UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

Trabalho de Conclusão de Curso

Utilização de diminuição na quantidade de *statements* em métodos como indício para detecção de refatoramento

Ana Carla Gomes Bibiano

ana.carlagb@gmail.com

Orientador:

Prof. Dr. Baldoino Fonseca Neto

MACEIÓ, FEVEREIRO DE 2017

Ana Carla Gomes Bibiano

Utilização de diminuição na quantidade de *statements* em métodos como indício para detecção de refatoramento

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:
Prof. Dr. Baldoino Fonseca Neto

MACEIÓ, FEVEREIRO DE 2017

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Prof. Dr. Baldoino Fonseca dos Santos Neto - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Prof. Dr.
Instituto de Computação
Universidade Federal de Alagoas

Prof. Msc.
Instituto de Computação
Universidade Federal de Alagoas

Agradecimentos:

Acima de tudo, quero agradecer a Causa de todas as causas e potências (Aristóteles), para que assim este momento tão desejado chegasse.

Em especial, agradecer aos meus avós, pais e irmãos por toda dedicação, apoio direto e indireto, frutos do nosso amor familiar, ao meu namorado pela paciência e auxílio em diversos momentos desse trabalho.

Agradeço ao corpo docente e discente do Instituto de Computação, em especial aos professores Baldoino Fonseca, Márcio Ribeiro, Rodrigo Paes por sempre motivarem a dedicação pela programação e pesquisa acadêmica. Aos professores Alcino Dall'Igna, Jaime Evaristo e Evandro Costa por me motivarem a ser exemplo de persistência, ousadia e dedicação aos estudos em buscas de avanços para o bem comum.

Aos meus amigos de graduação Tamirys Pino, Jairo José, Armando Barbosa, Laysa Silva, Karla Elizabeth e Anderson Santos, Anthony Jatobá, João Lucas e Gabriel Nunes por sempre estarem dispostos a trocar conhecimento e experiências; ao Paulo Victor e Rafaele Oliveira por me proporcionarem a oportunidade de aumentar meus conhecimentos através da experiência de estágio e pela liberação dos projetos privados para esse estudo.

"Nada do que está em potência,

passa ao ato senão por outra coisa que já está em ato. "

Aristóteles

Resumo:

A medida que o software evolui, surge a necessidade de mantê-lo a fim de refletir a evolução dos requisitos dos usuários e corrigir erros detectados no programa, se os desenvolvedores realizam mudanças no software sem considerar o projeto do sistema, a estrutura interna do programa tende a sofrer degradação. Os efeitos acumulativos de tais mudanças podem levar a sistemas não confiáveis e difíceis de manter [10], ocasionando a presença dos chamados *Bad Smells* que são, basicamente, partes do código que indicam que podem ser otimizadas, e que caso não, podem ocasionar problemas no sistema.

A técnica mais utilizada para essa otimização é o *Refactoring* ou refatoramento que é o processo de mudar um sistema de software de tal forma que não altera o comportamento externo do código, mas melhora a sua estrutura interna. [1]. Estudos empíricos mostram que o refatoramento pode melhorar a manutenibilidade e reusabilidade de um software [2][3]. Porém, como citado acima, a ausência de técnicas como refatoramento durante o desenvolvimento do software pode trazê-lo sérios problemas.

Atualmente há várias ferramentas na literatura, que detectam técnicas de refatoramento em softwares, auxiliando avaliar quais técnicas foram aplicadas, se as técnicas melhoraram ou não a qualidade do software e mais ainda, avaliando a preocupação do desenvolvedor em melhorar o código durante o processo de desenvolvimento. Porém, de acordo com [6], o conceito de refatoramento entre desenvolvedores ainda é muito subjetivo comparado as técnicas de detecção de refatoramento utilizada por ferramentas.

Baseado nessa problemática esse estudo busca investigar se a diminuição (“queda”) da quantidade de statements em métodos pode estar relacionada à técnica de refatoramento (pois de acordo com [1], refatoramento em métodos podem solucionar diversos *Bad Smells* relacionados a complexidade, tamanho e design do software). Para essa investigação foi realizado uma coleta de histórico de diminuição de *statements* em métodos e posteriormente desenvolvedores analisaram se essas quedas foram provocadas por refatoramento, caso sim, descreveram quais foram as técnicas de refatoração, e depois foi utilizado o uso de ferramenta de detecção de refatoramento para também detectar se essas quedas foram causadas por refatoração. Por conseguinte, foi feito uma análise entre as respostas dos desenvolvedores e a percepção de detecção da ferramenta, dessa forma também analisar quão próximo está o conceito de refatoramento entre ferramenta e o desenvolvedor.

Palavras chave: Detecção de refatoramento, Mudanças em métodos, Histórico de Desenvolvimento de software.

Lista de Figuras:

Figura 2.2.1: Estrutura Geral de um método...17

Figura 2.2.3.a: UML da aplicação “LojaXY Project”.. 19

Figura 2.2.3.b: Exemplo de dois métodos que podem ser agrupados em um único método...19

Figura 2.2.3.c: Aplicação do Extract Method sobre fragmentação proposta............................20

Figura 2.2.3.d: Exemplo de método que pode estar usando mais características de outra classe. 21

Figura 2.2.3.e: Exemplo após a aplicação do Move Method..22

Figura 2.2.3.f: Corpo do método que foi movido para outra classe..23

Figura 2.2.3.g: Exemplo de método que pode o nome do método não condiz com sua real função...23

Figura 2.2.3.h: Método após a renomeação...24

Figura 3.1.a: Controle de versão centralizado. Imagem baseada em [14].................................26

Figura 3.1.b: Controle de versão descentralizado. Imagem baseada em [14]...........................27

Figura 3.2: Fluxo de desenvolvimento usando Git..28

Figura 4.1.1.1.a: Dependência Maven do Metric Miner..29

Figura 4.1.1.1.b: Dependência Maven do JDT...29

Figura 4.1.1.1.c: Eclipse AST *..31

Figura 4.1.2.a: UML “Extract Minerator Tool”...35

Figura 4.1.2.b: Classe “MyStudy”...36

Figura 4.1.2.c: Clase “JavaParserVisitor” 37

Figura 4.1.2.d: Classe “Method Visitor”...39

Figura 4.2.1: Fluxo projetos Clip 42

Figura 4.3.1: Processo de construção de histórico de métodos...43

Figura 4.3.2.a: Total de métodos encontrados no projeto Clip 44

Figura 4.3.2.b: Total de métodos encontrados no projeto Painel Admin..................................45

Figura 4.3.2.c: Total de métodos encontrados no projeto Document Generator.......................45

Figura 4.3.2.d: Total de métodos encontrados no projeto Meyer Control................................46

Figura 4.3.2.e: Total de métodos encontrados no projeto Clip Commons................................46

Figura 4.3.2.f: Total de métodos encontrados no projeto Clip OCR..47

Figura 4.3.2.1a: Histórico do método 1*...48

Figura 4.3.2.1.b: Histórico do método 2...48

Figura 4.3.2.1.1.1.a: Fragmento do resultado do Refactoring Minner.......................................50

Figura 4.3.2.1.1.1.b: Total de Rename Method detectados pelo Refactoring Miner por projeto...51

Figura 4.3.2.1.1.2.a: Classe "HistoricProject"...52

Figura 4.3.2.1.1.2.b: Total de métodos válidos por projeto para análise de statements..............54

