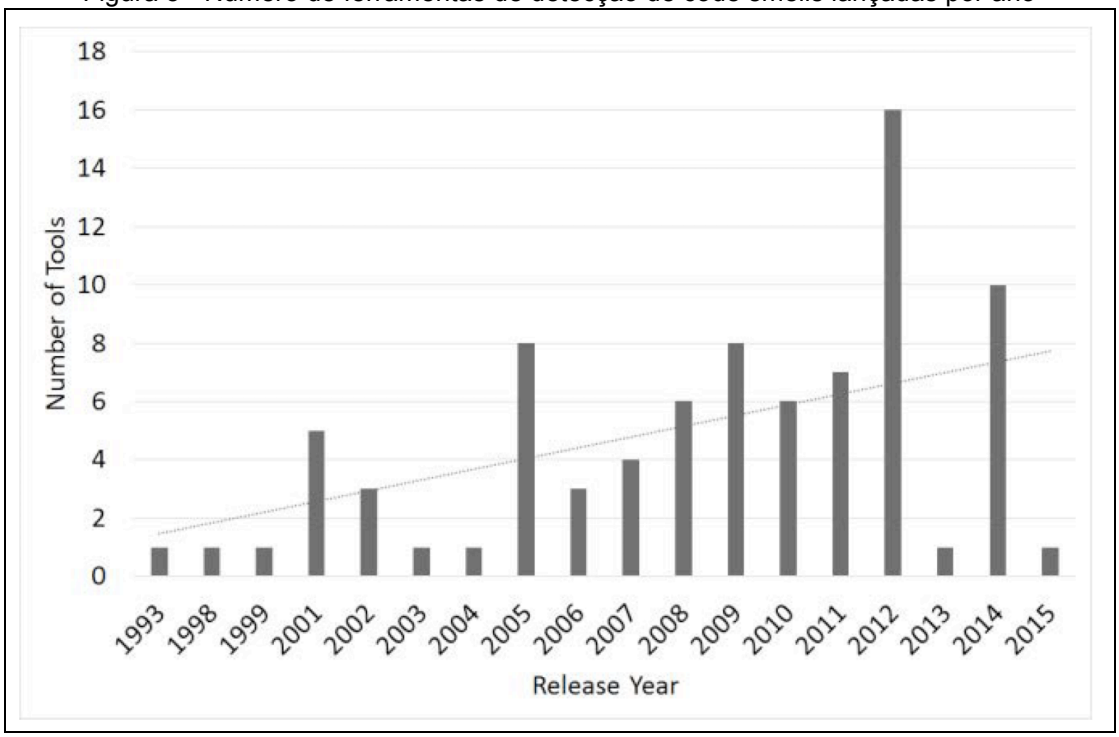
As ferramentas de detecção de *code smells*, a saber, inFusion, JDeodorant, PMD, aDoctor, JQuality e JSpIRIT analisam programas Java, podem ser instaladas e configuradas a partir da internet, executam detecção diretamente no código fonte ou byte code e geram uma saída (*output*) com uma lista de ocorrências de *code smells*.

Fernandes (FERNANDES et al., 2016) encontrou 84 (oitenta e quatro) ferramentas na literatura em seu trabalho, das quais 29 (vinte e nove) estavam disponíveis para download em 2016. A Figura 2-6 ilustra o número de ferramentas de detecção de *code smell* que foram lançadas por ano.

Uma das ferramentas mais utilizadas é o PMD e o JDeodorant. O PMD é um analisador de código fonte que procura e identifica falhas comum durante o

desenvolvimento de código (“PMD Source Code Analyzer”, [s.d.]). Já o JDeodorant é um plugin do Eclipse que detecta code smells e recomenda uma refatoração apropriada (TSANTALIS, [s.d.]). O inFusion é comercial, aceitando Java, C e C ++ que detecta 22 tipos de problemas de código, incluindo *God Class, God Method e Feature Envy*. Como produto comercial, o inFusion não está disponível para download. No entanto, existe a versão de código aberto da ferramenta, chamada iPlasma[1], ainda está disponível.

Figura 5 - Número de ferramentas de detecção de code smells lançadas por ano



Fonte: (FERNANDES et al., 2016)

[1] http://loose.upt.ro/iplasma/

2.3 MEDIDAS DE SOFTWARE

Uma medida de software é uma das características do software que são quantificáveis ou contáveis. As métricas de software são importantes por vários motivos, incluindo medir o desempenho do software, planejar itens de trabalho, medir a produtividade e muitos outros usos. Bunge (BUNGE, 1979) definiu a noção de que objetos possuem propriedades.

Uma propriedade é uma característica que um determinado objeto possui inerentemente. Ainda segundo Bunge (BUNGE, 1979), todos os objetos que importam possuem um conjunto finito de propriedades. Este é o princípio básico de uma métrica de software, captar essa característica e representar através de uma quantificação.

O objetivo de rastrear e analisar as métricas de software é determinar a qualidade do produto ou processo atual, melhorar essa qualidade e inspecionar a qualidade assim que o software estiver concluído. Em um nível mais granular, a utilização das métricas de software nos projetos tem os seguintes propósitos:

- Aumentar o retorno do investimento
- Identificar áreas de melhoria
- Identificar desvios nos padrões estabelecidos para o projeto.

Esses objetivos podem ser alcançados através do uso de métricas que forneçam informações com clareza em todos os momentos do desenvolvimento ou manutenção dos sistemas.

Tiago (TIAGO, 2006), menciona que os termos usados para descrever as métricas de software geralmente têm várias definições e maneiras de contar ou medir características. Por exemplo, linhas de código (LOC) é uma medida comum do desenvolvimento de software. Mas existem duas maneiras de contar cada linha de código:

- Contar cada linha física
- Contar cada declaração lógica

Portanto, um único artefato de software pode ter duas contagens LOC muito diferentes, dependendo do método de contagem usado. Isso dificulta a comparação de um software simplesmente utilizando como métrica as linhas de código ou qualquer

outra métrica sem uma definição padrão. É por isso, que é crucial estabelecer um método de medição e unidades de medida consistentes para serem usadas durante toda a vida do projeto (TIAGO, 2006).

Há também um problema de como as métricas de software são usadas. Se uma organização usa métricas de produtividade que enfatizam o volume de código e erros, os desenvolvedores de software podem evitar problemas complicados para manter seu LOC ativo e a contagem de erros baixa.

Estolano (ESTOLANO, 2005), menciona que os desenvolvedores de software que escrevem uma quantidade de código simples podem ter uma produtividade impressionante, mas não demonstrar habilidades de desenvolvimento que agreguem valor ao produto de software. Além disso, as métricas de software não devem ser monitoradas simplesmente porque são fáceis de obter e exibir - somente as métricas que agregam valor ao projeto e processos devem ser coletadas e rastreadas.

As métricas são úteis para as equipes de gerenciamento, pois oferecem uma maneira rápida de acompanhar o desenvolvimento, definir metas e medir o desempenho. Porém, o desenvolvimento simplificado e orientado apenas a métricas e a indicadores quantitativos, pode desviar os desenvolvedores do real objetivo do negócio, que é a construção de funcionalidade que atenda as expectativas do usuário final ou do cliente (ESTOLANO, 2005).

As métricas de software devem funcionar para as equipes, devem fazer sentido e que sejam percebidas como um norte para o aprimoramento tanto do produto quanto dos desenvolvedores. Medir e analisar não precisa ser oneroso ou algo que atrapalha a criação de código. As métricas de software devem ter várias características importantes. Mediante apontamentos realizados por Estolano (ESTOLANO, 2005), elas deveriam ser:

- Simples e computável;
- Consistente e inequívoca (objetiva);
- Usar unidades de medida consistentes;
- Independente de linguagem de programação;
- Fácil de calibrar e adaptável;

- Fácil e barata de se obter;
- Capaz de ser validada quanto à precisão e confiabilidade;
- Relevante para o desenvolvimento de produtos de software de alta qualidade;

2.3.1 Métricas de Software Orientado a Objetos

As métricas de software orientadas a objeto, conhecida como “Métricas de *Chidamber* e *Kemerer (C&K)” ou C&K*, foram propostas pela primeira vez em 1994, quando as únicas linguagens orientadas a objetos comercialmente significativas eram Smalltalk e C++. A C&K usou duas aplicações comerciais (uma em C ++ e outra em Smalltalk) para avaliar a eficácia das suas métricas (CHIDAMBER; KEMERER, 1991, 1994).

Embora a reutilização seja frequentemente mencionada, esse não era o objetivo inicial do conjunto de métricas originais de *Chidamber e Kemerer*. A partir de *Design Orientado a Objetos com Aplicações*, de *Grady Booch*, e procuravam um conjunto de métricas que ajudassem a obter algo que refletisse o *design* das aplicações OO (CHIDAMBER; KEMERER, 1994).

O uso de medidas permitiria comparar um *design* em potencial com outro e prever qual seria o melhor. Isso significava que as medidas teriam que ser capazes de se basear em um design, e não em um código. Outro ponto interessante foi que C&K acreditava firmemente em uma abordagem comercial prática, afirmando que a orientação das métricas deveriam oferecer as informações necessárias para indicar se os desenvolvedores estão seguindo os princípios definidos em projeto (CHIDAMBER; KEMERER, 1994).

Esse uso de métricas foi crítico, pois se tornou referência dos princípios do paradigma orientado a objetos. A partir daí *Chidamber e Kemerer* propuseram em 1991 um conjunto de métricas específicas para softwares orientado à objetos (CHIDAMBER; KEMERER, 1991) e em 1994 as aperfeiçoaram (CHIDAMBER; KEMERER, 1994). A Figura 2-7 mostra como as métricas foram mapeadas:

Figura 6 - Mapeamento das medidas por Chidamber e Kemerer em 1991 e 1994

C-K Metrics coverage of OO Design Steps

Metric	Identification	Semantics	Relationships
WMC	X	X	
DIT	X		
NOC	X		
CBO			X
RFC		X	X
LCOM		X	

Fonte: (CHIDAMBER; KEMERER, 1994)

2.3.2 Métricas C&K

Chidamber e Kemerer usaram um processo formal com base nos critérios da *Weyuker* (MISRA; AKMAN, 2008; WEYUKER, 1988) para avaliar métricas de software que permitissem avaliar seu próprio conjunto de métricas. Em geral, *Chidamber e Kemerer* achavam que seus critérios incentivavam a proliferação de classes, mas que mais classes tornavam o software mais complexo sob os critérios de *Weyuker* (WEYUKER, 1988).

Os desenvolvedores de C ++ e Smalltalk envolvidos nos softwares usados em seu estudo perceberam o mesmo: mais classes tornaram mais difícil rastrear erros. Isso contraria a nossa visão atual e de *Chidamber e Kemerer* de que atribuímos a cada classe uma única responsabilidade coerente (geralmente por um pequeno conjunto de dados e seus métodos associados) e que isso pode resultar em mais classes.

Chidamber & Kemerer (CHIDAMBER; KEMERER, 1991, 1994), definem as métricas OO da seguinte forma:

CBO (*C*oupling *B*etween *O*bjects) - acoplamento entre objetos: é uma contagem do número de classes que são acopladas a uma classe específica, isto é, onde os métodos de uma classe chamam os métodos ou acessam as variáveis de uma outra. Essas chamadas precisam ser contadas nas duas direções para que o CBO da classe **A,** tenha o tamanho do conjunto de classes que a classe **A** faz referência e as

classes que fazem referência à classe **A**. Como esse é um conjunto - cada classe é contada apenas uma vez, mesmo que a referência de **A** opere em ambas as direções, isto é, se **A** referenciar **B**, referências A e B são contadas apenas uma vez (CHIDAMBER; KEMERER, 1991).

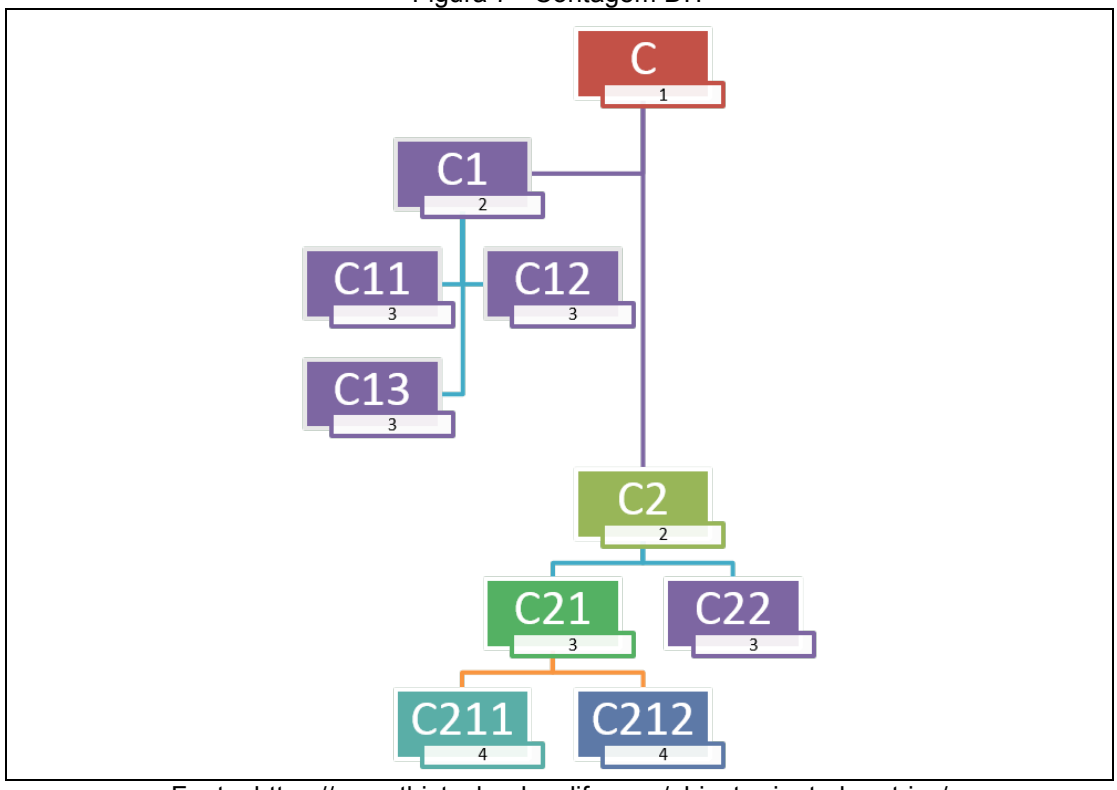
Chidamber & Kemerer consideraram que a CBO deveria ser a mais baixa possível por três razões:

- O aumento do acoplamento aumenta as dependências entre classes, tornando o código menos modular e menos adequado para reutilização. Em outras palavras, se você deseja empacotar o código para reutilização, pode acabar precisando incluir um código que não era realmente fundamental para a funcionalidade principal.
- Mais acoplamento significa que o código se torna mais difícil de manter, pois uma alteração no código em uma área corre um risco maior de afetar o código em outra área (vinculada).
- Quanto mais vínculos entre classes, mais complexo o código e mais difícil será de testar e garantir o comportamento em casos de alteração.

A única consideração no cálculo dessa medida é se apenas as classes escritas para o projeto devem ser consideradas ou se as classes da biblioteca devem ser incluídas.

DIT (*D*epth *I*nheritance *T*ree) – Profundidade da Árvore de Herança: é uma contagem das classes das quais uma classe específica herda (CHIDAMBER; KEMERER, 1991). A Figura 2-8 demonstra o valor do DIT para uma dada hierarquia de classes. É possível observar que as classes em níveis mais profundos possuem sempre uma contagem maior, ou seja, as subclasses sempre terão valores superiores as classes que herdam.

Figura 7 - Contagem DIT



Fonte: https://www.thistechnologylife.com/object-oriented-metrics/

A métrica possui as seguintes consequências com base na profundidade da herança:

- Quanto mais profunda a classe estiver na hierarquia, maior o número de métodos que provavelmente herdará, tornando mais complexo prever seu comportamento.
- Árvores mais profundas constituem maior complexidade de projeto, pois há mais métodos e classes envolvidos.
- Quanto mais profunda uma classe estiver na hierarquia, maior será a reutilização potencial dos métodos herdados. A classe também passa a se tornar dependente e acoplada, dificultando o reuso.

Os dois primeiros pontos parecem ser interpretados como indicadores "ruins", enquanto o terceiro parece ser sugerido como "bom" - tornando difícil decidir se o DIT deve ser grande ou pequeno. Os grandes sistemas são projetados para abrigarem dezenas e até centenas de classes. Em razão disto, os arquitetos e projetistas lançam mão do uso intenso de herança, *design patterns* e recursos estruturais para tirar o máximo de proveito das características do paradigma orientado a objetos.

Sobre o DIT, há ainda algumas considerações:

- A Árvore de Herança deve parar na fronteira do sistema? Ou deve incluir também classes externas ao sistema? *Chidamber & Kemerer* colocam DIT em seu trabalho de concepção (conforme trecho abaixo) – como o número de ancestrais pode afetar a classe em questão (CHIDAMBER; KEMERER, 1991).

> *"Theoretical basis: DIT relates to the notion of scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class."*

  Chidamber & Kemerer consideram que o número de métodos herdados aumentam a complexidade, possivelmente de todas as classes (dentro e fora do código).
- As classes Java têm uma única árvore de herança, mas isso também deve incluir interfaces implementadas e pode incluir dados estáticos definidos nessas interfaces.
- As interfaces Java têm herança múltipla. Se a medida considera a árvore mais profunda em seus cálculos. Este conceito está relacionado ao escopo das propriedades de uma classe definidas por Bunge (BUNGE, 1977, 1979).
- A partir da versão 1.8 do Java já é possível utilizar *default methods* nas interfaces e obter um nível mais aprimorado de herança múltipla.

Mas se a característica indicada pela métrica é o maior número de propriedades (atributos e métodos), o que indica um aumento na complexidade, adotar um único caminho (mais profundo) pode demonstrar um desvio no que a métrica representa.

LCOM (*Lack of **C**ohesion of **M**ethods*) - Falta de Coesão de Métodos**: *Chidamber & Kemerer* definiram o LCOM com base no número de pares de métodos que compartilhavam referências a variáveis de instância (CHIDAMBER; KEMERER, 1991, 1994). Todas as combinações de pares de métodos da classe são avaliadas. Se o par não compartilhar referências a nenhuma variável de instância, a contagem será aumentada em 1 e se eles compartilharem alguma variável de instância, a contagem será reduzida em 1.

O LCOM é visto como uma medida de quão bem os métodos da classe cooperam para atingir os objetivos da classe. Um valor baixo de LCOM sugere que a

classe é mais coesa e é vista como “melhor”. A lógica da C&K para o método LCOM foi a seguinte:

- A coesão dos métodos dentro de uma classe é desejável, pois promove o encapsulamento.
- A falta de coesão implica que as classes provavelmente devem ser divididas em duas ou mais subclasses.
- Qualquer medida de disparidade de métodos ajuda a identificar falhas no design de classes.
- A baixa coesão aumenta a complexidade, aumentando a probabilidade de erros durante o processo de desenvolvimento.

Uma das dificuldades com a medição da métrica LCOM foi que seu valor máximo dependia do número de pares de métodos - com 3 métodos se pode ter 3 pares de métodos, mas com 4 se pode ter 6. Isso significa que, para avaliar se o seu valor LCOM é baixo, é necessário levar em consideração o número de pares de métodos - um LCOM de 2 com 3 pares de métodos deve ser considerado diferente de um LCOM de 2 com 100 pares (HITZ; MONTAZERI, 1996). Este entendimento é abordado por *Chidamber & Kemerer* (CHIDAMBER; KEMERER, 1994) da seguinte forma: o LCOM de uma classe ser igual a 0 (zero) não implica necessariamente que esta classe é coesa, algumas classes com LCOM igual a 0 podem ser mais coesas que as outras.

Isso foi abordado por *Henderson-Sellars* (HENDERSON; SELLERS, 1996) quando ele introduziu uma variação do LCOM, às vezes conhecida como LCOM3 ou LCOM-HS, com o LCOM-C&K sendo conhecido como LCOM1. O LCOM-HS é calculado pela equação apresentada na Figura 2-9:

Figura 8 - Equação da medida LCOM-HS

$$LCOM = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1 - m}$$

Fonte: (LEITNER, 2017)

Onde o número de métodos é representado por *m* e os atributos por *a*, o número de métodos que acessam um atributo é definido por $\mu(A_j)$. LCOM pode estar no intervalo de 0 a 2, com valores a partir de 1 sendo vistos como sugestivos de um design inadequado.

Segundo Leitner (LEITNER, 2017), se todos os métodos acessam todos os atributos, ou seja, este é o melhor cenário possível para coesão, o LCOM-HS será 0. O pior cenário é quando cada atributo é acessado apenas por um método, neste caso o LCOM-HS será 1. A métrica só irá apresentar valores superiores a 1 quando existirem atributos mortos ou não utilizados pelos métodos da classe.

Existem outras derivações da métrica LCOM, a primeira versão do LCOM (LCOM1 ou LCOM-CK) sendo a proposta por *Chidamber & Kemerer* (CHIDAMBER; KEMERER, 1991), LCOM2 sendo a variação proposta por *Henderson-Sellers, Constantine & Graham* em seu trabalho *Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)*, LCOM3 ou LCOM-HS sendo a derivação já abordada acima, e por fim, a versão LCOM4 ou LCOM-HM que considera as melhorias propostas por Hitz and Montazeri (HITZ; MONTAZERI, 1995).