Figura 5.1.a: Método "minerateExtactMethod"..55

Figura 5.1.b: Método "findExtactMethod"...56

Figura 5.2.2: Total de quedas de statements por projeto...58

Figura 6.a: Total de quedas de statements e métodos por projeto..63

Figura 6.b:Total de refatoramentos encontrados – Projeto Clip...64

Figura 6.c:Total de refatoramentos encontrados – Projeto Document Generator.......................65

Figura 6.d:Total de refatoramentos encontrados – Projeto Clip OCR.......................................65

Figura 6.e:Total de refatoramentos encontrados – Projeto Painel Admin..................................66

Figura 6.f: Total de refatoramentos encontrados – Projeto Meyer Control...............................67

Figura 6.g: Causas da diminuição de statements nos métodos analisados do Projeto Clip...68

Figura 6.h: Causas da diminuição de statements nos métodos analisados do Projeto Meyer

Control..,..............69

Figura 6.i: Causas da diminuição de statements nos métodos analisados do Projeto Painel Admin...69

Figura 6.j: Causas da diminuição de statements nos métodos analisados do Projeto Document Generator..70

Figura 6.k: Causas da diminuição de statements nos métodos analisados do Projeto Clip OCR...70

Sumário:

1 Introdução.....12

1.1 Contextualização.....12

1.2 Motivação.....13

1.3 Objetivos.....13

1.4 Estrutura.....14

2 Fundamentação teórica sobre métricas de software relacionadas a técnicas de refatoramento.....15

2.1 Visão Geral sobre métricas de software.....15

2.1.1 Grupos de Métricas.....15

2.1.2 Métrica de *Statements*.....15

2.2 Técnicas de Refatoramento em métodos.....17

2.2.1 Estrutura geral de método.....17

2.2.2 Refatoramento aplicado em métodos.....17

2.2.3 Mudanças e métricas que podem ser causadas pela aplicação de refatoramento em métodos.....18

2.2.4 Diminuição de statements em métodos causados por refatoramento.....24

3 Fundamentação teórica sobre sistemas de controle de versão.....25

3.1 Sistema de controle de versão (VCS).....25

3.2 Introdução à tecnologia Git.....27

4 Materiais e métodos utilizados para a recuperação do histórico de softwares.....28

4.1 Técnicas e ferramentas utilizadas para recuperação do histórico de software.....28

4.1.1 Metric Miner.....29

4.1.1.1 Visão Geral sobre AST e JDT.....29

4.1.2 Implementação *Extract Minerator Tool* utilizando Metric Miner API.....35

4.2 Visão Geral sobre o histórico de desenvolvimento dos softwares escolhidos.....40

4.2.1 Domínio de *softwares* escolhidos.....40

4.2.1.1 Clip.....40

4.2.1.2 Clip OCR.....40

4.2.1.3 Clip Commons.....41

4.2.1.4 Meyer Control.....41

4.2.1.5 Painel Admin.....41

4.2.1.6 Document Generator.....41

4.3 Base de dados com o histórico de métodos.....43

4.3.1 Processo de construção.....43

4.3.2 Visão geral sobre o histórico de métodos obtido.....44

4.3.2.1 Histórico de métodos renomeados.....47

4.3.2.1.1 Técnicas e ferramentas utilizadas para a recuperação de histórico de métodos renomeados.....49

4.3.2.1.1.1 Refactoring Miner.....49

4.3.2.1.1.2 Rename Minerator Tool.....51

5. Materiais e métodos utilizados para o histórico de diminuição de *statements* em métodos......54

5.1 Técnica utilizada para detectar diminuição de *statements* em métodos......54

5.2 Base de dados com o histórico de diminuição de *statements* em métodos......57

5.2.1 Estrutura da Base de dados......57

5.2.2 Visão geral sobre o histórico de diminuição de statements obtido......58

5.3 Investigação das causas da diminuição de *statements*......58

5.3.1 Visão geral sobre o processo de investigação......59

5.3.1.1 Processo de Detecção......59

5.3.1.1.1 Detecção Automática......61

5.3.1.1.2 Detecção Manual......62

6. Resultados e discussões 62

7. Trabalhos Relacionados......71

7.1 "Do they Really Smell Bad? A Study on Developes' Perception of Bad Code Smells" 71

7.2 "Why we refactor? Confession of Github Contributor"......72

8. Considerações Finais......73

8.1 Trabalhos Futuros......74

REFERÊNCIAS BIBLIOGRÁFICAS......75

1. Introdução

1.1 Contextualização

Em softwares, durante o desenvolvimento, podem ocorrer várias mudanças, tais como a evolução dos requisitos dos usuários e a correção de erros detectados no programa. Logo, estudos mostram que a manutenção de software é uma tarefa complexa e custosa, sendo responsável por cerca de 40% a 90% do custo total do software [11]. Empiricamente, é observado que a maior parte desse tempo em manutenção é causada porque muitas vezes são adicionadas novas funcionalidades no software sem o melhoramento do código antigo, ou seja, os desenvolvedores realizam mudanças no software sem considerar a estrutura interna do programa, causando diversos problemas relacionados ao design e desempenho funcional do software.

Dentre esses problemas podem ocorrer os chamados *Bad Smells* que são, basicamente, partes do código indicam que podem ser otimizadas, e caso não, podem ocasionar problemas no sistema [1] como lentidão no tempo de resposta, vulnerabilidades de segurança de dados, elegibilidade no código [13].

Neste contexto, a utilização de técnicas de refatoramento têm se mostrado um dos caminhos mais promissores para lidar com a manutenção de software. Refatoramento ou *Refactoring* é o processo de mudar um sistema de software de tal forma que não altera o comportamento externo do código, mas melhora a sua estrutura interna. [1]. A aplicação de um refatoramento está condicionada a realização de pelos menos duas atividades [4]: (i) identificar as partes do software que precisam ser melhoradas e (ii) determinar as mudanças (ou correções) que devem ser aplicadas ao software a fim de melhorar sua qualidade.

Dessa forma, é de suma importância acompanhar os refatoramentos realizados durante o desenvolvimento para avaliar quais técnicas foram aplicadas, se as técnicas melhoraram ou não a qualidade do software e mais ainda, avaliando a preocupação do desenvolvedor em melhorar o código durante o processo de desenvolvimento. Para este acompanhamento, muitas vezes são utilizadas ferramentas de detecção de refatoramento. Na literatura é possível encontrar ferramentas tais como *Refactoring Miner,* RefFinder e JDevAn, descritos em [6], que utilizam técnicas para detecção como métricas, métodos formais, mineração de dados e etc.

1.2 Motivação

Como apresentado na seção acima, na literatura é possível encontrar diversas ferramentas de detecção de refatoramento que utilizam diversas técnicas para detecção. Porém o quão próximo está o conceito de refatoramento do desenvolvedor comparado ao conceito automatizado da ferramenta? Ou seja, em outras palavras, será aquilo que é refatoramento para a ferramenta, também é refatoramento para o desenvolvedor?

Portanto, há uma necessidade de estudos comparativos entre resultados de detecção de ferramentas e a avaliação de desenvolvedores em mudanças código que possam indicar a aplicação de refatoramento(s), e assim "quantificar e qualificar" quão próximo está este conceito de refatoramento entre humano e máquina.

1.3 Objetivos

Objetivo Geral: Estudo busca investigar se a diminuição ("quedas") da quantidade de statements em métodos pode ter sido provocada por técnicas de refatoramento no método. Para essa investigação foi realizado uma coleta de histórico de diminuição de *statements* em métodos e posteriormente desenvolvedores e ferramenta de detecção de refatoramento analisaram se essas quedas foram provocadas por refatoramento, caso sim, descreveram quais foram as técnicas de refatoração. Por conseguinte, foi feito uma comparação entre as respostas dos desenvolvedores e a percepção de detecção da ferramenta e dessa forma também analisar quão próximo está o conceito de refatoramento entre ferramenta e o desenvolvedor.

Objetivos Específicos:

1. Encontrar o histórico de diminuição na quantidade de statements em método;
2. Utilizar o histórico de diminuição na quantidade de statements em métodos para desenvolvedores analisarem se essa diminuição foi provocada por técnicas de refatoramento e caso houve refatoração, então descrever quais foram as técnicas de refatoramento;
3. Utilizar ferramenta de detecção de refatoramento para detectar se a diminuição na quantidade de statements em métodos foi provocada por técnicas de refatoramento e caso houve refatoração, então descrever quais foram as técnicas de refatoramento;
4. Avaliar as respostas obtidas por desenvolvedores e ferramenta de detecção e comparar se ambas perceberam a mesma quantidade de refatoramento.

5. Avaliar se desenvolvedores e ferramenta detectaram as mesmas técnicas de refatoração.

1.4 Estrutura

Esse trabalho está dividido da seguinte forma:

- **Cap. 2: Fundamentação teórica sobre métricas de software relacionadas a técnicas de refatoramento:** esse capítulo apresenta uma visão geral sobre o conceito de métricas de software e também apresenta o conceito das técnicas de refatoramento utilizadas nesse estudo: *Extract Method, Move Method* e *Rename Method,* apresentando exemplos de métricas que podem sofrer alterações por causas da aplicação dessas técnicas, especificamente a métrica de *statements.*

- **Cap. 3: Fundamentação teórica sobre sistemas de controle de versão:** esse capítulo apresenta o conceito de sistemas de controle de versão (VCS), utilizado para recuperar o histórico das mudanças ocorridas durante o desenvolvimento de softwares, apresentando uma visão geral sobre o VCS utilizado nesse estudo, o Git.

- **Cap. 4: Materiais e métodos utilizados para a recuperação do histórico de softwares:** apresenta ferramentas e técnicas utilizadas para recuperar o histórico de métodos em softwares. Apresentando uma visão geral sobre o histórico de desenvolvimento e o domínio de cada software escolhido para esse estudo e uma visão geral sobre o histórico de métodos obtido.

- **Cap. 5: Materiais e métodos utilizados para o histórico de diminuição de *statements* em métodos:** apresenta ferramentas e técnicas utilizadas para procurar a diminuição de statements entre o histórico de métodos, também apresenta uma visão geral sobre o processo de detecção de refatoramento e demais causas que podem ter provocado a diminuição dos statements, detectados através da percepção de desenvolvedores e ferramenta de detecção.

- **Cap. 6: Resultados e discussões:** apresenta os resultados obtidos através do processo de detecção de refatoramento realizados por desenvolvedores e ferramenta de detecção, apresentando também as possíveis causas que podem ter provocados a diminuição de statements.

- **Cap. 7: Trabalhos relacionados:** apresentam trabalhos que utilizaram metodologias e/ou ferramentas utilizadas por esse estudo e que auxiliaram como embasamento teórico e metodológico.

- **Cap. 8 Considerações finais:** será apresentado a conclusão dos resultados obtidos, dificuldades encontradas e aprendizado para trabalhos futuros.

2. Fundamentação teórica sobre métricas de software relacionadas a técnicas de refatoramento

2.1 Visão Geral sobre métricas de software

Durante o desenvolvimento do software, são aplicadas várias mudanças que podem ser adição ou remoção de funcionalidade, conserto de erros[1] ou melhoramento de código, especificamente refatoramento, estas mudanças causam a variação das chamadas métricas de software[2], na literatura há várias métricas e existem várias ferramentas automatizadas para coletar métricas do software tais como Understand[3], PMD[4], SDMetrics[5], em resumo a maioria das métricas que existem podem ser agrupadas em 4 grupos: Tamanho, Acoplamento, Coesão e Complexidade, segue abaixo as definições[6] de cada grupo:

2.1.1 Grupos de Métricas

- **Tamanho:** medem o tamanho dos elementos de projeto, normalmente contando os elementos contidos. Por exemplo, o número de classes, a quantidade de métodos por classe, o número de linhas de código.
- **Coesão:** É o grau em que os elementos em uma unidade de projeto (pacote, classe, etc.) estão logicamente relacionados, por exemplo, o grau de relacionamento entre diferentes pacotes.
- **Acoplamento:** é o grau em que os elementos de um projeto estão conectados, por exemplo, o grau de relacionamento entre classes de diferentes

pacotes.

- **Complexidade:** A complexidade mede o grau de conectividade entre os elementos de uma unidade de projeto, por exemplo, o grau de relacionamento entre métodos de diferentes classes.

2.1.2 Métrica de *Statements*

Dentre esses grupos descritos acima, são vários tipos de métricas que a compõem, como número de linha de código, quantidade de métodos por classe. Para esse estudo, foi utilizado, a quantidade de *statements* em métodos. Nesta seção, será explicado sobre o conceito de statements e na seção 4.1.1 será descrito quais tipos de statements foram aceitos para esse estudo.

De acordo com [8], "*statement*, na linguagem de programação, é o menor elemento autônomo de uma linguagem de programação imperativa[7] que expressa alguma ação a ser realizada. É uma instrução escrita em uma linguagem de alto nível que comanda o computador para executar uma ação especificada. Um programa escrito em tal linguagem é formado por uma sequência de uma ou mais declarações.".

A mudança na quantidade de statements foi escolhida para esse estudo, por que empiricamente pode indicar várias informações sobre quais tipos de mudanças estão ocorrendo naquele método, tais como se o código está sendo refatorado (pois pode estar relacionado a vários tipos de refatoramento como será descrito na próxima seção), se o domínio[8] está sofrendo alterações. Porém, para esse estudo, o foco é detectar se está havendo refatoramento.

---

1. Erros: para esse estudo, o conceito de erros é livre, inclui erros de sintaxe, semântica e etc.
2. Métrica de software: disponível em https://pt.wikipedia.org/wiki/M%C3%A9trica_de_software
3. Understand: disponível em https://scitools.com/
4. PMD: disponível em http://pmd.sourceforge.net/snapshot/pmd-java/
5. SDMetrics: disponível em http://www.sdmetrics.com/
6. Tipos de métricas: conceitos baseados em: http://www.sdmetrics.com/DProp.html

7.Linguagem de programação imperativa:
disponível em: https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_imperativa

8. Domínio de Software, disponível em: https://pt.wikipedia.org/wiki/An%C3%A1lise_de_dom%C3%ADni

2.2 Técnicas de Refatoramento em métodos.

Nesta seção, serão apresentadas técnicas de refatoramentos escolhidas para esses estudos, seus respectivos conceitos e casos de usos.

2.2.1 Estrutura geral de método

Segue abaixo, os termos utilizados para a estrutura de um método.

Figura 2.2.1: Estrutura Geral de um método

2.2.2 Refatoramento aplicado em métodos

Na literatura, há várias técnicas de refatoramento em métodos, mas para esse estudo foram escolhidas 3 técnicas, são estas: ***Extract Method, Rename Method*** e ***Move Method***, pois de acordo com [6] são as mais comuns e mais conhecidas por desenvolvedores e na prática as demais técnicas são variações destas.