NOC (*N*umber *O*f *C*hildren) - Número de Filhos: é definido por *Chidamber & Kemerer* como o número de subclasses imediatas de uma classe. A opinião de *Chidamber & Kemerer* é que quanto maior o número de filhos, maior o nível de reutilização, pois a herança permite que os métodos e atributos sejam reutilizados em toda hierarquia das subclasses (CHIDAMBER; KEMERER, 1991). Algumas observações acerca da métrica são:

- Quanto maior o número de subclasses, maior a probabilidade de abstração indevida na superclasse. Uma classe com o número excessivo de subclasses pode ser um indicativo para uma distorção na especialização destas subclasses.
- Segundo *Chidamber & Kemerer* (CHIDAMBER; KEMERER, 1991), o número de subclasses é um indicativo da importância da classe no design.
- Uma hierarquia de subclasses promove uma característica desejável em sistemas orientado a objetos por promover a reutilização de código. Em

contrapartida, quanto maior o número de subclasses, maior a complexidade na construção e execução dos testes do código.

- Não é uma boa prática e pode não corresponder à realidade que todas as classes tenham um mesmo valor para o NOC (CHIDAMBER; KEMERER, 1991).

Portanto, o ponto de vista geral é que aumentar o NOC indica um aumento na capacidade de reutilização, mas se isso não for executado corretamente, pode prejudicar o design do sistema e acarretar em um aumento na atividade de testes.

RFC (*R*esponse *F*or a *C*lass) - Respostas da Classe: esse é o tamanho do conjunto de respostas de uma classe. O conjunto de respostas para uma classe é definido por *Chidamber & Kemerer* como:

> "um conjunto de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto dessa classe"

Isso significa que todos os métodos na classe e todos os métodos chamados por métodos nessa classe são contados. Como é um conjunto, cada método chamado é contado apenas uma vez, não importa quantas vezes seja chamado. *Chidamber & Kemerer* (CHIDAMBER; KEMERER, 1991) coloca ainda os seguintes pontos a respeito da métrica RFC:

- Se um grande número de métodos puder ser chamado em resposta a uma mensagem, o teste e a depuração da classe se tornarão mais complicados, pois requer um maior nível de entendimento necessário da parte do testador.
- Quanto maior o número de métodos que podem ser invocados de uma classe, maior a complexidade.

Este é provavelmente o corte mais claro das métricas - uma RFC alta sugere que algo está errado. Podemos olhar para o RFC como um complemento do CBO, pois este mede o acoplamento de uma determinada classe sem dizer em que grau. O RFC vai indicar quantos métodos podem ser executados a partir da invocação de uma mensagem que foi recebida pelo objeto.

WMC (*W*eighted *M*ethods per *C*lass) - Métodos Ponderados para Classe: os métodos ponderados para classe (WMC) foram originalmente propostos por *Chidamber & Kemerer* como a soma de todas as complexidades dos métodos na classe. Em vez

de usar a complexidade ciclomática, eles atribuíram a cada método uma complexidade de um, tornando o WMC igual ao número de métodos na classe. A maioria das implementações clássicas segue essa regra. As seguintes ponderações foram feitas por C&K (CHIDAMBER; KEMERER, 1991) sobre WMC, são elas:

- O número de métodos e a complexidade dos métodos envolvidos é um indicador de quanto tempo e esforço são necessários para desenvolver e manter a classe.
- Quanto maior o número de métodos em uma classe, maior o impacto potencial nas subclasses, pois elas herdarão todos os métodos definidos na classe.
- Classes com grande número de métodos provavelmente serão mais específicas para uso geral no contexto de aplicação, limitando a possibilidade de reutilização.

Existem alguns problemas com essa medida, começando com o nome - não chame uma medida que a princípio deveria indicar um peso por classe se é atribuído imediatamente a cada método um valor igual e arbitrário de 1. Desta forma, o WMC original reflete a quantidade métodos que uma classe possui. As classes com mais métodos são menos suscetíveis a reutilização em razão do aumento da complexidade e da perda de coesão. Parece estranho, pois, potencialmente, há mais para reutilizar.

2.3.3 Utilização de métricas para avaliação de software

Uma visão geral pode ser analisada a partir as referências sobre *Chidamber & Kemerer*, assim resumida:

- CBO: cálculo direto, onde valores baixos teoricamente são bons.
- DIT: em geral, um DIT alto é visto como um indicador bom. O uso deve ser feito sob julgamento após um exame da hierarquia da classe. Há algum debate sobre cálculo, e no caso do JAVA, tem de se avaliar o uso de interfaces.
- LCOM: ampla variedade de abordagens de medição. Em geral, altos níveis são vistos como ruins.
- NOC: quanto maior o número de filhos, maior o nível de reutilização, pois a herança permite que os métodos e atributos sejam reutilizados em toda hierarquia das subclasses.

- RFC: uma contagem alta é um indicador útil de problemas em potencial, em razão da quantidade de métodos que podem ser executados por uma simples mensagem entre objetos.
- WMC: C&K fazem uma prova para constatar que a medida simplesmente pondera igualmente todos os métodos da classe.

2.4 Considerações finais

Apresentamos neste capítulo conceitos sobre refatoração, *code smell* e medidas de software, abordando suas principais características de forma a construir um entendimento acerca dos assuntos abordados neste trabalho. A pesquisa em refatoração e *code smells* sempre estiveram intimamente ligadas por tratar de aprimoramento de código e aspectos de qualidade. Neste sentido, a utilização de medidas de software como ferramenta de entendimento de tais aspectos, demonstra ser bastante útil na análise de como o código se comporta quando se depara com situações onde refatorações ou *code smells* foram introduzidos ou removidas.

3 Material e Método

Neste capítulo descrevemos o que utilizamos para o desenvolvimento do trabalho, desde o repositório de software, ferramentas e o processo de desenvolvimento. Primeiramente é explicado o processo de seleção do material, passando pelas etapas de escolha do repositório de software, seleção da ferramenta de detecção de *code smells* e refatoração. Neste Capítulo ainda abordamos o método para construção de uma solução para identificação de refatoração que resulta em eliminação de *code smells* e obtenção de medidas de software. Por fim, fazemos as considerações finais acerca dos critérios para obtenção do material e o método proposto.

3.1 SELEÇÃO DE REPOSITÓRIO DE SOFTWARE

O primeiro passo deste estudo foi definir que critérios serão adotados para selecionar os repositórios de software. Antes de efetivamente definir os critérios, adotou-se algumas premissas que são intrínsecas a este estudo com o propósito de facilitar o processo de escolha e utilizar tecnologias atuais e informações disponíveis publicamente. As premissas definidas foram:

- O repositório deve utilizar o GIT como sistema de controle de versões;
- O repositório deve ser público e permitir acesso a todo o histórico de evolução do respectivo software;
- O repositório deve estar hospedado por alguma solução relevante no meio acadêmico e/ou comercial. Em outras palavras, o *hosting* do repositório deverá ser uma empresa/marca sólida e amplamente utilizada.

As premissas acima foram definidas em razão de se evitar algum conflito de interesse entre os desenvolvedores e proprietários dos softwares, de forma que nos deparássemos com algum projeto com diversos *code smells* detectados e isto viesse a criar algum desconforto com os desenvolvedores. Também ponderou-se em evitar um esforço na compreensão e abrangência do modelo de licenciamento disponível para cada repositório ou projeto de software. Há dezenas de modelos de licenciamento que por muitas vezes são confusos, proibitivos e desconexos com a realidade. O maior

interesse neste estudo é constituído pelas informações sobre o código fonte e todo o histórico de evolução do repositório.

Desta forma, este trabalho definiu alguns critérios para selecionar os repositórios de software que serão analisados pelas ferramentas de detecção de *code smell*, detecção de *refactoring* e coleta de métricas de software. Os critérios foram:

- Repositórios que foram marcados como projetos desenvolvidos na linguagem de programação Java;
- Repositórios com diferentes níveis de colaboração e popularidade; onde popularidade será definida de acordo com as informações disponibilizadas pelo *hosting* público dos repositórios que será escolhido com base nas premissas adotadas anteriormente;
- Repositórios com tamanhos variáveis.

Diante destes critérios, vale ressaltar que uma premissa implícita para seleção dos repositórios é a condição de que as informações dos repositórios sejam passíveis exportação ou de integração através de alguma **API *(Application Programming Interface)***.

3.2 SELEÇÃO DE FERRAMENTA DE DETECÇÃO DE CODE SMELLS

Assim como o processo de escolha dos repositórios de software, definimos um conjunto de critérios para a seleção de uma ferramenta de detecção de *code smell*. Este passo é fundamental para alcançar nossos objetivos, tendo em vista que todo o processo de análise usará como insumo as informações coletadas a partir da execução de uma ferramenta de detecção de *code smell*. Desta forma, definimos os seguintes critérios para seleção da ferramenta:

- Ferramenta com código fonte disponível;
- Ferramenta que identifique qual o recurso/classe foi identificado o *code smell*; e se possível, em qual o trecho;
- Ferramenta desacoplada de qualquer IDE;
- Ferramenta que permita ser invocada ou incorporada a um projeto e que possa ter uma execução manipulada livremente;
- Ferramenta que não tenha como pré-requisito que o projeto contido no repositório de software precise ser compilado.

Um dos focos deste trabalho é atuar na investigação de como o código com *code smells* se comporta na presença de *refactoring*, e como suas respectivas métricas são afetadas em razão disto. Portanto, não é objetivo deste projeto construir uma ferramenta de detecção do zero, ou até mesmo dedicar esforços para adaptar soluções existentes no mercado.

De acordo com a revisão literária, a construção de ferramentas de detecção de *code smells* é assunto explorado por pesquisadores ao longo dos últimos anos como pode ser visto em (DIG et al., 2006b; FONTANA et al., 2013; HAMID et al., 2013; NONGPONG; BOYLAND, 2012; PALOMBA et al., 2017; PRETE et al., 2010; SOBRINHO, 2019; SOUZA, 2017; SZOKE et al., 2015; TSANTALIS, [s.d.]) . Também é senso comum que o catalogo disponibilizado por Fowler (FOWLER, 2006, 2013), é amplamente aceito como os tipos de *code smells* mais conhecidos e explorados pela comunidade científica e pela indústria.

Alguns critérios essenciais foram adotados para esta etapa do estudo com o propósito de alinhar as ferramentas de detecção com os objetivos deste trabalho. Sendo assim, os seguintes critérios foram definidos como essenciais:

- A ferramenta de detecção deverá identificar *code smells* na linguagem Java
- A ferramenta deve possuir capacidade para detecção dos *code smells* presentes no catalogo de Fowler;

Apesar da rigidez dos critérios, tomamos como base a variedade e a diversidade de estudos que já propuseram soluções (ferramentas) de detecção de *code smells* em diferentes áreas de atuação e linguagens de programação.

3.3 SELEÇÃO DE FERRAMENNTA DE DETECÇÃO DE REFATORAÇÃO

O cuidado em definir critérios consistentes para a seleção das ferramentas utilizadas no estudo foi sempre parte presente no processo de planejamento e revisão. Desta forma, a utilização de ferramentas funcionais que agregassem agilidade na busca por respostas dos objetivos estabelecidos para este trabalho.

A ferramenta de detecção de refatorações é de extrema importância para confirmar se existiu uma ação de transformação no código que por ventura venha a ter solucionado um *code smells*. Também é importante ressaltar que, de posse do conhecimento de quando uma transformação (*refactoring*) foi aplicado no código,

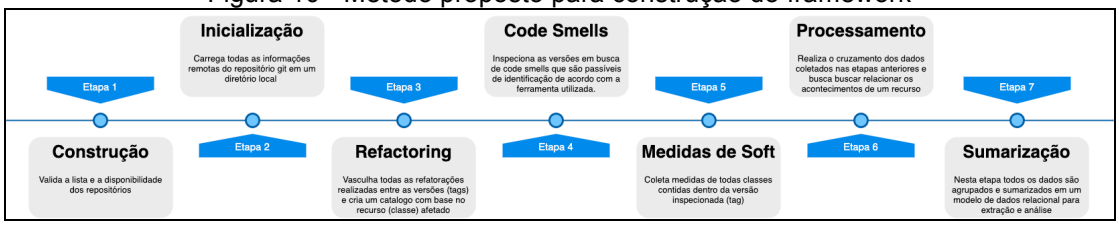
podemos, por conseguinte, inferir se as métricas de software tiveram uma sinalização clara da respectiva transformação, ou até mesmo se os indicadores convergiram na mesma direção para diferentes projetos.

Sendo assim, buscamos selecionar ferramentas de detecção de refatorações que executassem análise do código fonte e não do código compilado (*bytecode*). Esta abordagem tem como propósito evitar situações complexas que envolvem a construção de um ambiente para a compilação adequada dos projetos. Assim como nas ferramentas anteriores, também optamos por selecionar ferramentas que possam ser integradas ao código fonte e que possuam mecanismos para manipular as informações processadas. A intenção é ter acesso as métricas de maneira individualizada para posterior cruzamento.

3.4 MÉTODO

Esta seção apresenta o método proposto para construir uma solução para identificação de refatoração que resulta em eliminação de *code smells* e obtenção de medidas de software. As seções subsequentes descrevem as etapas referentes à Construção, Inicialização, *Refactoring*, *Code Smell,* Medidas de Software, Processamento e Sumarização. A Figura 3-1 demonstra a sequência de etapas do método e o que pretende-se abordagem em cada momento.

Figura 10 - Método proposto para construção do framework



3.4.1 Etapa 1 - Construção

Nesta etapa, avaliamos os repositórios de software público, verificando a validade, integridade e disponibilidade dos dados. A ideia nesta fase inicial é garantir que os dados de entrada estejam consistentes. Nesta etapa serão coletadas informações iniciais do repositório, tais como: nome do projeto (sistema), URL (fornecida no input), estado do repositório (se está apto a avançar para outras fases).

O propósito da principal na etapa de construção é garantir que apenas repositórios do GIT válidos venham a ser processados. Também são verificados todos os requisitos de ambiente e sistema que são necessários para próxima etapa.

3.4.2 Etapa 2 - Inicialização

Nesta etapa, todas as informações do repositório serão carregadas em um diretório local. Como já explanado anteriormente, nossa base são os repositórios que utilizam o GIT como sistema de controle de versões. Logo, está no planejamento tirar proveito das informações fornecidas através do uso de um cliente do GIT. Deixando claro que a escolha da ferramenta será feita posteriormente no Capítulo 6.

Esta etapa irá obter todo o histórico de evolução do software que está sendo avaliado, carregando todas as informações sobre *commits, merges, tags*, classes e desenvolvedores.

As informações sobre as versões (*tags*) e seus respectivos *commits* são fundamentais para o método estabelecido neste trabalho, tendo em vista que o reflexo da evolução de uma aplicação está contido nas alterações que são realizadas por seus desenvolvedores e submetidas ao repositório através de *commits*. As *tags* são marcos importantes que indicam quando ocorreu uma liberação para o público de uma nova versão, com adição de funcionalidades ou correção de problemas.

A etapa de inicialização reforça a premissa deste trabalho em só utilizar repositórios públicos, onde todo o histórico de evolução e seus envolvidos está disponível para ser obtido e analisado de forma mais profunda.

3.4.3 Etapa 3 - Refactoring

De posse do código fonte disponível no repositório que foi validado e inicializado nas etapas anteriores, este método propõe que seja estabelecida uma *timeline* do software através de suas versões, vasculhando o código em busca de refatorações que tenham sido aplicadas durante a evolução do software. Partindo do princípio que a refatoração é uma transformação realizada no código, fica claro que para desempenhar esta etapa é necessário olhar para o código fonte em momentos distintos.

De forma mais direta, para vascular o repositório em busca de refatorações, estabelecemos as versões como pontos de aferição, onde as alterações realizadas

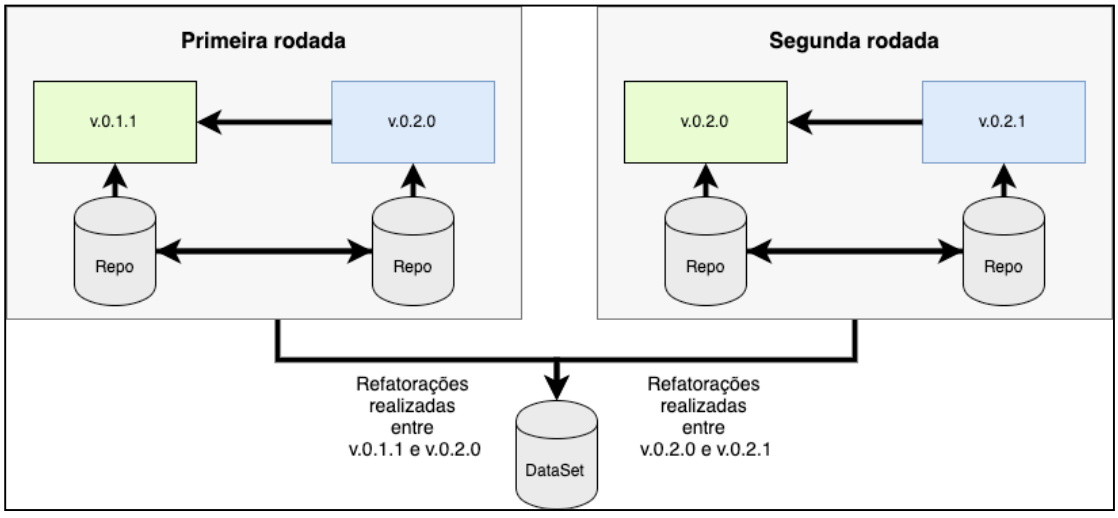
entre duas versões serão comparadas a partir de suas *tags*. Esta é uma abordagem mais prática para evitar o cruzamento entre *commits,* pois alguma modificação pode ser desfeita em *commits* próximos. Uma versão é algo mais sólido, o *commit* está limitado ao escopo de um desenvolvedor ou equipe, enquanto uma versão é algo que é uma formalização do final de um ciclo de desenvolvimento.

As refatorações devem ser vasculhadas sob a ótica das *tags* disponíveis no histórico do GIT. Vamos exemplificar com o projeto PhotoEditor: uma biblioteca para edição de imagens com alguns recursos de filtros, *emojis* e *stickers*. O PhotoEditor possui 6 (seis) versões disponíveis conforme a Tabela 3-1. A etapa de *refactoring* deverá vascular por refatorações como demonstrado na Figura 3-2, inspecionando as alterações que foram aplicadas entre as versões. A Etapa 3 irá realizar ciclos de verificação entre as *tags*, partindo das alterações realizadas entre a v.0.2.0 e a v.0.1.1, em seguida v.0.2.1 e a v.0.2.0 e seguir por este processo até que a última versão tenha sido inspecionada.

Tabela 4 - Versões do PhotoEditor

PhotoEditor	
Versões (tags)	**Data de publicação**
v.0.4.0	12/06/2019
v.0.3.3	28/08/2018
v.0.3.2	08/08/2018
v.0.2.1	03/06/2018
v.0.2.0	29/05/2018
v.0.1.1	25/05/2018

Figura 11 - Exemplo da Etapa 3 com versões do PhotoEditor



Vamos construir um framework que utilize a ferramenta de detecção de refatorações de maneira transparente, onde a responsabilidade de manipulação da *timeline* dos repositórios seja independente da ferramenta de detecção selecionada. Nesta fase do desenvolvimento deste trabalho, conseguimos identificar um requisito para abordagem discutida acima e ilustrada na Figura 3-2: todos os repositórios devem conter no mínimo duas *tags* criadas para que seja possível estabelecer as transformações que foram executadas no código.

3.4.4 Etapa 4 – Code Smells

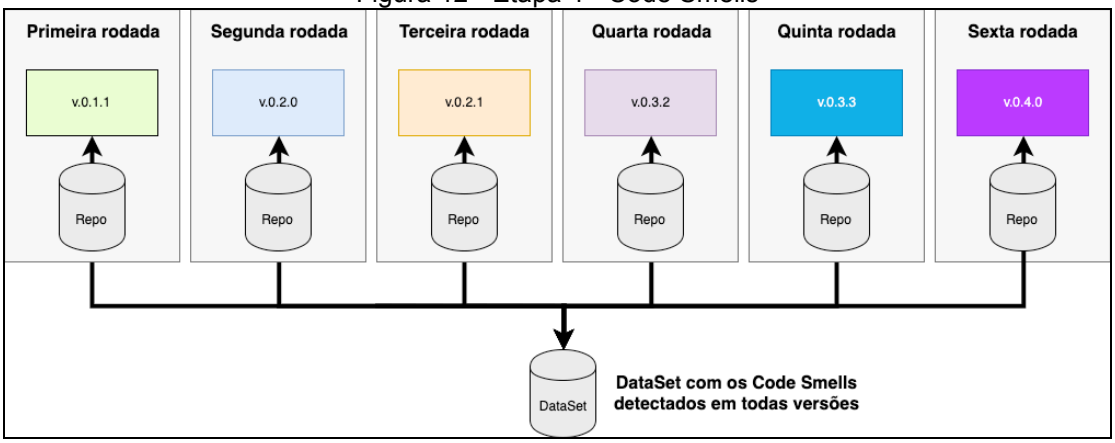
Nesta etapa, inspecionamos todos os recursos (classes) existentes em cada versão do software. De forma semelhante ao processo realizado na Etapa 3 – *Refactoring*, todas as versões serão vasculhadas em busca de *code smells*. A intenção é inicialmente conseguir trabalhar com o catálogo de *code smells* apresentados por Fowler (FOWLER; BECK, 1999).

Os tipos e a quantidade de problemas que serão detectados estará diretamente ligado as ferramentas utilizadas nesta etapa. O objetivo é construir um framework que permita a identificação de quais problemas (*code smells*) foram eliminados quando associado a ocorrência de refatorações.