Segue abaixo a definição, de acordo com [1], de cada uma dessas técnicas e descrição de mudanças e métricas que podem ser causadas por esses refatoramentos.

- ***Extract Method*** deve ser utilizado quando no corpo de um método, há fragmentos de código que podem ser agrupados em outro método, cujo nome deve explicitar a finalidade deste.

- ***Move Method:*** quando um método está usando mais características de outra classe do que a classe que ele foi definido, então deve-se mover o método para a classe que o método usa mais e o método antigo (da classe que o método estava definido antes) pode tornar-se uma delegação simples ou ser removido totalmente.
- ***Rename Method*** quando o nome do método não condiz com sua finalidade, então deve-se mudar o nome do método.

2.2.3 Mudanças e métricas que podem ser causadas pela aplicação de refatoramento em métodos.

Os exemplos abaixo apresentarão situações em que as técnicas de refatoramento podem ser aplicadas e as mudanças e métricas que essas técnicas podem causar.

Os fragmentos de código a seguir pertencem ao código aberto *LojaXY Project*[9], (código de autoria própria), esses fragmentos apresentam situações antes e depois das técnicas de refatoramento. Segue abaixo a UML do projeto que será utilizado como exemplo:

9. LojaXY Project: código aberto,
disponível em: https://github.com/anacarlagb/LojaXY_Project

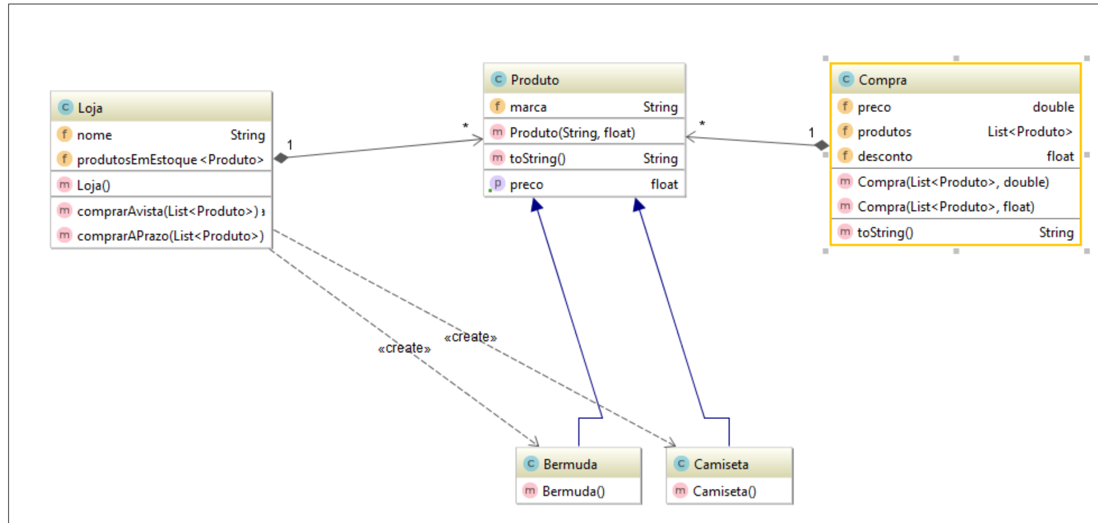

Figura 2.2.3.a: UML da aplicação “*LojaXY Project*”

❖ Antes do *Extract Method*

```
28    public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
29
30        if(listaProdutos.size() < produtosEmEstoque.size()){
31
32           double total = 0;
33           for(Produto produto : listaProdutos){
34
35               total += produto.preco;
36           }
37
38           double desconto = total * 0.1;
39           double totalComDesconto = total - desconto;
40           return new Compra(listaProdutos, totalComDesconto);
41        }
42
43        throw new CompraInvalidaException();
44    }
45
46    public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
47
48        if(listaProdutos.size() < produtosEmEstoque.size()){
49
50               double total = 0;
51               for(Produto produto : listaProdutos){
52
53                   total += produto.preco;
54               }
55
56               double desconto = total * 0.05;
57               double totalComDesconto = total - desconto;
58               return new Compra(listaProdutos, totalComDesconto);
59           }
60
61           throw new CompraInvalidaException();
62
63    }
64
```

Figura 2.2.3.b: Exemplo de dois métodos que podem ser agrupados em um único método.

Esse fragmento de código pertence à classe da “Loja”, há dois métodos com

aproximadamente 95% do código duplicado, essa é uma das situações em que o código pode ser agrupado em um único método como segue abaixo.

❖ Depois do *Extract Method*

```
30
31    public Compra comprar(List<Produto> listaProdutos, double desconto) throws CompraInvalidaException{
32
33        if(listaProdutos.size() < produtosEmEstoque.size()){
34
35                double total = 0;
36                for(Produto produto : listaProdutos){
37
38                    total += produto.getPreco();
39                }
40
41                double valorDescontado = total * desconto;
42
43                double totalComDesconto = total - valorDescontado;
44
45                return new Compra(listaProdutos, totalComDesconto);
46        }
47
48        throw new CompraInvalidaException();
49    }
50
51
52    public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
53        return comprar(listaProdutos, 0.1);
54    }
55
56    public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
57        return comprar(listaProdutos, 0.05);
58    }
59
```

Figura 2.2.3.c: Aplicação do *Extract Method* sobre fragmentação proposta

Como pode ser observado acima, o código que estava em comum entre os dois métodos foi extraído, gerando um novo método e os “métodos antigos” usam o “novo método” passando como parâmetro o que é específico de cada método.

❖ Mudanças e Métricas causadas pelo Refatoramento:

- A quantidade de *statements* nos “métodos antigos” diminuiu, assim como a quantidade de linhas;
- A classe “Loja” possui 3 métodos;
- Variáveis locais do “novo método” foram renomeadas comparadas as variáveis do “método antigo”.

❖ Antes do *Move Method*

```
30
31    public Compra comprar(List<Produto> listaProdutos, double desconto) throws CompraInvalidaException{
32
33        if(listaProdutos.size() < produtosEmEstoque.size()){
34
35            double total = 0;
36            for(Produto produto : listaProdutos){
37
38                total += produto.getPreco();
39            }
40
41            double valorDescontado = total * desconto;
42
43            double totalComDesconto = total - valorDescontado;
44
45            return new Compra(listaProdutos, totalComDesconto);
46        }
47
48        throw new CompraInvalidaException();
49    }
50
51
52    public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
53        return comprar(listaProdutos, 0.1);
54    }
55
56    public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
57        return comprar(listaProdutos, 0.05);
58    }
59
```

Figura 2.2.3.d: Exemplo de método que pode estar usando mais características de outra classe.

De acordo com a Figura 3 e o UML apresentado acima, os métodos:

- ❖ `comprar(List<Produto> produtos, double desconto)`
- ❖ `comprarAvista(List<Produto> produtos)`
- ❖ `comprarAPrazo(List<Produto> produtos)`

Estão implementados na classe “Loja” sendo que está retornando uma instância da classe “Compra”, esse método recebe parâmetros que são atributos da classe “Compra”, ou seja, “*o método está usando mais características de outra classe*”, podendo então ser aplicado o *Move Method.*

❖ **Depois do *Move Method***

Uma das alternativas de mudança, é mover o método para a classe “Compra”, mas como o atributo “produtosEmEstoque” pertence à classe “Loja”, a implementação ficou a seguinte:

❖ Classe “Loja”

```
public Compra comprar(List<Produto> listaProdutos, double desconto) throws CompraInvalidaException{
    if(listaProdutos.size() < produtosEmEstoque.size()){

            Compra compra =  new Compra(listaProdutos, desconto);
            compra.comprar();
            return compra;
    }
    throw new CompraInvalidaException();

}

public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
    return comprar(listaProdutos, 0.1);
}

public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
    return comprar(listaProdutos, 0.05);
}
```

Figura 2.2.3.e: Exemplo após a aplicação do *Move Method*

Uma das alternativas, foi manter o método

❖ `comprar (List<Produto> produtos, double desconto)`

Na classe “Loja”, porém toda a lógica de negócio da operação “comprar” foi movida para a classe “Compra”, como apresentará a figura abaixo. Sendo que na classe “Loja”, ficou a parte de verificação se há produtos disponíveis na “Loja”.

❖ Mudanças e Métricas causadas pelo Refatoramento:

- A quantidade de *statements* do método “comprar” da classe “Loja” diminuiu, assim como a quantidade de linhas;
- A classe “Loja” também ficou menos acoplada da classe “Compra”, aumentando assim a coesão entre ambos.

```java
1  package br.ufal.ic.easy.refactoring.examples.after.compra;
2
3⊕ import java.util.List;
7
8  public class Compra {
9
10     private double preco;
11     private List<Produto> produtos;
12     private float desconto;
13
14
15
16⊖    public Compra(List<Produto> produtos, float desconto){
17         this.produtos = produtos;
18         this.desconto = desconto;
19     }
20
21⊖    public Compra(List<Produto> produtos) {
22         // TODO Auto-generated constructor stub
23     }
24
25
26⊖    public void comprar(){
27
28        double total = 0;
29        for(Produto produto : produtos){
30
31            total += produto.getPreco();
32        }
33
34         double valorDescontado = total * desconto;
35         preco = total - valorDescontado;
36     }
37
```

Na figura ao lado, vemos o um fragmento da classe “Compra”, como pode ser observado o método “comprar” foi movido para esta classe, sendo que não precisava mais da passagem de parâmetros, pois foram substituídos por atributos da classe como “produtos”, “preco” e “desconto”.

Figura 2.2.3.f: Corpo do método que foi movido para outra classe

- Mudanças e Métricas causadas pelo Refatoramento:
  - A classe “Compra” ficou com 4 métodos (contando com os construtores);
  - A classe “Compra” ficou com mais um atributo “desconto”.

- Antes do *Rename Method*

```java
public Compra comprar(List<Produto> listaProdutos, double desconto) throws CompraInvalidaException{
    if(listaProdutos.size() < produtosEmEstoque.size()){

           Compra compra =  new Compra(listaProdutos, desconto);
           compra.comprar();
           return compra;
    }
    throw new CompraInvalidaException();

}

public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
    return comprar(listaProdutos, 0.1);
}

public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
    return comprar(listaProdutos, 0.05);
}
```

Figura 2.2.3.g: Exemplo de método que pode o nome do método não condiz com sua real função

O método “comprar” da classe “Loja” pode ser renomeado, pois toda a lógica da operação “comprar” pertence a classe “Compra”.

❖ Depois do *Rename Method*

```
29
30
31⊖    public Compra gerarCompra(List<Produto> listaProdutos, double desconto) throws CompraInvalidaException{
32        if(listaProdutos.size() < produtosEmEstoque.size()){
33
34            Compra compra =  new Compra(listaProdutos, desconto);
35            compra.comprar();
36            return compra;
37        }
38        throw new CompraInvalidaException();
39
40    }
41
42⊖    public Compra comprarAvista(List<Produto> listaProdutos) throws CompraInvalidaException{
43        return gerarCompra(listaProdutos, 0.1);
44    }
45
46⊖    public Compra comprarAPrazo(List<Produto> listaProdutos) throws CompraInvalidaException{
47        return gerarCompra(listaProdutos, 0.05);
48    }
49
50
```

Figura 2.2.3.h: Método após a renomeação

A sugestão é o método possa ser renomeado para “gerarCompra”, já que de fato é nova real função do método.

❖ Mudanças e Métricas causadas pelo Refatoramento:

- Os métodos “comprarAvista” e “comprarAPrazo” tiveram que sofrer alterações, pois o método utilizado por ambos foi renomeado.

2.2.4 Diminuição de statements em métodos causados por refatoramento

Como descrito na seção acima, mudanças no código podem ter várias motivações, dentre estas, o refatoramento, para este estudo, a mudança a ser investigada em métodos será, especificamente, a diminuição de *statements*, pois empiricamente é visto que esse tipo de mudança pode está ligada a vários refatoramentos. Nas próximas seções serão apresentadas: o processo de recuperação do histórico de métodos e a investigação de possíveis refatoramentos.

3. Fundamentação teórica sobre sistemas de controle de versão

Nesta seção será apresentado o processo e resultados utilizados para a coleta do histórico de métodos de softwares em desenvolvimento.

3.1 Sistema de controle de versão (VCS)

O controle de versão é um conceito utilizado durante o desenvolvimento de software que permite salvar a cópia de cada versão em um repositório armazenado em um servidor, compartilhamento dessas cópias entre computadores e registro das mudanças realizadas permitindo assim consultas sobre quem e quando foi realizada a mudança. Há vários sistemas que são utilizados para o controle de versão, que são chamados de sistema de controle de versão, há um vocabulário comum entre esses sistemas, segue abaixo o conceito dos termos mais relevantes para esse estudo, de acordo com [5]:

- **Repositório** é onde os dados atuais e históricos dos arquivos são armazenados, geralmente em um servidor.
- **clone:** é quando um repositório é criado contendo o mesmo histórico de outro repositório, porém dois repositórios só são considerados clones se eles são mantidos sincronizados, contendo as mesmas versões.
- **branch (ramo)**: é conjunto de arquivos sob controle de versão pode ser ramificado, dessa forma duas ou mais cópias desses arquivos podem ser desenvolvidos em velocidades e maneiras diferentes, independentemente uns dos outros;
- **Commit**: é escrever ou mesclar as alterações feitas no repositório local[10] para o repositório remoto[11].
- **Pull:** é quando o repositório local recebe as modificações do repositório remoto.
- **Push:** é quando o repositório local envia as modificações locais para repositório remoto.
- **Merge:** é a operação na qual dois ou mais conjuntos de alterações são aplicados a um arquivo ou conjunto de arquivos.
- **Conflito:** ocorre quando diferentes alterações são feitas no mesmo documento e o sistema não consegue conciliar as alterações.
- **Diff:** operação que permite mostrar as diferenças entre arquivos de diferentes

versões.

Os sistemas de controle de versão podem ser centralizados e descentralizados, segue abaixo as diferenças entre os dois tipos de acordo com [14][15]:

- **Centralizado:** quando há uma única cópia remota do repositório e todas as alterações das cópias locais são automaticamente sincronizadas à cópia remota. A centralização do sistema pode ser representada pela figura abaixo:

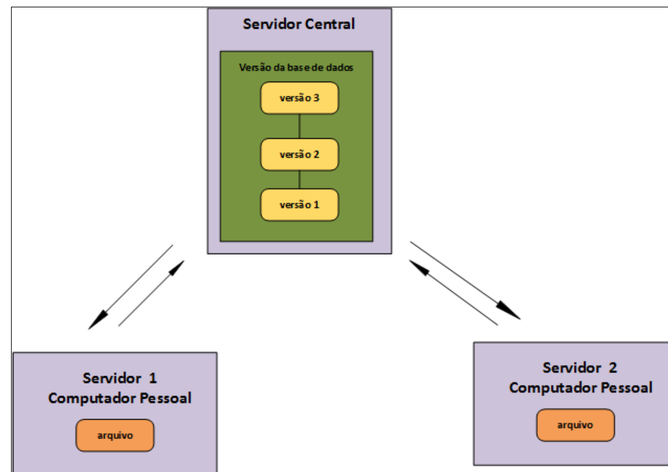

Figura 3.1.a: Controle de versão centralizado. Imagem baseada em [14].