Buscamos estabelecer uma relação entre recursos (classes), refatorações, *code smells* e medidas de software que permita construir uma fonte de dados que proporcione informações sobre os efeitos das refatorações ao longo da vida de um software.

Existe uma diferença sútil entre a etapa anterior e esta. Todas as versões devem ser analisadas e terem seus dados coletados, contudo, o *code smell* pode ser encarado como um problema que existe ou não em um dado momento do software. A Figura 3-3 mostra que executaremos uma inspeção em cada versão, diferente da Etapa 3 que no momento de sua execução é preciso duas versões para cada rodada.

Figura 12 - Etapa 4 - Code Smells



3.4.5 Etapa 5 – Medidas de Software

Neste etapa, coletamos medidas de software acerca dos recursos (classes) existentes em cada versão do repositório analisado. O objetivo é permitir que um histórico seja traçado a partir do momento que as versões evoluem, sofrem alterações, refatorações e tem seus indicadores afetados.

Ao final desta etapa, teremos em nosso *dataset* as seguintes informações acerca do repositório público avaliado:

- código fonte de todas as versões do software ao longo da sua vida;
- *commits* com informações sobre o que foi alterado (diff), quando foi alterado (data), por quem foi alterado (autor) e com qual justificativa (comentário);
- medidas de software que são utilizadas para avaliar diversos aspectos de um software;
- *code smells* que podem indicar problemas no código;
- refatorações que foram aplicadas em determinada classe durante o desenvolvimento de um software.

A partir destes dados, vamos estabelecer relações entre eles, realizando o processamento e extraindo informações. A infraestrutura de software que vamos construir permitirá tais atividades.

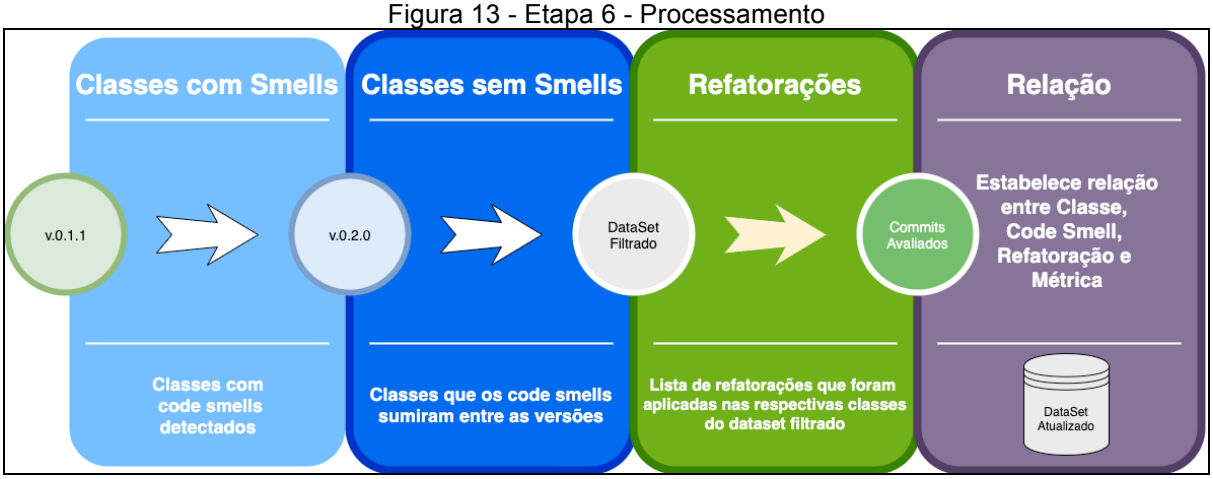
3.4.6 Etapa 6 – Processamento

O processamento deve consistir em, a partir de todos os dados que já foram coletados, buscar por informações ou indícios que estejam alinhados com os objetivos deste

trabalho. Nesta etapa, o foco é buscar por situações previamente identificadas, similaridades entre dados de projetos distintos, cruzamento entre características obtidas nos dados, relacionamentos através de identificadores chaves, entre outros. Contudo, o principal direcionamento no processamento são os passos abaixo (ilustrado na Figura 3-4):

- Realizar um cruzamento de todos os recursos (classes) que tiveram *code smells* detectados em uma versão, e na seguinte o mesmo não foi detectado, ou seja, não consta mais no projeto;
- Executar rotina de comparação entre código afetado para identificar se é o mesmo *code smell* ou um novo foi inserido no mesmo recurso;
- Buscar se ocorreram refatorações entre uma versão e a anterior (conforme demonstrado na Figura 3-2);
- Buscar indícios sobre decisões de refatoração nos *commits* que realizaram a refatoração;
- Relacionar o recurso com as refatorações, *code smells* e métricas;

Figura 13 - Etapa 6 - Processamento



Ao final desta etapa todos os dados já foram cruzados, comparados e tiveram suas respectivas relações estabelecidas. O objetivo é concluir a Etapa 6 com o *dataset* reduzido apenas aos casos que podem indicar que a ocorrência de refatoração levou à eliminação de *code smells*.

3.4.7 Etapa 7 – Sumarização

Neste etapa, todos os dados são agrupados e sumarizados em um modelo de dados. Este trabalho está baseando sua proposta em um modelo de dados relacional. O objetivo é estruturar os dados de maneira simples e fácil, permitindo representar as relações encontradas na etapa de processamento.

3.5 FERRAMENTAS SELECIONADAS

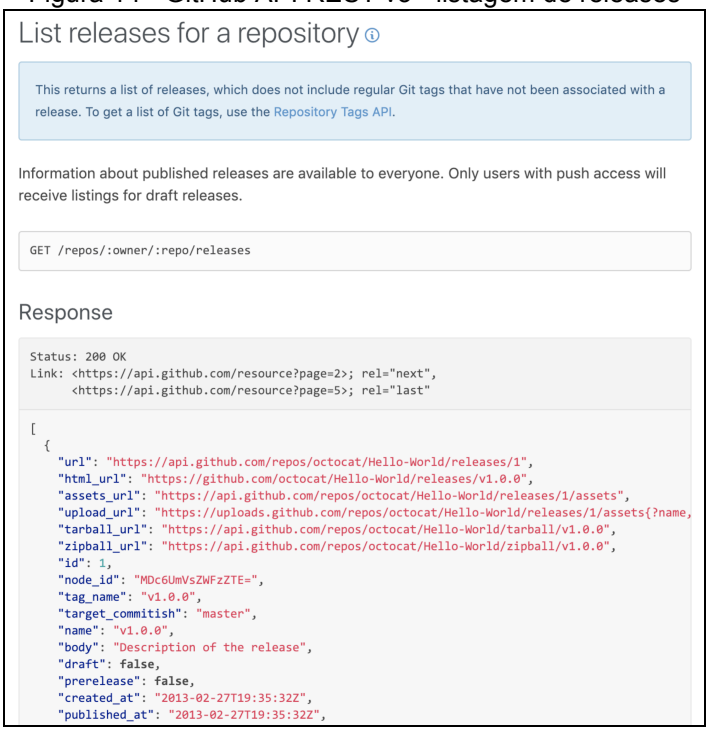
Este estudo definiu o Github[2] como fonte para a busca de repositórios de software que foram selecionados com base nas premissas e critérios estabelecidos no método utilizado por este trabalho. O Github é a maior comunidade *open source* do mundo. Em 2018, o Github contou com mais de 31 milhões de desenvolvedores, só em 2018 o Github viu sua base de usuários aumentar mais do que o total de usuários combinados dos seus seis primeiros anos de existência.

O Github conta com aproximadamente 100 milhões de repositórios de diferentes tipos de linguagens de programação, finalidade e tamanho. Mais de 2.1 milhões de organizações já utilizam o Github como solução de repositório de código fonte.

Além disso, este estudo resolveu utilizar o Github porque ele oferece ferramentas de pesquisa avançada, onde é possível definir um conjunto de filtros e critérios para seleção dos repositórios. O Github ainda oferece uma API para integração que permite obter informações sobre o repositório de forma direta e automatizada (conforme pode ser visto na Figura 3-5).

[2] https://github.com

Figura 14 - GitHub API REST v3 - listagem de releases



Fonte: https://github.com

3.5.1 JQuality/SmartSmell

Para a escolha da ferramenta de detecção de *code smells*, houve diversas dificuldades no que concerne à obtenção de uma ferramenta funcional ou até mesmo disponível. A literatura tem diversos relatos de soluções de detecção de *code smells* (FERNANDES et al., 2016). A grande maioria das ferramentas nunca foi disponibilizada publicamente ou quando está disponível, não funciona em conformidade com o seu propósito.

Desta forma, o processo de seleção da ferramenta de detecção levou mais tempo do que o esperado e acabou exigindo um esforço extra. Cogitamos a alternativa de criar uma solução do zero, caso nenhuma ferramenta fosse encontrada dentro dos critérios de seleção.

Analisamos diversas ferramentas, buscando apoio em trabalhos que fizeram revisão das soluções existentes na literatura. Fernandes (FERNANDES et al., 2016) encontrou 84 (oitenta e quatro) ferramentas na literatura, das quais 29 (vinte e nove) estavam disponíveis para download em 2016. Outros trabalhos como (BOSCH, 2018; EMDEN; MOONEN, 2002; HAMID et al., 2013; PALOMBA et al., 2017; ROPERIA, 2009; TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2018; VAN EMDEN; MOONEN, 2002) também exploraram e propuseram suas respectivas ferramentas.

Selecionamos duas ferramentas de detecção de *code smells*, a *Jquality/SmartSmell* (BOSCH, 2018) e o *aDoctor* (PALOMBA et al., 2017).

O *Jquality*[3] ou *SmartSmell*[2] (como foi chamado até ser renomeado), é uma ferramenta de análise estática de código Java. O *Jquality* é um detector de *code smell* que foi desenvolvido utilizando o *JavaParser*. No momento em que avaliamos a ferramenta, sua última versão era 1.0.0.RC8. A Tabela 3-2 apresenta a lista de *code smell* que são detectados pela versão utilizada.

Tabela 5 - JQuality Code Smell List

BrainMethod	*ClassDataShouldBePrivate*	*CommentSmell/Javadoc*
ComplexCondition	*ComplexMethod*	*Cycle*
Deadcode	*GodClass*	*DataClass*
FeatureEnvy	*LargeClass*	*LongMethod*
LongParameterList	*MessageChain*	*MiddleMan*
NestedBlockDepth	*RefusedParentBequest*	*ShotgunSurgery*
StateChecking	*TraditionBreaker*	

O *Jquality* possui características alinhadas com nossos requisitos: possui repositório público, integração com gerenciador de compilação e dependências Gradle[4], mecanismo de seleção de quais *code smells* deseja-se detectar, e o mais importante, parametrização de limites de detecção (*threshold)* que permite definir diferentes níveis e formatos de detecção dos *code smells*. Por exemplo, alguns *code smells* do catalogo de Fowler podem ser vistos de formas distintas. O *LongMethod* ou *LargeClass* são *code smells* que não existe um consenso do valor limite (*threshold*) ideal para determinar se é ou não um *code smell*.

O *Jquality* permite que você parametrize o valor de acordo com a perspectiva pessoal ou com literatura que esteja seguindo. Por padrão, o limite para *LongMehotd e LargeClass* são 20 linhas e 150 linhas respectivamente.

3.5.2 aDoctor

A segunda ferramenta que selecionamos foi a *aDoctor*[4], é uma ferramenta para detecção de *code smells* específicos de aplicativos *Android*. O *aDoctor* é um projeto desenvolvido na linguagem Java, disponível em repositório público e com capacidade de detectar um conjunto de 15 falhas de design em aplicativos *Android*. No momento em que avaliamos está ferramenta, sua última versão era um *merge commit* realizado em 17 de agosto de 2017. A Tabela 3-3 apresenta a lista de *code smell* que são detectados:

[3] https://github.com/arturbosch/jquality

[4] https://github.com/fpalomba/aDoctor

Tabela 6 - aDoctor Code Smell List

Debuggable Release	*Slow Loop*	*Data Transmission Without Compression*
Inefficient Data Format and Parser	*Inefficient Data Structure*	*Inefficient SQL Query*
Internal Getter and Setter	*Leaking Thread*	*Leaking Inner Class*
No Low Memory Resolver	*Unclosed Closable*	*Durable Wakelock*
Member Ignoring Method	*Public Data*	*Rigid Alarm Manager*

Da mesma forma que o *Jquality*, o *aDoctor* possui a possibilidade de indicação de quais *code smells* deseja-se detectar. Possui também um formato estruturado de exportação dos resultados encontrados. Com a combinação destas duas ferramentas, a solução desenvolvida por este trabalho estará apta a detectar 35 *code smells* de propósitos distintos. Isto permitirá a construção de um conjunto de dados (*dataset*) significativo para o cruzamento e associação com as refatorações e as métricas de software.

3.5.3 Ferramenta de Identificação de Refatoração

Com relação a ferramenta de detecção de refatorações, analisamos diversos trabalhos científicos que avaliaram ou propuseram uma solução acerca de detecção automática de refatorações (CARNEIRO, 2003; CEDRIM et al., 2017; CORNÉLIO; CAVALCANTI; SAMPAIO, 2005; SOARES et al., 2011; TOURWE; MENS, 2003; WEISSGERBER; DIEHL, 2006). A intenção foi selecionar uma ferramenta que identifique as transformações realizadas em variáveis, métodos e classes ao longo da história do projeto avaliado. Como já dito anteriormente, a finalidade destas transformações é melhorar a estrutura do software e torná-lo mais simples e fácil de manter. Portanto, uma ferramenta que detecte de maneira eficaz estas transformações, é fundamental para o desenvolvimento deste estudo.

Existem diversas técnicas para detecção de refatoração. A maioria delas utiliza como base para seu algoritmo o processo de "*code clone detection*". Também é válido mencionar a técnica "*signature-based*" de Weißgerber (WEISSGERBER; DIEHL, 2006), que já demonstrou ser uma técnica de sucesso (BIEGEL et al., 2011).

É importante destacar que Weißgerber também adota *code clone detection* como estratégia para construir um *rank* de possíveis candidatos. Para Kim et al (KIM et al., 2010), as técnicas que se baseiam apenas em duas entradas do código fonte e uma inspeção neste para detectar as alterações (princípio de *code clone detection)* são

inadequadas em termos de cobertura por estarem limitadas a identificar um conjunto específico de transformações.

Avaliamos três ferramentas de detecção de refatoração, analisando os aspectos e premissas definidos em nosso critério de seleção. As ferramentas avaliadas foram: RefactoringCrawler, RefactoringMiner e Ref-Finder.

O RefactoringCrawler (DIG et al., 2006b) é uma ferramenta que detecta refatorações a partir de duas entradas, a original, onde o código fonte ainda não sofreu nenhuma transformação, e a refatorada, onde o código já foi alterado. A ferramenta combina análise sintática para identificar possíveis refatorações e, utiliza análise semântica para refinar a detecção. As transformações detectadas no RefactoringCrawler estão na Tabela 3-4.

Tabela 7 - RefactoringCrawler refactoring

Rename Method	*Rename Class*	*Rename Package*
Move Method	*Pull Up Method*	*Push Down Method*
Change Method Signature		

A segunda ferramenta avaliada foi *RefactoringMiner*. Basicamente ela detecta refatorações que foram aplicadas ao longo do desenvolvimento de um software escrito em Java. O RMiner (abreviação utilizada pelos próprios autores (TSANTALIS et al., 2018)) suporta a detecção de 40 (quarenta) refatorações e a lista completa pode ser verificada no Apêndice C. A ferramenta já foi utilizada por dezenas de pesquisadores e tem como um dos autores Danny Dig, que também é autor da ferramenta RefactoringCrawler.

O RMiner é baseado em encontrar padrões a partir da construção de árvores sintáticas abstratas (AST). Também possui um diferencial que dispensa calibrações dos valores limites de detecção (*thresholds)*. Este fato, aliado à precisão de 98% no processo de detecção (TSANTALIS et al., 2018), e ao fato de poder ser integrada diretamente ao nosso código através de injeção de dependência via Maven, foram as razões para escolhermos como ferramenta dentre as três opções avaliadas.

Por fim, avaliamos a ferramenta Ref-Finder, que identifica refatorações entre duas versões de um código usando uma abordagem "*template-based refactoring reconstruction*" (PRETE et al., 2010). O Ref-Finder expressa cada tipo de refatoração em termos de um modelo de regras lógicas, utilizando um mecanismo lógico para inferir instâncias de refatoração concretas.

Apesar da ferramenta suportar sessenta e três (63) tipos de refatorações do catálogo de Fowler (FOWLER, 2013), e isto demonstrar uma cobertura aparentemente mais abrangente, avaliamos a ferramenta e analisamos estudos empíricos acerca da precisão da ferramenta (HEGEDŰS et al., 2018; KADAR et al., 2016; SOARES et al., 2013). Na avaliação inicial já resolvemos descartar a utilização do Ref-Finder em razão desta ferramenta ser um *plugin* do eclipse e por encontrar evidências em outros estudos acerca da qualidade na identificação das refatorações, onde a precisão chegou a variar entre 27% e 35% (HEGEDŰS et al., 2018; KADAR et al., 2016; SOARES et al., 2013).

3.5.4 CK

A ferramenta selecionada para coletar as medidas de software foi a *CK Tool.* Ela calcula medidas de software para classes, métodos, atributos e variáveis locais em projetos Java através de análise estática. A grande vantagem é que a ferramenta dispensa a necessidade de compilação do código. A Tabela 3-5 apresenta a lista de medidas de software que são coletadas pela ferramenta. A versão selecionada coleta um total de 37 medidas, incluindo aquelas propostas por *Chidamber & Kemerer* (CHIDAMBER; KEMERER, 1991).

Tabela 8 - Medidas coletadas pela CK Tool

cbo	wmc	dit	rfc
lcom	totalMethods	staticMethods	publicMethods
privateMethods	protectedMethods	defaultMethods	abstractMethods
finalMethods	synchronizedMethods	totalFields	staticFields
publicFields	privateFields	protectedFields	defaultFields
finalFields	synchronizedFields	nosi	loc
returnQty	loopQty	comparisonsQty	tryCatchQty
parenthesizedExpsQty	stringLiteralsQty	numbersQty	assignmentsQty
mathOperationsQty	variablesQty	maxNestedBlocks	anonymousClassesQty
subClassesQty	lambdasQty	uniqueWordsQty	

O conjunto de medidas de software coletado foi utilizado para construção do *dataset* e na avaliação em como as refatorações e *code smells* impactaram em seus respectivos valores. Uma das contribuições deste trabalho é construir um modelo de dados estruturados com um *dataset* de projetos hospedados em repositórios de software público e que tiveram seus dados processados e coletados. Este conjunto de dados pode ser utilizado para ampliar a compreensão dos impactos dos *code smells* e das práticas de refatoração sob a ótica de medidas de software, principalmente as voltas para sistemas orientado a objetos.

3.6 CONSIDERAÇÕES FINAIS

Neste Capítulo apresentamos todos os critérios que definimos para obter os materiais necessários para o desenvolvimento deste trabalho, desde a definição de como selecionar o repositório de software, como os critérios para seleção das ferramentas de refatoração e *code smell*. Também apresentamos o método que desenvolvemos para a construção do framework que permita o desenvolvimento de um software que faça integração de mineração de repositórios de projetos versionados através GIT, com ferramentas de detecção de *code smells* e refatorações. Por fim, apresentamos as opções de ferramentas que selecionamos para utilizar neste trabalho.

4 RESULTADOS E DISCUSSÃO

Neste capítulo apresentaremos os resultados obtidos a partir da construção de uma solução chamada de Refactoring Link (Reflink) e um *dataset* com dados coletados de 1000 projetos. A Seção 4.1 irá apresentar a ferramenta *Refactoring Link (RefLink Tool),* solução que foi desenvolvida para processar de forma automatizada os repositórios de software em busca da relação entre refatorações e *code smells*, bem como coletar as medidas ao longo da história do software. A Seção 4.2 aborda o conjunto de dados construído a partir do processamento do RefLink. Apresentamos o modelo de dados utilizado para estruturar e armazenar as informações do *dataset*. A Seção 4.3 apresenta os dados coletados, sumarizando projetos, versões, classes, refatorações, *code smells* e medidas de software. Ainnda na Seção 4.3, discorremos na discussão acerca das medidas de software e seus reflexos em alguns cenários. A Seção 4.4 discute as principais ameaças à validade deste trabalho.

O objetivo deste trabalho é identificar a relação entre as refatorações e *code smells,* observando seus reflexos em medidas de software. Para alcançar este objetivo, o trabalho foi dividido em quatro fases incrementais. A primeira fase consistia em compreender as soluções de identificação de refatorações, *code smells* e métricas de software. Buscando definir critérios práticos para que no processo de seleção, apenas ferramentas alinhadas com o objetivo do trabalho fossem utilizadas.

A segunda fase, era construir uma ferramenta que fosse capaz de conciliar a mineração de dados em repositórios públicos de software, a integração e execução das ferramentas selecionadas na primeira fase. A terceira fase consistia em consolidar todos os dados obtidos na fase anterior, construindo um conjunto de dados que fosse passível de análise, um *dataset* estruturado com as informações correlacionadas. A quarta e última fase foi a

análise efetiva dos dados, buscando identificar os reflexos que as refatorações e os *code smells* causam nas medidas de software.