Alguns sistemas de controle de versão centralizado são CVS, Subversion (ou SVN[12]) e Perforce[13].

10.Local: utilizado na literatura para indicar o repositório que está sofrendo alterações no computador (servidor) pelo desenvolvedor.

11.Remoto: utilizado na literatura para indicar que o repositório que vai receber as alterações feitas por um repositório de outro computador (servidor).

12. SVN: disponível em: https://subversion.apache.org/

13. Perforce: disponível em: https://www.perforce.com/

- **Descentralizado:** quando o sistema permite que histórico de todas as versões não fiquem concentrados em um único repositório central, dessa forma as cópias podem ter o histórico completo e podem trocar alterações entre si, como representa a figura abaixo.

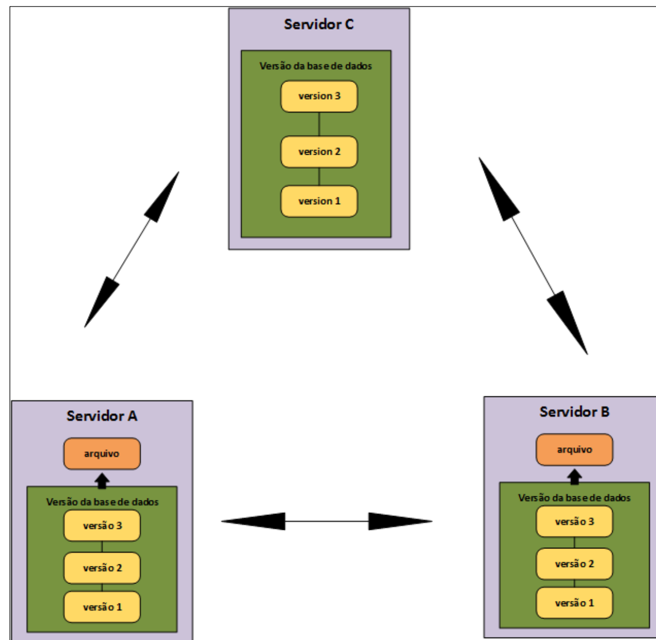

Figura 3.1.b: Controle de versão descentralizado. Imagem baseada em [14]

Exemplos de sistemas descentralizados são: Git, Mercurial[14].

3.2 Introdução à tecnologia Git

O sistema de controle de versão escolhido para esse estudo foi o Git[15], pois por ser um sistema descentralizado é possível obter histórico completo a partir de repositórios locais e aplicar técnicas de mineração de dados para coletar informações sobre mudanças aplicadas em métodos.

14. Mercurial: disponível em: https://www.mercurial-scm.org/

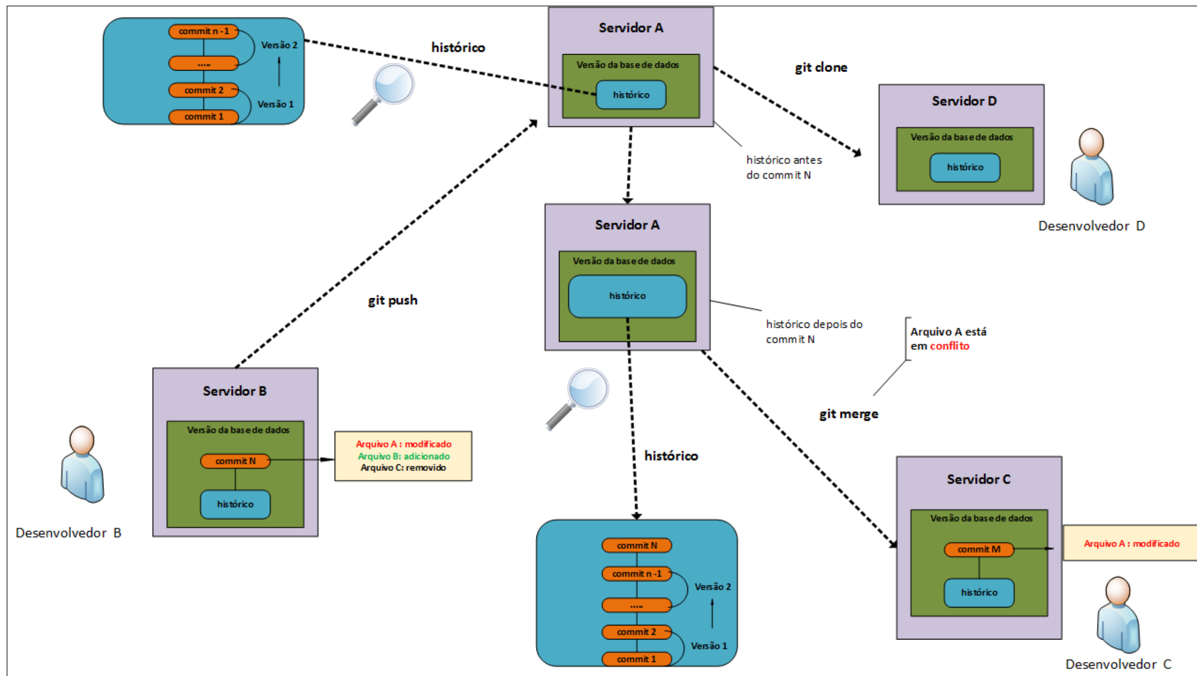

Figura 3.2: Fluxo de desenvolvimento usando Git

A figura acima apresenta o fluxo de desenvolvimento usando Git, com o os principais comandos utilizados entre os desenvolvedores.

4. Materiais e métodos utilizados para a recuperação do histórico de softwares

Essa seção será apresentada a metodologia utilizada para recuperar o histórico de desenvolvimento de cada software, a fim de encontrar armazenar, em uma base dados, os métodos e a quantidade de statements de cada método durante o desenvolvimento.

4.1 Técnicas e ferramentas utilizadas para recuperação do histórico de software

Nesta seção será apresentada uma visão geral sobre as ferramentas utilizadas para minerar o histórico do desenvolvimento de softwares.

4.1.1 Metric Miner

Em projetos Git, cada mudança no histórico do repositório relatada pelos desenvolvedores é chamada de *commit*, desta forma foi preciso coletar todos os *commits* de cada projeto para assim obter um histórico mais completo possível de mudanças, para isto utilizamos uma ferramenta chamada Metric Miner[7], com a API disponibilizada em Java, utiliza comandos Git para consultar histórico de repositório.