4.1 REFACTORING LINK (REFLINK TOOL)

Um dos objetivos deste trabalho é propor uma solução automatizada capaz de minerar repositório de softwares, coletando todas as informações ao longo do ciclo de vida de um sistema e de identificar *code smells* que foram introduzidos ao longo do desenvolvimento, bem como aqueles que desapareceram. Outra funcionalidade da ferramenta é a de capturar as refatorações que ocorreram entre as versões do sistema, de forma que seja possível identificar quais foram os recursos afetados por essas transformações e assim criar um link entre a refatoração e o *code smells*. Além destas funcionalidades, construímos a ferramenta RefLink (Refactoring Link Tool) que também é capaz de coletar 37 (trinta e sete) medidas de software por recurso.

O conceito de recurso no *RefLink* está associado a uma classe Java, seja ela externa ou interna. O RefLink foi projetado para relacionar as informações sobre code smell, refatorações e medidas de software com o recurso, ou seja, cada classe estará associada com um conjunto de medidas específicas em um dado momento da sua existência (versão do software). Cada classe estará relacionada a um conjunto de *code smells* que foram identificados em cada versão, e cada refatoração estará associada a duas versões do mesmo recurso.

4.1.1 Integração de ferramentas

Várias ferramentas foram utilizadas para a construção do núcleo do *RefLink*. Estas ferramentas foram integradas ao código fonte através de injeção de dependências com MAVEN[5] ou chamadas de sistema, externa a aplicação através de invocações de binários dentro de um contexto previamente definido na própria aplicação.

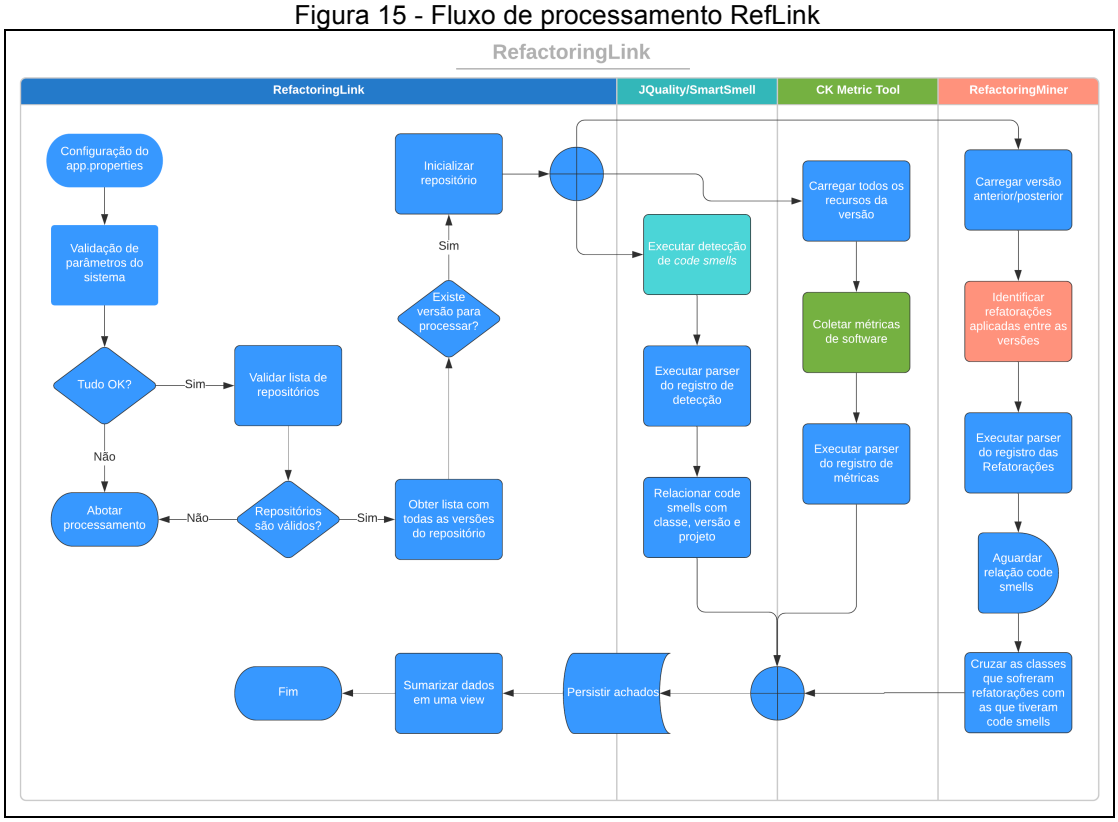
A integração das ferramentas *JQuality/SmartSmell, aDoctor, RefactoringMiner e CK Tool,* permitiu que a nossa ferramenta

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

(RefactoringLink – RefLink) tivesse acesso a um conjunto de dados e que o esforço no seu desenvolvimento fosse focado em estabelecer a correlação entre os recursos com as refatorações e *code smells*, utilizando a mineração dos repositórios de software para obter as informações.

O fluxo de processamento pode ser visto conforme demonstrado na Figura 4-1. É possível identificar que a *RefLink* possui uma estratégia de execução orientada a garantir a integridade dos dados em cada etapa do modelo de processamento. A integração com as ferramentas acontece em três momento específicos, durante os seguintes processos: Executar detecção de code smells, coletar medidasde software e identificar refatorações aplicadas entre as versões. Apenas nestes pontos é que existe interação com outras ferramentas.

Figura 15 - Fluxo de processamento RefLink



Os principais processos do fluxo de execução do RefLink serão brevemente explicados abaixo, maiores detalhes poderão ser explorados diretamente no código fonte da ferramenta.

- **Configuração do app.properties** este processo realiza toda a inicialização da aplicação, definindo diretório de dados, caminho das aplicações que serão integradas, parâmetros de execução, versão do Java, versão do Git e a lista de repositórios de software que serão minerados e processados pelo RefLink. A configuração deste arquivo é obrigatória para execução da ferramenta. O processo de **Validação de parâmetros do sistema** irá realizar checagens acerca do sistema operacional, espaço em disco, conectividade com internet e outros requisitos necessários para a correta execução da ferramenta.
- **Validar lista de repositórios** verifica se todos os repositórios que foram indicados no arquivo definido no parâmetro GITHUB_PROJECT_LIST são válidos: possuem repositórios e são públicos.
- **Obter lista com todas as versões do repositório** carrega a lista com todas as versões (tags) disponíveis no repositório. Um *shell script* é responsável por obter essa lista de versões de maneira cronológica. Aqui já destaca-se um problema encontrado em alguns repositórios que as versões não refletem uma temporalidade na evolução do software. Este script é parametrizado e pode ser definido na variável GIT_TAG_SCRIPT no arquivo app.properties.
- **Inicializar repositório** neste processo o repositório já foi validado e agora será inicializado com seus dados, commits e versões. O RefLink possui um *initializer* que irá construir toda a estrutura local para abrigar os dados do repositório e irá carregar todas as informações remotas através de comandos do GIT.
- **Executar detecção de code smells** neste processo as ferramentas de detecção de *code smells* são invocadas e um parse é realizado para conversão dos dados no formato que foi estruturado neste trabalho. A Figura 4-2 demonstra como as entidades de dados foram modeladas. Durante o parse, são coletadas as seguintes informações do *code smell*: tipo do *smell*, classe onde ocorreu, linha, coluna (quando aplicável), qual a ferramenta que identificou, versão (*tag*) e detalhes sobre o smell.

- **Coletar métricas de software** para cada versão do sistema (tag), são coletadas 37 medidas para classe presente na versão, a Tabela 3-5 contém a lista de todas as medidas. É importante ressaltar que este processo é o responsável por “fotografar” o estado do software durante todo o seu ciclo de vida. É aqui que coletamos as medidas de todas as classes durante a evolução do software. Com esta fotografia é que poderemos analisar os reflexos que as refatorações e os *code smells* exercem nas medidas de software.
- **Identificar refatorações aplicadas entre as versões** aqui o *RefactoringMiner* é invocado. Antes disso, duas versões são definidas, a atual e a anterior. Esta informação será utilizada pelo *RefactoringMiner* para detectar se ocorreram refatorações entre as versões. Conforme já mencionado, o *RefactoringMiner* detecta um total de 40 tipos de refatorações e pode ser integrado através de um API no código fonte ou diretamente através de uma chamada de sistema. Para a nossa integração no *RefLink*, optamos por realizar uma chamada de sistema para diminuir o acoplamento da solução.
- **Cruzar as classes que sofreram refatorações com as que tiveram *code smells*** este processo é responsável por relacionar a refatoração que foi aplicada a uma classe que tinha um *code smell* em uma versão, e desapareceu na mesma versão em que a refatoração foi aplicada. Adiante desta relação, também é realizado um match dos *commits* com uma base de palavras chaves para se buscar alguma correlação entre a intenção do desenvolvedor ao realizar tal *commit*. O objetivo desta etapa, era verificar se a refatoração foi conscientemente realizada pelo desenvolvedor.

  Outra questão é como o RefLink valida que um *code smell* que existia em uma classe na versão anterior, desapareceu na versão seguinte. O trecho de código onde foi detectado o *code smell* é comparado utilizando o algoritmo de similaridade Levenshtein (WIELING; NERBONNE, 2009). O trecho de código é comparado e para qualquer resultado acima de 80% de similaridade, o mesmo tipo de *code smell*,

identificado pela mesma ferramenta e na mesma classe, o RefLink assume que é o mesmo *code smell*.

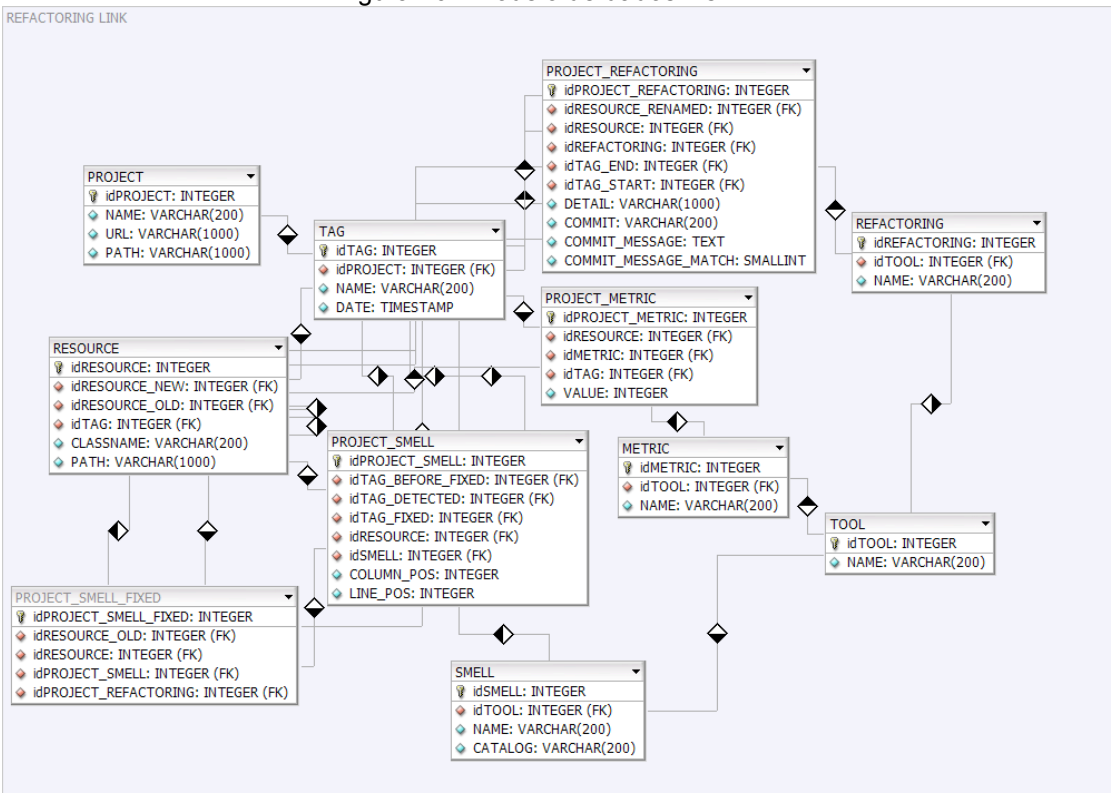
- **Sumarizar dados em uma view** os dados processados pelo RefLink foram agrupados em uma *view* para facilitar a busca e análise dos dados. Esta *view* apresenta todas as classes agrupadas por versões com os dados referentes a *code smells*, refatorações e medidas de software.

4.2 DATASET REFLINK

O modelo de dados ER (Entidade Relacionamento) utilizado para representar as entidades de dados que foram coletadas neste trabalho, está representado na Figura 4-2. O modelo traz um conjuto de 10 tabelas, estruturadas da seguinte forma:

- *Project*: responsável por representar a entidade projeto. Cada repositório terá um registro.
- *Tag*: representa as versões de cada projeto. No github as versões são obtidas através do conceito de tags. *Tag* nada mais é do que uma marcação para um *commit* específico.
- *Resource*: um registro nesta tabela representa uma classe java. Um recurso pode estar associado a um novo recurso caso ele tenha sido movido ou renomeado.
- *Tool*: representa as ferramentas que estão cadastradas no RefLink.
- *Refactoring*: cada registro representa um tipo de refatoração suportado pela ferramenta relacionada.
- *Smell*: idêntico a tabela *refactoring*. Cada registro representa um tipo de *code smell* suportado pela ferramenta.
- *Metric*: idêntico as tabelas *smell* e *refactoring*.
- PROJECT_REFACTORING, PROJECT_METRIC, PROJECT_SMELL: representam o relacionamento com o recurso, versão e seu respectivo sufixo.
- PROJECT_SMELL_FIXED: representa uma ocorrência sumarizada de um smell que foi corrigido por uma ou mais refatorações.

Figura 16 - Modelo de dados RefLink



4.3 DADOS COLETADOS

Os dados coletados serão apresentados de forma analítica e com valores agrupados por projeto. Nós selecionamos 1000 projetos utilizando os seguintes critérios na API do Github:

q=**android**+language:**java**+stars:**>=1000**&sort=**stars**&order=**desc**

Foi utilizado o curl para obter um arquivo json a partir da string de busca definida acima e limitou-se a utilizar os 1000 (mil) repositórios com maior classificação (critério de stars). A lista completa com os repositórios encontra-se no Apêndice B.

```
curl -o
android_start_gt1000_repo_github.json https://api.github.com/search/repositories?q=android+language:java+stars:>=1000&sort=stars&order=desc
```

Após a obtenção da lista dos projetos e durante o processo de inicialização do repositório no RefLink (vide Figura 4-1 – Fluxo de

processamento RefLink), foram excluídos os projetos que se enquadraram nos seguintes critérios:

- Não possuir nenhuma versão publicada
- Possuir apenas uma versão publicada
- Não possuir código fonte Java
- Ocorreu algum erro durante a operação de *git clone* ou *git fetch --tags*

A Tabela 4-1 demonstra o quantitativo de projetos válidos e inválidos, que estão quantificados nos critérios descritos acima.

Tabela 9 - Quantitativo de projetos

Válidos	427
Inválidos	573
Não possui versão	335
Possui apenas uma versão	74
Não possui código fonte	21
Erro durante clone ou fetch	143

A Tabela 4-2 demonstra o total de versões analisadas, total de classes inspecionadas, média de versões por projeto e a média de classes por projeto.

Tabela 10 - Quantitativo de versões e classes

Total de versões válidas	11.593
Total de versões inválidas	319
Total de classes	2.141.207
Média de versões por projeto	≅27
Média de classes por projeto	≈98

Apresentamos na Tabela 4-3 os quantitativos das refatorações e *code smells.* Também apresentamos a média por projeto e a quantidade de refatorações que resultaram em correção de *code smells* (indiretamente). Uma informação importante na Tabela 4-3 é a quantidade de projetos sem refatorações, logo a média calculada de refatorações por projeto equivale a seguinte fórmula:

Média de refatorações por projeto =

$$MFP = \frac{TR}{(TPV - TPSR)}$$

Onde:
TR = total de refatorações
TPV = total de projetos válidos
TPSR = total de projetos sem refatorações

192.525/(427-44)
502,67

Tabela 11 - Quantitativo de refatorações e code smells

Total de refatorações	197.525
Total de *code smells*	4.555.351
Média de refatorações por projeto	≈502
Média de refatorações por versão	≈13
Média de *code smells* por projeto	≈9.541
Média de *code smells* por versão	≈395
Refatorações que resolveram *code smells**	4.508
Code smells resolvidos por refatoração*	5.354
Projetos sem refatorações identificadas	44

* A contagem assume que se um mesmo recurso em uma mesma versão (anterior/seguinte) sofreram transformações e o *smell* desapareceu, todas as refatorações aplicadas naquela classe são relacionados ao *smell* que desapareceu.

A Tabela 4-4 apresenta os dados sobre as refatorações e *code smells* mais comuns, os tipos de refatorações que mais refletiram em correções e quais os *code smells* mais corrigidos.

Tabela 12 - Refatorações e Code Smells mais comuns e corrigidos

Mais comum			
refatoração	**qtd**	***code smells***	**qtd**
Extract Method	30.070	LongMethod	1.146.101
Move Attribute	26.665	DeadCode	634.791
Move Class	26.011	FeatureEnvy	425.479
Move Method	24.570	DataClass	422.721
Extract Variable	12.645	StateChecking	315.609
Pull Up Method	11.981	NestedBlockDepth	301.261
Extract And Move Method	9.973	LargeClass	294.961
Rename Class	8.396	ComplexMethod	209.141
Inline Method	8.026	LongParameterList	156.783
Pull Up Attribute	7.850	ClassDataShouldBePrivate	123.615
Correção			
refatoração	**qtd**	***code smells***	**qtd**
Extract Method	1.011	LongMethod	1.319
Extract Variable	758	FeatureEnvy	861
Inline Method	645	NestedBlockDepth	644
Extract And Move Method	503	ComplexCondition	551
Move Method	501	ComplexMethod	542
Pull Up Method	415	DeadCode	353
Pull Up Attribute	251	BrainMethod	317
Extract Class	137	StateChecking	267
Inline Variable	128	GodClass	124
Parameterize Variable	76	LargeClass	95

Foram coletadas 37 medidas de software que foram apresentadas na Tabela 3-5. As medidas foram coletadas para cada classe de cada versão existente no repositório. Apresentar uma tabela com a quantidade de

métricas que foram coletadas não faz sentido porque precisamos colocá-las dentro de uma contexto. Desta forma, apresentamos na Tabela 4-5 uma relação com as medidas de software OO dentro de três cenários.

O primeiro apresenta as medidas acerca da refatoração, trazendo cinco exemplos com as medidas antes e após a refatoração. O segundo cenário também apresenta cinco exemplos com as medidas de classes sem *code smell* e após sua introdução. Por fim, o terceiro cenário traz cinco exemplos de medidas de software em classes que tiveram uma refatoração que resultou na correção de um *code smell*. Para os três cenários, optamos por selecionar exemplos da refatoração e *code smell* mais comuns, ou seja, *Extract Method* e *Long Method* respectivamente.

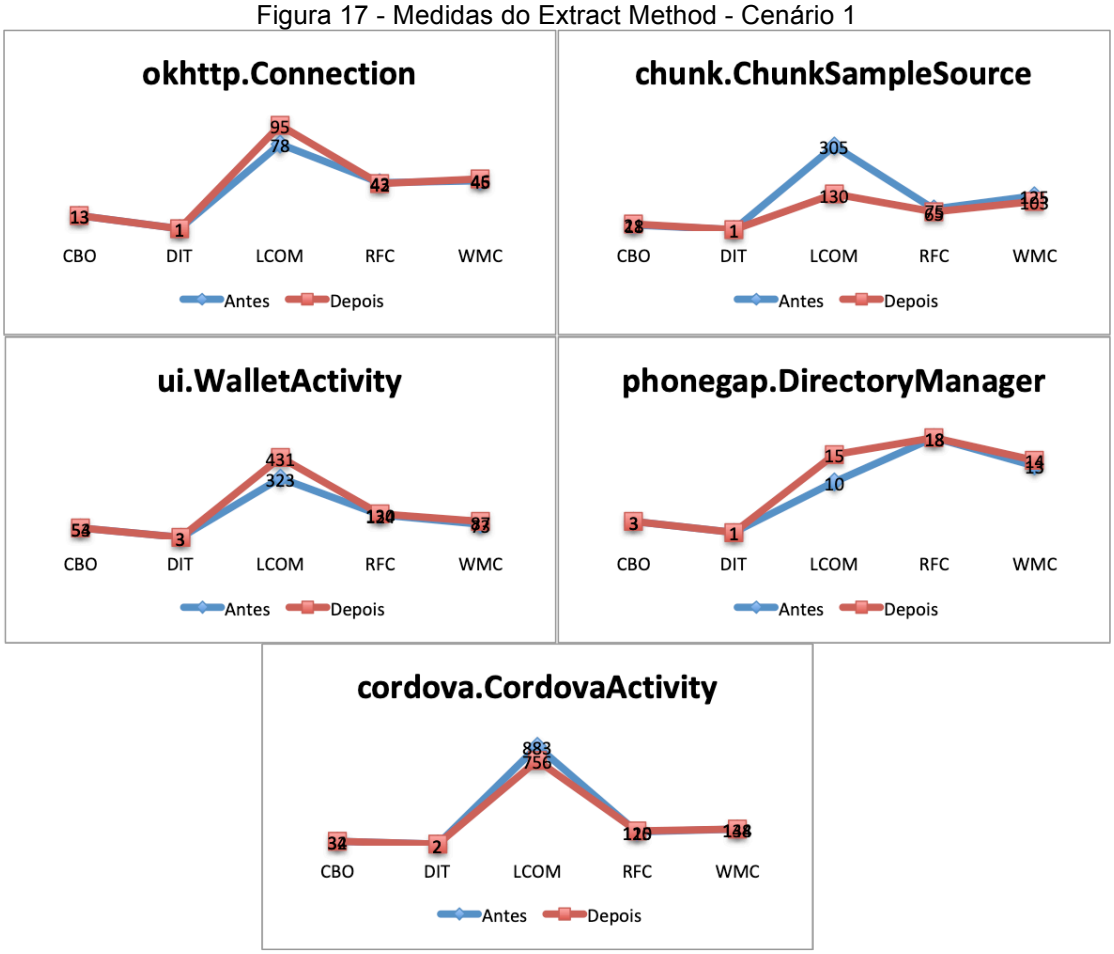
Tabela 4-5 - Medidas de Software para Extract Method e Long Method

Cenário 1: medidas nas refatorações – *Extract Method*													
Classe	**Método**	**Antes**						**Após**					
		CBO	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
com.squareup.okhttp.Connection	upgradeToTls → streamWrapper	13	1	78	-	43	45	13	1	95	-	42	46
com.squareup.okhttp.Connection	connect → streamWrapper	13	1	78	-	43	45	13	1	95	-	42	46
com.google.android.exoplayer.chunk.ChunkSampleSource	clearCurrentLoadable → clearCurrentLoadableException	18	1	305	-	75	125	21	1	130	-	65	103
de.schildbach.wallet.ui.WalletActivity	onOptionsItemSelected → handleRestoreWallet	53	3	323	-	124	73	54	3	431	-	130	87
com.phonegap.DirectoryManager	getFreeDiskSpace → freeSpaceCalculation	3	1	10	-	18	13	3	1	15	-	18	14
org.apache.cordova.CordovaActivity	onCreate → createViews	32	2	883	-	115	144	34	2	756	-	120	138
Cenário 2: medidas *code smell – Long Method*													
Classe	**Método**	**Antes**						**Após**					
		CBO	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
com.tundem.aboutlibraries.ui.LibsActivity	onCreate	10	2	1	-	32	12	11	2	1	-	32	13
de.danoeh.antennapod.storage.DownloadRequester	download	15	1	118	-	46	34	15	1	118	-	46	34
org.jivesoftware.smackx.muc.packet.MUCUser	toXML	7	1	93	-	15	21	7	1	93	-	15	21
cn.nekocode.camerafilter.CameraRenderer	run	30	2	22	-	31	23	36	2	22	-	31	23
com.omkarmoghe.pokemap.map.LocationManager	LocationManager	9	1	8	-	12	14	9	1	8	-	12	14
io.reactivex.android.schedulers.HandlerScheduler	schedule	6	3	0	-	3	5	8	3	0	-	5	6
Cenário 3: medidas de refatorações com correção de *code smell – Extract Method / Long Method*													
Classe	**Método**	**Antes**						**Após**					
		CBO	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
com.squareup.okhttp.Connection	connect upgradeToTls	13	1	78	-	43	45	13	1	95	-	42	46
com.alibaba.fastjson.parser.JSONLexer	scanISO8601DateIfMatch	9	1	218	-	61	845	9	1	0	-	65	869
com.bumptech.glide.ListPreloader	preload	9	2	33	-	14	20	11	2	38	-	15	25
jadx.core.utils.StringUtils	escapeResValue escapeResStrValue	2	1	26	-	8	63	2	1	53	-	11	56
me.ccrama.redditslide.Reddit	getSortingStrings	35	3	746	-	71	117	40	3	1177	-	80	114
org.thoughtcrime.securesms.jobs.RetrieveProfileJob	handleIndividualRecipient	22	3	20	-	31	18	23	3	7	-	39	25

* O cálculo do NOC (Number of Children) foi desativado na ferramenta em razão da quantidade de consumo de memória.
* O cenário três obtém as medidas a partir da versão que o *code smell* foi detectado e na versão em que uma refatoração ocorreu e o mesmo *code smell* não foi detectado.
* Neste trabalho foram identificados 737 casos de Long Method que desapareceram após uma refatoração Extract Method

A seguir apresentamos os dados demonstrados na Tabela 4-5 em formato de gráfico.

Figura 17 - Medidas do Extract Method - Cenário 1



A Figura 4-3 mostra o comportamento das medidas de software CBO, DIT, LCOM, RFC e WMC para as classes que sofreram a refatoração *Extract Method*. É possível observar que as medidas permanecem próximas ao valor original após passar por transformação, com poucas exceções.

Podemos observar que para o caso das classes *ChunkSampleSource* do projeto ExoPlayer e *WalletActivity* do projeto Bitcoin-Wallet, os valores da medida LCOM apresentaram comportamentos distintos. Para a classe *ChunkSampleSource*, o valor anterior à refatoração era aproximadamente 3x maior do que o valor atual. Analisando as refatorações e os *commits* da classe, podemos identificar que a classe sofreu 466 mudanças: 138 adições e 308 exclusões. Já para classe WalletActivity o LCOM aumentou aproximadamente 25% em relação à versão anterior, identificamos que a classe sofreu 75 mudanças: 67 adições e 8 exclusões.

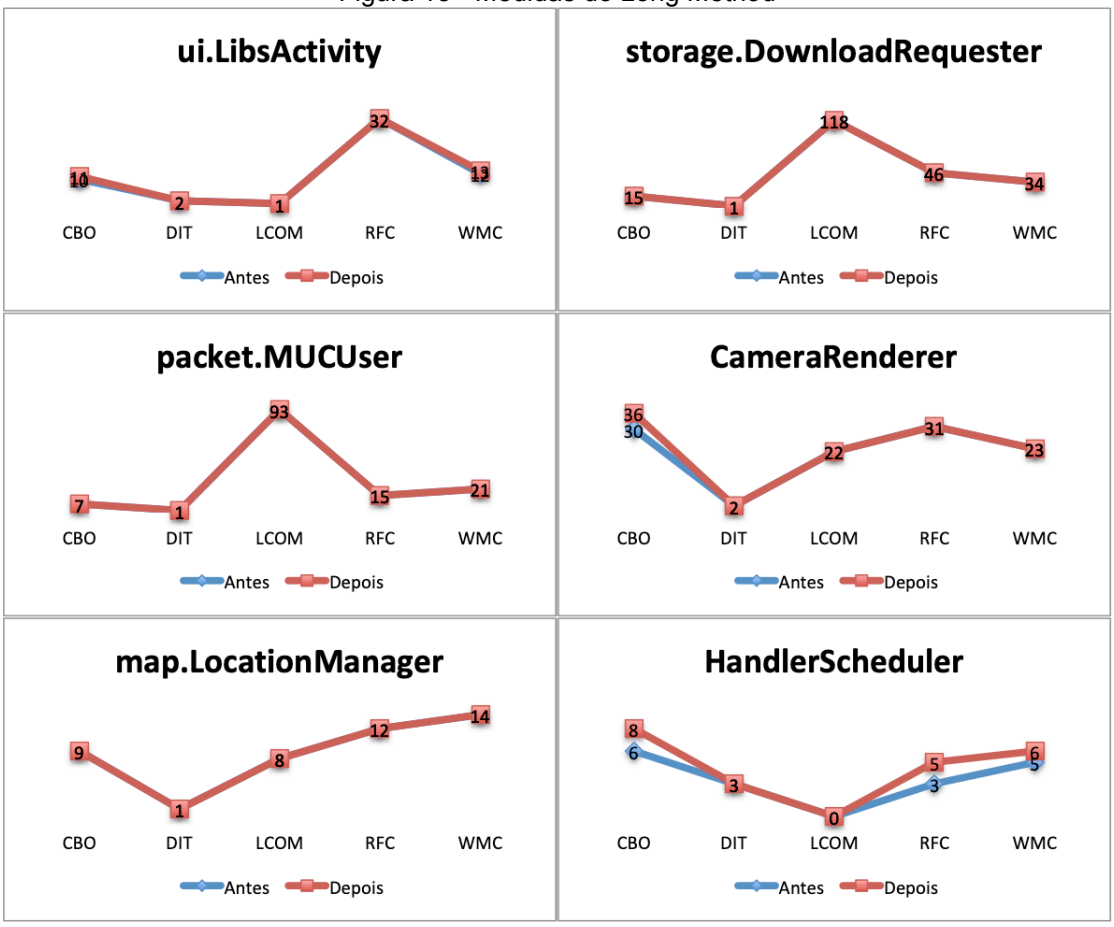
Recapitulando o que vimos acerca de LCOM ser uma medida de quão bem os métodos da classe cooperam para atingir os objetivos da classe, e um valor baixo de

LCOM sugere que a classe é mais coesa e é vista como “melhor”. A partir dos números apresentados, podemos interpretar para este caso o seguinte:

(I) Após uma intervenção profunda na classe *ChunkSampleSource* (466), a cooperação entre os métodos melhorou e sugere que a classe é mais coesa após o conjunto de mudanças e a refatoração sofrida;

(II) O LCOM da classe WalletActivity indica que está menos coesa em razão da inclusão de diversos métodos e na baixa colaboração entre seus métodos. Olhando para o *commit* específico da refatoração, observamos que a transformação proporcionada pelo *Extract Method* é pontual, afetando apenas duas linhas. Avaliando o aspecto de qualidade, um trecho de código foi extraído e foi reutilizado, é inegável que temos um aprimoramento daquele trecho do código. Contudo, a medida de software avalia o aspecto da cooperação dos métodos da classe.

Figura 18 - Medidas do Long Method



A Figura 4-4 mostra como as medidas se comportaram com relação à presença do *code smell Long Method*. É possível observar que a maioria das medidas mantém

seus valores após a detecção do *Long Method*. Observamos que nas classes CameraRenderer e HandlerScheduler a medida CBO aumentou. Na classe HandlerScheduler a medida RFC também aumentou. Com relação ao CBO podemos interpretar com base no conceito definido por Chidamber & Kemerer (CHIDAMBER; KEMERER, 1991): CBO é uma contagem do número de classes acopladas a uma classe específica (neste caso CameraRenderer e HandlerScheduler).

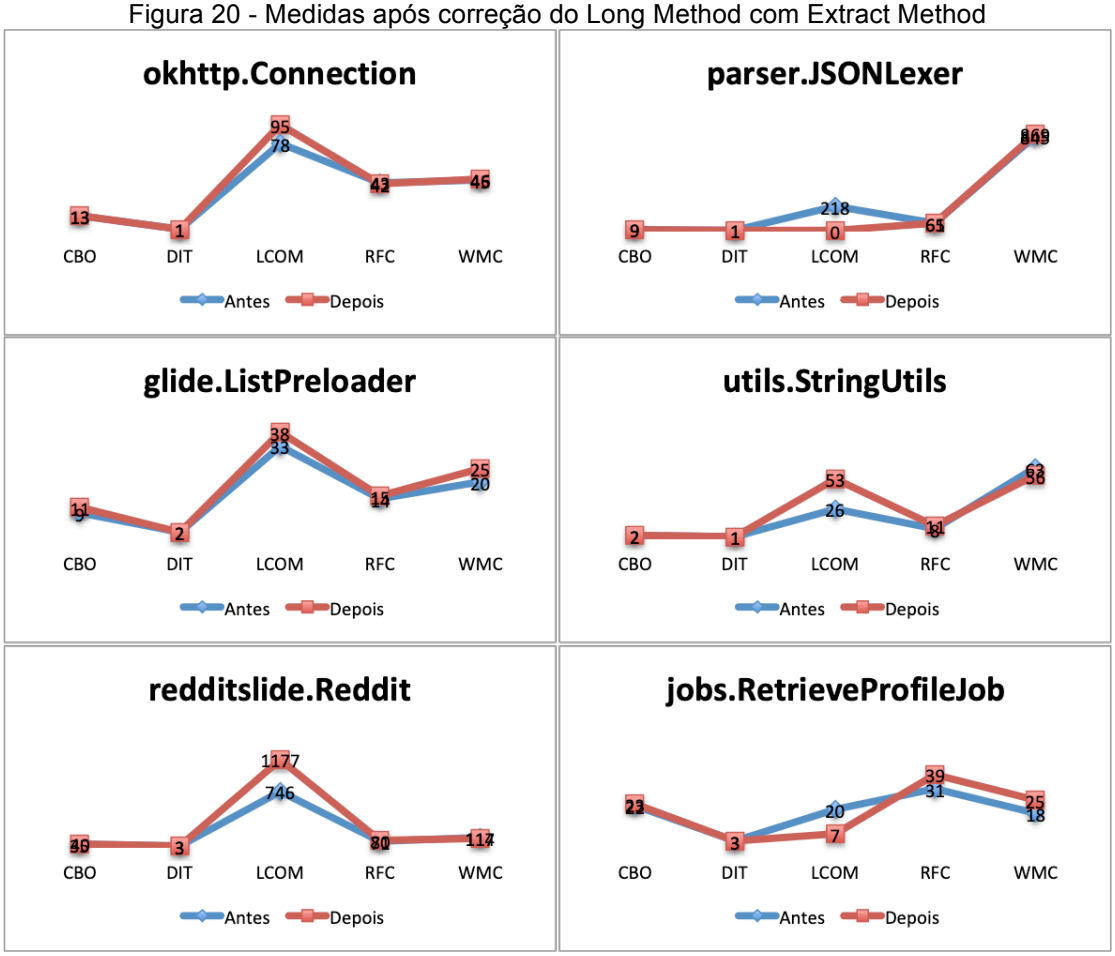
Analisamos os respectivos commits e observamos que algumas linhas foram adicionadas aos respectivos métodos (indicados na Tabela 4-5), ativando o *threshold* de identificação do *code smell*. A alteração no CBO justifica-se em razão das novas linhas adicionarem novas referências a classes que ainda não estavam acopladas. Por exemplo: no método CameraRenderer.run() foram adicionadas as linhas destacadas na Figura 4-5. De fato, referências a CrackedFilter e PolygonizationFilter foram adicionadas, aumentando o acomplamento de CameraRenderer e justificando o aumento na medida CBO.

Figura 19 - Long Method em CameraRenderer.run()

```
    public void run() {
        initGL(surfaceTexture);

        // Create texture for camera preview
        cameraTextureId = MyGLUtils.createCameraTextureID();
        cameraSurfaceTexture = new SurfaceTexture(cameraTextureId);

        cameraFilterMap.append(R.id.filter0, new OriginalFilter(context));
        cameraFilterMap.append(R.id.filter1, new EdgeDetectionFilter(context));
        cameraFilterMap.append(R.id.filter2, new PixelizeFilter(context));
        cameraFilterMap.append(R.id.filter3, new EMInterferenceFilter(context));
        cameraFilterMap.append(R.id.filter4, new TrianglesMosaicFilter(context));
        cameraFilterMap.append(R.id.filter5, new LegofiedFilter(context));
        cameraFilterMap.append(R.id.filter6, new TileMosaicFilter(context));
        cameraFilterMap.append(R.id.filter7, new BlueorangeFilter(context));
        cameraFilterMap.append(R.id.filter8, new ChromaticAberrationFilter(context));
        cameraFilterMap.append(R.id.filter9, new BasicDeformFilter(context));
        cameraFilterMap.append(R.id.filter10, new ContrastFilter(context));
        cameraFilterMap.append(R.id.filter11, new NoiseWarpFilter(context));
        cameraFilterMap.append(R.id.filter12, new RefractionFilter(context));
        cameraFilterMap.append(R.id.filter13, new MappingFilter(context));
        cameraFilterMap.append(R.id.filter14, new CrosshatchFilter(context));
        cameraFilterMap.append(R.id.filter15, new LichtensteinEsqueFilter(context));
        cameraFilterMap.append(R.id.filter16, new AsciiArtFilter(context));
        cameraFilterMap.append(R.id.filter17, new MoneyFilter(context));
+       cameraFilterMap.append(R.id.filter18, new CrackedFilter(context));
+       cameraFilterMap.append(R.id.filter19, new PolygonizationFilter(context));
        cameraFilter = cameraFilterMap.get(R.id.filter0);

        try {
```

Fonte: https://github.com/nekocode/CameraFilter

Figura 20 - Medidas após correção do Long Method com Extract Method



Com relação às medidas de software para as classes que tiveram a correção de Long Method associados à refatoração *Extract Method*, podemos observar na Figura 4-6 comportamentos distintos entre as medidas:

CBO: permanceu inalterado em três das seis classes, as outras três tiveram aumento na medida. Isto nos leva a pensar que o fato de um método longo da classe ter sido refatorado, não implica que a classe estará menos acoplada, ou seja, que existiria uma redução do CBO, pelo menos não é o que os dados nos apresentam para a refatoração e code smell analisados;

DIT: permance inalterado, algo que é o esperado para o tipo de transformação que o Extract Method emprega;

LCOM: O esperado é que a medida tenha uma redução, uma vez que um novo método é criado e uma chamada é adicionada ao método original, aumentando a colaboração entre os métodos da classe. Contudo, como já evidenciamos na análise dos dados da Figura 4-3,

as classes passam por adições e exclusões de código entre suas versões, o que na prática, mascara o valor real da medida LCOM;

RFC: nesta medida a expectativa era de aumento como consequência direta da criação de um novo método. Conforme podemos observar, a medida cresceu em todas classes analisadas. Em alguns casos tivemos um crescimento acentuado da medida, isto pode ser explicado pelos motivos já expostos anteriormente, cada classe pode sofrer diversas mudanças entre as versões;

WMC: baseada na complexidade definida por McCabe (MCCABE, 1976), contabiliza a quantidade de caminhos que uma instrução pode tomar dentro da classe. Conforme visto na Tabela 4-5, a medida WMC aumentou em 4 casos e diminuiu em 2. Relacionamos este comportamento às mudanças sofridas pela classe que não podem ser associadas de maneira exclusiva à refatoração.

Por fim, a Tabela 4-6 traz um complemento à Tabela 4-4 com um ranking de *code smells* que foram identificados pela ferramenta RefLink com a listagem daqueles que mais sumiram quando relacionados a uma transformação realizada por uma refatoração do código.

Tabela 14 - Ranking de code smells

Rank	Refatoração	Code Smells	Ocorrências
1º	Extract Method	LongMethod	737
2º	Extract Variable	LongMethod	641
3º	Inline Method	LongMethod	627
4º	Move Method	LongMethod	510
5º	Extract Variable	FeatureEnvy	432
6º	Extract Method	ComplexCondition	424
7º	Inline Method	ComplexCondition	400
8º	Extract And Move Method	LongMethod	387
9º	Extract Variable	ComplexCondition	372
10º	Move Method	ComplexCondition	331
11º	Extract Method	NestedBlockDepth	324
12º	Extract Variable	NestedBlockDepth	298
13º	Inline Method	NestedBlockDepth	290
14º	Extract Method	FeatureEnvy	288
15º	Extract And Move Method	FeatureEnvy	276
16º	Move Method	NestedBlockDepth	265
17º	Extract Method	ComplexMethod	250
18º	Extract Variable	ComplexMethod	234
19º	Inline Method	ComplexMethod	227
20º	Inline Method	FeatureEnvy	224

Com base nos dados coletados, podemos então afirmar que é possível estabelecer uma relação entre code smells e refatorações. Conseguimos inclusive

indicar qual o conjunto de refatorações afetou um determinado *code smell*, apontando o *commit* responsável pela alteração, versão do software e os tipos de transformações aplicadas a classe.

Contudo, não foi possível determinar se as refatorações que eliminam *code smells* causam reflexos nas medidas de software. Conforme explanado nas análises das Figuras 4-3, Figura 4-4 e Figura 4-6, as classes sofrem diversas mudanças durante o desenvolvimento de uma versão, dificultando estabelecer se a hipótese é válida ou se os dados dão indícios para refutá-la. Sendo assim, não conseguimos avaliar a hipótese levantada neste trabalho. Ainda podemos destacar que nossa análise concentrou-se em um escopo que avaliou a refatoração e o *code smell* com maiores ocorrências, fazendo um corte menor de todo o conjunto de dados que construímos durante o desenvolvimento desta dissertação. Analisar todo o *dataset* poderá dar subsídios suficientes para validar ou refutar nossa hipótese no futuro.

4.4 AMEAÇAS À VALIDADE

As principais ameaças relacionadas à validade deste trabalho estão listadas com as respectivas ações que foram adotadas para mitigar os riscos.

- **Definição dos critérios de seleção**

  Com relação a definição dos critérios de seleção das ferramentas, buscamos utilizar critérios claros e bem definidos, baseados em outros trabalhos da literatura. Por exemplo, ao definir os critérios para seleção do repositório de software, adotamos algumas premissas alinhadas com este trabalho para direcionar o processo de seleção. A mesma lógica foi utilizada para a seleção das demais ferramentas e soluções.

- **Precisão das ferramentas**

  Outro ponto de alerta e atenção deste trabalho com relação às ferramentas foi no âmbito da precisão dos resultados e saídas obtidas, das ferramentas que estavam sendo incorporadas ao RefLink. Desta forma, todas as ferramentas utilizadas passaram por uma verificação manual de sua acurácia com relação à identificação das refatorações e da detecção de *code smells*. O procedimento para verificação passava desde a obtenção do *commit* ou versão do projeto, verificação dos *merge commits* e comparação de conteúdo através de ferramenta de diff. Também foram levadas em considerações as menções de outros trabalhos científicos que já utilizaram e publicaram seus respectivos resultados.

- **Representatividade da amostra**

  Utilizamos uma listagem com 1000 projetos voltados para Android na linguagem Java. Inicialmente tínhamos a pretensão de utilizar aproximadamente 50 projetos, porque era o número que ficava dentro da média dos trabalhos encontrados na mesma linha. Contudo, resolvemos expandir a amostra para garantir alguma representatividade estatística e para aferir o funcionamento da ferramenta RefLink sob diversas circunstancias, tais como: tamanho de projeto, tamnho de equipe, finalidade e quantidade de versões. Isto permitiu que várias correções e ajustes no código responsável pelo fluxo de execução fossem realizados.