4.1.1.1 Visão Geral sobre AST e JDT:

O Metric Miner recupera todos os arquivos daquele determinado “momento” do software, ou seja, o *commit*, porém é preciso um mecanismo em conjunto para mapear estruturas dos arquivos Java, tais estruturas como: pacotes, classes, atributos, métodos, interfaces. Para isso, é utilizado a tecnologia JDT [16][17], é uma API desenvolvida pelo Eclipse[15], que significa Java *Development Tools*, ou seja, é uma ferramenta para desenvolver aplicativos Java, que fornece bibliotecas para criar e manipular código-fonte de projetos Java. O JDT pode ser instalado como plugin[16] da Eclipse IDE ou pode ser adicionado como biblioteca externa (adicionando o .jar por exemplo), para esse estudo o JDT foi adicionado como uma dependência Maven[17], como segue a imagem abaixo:

```xml
<dependency>
    <groupId>br.usp</groupId>
    <artifactId>metricminer</artifactId>
    <version>2.4.0</version>
</dependency>
```

Figura 4.1.1.1.a: Dependência Maven do Metric Miner

```xml
<dependency>
    <groupId>org.eclipse.jdt</groupId>
    <artifactId>core</artifactId>
    <version>3.3.0-v_771</version>
</dependency>
```

Figura 4.1.1.1.b: Dependência Maven do JDT

O JDT cria e manipula códigos fonte de Java através de duas

maneiras: *Java Model* e *Abstract Syntax Tree* (AST) [17][18],

- **Java Model** é responsável por mapear cada componente do projeto Java para a árvore de estrutura de hierarquia de componentes.
- **AST a** árvore abstrata sintática é a representação do código criada pelo compilador após a análise sintática. A estrutura dessa árvore é a seguinte:

  - **Nós**: são diretamente valorados em seus símbolos terminais[18], ou seja: palavras reservadas da linguagem, variáveis, operadores lógicos, aritméticos e booleanos, dentre outros.
  - **Arestas**: indica o relacionamento entre os nós.

Cada compilador possui sua própria AST. Na figura abaixo, segue a AST do JDT e é chamada de Eclipse AST, como destaque para a estrutura de métodos e *statements* que serão as mais utilizadas nesse estudo.

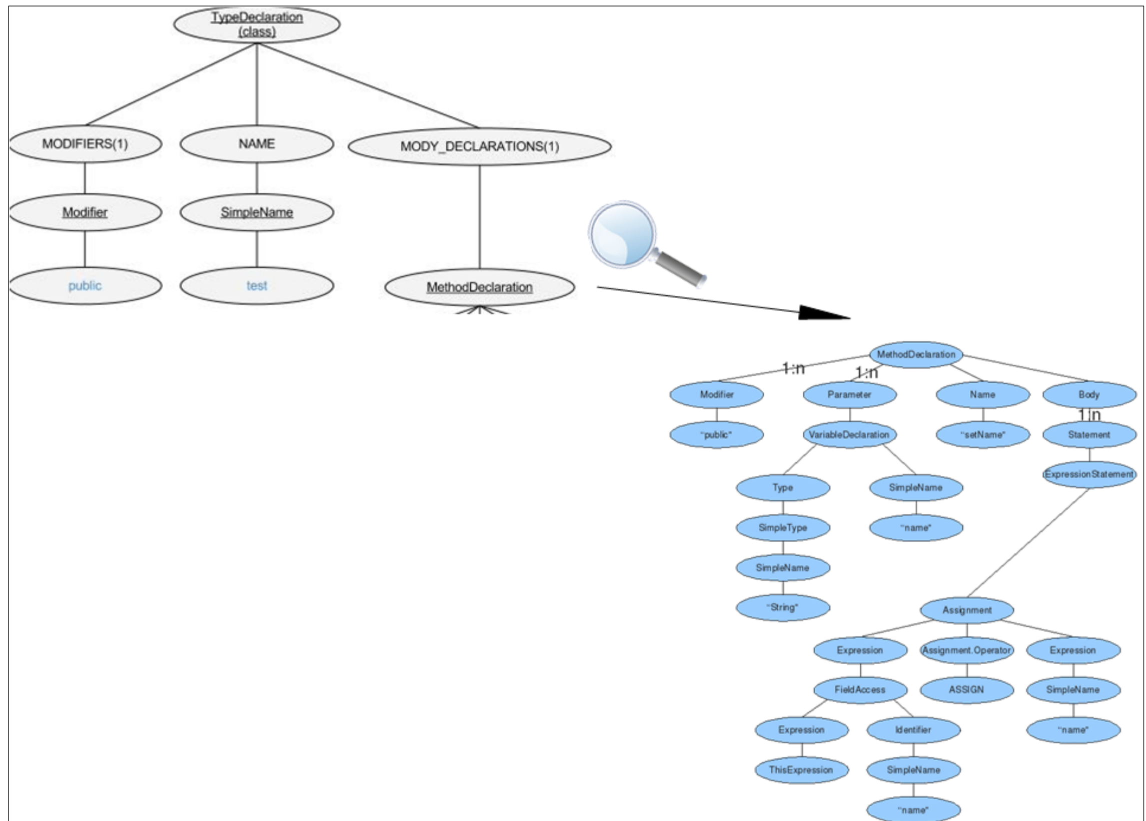

Figura 4.1.1.1.c: Eclipse AST *

*Imagem baseada em :

http://sahits.ch/blog/blog/2008/05/23/yaat-yet-another-ast-tutorial/#link3

http://www.programcreek.com/2012/04/represent-a-java-file-as-an-astabstract-syntax-tree/

15. Eclipse: disponível em: https://www.eclipse.org/
16. Plugin: disponível em: https://pt.wikipedia.org/wiki/Plug-in
17. Maven: disponível em: https://maven.apache.org/
18. Símbolos terminais: disponível em: https://pt.wikipedia.org/wiki/S%C3%ADmbolos_terminais_e_n%C3%A3o_terminais

De acordo, com a documentação do JDT, segue as classes abaixo que implementam a super classe *Statement,* e que descrevem os tipos de statments que serão aceitos nesse estudo, cada uma é encontrada na AST através de uma expressão regular:

Tabela 1: Statments implementados pelo JDT

Statement	Expressão Regular
AssertStatement	`AssertStatement:` `assert Expression [ : Expression ] ;`
Block	`Block:` `{ { Statement } }`
BreakStatement	`BreakStatement:` `break [ Identifier ] ;`

ConstructorInvocation	`ConstructorInvocation:`
	`[ < Type { , Type } > ]`
	`this ( [ Expression { , Expression } ] ) ;`

ContinueStatement	`ContinueStatement:` `continue [ Identifier ] ;`
DoStatement	`DoStatement:` `do Statement while ( Expression ) ;`
EmptyStatement	`EmptyStatement:` **;**

EnhancedForStatement}	`EnhancedForStatement:`
	`for ( FormalParameter : Expression )`
	`Statement`

ExpressionStatement	`ExpressionStatement:` StatementExpression **;**

<table>
<tr><td>ForStatement</td><td><pre>ForStatement:
    for (
                     [ForInit];
                     [Expression] ;
                     [ForUpdate])
                     Statement
 ForInit:
             Expression { , Expression }
 ForUpdate:
             Expression { , Expression }</pre></td></tr>
<tr><td>IfStatement</td><td><pre>IfStatement:
    if (Expression) Statement [else
Statement]</pre></td></tr>
<tr><td>LabeledStatement</td><td><pre>LabeledStatement:
    Identifier : Statement</pre></td></tr>
<tr><td>ReturnStatement</td><td><pre>ReturnStatement:
    return [Expression] ;</pre></td></tr>
<tr><td>SuperConstructorInvocation</td><td><pre>SuperConstructorInvocation:
     [Expression .]
         [< Type { , Type } >]
         super ([Expression { , Expression
}]) ;</pre></td></tr>
<tr><td>SwitchCase</td><td><pre>SwitchCase:
             case Expression :
             default :</pre></td></tr>
<tr><td>SwitchStatement</td><td><pre>SwitchStatement:
             switch (Expression)
                     { { SwitchCase |
Statement } } }
 SwitchCase:
             case Expression :
             default :</pre></td></tr>
<tr><td>SynchronizedStatement</td><td><pre>SynchronizedStatement:
    synchronized (Expression) Block</pre></td></tr>
<tr><td>ThrowStatement</td><td><pre>ThrowStatement:
    throw Expression ;</pre></td></tr>
</table>

TryStatement	`TryStatement:`
	`try [ ( Resources ) ]`
	`Block`
	`[ { CatchClause } ]`
	`[ finally Block ]`

TypeDeclarationStatement	`TypeDeclarationStatement:`
	`TypeDeclaration`
	`EnumDeclaration`

VariableDeclarationStatement	`VariableDeclarationStatement:`
	`{ ExtendedModifier } Type VariableDeclarationFragment`
	`{ , VariableDeclarationFragment } ;`

WhileStatement	`WhileStatement:` `while ( Expression ) Statement`

Para utilizar a Eclipse AST, o JDT fornece uma classe chamada ASTVisitor com funções que permitem:

- Analisar qualquer fonte de informação de código da AST;
- Manipular AST para inserir/excluir código.