- **Validade das medidas de software**

  Uma ameaça a validade deste trabalho são as medidas obtidas a partir das métricas utilizadas neste trabalho. Uma classe evolui dentro de um projeto de software, logo, é difícil aferir métricas de um determinado recurso sem que o mesmo possa ter sido alterado. Por exemplo: uma classe que teve uma *Long Method* detectado na versão 2.0.0, e este mesmo *code smell* desapareceu na versão 3.5.1 em razão de um *Extract Method*, não podemos inferir que as medidas obtidas na versão 2.0.0 permaneceram imutáveis durante as versões subsequentes e que nenhuma alteração foi realizada.

  Além do exposto, uma mesma classe pode conter inúmeros casos de *code smells*, com inúmeras intervenções evolutivas ou corretivas, e suas medidas permanecerem intactas, sem nenhum reflexo direto. Em razão deste cenário, uma das ações foi construir um *dataset* com um conjunto significativo de medidas (37 ao todo) para permitir que no futuro estes dados possam ser correlacionadas e analisados sob outras perspectivas.

5 CONSIDERAÇÕES FINAIS

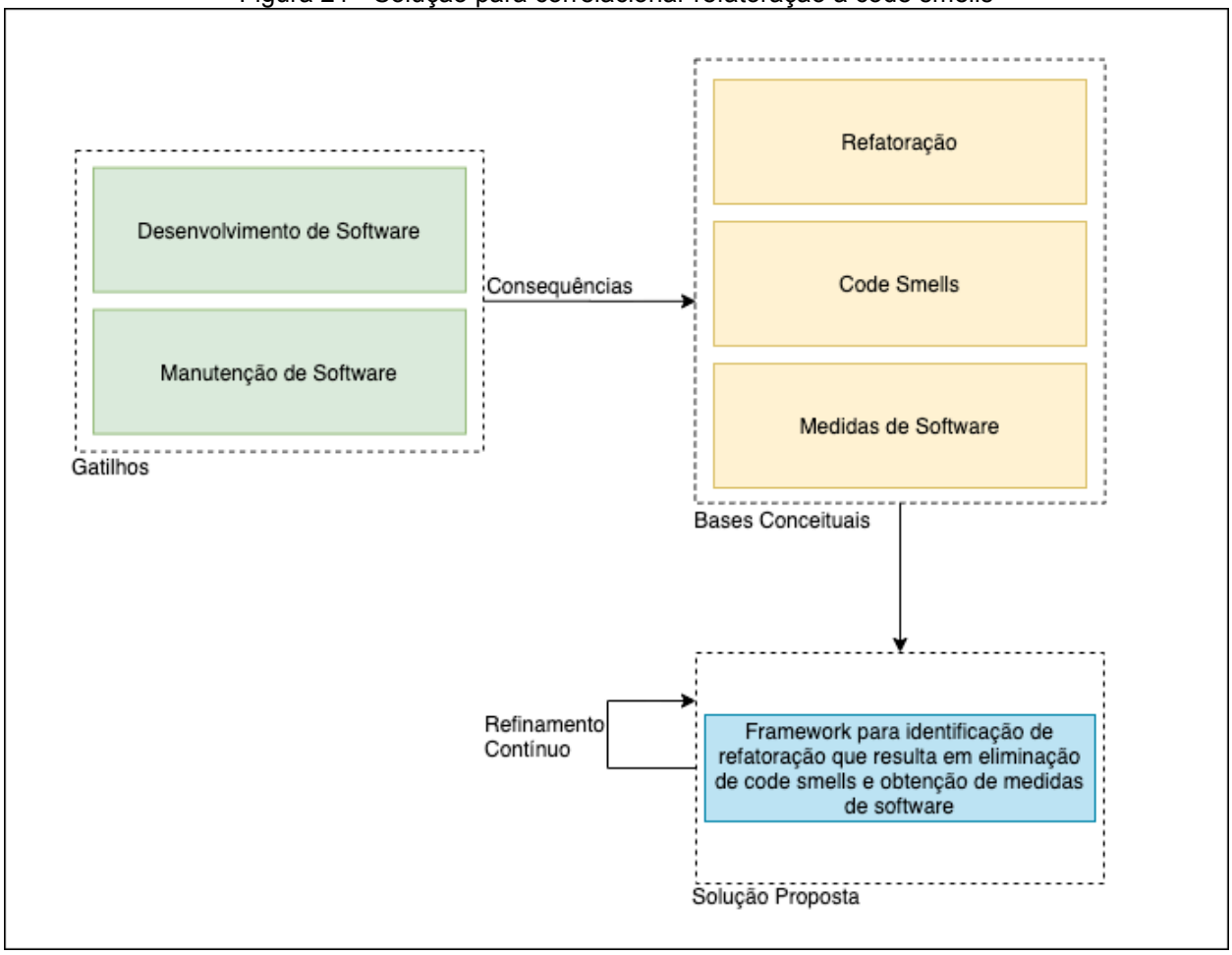
Neste capítulo apresentamos nossas considerações finais, as principais contribuições desta dissertação, as limitações do trabalho e as possibilidades de evolução em trabalhos futuros a partir desta dissertação.

A ideia central desse trabalho consistiu em investigar a manutenção de sistemas em repositórios públicos de software, buscando sinalizar se são efetivas ou não sob a perspectiva de medidas de software. Alguns processamentos foram realizados nos repositórios públicos de software como forma de avaliar a qualidade dos encaminhamentos sugeridos. Com foco na manutenção de software, duas linhas correlacionadas foram concebidas: *code smells* e refatoração, uma vez que refatorações transformam o código, resolvendo o *code smells* (FOWLER; BECK, 1999). Como consequência desta correlação e fator de validação da qualidade do código, foi norteado por medidas de software.

Investigando essa duas questões, alinhadas a medidas de software, seria possível elaborar um solução que indicasse se a refatoração, verdadeiramente, provocou a melhoria no código.

Porém, outros temas tangentes acabaram sendo abordados no trabalho: análise estática de código, mineração de repositórios, árvores sintáticas (AST) e ferramentas de visualização de código, o que levou à construção de uma ferramenta para realizar a atividade de mineração de repositórios públicos de software, com o objetivo de correlacionar a refatoração à correção de *code smells*.

Figura 21 - Solução para correlacionar refatoração a code smells



Após a seleção de um conjunto de ferramentas necessárias para o desenvolvimento da pesquisa, avançamos na definição de um fluxo de trabalho para correlacionar refatorações e *code smells*. Através do cruzamento dos dados minerados, estabelecemos relações a partir da presença de refatorações em classes que deixaram de possuir um determinado *code smell*, sempre verificando a identidade através de algoritmo de similaridade de Levenshtein.

Antes da construção da ferramenta automatizada, avaliamos a integridade das soluções escolhidas (*RefatoringMiner, Jquality/SmartSmell, CK Tool*) acerca de sua precisão, afinal era um pré-requisito básico para algumas das capacidades do core da ferramenta RefactoringLink, ou seja, inspecionar repositórios de software, identificar refatorações e *code smells*, buscando correlaciona-los durante toda a vida do software.

Apresentamos o método e fluxo de processamento que foram utilizados na construção do RefLink. Estes artefatos podem ser utilizados como base na construção de novas ferramentas, através do detalhamento do fluxo de execução e do código fonte, é possível estabelecer uma base promissora para novas ferramentas ou evolução das funcionalidades do RefLink. Por fim, esta dissertação ainda construiu um

modelo de dados estruturados com um *dataset* com centenas de projetos de repositórios de software público. Onde conseguimos indicar qual o conjunto de refatorações afetou um determinado code smell, apontando o commit responsável pela alteração, versão do software e os tipos de transformações aplicadas a classe. Contudo, não foi possível determinar se as refatorações que eliminam code smells causam reflexos nas medidas de software.

Consideramos que esta dissertação apresenta duas contribuições principais: (I) A ferramenta RefactoringLink – RefLink, capaz de minerar repositórios de software hospedados em servidores de controle de versão que utilizam GIT como ferramenta, inspecionando todo o processo evolutivo do software, versão por versão, classe por classe, construindo um catalogo de identificação de refatorações, *code smells* e medidas de software; (II) Conjunto de dados (*dataset)* com 1.000 projetos de software na linguagem Java para a plataforma Android, dos quais 427 foram válidos, foram computadas 197.525 refatorações, detectamos 4.738.238 *code smells* e coletamos 75.257.799 de medidas de software. Onde cada projeto teve suas versões, classes e *commits* esmiuçados a fundo, disponibilizados em um formato estruturado e passível de utilização em outras pesquisas cientificas.

5.1 TRABALHOS RELACIONADOS

Esta Seção descreve trabalhos relacionados encontrados na literatura durante a pesquisa desta dissertaçãoo.

5.1.1 Refatoração e repositórios de software

Com relação a refatoração e repositórios de software, Soares et al. (SOARES et al., 2011) faz uma análise dos aspectos de refatoração em repositórios de software, propondo uma técnica automatizada para entender a frequência, granularidade e escopo das refatorações nos repositórios de software. Soares et al. (SOARES et al., 2011) analisou cinco projetos Java de código aberto, processando 40.723 versões e um total de 80 desenvolvedores envolvidos. A técnica apresentada é focada em entender os seguintes fatores:

- Frequência de refatoração: objetivo é medir quão frequente as refatorações são aplicadas durante o ciclo de vida do projeto;
- Granularidade: analisa o impacto da refatoração na estrutura do software, avaliando se as transformações são internas a um método ou se espalham por vários métodos e classes;

- Escopo: avalia se a refatoração afeta um único pacote ou vários.

Neste sentido, (SOARES et al., 2011) utiliza o repositório do código fonte de um projeto como entrada para obter o total de refatorações aplicadas durante o ciclo de vida. Para isso, a técnica proposta é divida nas seguintes etapas:

- Etapa 1: analisa um par de versões e as classifica em duas categorias: refatoradas ou não refatoradas. Uma particularidade desta etapa é que a classificação se baseia em as transformações não terem afetado o comportamento. Para esta verificação é utilizado a ferramenta SafeRefactor que indica se um comportamento foi ou não mantido entre dois códigos.
- Etapa 2: verifica se as refatorações encontradas impactam na estrutura do software, verificando o fator de granularidade.
- Etapa 3: verifica a questão de escopo da mesma forma que na Etapa 2.

Também é importante destacar outra característica do trabalho de (SOARES et al., 2011). Foram utilizadas duas ferramentas de controle de versão, SVN e CVS. O SVN foi utilizado em quatro projetos e o CVS em apenas um. O conceito de versão para o SVN foi definido como um simples *commit*, ou seja, uma versão para este trabalho não está associado ao conceito de release. Para o CVS, uma versão foi definida como um conjunto de arquivos alvo de *commit* pelo mesmo autor, com a mesma mensagem, em um intervalo de até um minuto.

Neste cenário, o trabalho avaliou aproximadamente 41.000 versões e aproximadamente 72.73% das transformações encontradas, alteravam o comportamento, ou seja, não foram classificadas como refatorações. Percebe-se que o conceito de transformação utilizado por (SOARES et al., 2011) não está ligado a um catálogo ou alguma definição de tipo de refatorações. Sua abordagem assume que qualquer alteração no código que preserve o comportamento é uma refatoração.

Com relação ao nosso trabalho, podemos destacar algumas distinções. Investigamos a relação entre *code smells* e refatorações e qual seu reflexo nas medidas de software, realizando mineração automática de repositórios, classificando as refatorações de acordo com catálogo estabelecido por Fowler e colentado medidas de software para todas as classes de todas as versões do projeto, concebendo um *dataset* com mais de 1000 projetos. Também utilizamos repositórios GIT e estabelecemos um fluxo de processamento que permite que o RefLink seja expandido. Já Soares et al. limitou seu trabalho em tratar refatoração como um conceito mais amplo, sem tipificar

as transformações e tratando os *commits* como versões dos sistemas. A nossa abordagem estabeleceu que uma *tag (release)* é um marco de uma versão, estabelecendo uma referência próxima ao mundo real de como os softwares são controlados.

5.1.2 Code Smells e Medidas de Software

No trabalho de (BAVOTA et al., 2015) é abordada a relação entre medidas de software e a presença de *code smells*. É apresentado um estudo empírico onde foram utilizadas ferramentas para detectar refatorações, code smells e coletar métricas de software. O foco do trabalho foi em responder dois questionamentos: (I) as refatorações são realizadas nas classes com algum indicativo com base em métricas de qualidade de software? e (II) As refatorações são aplicadas em classes que possuem code smells?

(BAVOTA et al., 2015) investigou a evolução de três projetos Java de código aberto com o intuito de analisar se as refatorações ocorreram no código com algum indicativo, alguma medida de software. Ou entender o inverso, se as medidas de qualidade de software sugerem a presença de code smells ou a necessidade de refatoração.

No trabalho de (BAVOTA et al., 2015) foi utilizado a ferramenta Ref-Finder para detectar as operações de refatoração. Para a detecção de *code smell,* o autor só menciona que foi desenvolvida uma ferramenta simples para exibir uma relação de possíveis classes candidatas a *code smells.* Com base nessa relação, um estudante de mestrado e outro de doutorado fizeram a análise e classificação individual para determinar se as classes candidatas continham *code smells*. Para coletar as medidas de software foi utilizado o Eclipse JDT API para extrair todas as informações necessárias para coletar as medidas.

Como resultado obtido, o trabalho de (BAVOTA et al., 2015) indica que as medidas de software não demonstram claramente uma relação com refatoração. O trabalho analisou um total de 12.922 operações de refatorações, onde 5425 foram executadas em classes com *code smells (42%)*. Contudo, apenas 933 (7%) refatorações removeram o *code smell* da respectiva classe que foi afetada. Destes, 895 estão atribuídos a quatro *code smells*, são eles: Large Class, Long Method, Spaghetti Code e Feature Envy.

Com relação ao nosso trabalho podemos elencar de forma comparativa as seguintes distinções:

- Nossa pergunta de pesquisa é se é possível estabelecer uma relação entre code smells e refatorações, observando as medidas de software. Já o Bavota et al., busca entender se o agente motivador das refatorações teve como base os indicadores de métricas de qualidade.
- Com relação ao tamanho da amostragem, Bavota et al. analisou um cenário com três projetos. Processamos 1000 projetos e avaliamos 427. Ainda sobre a amostragem, avaliamos 11.593 versões, construindo um conjunto de dados relacionados entre projetos, versões, classes, refatorações, *code smells* e medidas de software.
- A solução utilizada por Bavota et al. contempla a utilização do Ref-Finder, Eclipse JDT API e um detector de possíveis classes candidatas a terem *code smells*. A solução desta dissertação apresentou um framework para construção de soluções como o RefLink (Refactoring Link Tool). Utilizamos uma composição de ferramentas selecionadas com base em critérios bem definidos e alinhados com a literatura, principalmente o preconizado por Fowler (FOWLER; BECK, 1999). Desenvolvemos o RefLink e incorporamos o JQuality, Refactoring Miner e CK Tools. Demonstramos como é o fluxo de processamento do RefLink e onde as ferramentas atuam, deixando claro onde é possível extender para outras soluções.
- Bavota et al. apresenta que 7% das refatorações que foram aplicadas removeram o *code smell* da classe. Com a nossa amostragem, encontramos que apenas 2.3% das refatorações pode ser associadas a remoção de code smell de forma efetiva. Ainda avançamos na análise e em nosso *dataset,* o conjunto de refatorações aplicadas a uma classe é relacionado ao respectivo *code smell* que foi removido. Realizamos uma verificação para garantir que o *code smell* resolvido de fato é o mesmo que existia na classe em versões anteriores.
- Por fim, os *code smells* mais representativos nos achados de Bavota et al. também aparecem em nosso top 10.

5.2 LIMITAÇÕES

Este trabalho investigou ferramentas de detecção de *code smells*, identificação de refatorações e coleta de métricas de software, contudo, a dissertação possui algumas limitações conhecidas. O método proposto para construção da ferramenta

automatizada foi apenas um passo inicial na direção da criação de suíte robusta que poderá ajudar no entendimento de quais reflexos os *code smells* e refatorações causam no software. Para avançarmos na evolução deste trabalho, podemos citar a necessidade de exploração de outros aspectos que impuseram as seguintes limitações:

- Investigação mais aprofundada acerca de outras ferramentas de detecção de *code smells*, permitindo a composição de uma solução que permita uma análise heurística do código fonte que está sendo analisado. Este trabalho enfrentou diversas dificuldades em obter soluções de detecção que fossem funcionais e com precisão nos resultados.
- Inclusão de uma ferramenta de refatoração que execute transformações no código de forma automática, permitindo incorporar no RefLink a capacidade de simular ou efetuar a correção de *code smells*. Com o RefLink seria possível avaliar se as transformações aplicadas, principalmente as propostas por Fowler (FOWLER; BECK, 1999).
- Exploração das métricas de software de maneira que se estabeleça uma correlação com as refatorações ou *code smells*.
- Expansão do *dataset* para um conjunto de repositórios maior.
- Aperfeiçoamento do *dataset* para regionalização, práticas de desenvolvimento utilizadas, time de desenvolvedores e outros fatores que podem determinar as características dos achados acerca dos *code smells* e da prática de refatoração.

5.3 TRABALHOS FUTUROS

Esta dissertação pode ser complementada e ganhar novas contribuições em futuros trabalhos de pesquisas nos seguintes aspectos:

- Atualizar o fluxo de processamento de dados para criar um gatilho quando determinadas situações forem encontradas. Por exemplo: uma classe que sofreu a refatoração de *Extract Method* e não sofreu outras intervenções no código, quais forem as mudanças nas medidas de software. A partir desta característica, realizar uma busca reversa por outras classes ou casos de refatoração semelhantes. É uma ideia similar ao que faz uma ferramenta de *Business Intelligence (BI)* ou de *Online Analytical Processing (OLAP)*;
- Incluir uma ferramenta de visualização de dados, permitindo destacar características por projeto, versões, *code smells*, refatorações e medidas;
- Realizar estudos para inclusão de novas ferramentas de detecção;

- Ampliar o *dataset* para outras linguagens e repositórios;
- Avaliar repositórios públicos e privados, permitindo criar um paralelo se existe diferença significativa nos achadas entre softwares públicos e privados.
- Atuar nas limitações enfrentadas por este trabalho e reportadas na Seção anterior.

REFERÊNCIAS

BARROS, V. **Um método para a refatoração de software baseado em frameworks de domínio**. [s.l.] Universidade Tecnológica Federal do Paraná, 2015.

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

BECK, K. **Extreme Programming Explained: Embrace Change.** First edit ed. [s.l.] Addison-Wesley, 1999.

BIEGEL, B. et al. Comparison of similarity metrics for refactoring detection. **Proceedings - International Conference on Software Engineering**, n. i, p. 53–62, 2011.

BOSCH, A. **Priorisierung von Code Smells**. [s.l.] University Bremen, 2018.

BROWN, W. J. et al. **AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis**. New York, NY, USA: John Wiley & Sons, 1998. v. 3

BUNGE, M. **Treatise on Basic Philosophy - Onthology I: The Furniture of the World**. Dordrecht, Holland: D. Reidel, 1977. v. 3

BUNGE, M. **Treatise on Basic Philosophy - Ontology II: A World of Systems**. Dordrecht, Holland: Springer Netherlands, 1979. v. 66

CARNEIRO, G. **Usando medição de código fonte para refactoring**. [s.l.] Universidade Salvador, 2003.

CEDRIM, D. et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**, v. Part F1301, p. 465–475, 2017.

CHIDAMBER, S. R.; KEMERER, C. F. Towards a metrics suite for object oriented design. **ACM SIGPLAN Notices**, v. 26, n. 11, p. 197–211, 1 nov. 1991.

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

CORNÉLIO, M.; CAVALCANTI, A.; SAMPAIO, A. Refactoring Towards a Layered Architecture. **Electronic Notes in Theoretical Computer Science**, v. 130, p. 281–300, 2005.

DIG, D. et al. Automated detection of refactorings in evolving components. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 4067 LNCS, p. 404–428, 2006a.

DIG, D. et al. **RefactoringCrawler**. Disponível em: <http://dig.cs.illinois.edu/tools/RefactoringCrawler/>. Acesso em: 2 dez. 2018b.

DIG, D.; JOHNSON, R. **The role of refactorings in API evolution**. 21st IEEE International Conference on Software Maintenance (ICSM'05). **Anais**...IEEE, 2005

EMDEN, E. VAN; MOONEN, L. Assuring software quality by code smell detection. **2002 9th Working Conference on Reverse Engineering**, p. 97–106, 2002.

ESTOLANO, M. **Base de Métricas para a Estação TABA**. [s.l.] Universidade Federal

do Rio de Janeiro, 2005.

FANTA, R.; RAJLICH, V. **Reengineering object-oriented code**. Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). **Anais**...IEEE Comput. Soc, 1998

FERNANDES, E. et al. **A review-based comparative study of bad smell detection tools**. Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16. **Anais**...New York, New York, USA: ACM Press, 2016

FONTANA, F. A. et al. **Code Smell Detection: Towards a Machine Learning-Based Approach**. 2013 IEEE International Conference on Software Maintenance. **Anais**...IEEE, set. 2013

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. **Journal of Object Technology**, v. 11, n. 2, p. 1–38, 2012.

FONTANA, F. A.; SPINELLI, S. Impact of refactoring on quality code evaluation. **Proceedings - International Conference on Software Engineering**, p. 37–40, 2011.

FOWLER, M. **CodeSmell**. Disponível em: <https://martinfowler.com/bliki/CodeSmell.html>. Acesso em: 1 jun. 2017.

FOWLER, M. **Catalog of Refactorings**. Disponível em: <https://www.refactoring.com/catalog/>. Acesso em: 1 jun. 2017.

FOWLER, M.; BECK, K. **Refactoring: Improving the Design of Existing Code**. 1ª ed. Boston, United States: Pearson Education, 1999.

HAMID, A. et al. A Comparative Study on Code Smell Detection Tools. **International Journal of Advanced Science and Technology**, v. 60, p. 25–32, 2013.

HEGEDŰS, P. et al. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. **Information and Software Technology**, v. 95, n. January 2017, p. 313–327, 2018.

HENDERSON; SELLERS. **Object-Oriented Metrics: measures of Complexity**. 1ª ed. Upper Saddle River, NJ, USA: Prentice Hall, 1996.

HITZ, M.; MONTAZERI, B. **Measuring Coupling and Cohesion in Object-Oriented Systems**. Proc. Int. Symposium on Applied Corporate Computing. **Anais**...1995

HITZ, M.; MONTAZERI, B. Chidamber and kemerer's metrics suite: A measurement theory perspective. **IEEE Transactions on Software Engineering**, v. 22, n. 4, p. 267–271, 1996.

KADAR, I. et al. A manually validated code refactoring dataset and its assessment regarding software maintainability. **ACM International Conference Proceeding Series**, p. 2–5, 2016.

KIM, M. et al. Ref-Finder: A refactoring reconstruction tool based on logic query templates. **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**, p. 371–372, 2010.

KIM, M.; CAI, D.; KIM, S. **An empirical investigation into the role of API-level refactorings during software evolution**. Proceeding of the 33rd international conference on Software engineering - ICSE '11. **Anais**...New York, New York, USA: ACM Press, 2011

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. [s.l.] Springer Science & Business Media, 2006.

LAVALLÉE, M.; ROBILLARD, P. N. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. **Proceedings - International Conference on Software Engineering**, v. 1, p. 677–687, 2015.

LEHMAN, M. M. Laws of software evolution revisited. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 1149, p. 108–124, 1996.

LEITNER, D. **A Model for Measuring Maintainability Based on Source Code Characteristics**. [s.l.] University of Applied Sciences Technikum Wien, 2017.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, v. SE-2, n. 4, p. 308–320, dez. 1976.

MENS, T.; TOURWE, T. A survey of software refactoring. **IEEE Transactions on Software Engineering**, v. 30, n. 2, p. 126–139, fev. 2004.

MISRA, S.; AKMAN, I. Applicability of weyuker's properties on OO metrics: Some misunderstandings. **Computer Science and Information Systems**, v. 5, n. 1, p. 17–24, 2008.

MOHA, N. et al. DECOR: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, v. 36, n. 1, p. 20–36, 2010.

MOHAN, M.; GREER, D.; MCMULLAN, P. Technical debt reduction using search based automated refactoring. **Journal of Systems and Software**, v. 120, p. 183–194, 2016.

NONGPONG, K.; BOYLAND, J. T. Integrating "Code Smells" Detection with Refactoring Tool Support. v. 3523928, n. August, p. 154, 2012.

OPDYKE, W. F.; B.S. **Refactoring Object-Oriented Frameworks**. [s.l.] University of Illinois at Urbana-Champaign, 1992.

PALOMBA, F. et al. **Lightweight detection of Android-specific code smells: The aDoctor project**. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). **Anais**...IEEE, fev. 2017

PINTO, G. **A Refactoring Approach to Improve Energy Consumption of Parallel Software Systems**. [s.l.] Universidade Federal de Pernambuco, 2015.

PMD Source Code Analyzer. Disponível em: <https://pmd.github.io/>. Acesso em: 20 jun. 2018.

PRETE, K. et al. **Template-based reconstruction of complex refactorings**. 2010 IEEE International Conference on Software Maintenance. **Anais**...Timisoara, Romania: IEEE, set. 2010

ROBERTS, D. B. **Practical analysis for refactoring**. [s.l.] University of Illinois at Urbana-Champaign, 1999.

ROPERIA, N. **JSMELL: A BAD SMELL DETECTION TOOL FOR JAVA SYSTEMS**. [s.l.] Maharishi Dayanand University, 2009.

SINGH, G.; CHOPRA, V. Design and Implementation of Testing Tool for Code Smell Rectification Using C-Mean Algorithm. **International Journal of Advanced Research**

in Computer Science, v. 04, n. 09, p. 108–114, 2013.

SOARES, G. et al. Analyzing refactorings on software repositories. **Proceedings - 25th Brazilian Symposium on Software Engineering, SBES 2011**, p. 164–173, 2011.

SOARES, G. et al. Comparing approaches to analyze refactoring activity on software repositories. **Journal of Systems and Software**, v. 86, n. 4, p. 1006–1022, 2013.

SOARES, G.; GHEYI, R.; MASSONI, T. Automated behavioral testing of refactoring engines. **IEEE Transactions on Software Engineering**, v. 39, n. 2, p. 147–162, 2013.

SOBRINHO, E. **Inter-relação Entre Smells — Uma Análise de Large Class , Complex Class e Clone**. [s.l.] Universidade Federal de Uberlândia, 2019.

SOMMERVILLE, I. **Engenharia de Software**. 9ª ed. [s.l.] PEARSON, 2011.

SOUZA, M. **Histrategy: uma técnica para a customização guiada de estratégias para a detecção de bad smells**. [s.l.] Universidade Federal de Campina Grande, 2017.

SZOKE, G. et al. FaultBuster: An automatic code smell refactoring toolset. **2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings**, p. 253–258, 2015.

TIAGO, A. G. **Fundamentação teórica das métricas de software**. [s.l.] UNICAMP - Universidade Estadual de Campinas, 2006.

TOURWE, T.; MENS, T. Identifying refactoring opportunities using logic meta programming. **Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR**, n. June 2014, p. 91–100, 2003.

TSANTALIS, N. **JDeodorant**. Disponível em: <https://github.com/tsantalis/JDeodorant>. Acesso em: 3 maio. 2018.

TSANTALIS, N. et al. **Accurate and efficient refactoring detection in commit history**. Proceedings of the 40th International Conference on Software Engineering - ICSE '18. **Anais**...New York, New York, USA: ACM Press, 2018

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. JDeodorant: Identification and removal of type-checking bad smells. **Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR**, p. 329–331, 2008.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Ten years of JDeodorant: Lessons learned from the hunt for smells. **25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings**, v. 2018- March, p. 4–14, 2018.

VALE, G. et al. Defining metric thresholds for software product lines: A comparative study. **ACM International Conference Proceeding Series**, v. 20-24- July, p. 176–185, 2015.

VAN EMDEN, E.; MOONEN, L. Java quality assurance by detecting code smells. **Proceedings - Working Conference on Reverse Engineering, WCRE**, v. 2002- Janua, p. 97–106, 2002.

WEISSGERBER, P.; DIEHL, S. Identifying refactorings from source-code changes. **Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006**, n. June, p. 231–240, 2006.

WEYUKER, E. Evaluating Software Complexity Metrics. **IEEE Transactions on Software Engineering**, v. 14, n. 9, p. 1357–1365, 1988.

WIELING, M.; NERBONNE, J. **Bipartite spectral graph partitioning to co-cluster**

varieties and soundcorrespondences in dialectology. Proceedings of the 2009 Workshop on Graph-based Methods for Natural Language Processing - TextGraphs-4. **Anais**...Morristown, NJ, USA: Association for Computational Linguistics, 2009

APÊNDICE A - CATÁLOGO DE CODE SMELLS DE FOWLER

Duplicated Code	Parallel Inheritance Hierarchies
Long Method	Lazy Class
Large Class	Speculative Generality
Long Parameter List	Temporary Field
Divergent Change	Message Chains
Shotgun Surgery	Middle Man
Feature Envy	Inappropriate Intimacy
Data Clumps	Alternative Classes with Different Interfaces
Primitive Obsession	Incomplete Library Class
Switch Statements	Data Class
Comments	Refused Bequest

APÊNDICE B - LISTAGEM DE PROJETOS SELECIONADOS DO GITHUB

square/retrofit	baoyongzhang/android-PullRefreshLayout
square/okhttp	gongwen/MarqueeViewLibrary
PhilJay/MPAndroidChart	huburt-Hu/NewbieGuide
bumptech/glide	elevenetc/TextSurface
airbnb/lottie-android	ApmeM/android-flowlayout
JakeWharton/butterknife	white-cat/ThinkAndroid
Blankj/AndroidUtilCode	novoda/android-demos
square/leakcanary	RomainPiel/Shimmer-android
zxing/zxing	seiginonakama/BlockCanaryEx
greenrobot/EventBus	XunMengWinter/CircularAnim
skylot/jadx	TakuSemba/Spotlight
ReactiveX/RxAndroid	team-supercharge/ShimmerLayout
alibaba/fastjson	alexvasilkov/GestureViews
scwang90/SmartRefreshLayout	igniterealtime/Smack
CymChad/BaseRecyclerViewAdapterHelper	yadav-rahul/TastyToast
square/picasso	pilgr/Paper
nostra13/Android-Universal-Image-Loader	Karumi/Rosie
facebook/fresco	hitherejoe/Android-Boilerplate
libgdx/libgdx	mthli/Knife
chrisbanes/PhotoView	mik3y/usb-serial-for-android
Tencent/tinker	nitaliano/react-native-mapbox-gl
android10/Android-CleanArchitecture	l123456789jy/Lazy
google/ExoPlayer	Yalantis/StarWars.Android
lgvalle/Material-Animations	nex3z/FlowLayout
hdodenhof/CircleImageView	Ramotion/cardslider-android
DrKLO/Telegram	JorgeCastilloPrz/AndroidFillableLoaders
signalapp/Signal-Android	Rajawali/Rajawali
jfeinstein10/SlidingMenu	timusus/Shuttle
greenrobot/greenDAO	gcacace/android-signaturepad
daimajia/AndroidSwipeLayout	Azoft/CarouselLayoutManager
orhanobut/logger	hidroh/materialistic
facebook/stetho	pakerfeldt/android-viewflow
androidannotations/androidannotations	matrixxun/ProductTour
daimajia/AndroidViewAnimations	janishar/PlaceHolderView
loopj/android-async-http	Muddz/StyleableToast
realm/realm-java	xmuSistone/AndroidPileLayout
mikepenz/MaterialDrawer	Yalantis/FlipViewPager.Draco
JakeWharton/ViewPagerIndicator	Ramotion/expanding-collection-android
Konloch/bytecode-viewer	florent37/ShapeOfView
liaohuqiu/android-Ultra-Pull-To-Refresh	HeZaiJin/SlantedTextView
Curzibn/Luban	firebase/firebase-jobdispatcher-android
Bigkoo/Android-PickerView	lyft/scissors
pockethub/PocketHub	Skykai521/ECTranslation
Tencent/VasSonic	wangdan/AisenWeiBo
CarGuo/GSYVideoPlayer	SufficientlySecure/html-textview
ksoichiro/Android-ObservableScrollView	mikepenz/LollipopShowcase
alibaba/vlayout	DreaminginCodeZH/MaterialProgressBar
alibaba/ARouter	wingjay/jianshi
zhihu/Matisse	LuckyJayce/LargeImage
wasabeef/recyclerview-animators	googlesamples/android-testing-templates
navasmdc/MaterialDesignLibrary	google/santa-tracker-android
chrisbanes/Android-PullToRefresh	harjot-oberai/MaterialShadows
vondear/RxTool	mmin18/RealtimeBlurView
permissions-dispatcher/PermissionsDispatcher	natario1/CameraView
lipangit/JiaoZiVideoPlayer	terrakok/Cicerone
81813780/AVLoadingIndicatorView	mathew-kurian/TextJustify-Android
Yalantis/uCrop	Luosunce/material-design-data
AppIntro/AppIntro	hongyangAndroid/android-percent-support-extend
umano/AndroidSlidingUpPanel	AigeStudio/DatePicker
YoKeyword/Fragmentation	federicoiosue/Omni-Notes
H07000223/FlycoTabLayout	plattysoft/Leonids
roughike/BottomBar	TakuSemba/MultiSnapRecyclerView
chrisjenx/Calligraphy	googlesamples/android-ConstraintLayoutExamples
tbruyelle/RxPermissions	AlexLiuSheng/CheckVersionLib
lingochamp/FileDownloader	microg/android_packages_apps_GmsCore
youth5201314/banner	rallat/EffectiveAndroid
florent37/MaterialViewPager	pbreault/adb-idea
HannahMitt/HomeMirror	artem-zinnatullin/qualitymatters
iBotPeaches/Apktool	CJT2325/CameraView
JessYanCoding/MVPArms	techGay/v9porn
Freelander/Android_Data	ankidroid/Anki-Android
chrisbanes/cheesesquare	Kaopiz/android-segmented-control

bilibili/DanmakuFlameMaster	card-io/card.io-Android-SDK
wasabeef/glide-transformations	iqiyi/Andromeda
Tencent/QMUI_Android	henrytao-me/smooth-app-bar-layout
google/agera	sephiroth74/ImageViewZoom
JakeWharton/ActionBarSherlock	jaredrummler/AnimatedSvgView
googlesamples/easypermissions	nekocode/CameraFilter
koral--/android-gif-drawable	bingoogolapple/BGASwipeBackLayout-Android
laobie/StatusBarUtil	elvishew/xLog
square/dagger	dingjikerbo/Android-BluetoothKit
cymcsg/UltimateRecyclerView	MatthiasRobbers/shortbread
Bearded-Hen/Android-Bootstrap	bitcoin-wallet/bitcoin-wallet
kaushikgopal/RxJava-Android-Samples	BeesX/BeesAndroid
alibaba/atlas	sqlcipher/android-database-sqlcipher
didi/VirtualAPK	Hitomis/transferee
GcsSloop/AndroidNote	Ramotion/garland-view-android
trello/RxLifecycle	RomainPiel/Titanic
geeeeeeeeek/WeChatLuckyMoney	medyo/android-about-page
aritraroy/UltimateAndroidReference	iPaulPro/aFileChooser
hongyangAndroid/AndroidAutoLayout	Tencent/VasDolly
material-components/material-components-android	tarek360/RichPath
facebook/buck	KingJA/LoadSir
pedant/sweet-alert-dialog	Kyson/AndroidGodEye
lecho/hellocharts-android	klinker24/talon-twitter-holo
daniulive/SmarterStreaming	friendlyrobotnyc/TinyDancer
JessYanCoding/AndroidAutoSize	android-notes/Cockroach
lzyzsd/JsBridge	yuliskov/SmartYouTubeTV
android-hacker/VirtualXposed	ZhaoKaiQiang/KLog
googlesamples/android-testing	airbnb/AirMapView
crazycodeboy/TakePhoto	tumblr/Backboard
DroidPluginTeam/DroidPlugin	eluleci/FlatUI
LitePalFramework/LitePal	maurycyw/StaggeredGridView
koush/AndroidAsync	czy1121/update
koush/ion	stfalcon-studio/FrescoImageViewer
ogaclejapan/SmartTabLayout	akexorcist/Android-RoundCornerProgressBar
didi/DoraemonKit	googlesamples/android-play-location
pedrovgs/EffectiveAndroidUI	andpor/react-native-sqlite-storage
cats-oss/android-gpuimage	anjlab/android-inapp-billing-v3
ybq/Android-SpinKit	athkalia/Just-Another-Android-App
futuresimple/android-floating-action-button	evollu/react-native-fcm
amitshekhariitbhu/Android-Debug-Database	UFreedom/FloatingView
LuckSiege/PictureSelector	zzz40500/android-shapeLoadingView
pxb1988/dex2jar	felipecsl/AsymmetricGridView
facebook/litho	Yalantis/Taurus
leolin310148/ShortcutBadger	avjinder/Minimal-Todo
rey5137/material	dexafree/MaterialList
nhaarman/ListViewAnimations	mqzhangw/JIMU
antoniolg/androidmvp	jokermonn/permissions4m
	mancj/MaterialSearchBar
ikew0ng/SwipeBackLayout	pwnall/chromeview
singwhatiwanna/dynamic-load-apk	Trinea/android-auto-scroll-view-pager
shwenzhang/AndResGuard	dinuscxj/RecyclerRefreshLayout
gyf-dev/ImmersionBar	wangchenyan/ponymusic
JakeWharton/u2020	inmite/android-selector-chapek
naman14/Timber	alexvasilkov/FoldableLayout
google/android-classyshark	redsolution/xabber-android
Qihoo360/RePlugin	delight-im/Android-AdvancedWebView
asLody/VirtualApp	medyo/Fancybuttons
TeamNewPipe/NewPipe	frogermcs/LikeAnimation
emilsjolander/StickyListHeaders	stephentuso/welcome-android
daimajia/NumberProgressBar	xiaoyanger0825/NiceVieoPlayer
Justson/AgentWeb	qdrzwd/VideoRecorder
daimajia/AndroidImageSlider	react-native-community/react-native-share
evant/gradle-retrolambda	osmdroid/osmdroid
wyouflf/xUtils3	GcsSloop/rclayout
airbnb/epoxy	kanytu/android-parallax-recyclerview
firebase/quickstart-android	brianwernick/ExoMedia
Nightonke/BoomMenu	zcweng/SwitchButton
Devlight/InfiniteCycleViewPager	liuguangqiang/SwipeBack
square/otto	ImmortalZ/StereoView
jgilfelt/SystemBarTint	square/assertj-android
dmytrodanylyk/circular-progress-button	Jungerr/GridPasswordView
yanzhenjie/AndPermission	JavaNoober/BackgroundLibrary
davemorrissey/subsampling-scale-image-view	dlazaro66/QRCodeReaderView
kickstarter/android-oss	kymjs/TheMVP
GrenderG/Toasty	parse-community/Parse-SDK-Android
sockeqwe/mosby	KeepSafe/ReLinker
	mancj/SlideUp-Android

evernote/android-job	nispok/snackbar
wyouflf/xUtils	harism/android-pagecurl
aa112901/remusic	JZXiang/TimePickerDialog
topjohnwu/Magisk	dmytrodanylyk/shadow-layout
prolificinteractive/material-calendarview	florent37/ExpansionPanel
smuyyh/BookReader	dersoncheng/MultipleTheme
JakeWharton/DiskLruCache	yydcdut/PhotoNoter
yixia/VitamioBundle	wasabeef/picasso-transformations
alibaba/freeline	boycy815/PinchImageView
markzhai/AndroidPerformanceMonitor	bingoogolapple/BGAPhotoPicker-Android
izzyleung/ZhihuDailyPurify	googlemaps/android-samples
Yalantis/Side-Menu.Android	cachapa/ExpandableLayout
Trinea/android-common	MindorksOpenSource/android-mvvm-architecture
MindorksOpenSource/android-interview-questions	omadahealth/LolliPin
ACRA/acra	stepstone-tech/android-material-stepper
commonsguy/cw-omnibus	jaredrummler/AndroidProcesses
traex/RippleEffect	yeriomin/YalpStore
gabrielemariotti/cardslib	Cutta/GifView
facebook/facebook-android-sdk	mayubao/Android-Pay
guardianproject/haven	hongyangAndroid/AndroidChangeSkin
etsy/AndroidStaggeredGrid	Q42/AndroidScrollingImageView
hongyangAndroid/FlowLayout	nkzawa/socket.io-android-chat
pardom-zz/ActiveAndroid	tangqi92/WaveLoadingView
wequick/Small	LyndonChin/AndroidRubberIndicator
Clans/FloatingActionButton	Zhaoss/WeiXinRecordedDemo
dm77/barcodescanner	yanzhenjie/AndServer
jwtk/jjwt	AAkira/ExpandableLayout
chentao0707/SimplifyReader	huxq17/XRefreshView
yarolegovich/DiscreteScrollView	apollographql/apollo-android
Devlight/NavigationTabBar	Arello-Mobile/Moxy
hanks-zyh/HTextView	burhanrashid52/PhotoEditor
barteksc/AndroidPdfViewer	heinrichreimer/material-intro
JakeWharton/NineOldAndroids	D-clock/AndroidDaemonService
huanghaibin-dev/CalendarView	gowong/material-sheet-fab
wasabeef/richeditor-android	Jasonette/JASONETTE-Android
danielzeller/Depth-LIB-Android-	Ramotion/circle-menu-android
mcxiaoke/android-volley	DreaminginCodeZH/MaterialRatingBar
robolectric/robolectric	florent37/GlidePalette
uber/RIBs	hehonghui/AndroidEventBus
h6ah4i/android-advancedrecyclerview	Qihoo360/ArgusAPM
orhanobut/dialogplus	yangchong211/LifeHelper
alibaba/UltraViewPager	schwabe/ics-openvpn
Ramotion/folding-cell-android	frogermcs/AndroidDevMetrics
gzu-liyujiang/AndroidPicker	majidgolshadi/Android-Download-Manager-Pro
ArthurHub/Android-Image-Cropper	sucese/android-interview-guide
square/android-times-square	JsonChao/Awesome-WanAndroid
aporter/coursera-android	android10/frodo
KeepSafe/TapTargetView	romainguy/road-trip
Meituan-Dianping/walle	mayubao/KuaiChuan
jdamcd/android-crop	Kelin-Hong/MVVMLight
k9mail/k-9	nisrulz/easydeviceinfo
zetbaitsu/Compressor	xiaoyaoyou1212/XSnow
AndroidBootstrap/android-bootstrap	commonsguy/cw-advandroid
google/cameraview	f2prateek/rx-preferences
jpush/aurora-imui	eschao/android-PageFlip
amitshekhariitbhu/Fast-Android-Networking	markushi/android-circlebutton
amitshekhariitbhu/RxJava2-Android-Samples	akexorcist/Android-BluetoothSPPLibrary
hongyangAndroid/baseAdapter	Yalantis/ToDoList
castorflex/SmoothProgressBar	dmytrodanylyk/folding-plugin
Bigkoo/Android-ConvenientBanner	houkx/android-pluginmgr
mikepenz/Android-Iconics	xingda920813/HelloDaemon
k0shk0sh/FastHub	hongyangAndroid/Android_Blog_Demos
Flipboard/bottomsheet	mmin18/FlexLayout
DreaminginCodeZH/Douya	j256/ormlite-android
alibaba/dexposed	thoughtbot/expandable-recycler-view
mcxiaoke/packer-ng-plugin	singwhatiwanna/android-art-res
diogobernardino/WilliamChart	BakerJQ/Android-InfiniteCards
wuhaoyu1990/MagicCamera	fossasia/phimpme-android
makovkastar/FloatingActionButton	blackfizz/EazeGraph
wdullaer/MaterialDateTimePicker	uccmawei/FingerprintIdentify
Yalantis/Phoenix	rosenpin/fading-text-view
HotBitmapGG/bilibili-android-client	Yalantis/pull-to-make-soup
yipianfengye/android-zxingLibrary	getActivity/AndroidProject
MindorksOpenSource/from-java-to-kotlin	siyamed/android-satellite-menu
bingoogolapple/BGARefreshLayout-Android	xuuhaoo/OkSocket
pedrovgs/AndroidWiFiADB	mrmans0n/smart-location-lib
wasabeef/Blurry	roomorama/Caldroid

roboguice/roboguice	flschweiger/SwipeStack
zo0r/react-native-push-notification	johnkil/Android-AppMsg
grantland/android-autofittextview	500px/greedo-layout-for-android
pili-engineering/PLDroidPlayer	limpoxe/Android-Plugin-Framework
nisrulz/android-tips-tricks	gelitenight/WaveView
Ashok-Varma/BottomNavigation	florent37/ArcLayout
Karumi/Dexter	Piasy/RxAndroidAudio
ximsfei/Android-skin-support	keklikhasan/LDrawer
JoanZapata/android-iconify	LuckyJayce/MVCHelper
jgilfelt/chuck	litesuits/android-lite-orm
hugeterry/CoordinatorTabLayout	kmshack/Android-ParallaxHeaderViewPager
Tencent/matrix	andyxialm/SmoothCheckBox
BoltsFramework/Bolts-Android	SundeepK/CompactCalendarView
MindorksOpenSource/android-mvp-architecture	wendux/DSBridge-Android
romandanylyk/PageIndicatorView	Genymobile/gnirehtet
ChadCSong/ShineButton	OCNYang/Android-Animation-Set
dinuscxj/LoadingDrawable	florent37/MaterialTextField
yanzhenjie/NoHttp	jrvansuita/MaterialAbout
youlookwhat/CloudReader	vekexasia/android-edittext-validator
TommyLemon/Android-ZBLibrary	DaxiaK/MyDiary
termux/termux-app	txusballesteros/bubbles-for-android
google/google-authenticator	Avocarrot/json2view
timehop/sticky-headers-recyclerview	mmin18/AndroidDynamicLoader
nytimes/Store	DmitryMalkovich/material-design-dimens
rockerhieu/emojicon	Tamicer/Novate
johncarl81/parceler	liaohuqiu/android-UCToast
journeyapps/zxing-android-embedded	UweTrottmann/SeriesGuide
SplashCodes/JAViewer	yyued/SVGAPlayer-Android
CellularPrivacy/Android-IMSI-Catcher-Detector	lopspower/CircularImageView
robinhood/ticker	daCapricorn/ArcMenu
Jacksgong/JKeyboardPanelSwitch	ukanth/afwall
avast/android-butterknife-zelezny	badoo/android-weak-handler
Manabu-GT/ExpandableTextView	qiujuer/ImageBlurring
bluelinelabs/Conductor	uber/okbuck
airbnb/DeepLinkDispatch	scottyab/secure-preferences
ryanhoo/StylishMusicPlayer	johannilsson/android-actionbar
facebook/shimmer-android	googlesamples/android-PictureInPicture
firebase/FirebaseUI-Android	Flowdalic/asmack
kikoso/android-stackblur	ac-pm/Inspeckage
yigit/android-priority-jobqueue	fossasia/open-event-droidgen
xcltapestry/XCL-Charts	square/phrase
yangfuhai/ASimpleCache	jdeferred/jdeferred
litesuits/android-common	harjot-oberai/VectorMaster
orhanobut/hawk	SalomonBrys/ANR-WatchDog
ivacf/archi	bravoborja/ReadMoreTextView
bingoogolapple/BGABanner-Android	GeekGhost/Ghost
bauerca/drag-sort-listview	blynkkk/blynk-server
seven332/EhViewer	hehonghui/Colorful
qstumn/BadgeView	liaohuqiu/android-GridViewWithHeaderAndFooter
bluelinelabs/LoganSquare	moagrius/TileView
arimorty/floatingsearchview	wangjiegulu/WheelView
yangfuhai/afinal	RoboBinding/RoboBinding
siacs/Conversations	mttkay/ignition
danikula/AndroidVideoCache	multidots/android-app-common-tasks
andremion/Music-Player	nntuyen/mkloader
tangqi92/Android-Tips	siwangqishiq/ImageEditor-Android
Meituan-Dianping/Robust	yahoo/squidb
huangyanbin/smartTable	zyyoona7/EasyPopup
bilibili/MagicaSakura	mdg-iitr/RotatingText
willowtreeapps/spruce-android	romannurik/Android-SwipeToDismiss
crazycodeboy/react-native-splash-screen	wuxudong/react-native-charts-wrapper
keyboardsurfer/Crouton	loopj/android-smart-image-view
hitherejoe/animate	hongyangAndroid/Android-StickyNavLayout
alamkanak/Android-Week-View	zserge/anvil
goldze/MVVMHabit	Tencent/soter
daimajia/AndroidViewHover	jberkel/sms-backup-plus
udacity/Sunshine-Version-2	eclipse/paho.mqtt.android
stephanenicolas/robospice	eleme/Amigo
gabrielemariotti/RecyclerViewItemAnimators	meituan/WMRouter
davideas/FlexibleAdapter	Bigkoo/Android-AlertView
jgilfelt/android-viewbadger	TakWolf/CNode-Material-Design
nicolasgramlich/AndEngine	anton46/Android-StepsView
objectbox/objectbox-java	facebook/TextLayoutBuilder
ragunathjawahar/android-saripaar	chenenyu/Router
pedrovgs/DraggablePanel	whilu/AndroidTagView
ManuelPeinado/FadingActionBar	antoniolg/MaterializeYourApp
saulmm/Android-Material-Examples	crossbario/autobahn-java

Skykai521/StickerCamera	blipinsk/RecyclerViewHeader
lovetuzitong/MultiImageSelector	doggycoder/AndroidOpenGLDemo
jasonross/Nuwa	chrisk44/Hijacker
psaravan/JamsMusicPlayer	franmontiel/PersistentCookieJar
dmytrodanylyk/android-process-button	wingjay/BlurImageView
iammert/ScalingLayout	lgvalle/android-flux-todo-app
razerdp/BasePopup	laobie/NineGridImageView
yipianfengye/android-adDialog	xamarin/XobotOS
Freeyourgadget/Gadgetbridge	6thsolution/EasyMVP
bilibili/boxing	pengyuantao/OnePush
JulienGenoud/android-percent-support-lib-sample	ogaclejapan/ArcLayout
CalebFenton/simplify	qyxxjd/MultipleStatusView
luojilab/DDComponentForAndroid	googlesamples/android-FingerprintDialog
CameraKit/blurkit-android	ikarus23/MifareClassicTool
Jasonchenlijian/FastBle	Kaopiz/KProgressHUD
HoraApps/LeafPic	stephanenicolas/Quality-Tools-for-Android
hmkcode/Android	ribot/ribot-app-android
lygttpod/SuperTextView	greenrobot/essentials
zzhoujay/RichText	Hitomis/FunGameRefresh
danylovolokh/VideoPlayerManager	matrixxun/MaterialBadgeTextView
SpecialCyCi/AndroidResideMenu	JessYanCoding/ArmsComponent
openaphid/android-flip	mozilla-mobile/focus-android
requery/requery	txusballesteros/welcome-coordinator
owncloud/android	jpardogo/GoogleProgressBar
googlemaps/android-maps-utils	hoang8f/android-flat-button
jiangqqlmj/FastDev4Android	facebook/screenshot-tests-for-android
Ereza/CustomActivityOnCrash	fyhertz/spydroid-ipcamera
edmodo/cropper	linglongxin24/DylanStepCount
code-troopers/android-betterpickers	mikepenz/Android-ActionItemBadge
square/wire	klinker41/android-slidingactivity
JakeWharton/ThreeTenABP	willowtreeapps/Hyperion-Android
qii/weiciyuan	yigit/dev-summit-architecture-demo
Yalantis/GuillotineMenu-Android	typ0520/fastdex
mzule/ActivityRouter	survivingwithandroid/Surviving-with-android
SimonVT/android-menudrawer	googlesamples/android-RuntimePermissions
Screetsec/TheFatRat	mixi-inc/AndroidTraining
LyndonChin/MasteringAndroidDataBinding	LarsWerkman/HoloColorPicker
BelooS/ChipsLayoutManager	ManuelPeinado/GlassActionBar
pushtorefresh/storio	pingpongboss/StandOut
north2016/T-MVP	MrEngineer13/SnackBar
w446108264/XhsEmoticonsKeyboard	jingle1267/android-utils
cyrilmottier/GreenDroid	john990/WaveView
luckybilly/CC	andreasschrade/android-design-template
florent37/ExpectAnim	connectbot/connectbot
beworker/pinned-section-listview	bignerdranch/expandable-recycler-view
geftimov/android-pathview	toddway/MaterialTransitions
apache/cordova-android	liaohuqiu/cube-sdk
stfalcon-studio/ChatKit	luizgrp/SectionedRecyclerViewAdapter
JohnPersano/SuperToasts	nisrulz/android-examples
chennaione/sugar	julian-klode/dns66
Todd-Davies/ProgressWheel	OCNYang/QBox
mikepenz/FastAdapter	romannurik/Android-WizardPager
JoanZapata/android-pdfview	oblador/react-native-keychain
googlesamples/android-vision	limedroid/XDroidMvp
RobotiumTech/robotium	thest1/LazyList
alibaba/Tangram-Android	k0shk0sh/PermissionHelper
oguzbilgener/CircularFloatingActionMenu	Pkmmte/CircularImageView
johannilsson/android-pulltorefresh	eoecn/android-app
mihaip/dex-method-counts	aritraroy/PinLockView
TangoAgency/material-intro-screen	iiordanov/remote-desktop-clients
yarolegovich/SlidingRootNav	Automattic/simplenote-android
jd-alexander/LikeButton	MindorksOpenSource/AndroidTensorFlowMachineLearningExa mple
MagicMashRoom/SuperCalendar	
rovo89/XposedInstaller	adrielcafe/AndroidAudioRecorder
fyhertz/libstreaming	f2prateek/dart
LawnchairLauncher/Lawnchair	venshine/BezierMaker
bmelnychuk/AndroidTreeView	evant/binding-collection-adapter
google/hover	yjfnypeu/UpdatePlugin
hussien89aa/AndroidTutorialForBeginners	j4velin/Pedometer
mpusher/mpush	JeroenMols/LandscapeVideoCamera
iwgang/CountdownView	todotxt/todo.txt-android
Comcast/FreeFlow	splitwise/TokenAutoComplete
balysv/material-menu	nextcloud/android
bin456789/Unblock163MusicClient-Xposed	linfaxin/TransitionPlayer
uber/NullAway	panpf/sketch
romainguy/ViewServer	ParkSangGwon/TedPermission
florent37/DiagonalLayout	open-keychain/open-keychain

path/android-priority-jobqueue	Neamar/KISS
wenmingvs/AndroidProcess	wooplr/Spotlight
lizhangqu/CoreLink	MiPushFramework/MiPushFramework
harjot-oberai/MusicDNA	bytedeco/javacpp-presets
pchmn/MaterialChipsInput	shaohui10086/AdvancedLuban
500px/500px-android-blur	Devlight/ArcProgressStackView
TeamAmaze/AmazeFileManager	kaelaela/VerticalViewPager
ozodrukh/CircularReveal	pedrovgs/Renderers
AriaLyy/Aria	pili-engineering/PLDroidMediaStreaming
Cleveroad/SlidingTutorial-Android	commonsguy/cw-android
flavioarfaria/KenBurnsView	kymjs/EmojiChat
ittianyu/BottomNavigationViewEx	GavinCT/AndroidMultiChannelBuildTool
nisrulz/sensey	Tibolte/AgendaCalendarView
googlesamples/google-services	GDG-Korea/PinterestLikeAdapterView
JakeWharton/Telecine	yale8848/CacheWebView
JackyAndroid/AndroidChromium	siriscac/RippleView
hyb1996/Auto.js	noties/Scrollable
WritingMinds/ffmpeg-android-java	zeapo/Android-Password-Store
iSoron/uhabits	mrwonderman/android-square-progressbar
ZieIony/Carbon	tiann/epic
siyamed/android-shape-imageview	eneim/toro
iPaulPro/Android-ItemTouchHelper-Demo	javiersantos/PiracyChecker
simple-android-framework/android_design_patterns_analysis	PureDark/H-Viewer
moezbhatti/qksms	dueeeke/dkplayer
JZ-Darkal/AndroidHttpCapture	saulmm/CoordinatorExamples
konifar/android-material-design-icon-generator-plugin	jasonpolites/gesture-imageview
aritraroy/PatternLockView	tumblr/Graywater
dlew/joda-time-android	scwang90/MultiWaveHeader
facebook/react-native-fbsdk	ShamylZakariya/StickyHeaders
web3j/web3j	glomadrian/RoadRunner
woxingxiao/BubbleSeekBar	limedroid/XDroid
jjoe64/GraphView	niniloveyou/StateButton
JackyAndroid/AndroidTVLauncher	dongjunkun/GanK
deano2390/MaterialShowcaseView	HackPlan/AndroidCharts
mikepenz/AboutLibraries	google/android-ui-toolkit-demos
r0adkll/Slidr	hitherejoe/Bourbon
FinalTeam/RxGalleryFinal	chiuki/advanced-textview
amitshekhariitbhu/awesome-android-complete-reference	sjwall/MaterialTapTargetPrompt
ldoublem/LoadingView	jonfinerty/Once
esoxjem/MovieGuide	Yasic/ParticleTextView
markushi/android-ui	bpellin/keepassdroid
theDazzler/droidicon	katzer/cordova-plugin-background-mode
square/tape	D-clock/AndroidSystemUiTraining
2dxgujun/AndroidTagGroup	githubwing/ByeBurger
qiujuer/Genius-Android	yankai-victor/Loading
AntennaPod/AntennaPod	fishwjy/MultiType-FilePicker
Frank-Zhu/PullZoomView	couchbase/couchbase-lite-android
bingoogolapple/BGABadgeView-Android	venshine/GoodView
facebook/device-year-class	mohak1712/UberUX
xinghongfei/LookLook	hongyangAndroid/Android-CircleMenu
daimajia/AnimationEasingFunctions	davidschreiber/FancyCoverFlow
chenBingX/SuperTextView	cn-ljb/rxjava_for_android
florent37/ViewAnimator	whataa/pandora
zhou-you/RxEasyHttp	suzeyu1992/repo
jaydenxiao2016/AndroidFire	pili-engineering/PLDroidShortVideo
balysv/material-ripple	sd6352051/NiftyNotification
zhouchaoyuan/excelPanel	omkarmoghe/Pokemap
stormzhang/9GAG	petrnohejl/Android-Templates-And-Utilities
iammert/MaterialIntroView	todoroo/astrid
vikramkakkar/SublimePicker	orgzly/orgzly-android
VictorAlbertos/RxCache	square/haha
Yalantis/Euclid	UCodeUStory/S-MVP
AweiLoveAndroid/CommonDevKnowledge	iammert/ExpandableLayout
AltBeacon/android-beacon-library	guardianproject/ChatSecureAndroid
avast/android-styled-dialogs	westnordost/StreetComplete
oasisfeng/condom	Chanven/CommonPullToRefresh
xfumihiro/ViewInspector	hongyangAndroid/ChangeSkin
zagum/Android-SwitchIcon	michael-rapp/ChromeLikeTabSwitcher
yqritc/RecyclerView-FlexibleDivider	FabianTerhorst/Isometric
Zomato/AndroidPhotoFilters	romannurik/Android-MonthCalendarWidget
tyzlmjj/PagerBottomTabStrip	philliphsu/BottomSheetPickers
Rukey7/MvpApp	sockeqwe/fragmentargs
WVector/AppUpdate	sahildave/Search-View-Layout
igreenwood/SimpleCropView	weidongjian/androidWheelView
	misakuo/3dTagCloudAndroid
	sendtion/XRichText
	jiang111/RxJavaApp

grandcentrix/tray	Trinea/android-demo
Diolor/Swipecards	lopspower/CircularFillableLoaders
googlesamples/android-Camera2Basic	syncthing/syncthing-android
fengjundev/Android-Skin-Loader	linkedin/dexmaker
Polidea/RxAndroidBle	fabioCollini/DaggerMock
florent37/CameraFragment	antonkrasov/AndroidSocialNetworks
chrisbanes/philm	guiying712/AndroidModulePattern
reark/reark	yshrsmz/KeyboardVisibilityEvent
AigeStudio/WheelPicker	JakeWharton/Android-DirectionalViewPager
Dimezis/BlurView	desmond1121/Android-Ptr-Comparison
H07000223/FlycoDialog_Master	roughike/SwipeSelector
ThirtyDegreesRay/OpenHub	timusus/RecyclerView-FastScroll
cSploit/android	lyft/scoop
zjw-swun/AppMethodOrder	tommybuonomo/dotsindicator
jiajunhui/PlayerBase	kofigyan/StateProgressBar
dmilicic/Android-Clean-Boilerplate	jinatonic/confetti
android-cjj/Android-MaterialRefreshLayout	lygttpod/AndroidCustomView
promeG/TinyPinyin	dmytrodanylyk/android-morphing-button
Ramotion/paper-onboarding-android	liaohuqiu/android-cube-app
pwittchen/ReactiveNetwork	square/seismic
mcharmas/Android-ReactiveLocation	cachecats/coderiver
androidquery/androidquery	iammert/MusicPlayerView
MindorksOpenSource/PRDownloader	bboyairwreck/PieMessage
kabouzeid/Phonograph	4thline/cling
jackpal/Android-Terminal-Emulator	daniel-stoneuk/material-about-library
yingLanNull/ShadowImageView	qinci/MarkdownEditors
jgilfelt/android-sqlite-asset-helper	lovedise/PermissionGen
evrencoskun/TableView	alibaba/Virtualview-Android
cymcsg/UltimateAndroid	luckybilly/PreLoader
BaronZ88/MinimalistWeather	lantouzi/WheelView-Android
balsikandar/Android-Studio-Plugins	felipecsl/GifImageView
Krupen/FabulousFilter	HpWens/MeiWidgetView
zcweng/ToggleButton	smuyyh/ImageSelector
wordpress-mobile/WordPress-Android	vilyever/AndroidSocketClient
TheFinestArtist/FinestWebView-Android	JoaquimLey/faboptions
mcharmas/android-parcelable-intellij-plugin	yewei02538/HiPermission
tiann/understand-plugin-framework	ccrama/Slide
sucese/android-open-source-project-analysis	SilenceDut/KnowWeather
steelkiwi/cropiwa	fossasia/pslab-android
MikeOrtiz/TouchImageView	tianshaojie/AndroidFine
EverythingMe/overscroll-decor	grandcentrix/ThirtyInch
konmik/nucleus	Ramotion/android-ui-animation-components-and-libraries
fython/MaterialStepperView	z56402344/BaseAnimation
Devlight/NavigationTabStrip	lucasr/smoothie
uber/AutoDispose	googlesamples/androidtv-Leanback
zzz40500/AndroidSweetSheet	robertlevonyan/materialChipView
yanzhenjie/Album	xiongwei-git/AndroidVideoPlayer
bonnyfone/vectalign	simpligility/android-maven-plugin
PomepuyN/BlurEffectForAndroidDesign	vlonjatg/progress-activity
lingcimi/jjdxm_ijkplayer	xdtianyu/CallerInfo
JingYeoh/FragmentRigger	PhilippC/keepass2android
tjerkw/Android-SlideExpandableListView	pungrue26/SelectableRoundedImageView
rmtheis/android-ocr	androiddevelop/AlignTextView
skyfishjy/android-ripple-background	Kennyc1012/MultiStateView
gotev/android-upload-service	
arcadefire/nice-spinner	
kymjs/KJFrameForAndroid	
Yalantis/Horizon	

APÊNDICE C - REFATORAÇÕES SUPORTADAS PELO REFACTORINGMINER

RefactoringMiner 1.0 & 2.0	RefactoringMiner 2.0
Extract Method	Move and Rename Class
Inline Method	Extract Class
Rename Method	Extract Subclass
Move Method	Extract Variable
Move Attribute	Inline Variable
Pull Up Method	Parameterize Variable
Pull Up Attribute	Rename Variable
Push Down Method	Rename Parameter
Push Down Attribute	Rename Attribute
Extract Superclass	Move and Rename Attribute
Extract Interface	Replace Variable with Attribute
Move Class	Replace Attribute (with Attribute)
Rename Class	Merge Variable
Extract and Move Method	Merge Parameter
Change Package (Move, Rename, Split, Merge)	Merge Attribute
	Split Variable
	Split Parameter
	Split Attribute
	Change Variable Type
	Change Parameter Type
	Change Return Type
	Change Attribute Type