A “ASTVisitor”[19] é responsável por “visitar” cada nós da AST, essa função é realizada através do método “visitor()”[19] e dentro desse método pode ser implementado qualquer manipulação com o nó “visitado”, para utilizar as funções fornecidas por “ASTVisitor” é preciso criar uma classe que herde da classe “ASTVisitor” e implementar o método “visitor()”. Na próxima seção será apresentada a implementação da ferramenta que utiliza a API do Metric Miner para coletar o histórico de cada método e a quantidade de *statements* de cada método por *commit.*

4.1.2 Implementação *Extract Minerator Tool* utilizando Metric Miner API

Abaixo, será apresentada a implementação da ferramenta *Extract Minerator Tool,* que foi utilizada para recuperar o histórico dos métodos de cada repositório Git e posteriormente criar uma base de dados com o histórico de quantidade de *statements* de cada método por *commit*.

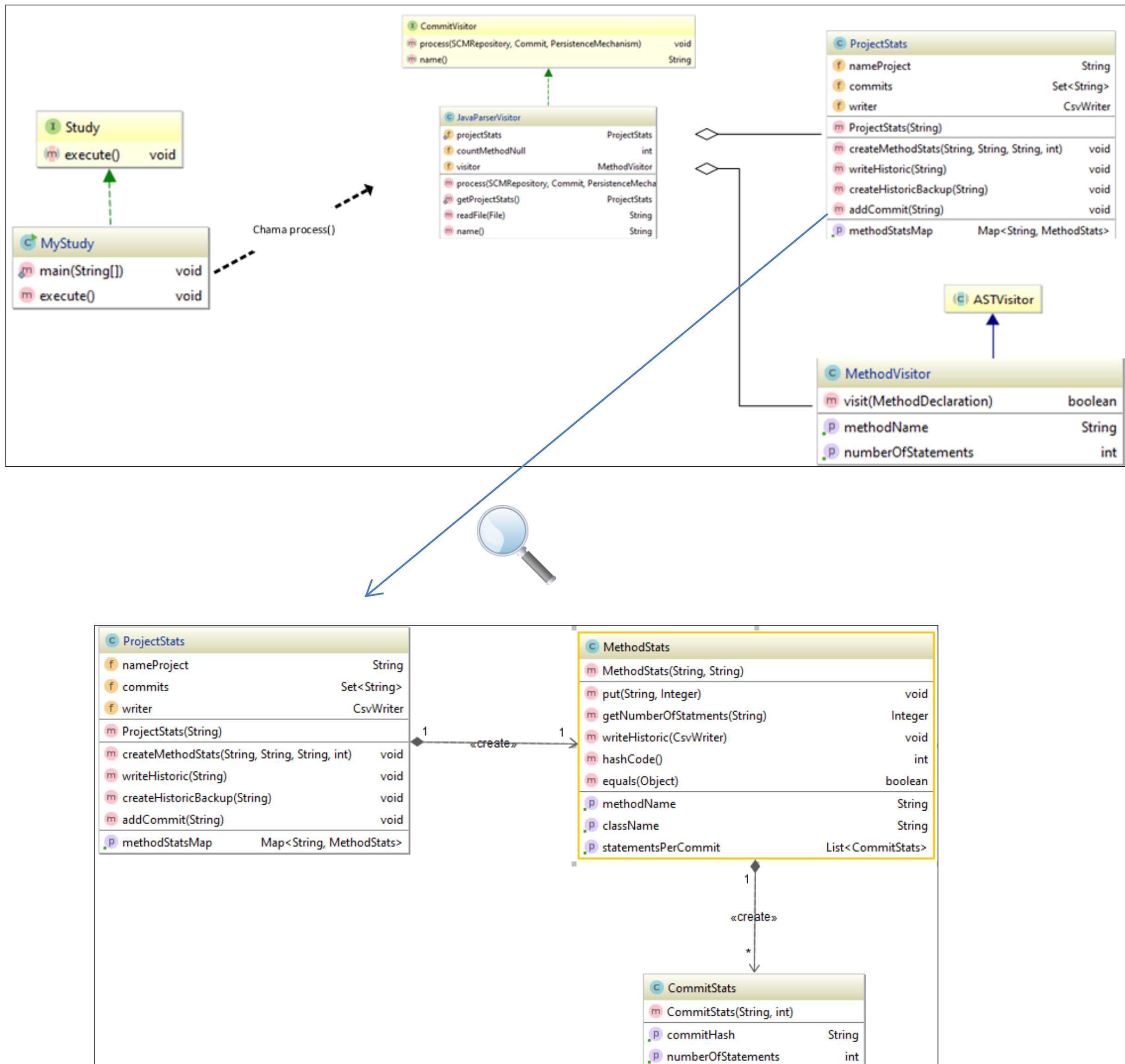

Figura 4.1.2.a: UML "*Extract Minerator Tool"*

Segue o comentário sobre a implementação das principais classses:

```
1   package br.ufal.ic.method.historic.minerator;
2
3   import ...
12
13  public class MyStudy implements Study{
14
15
16      public static void main(String[] args) { new MetricMiner2().start(new MyStudy()); }
19
20      public void execute() {
21          JavaParserVisitor visitor = new JavaParserVisitor();
22          new RepositoryMining()
23          .in(GitRepository.singleProject( path: "path/projeto.git"))
24          .through(Commits.all())
25          .process(visitor, new CSVFile(Utils.RESOURCE))
26          .mine();
27
28          try {
29
30              JavaParserVisitor.getProjectStats().createHistoricBackup( path: "path/historico_metodos.csv");
31          } catch (IOException e) {
32              // TODO Auto-generated catch block
33              e.printStackTrace();
34          }
35      }
36
37  }
38
```

Figura 4.1.2.b: Classe “*MyStudy*”

A classe “MyStudy” implementa a interface “Study”[19],“MyStudy” é a responsável pela inicialização da ferramenta.

- Na linha 16, é feita a inicialização através do método “start(Study study)”[19] que chama o método “execute”[19] da interface “Study”;
- Na linha 20, está declarado o método “execute”[19] da interface “Study” que por sua vez está sendo implementado pela classe “MyStudy”;
- Na linha 21 está a declaração da instância da classe “JavaParserVisitor”, classe que será apresentada na figura 18, e que implementa a interface “CommitVisitor”[19];
- Na linha 22, é iniciado o processo de mineração de dados do repositório, através da declaração da instância da classe “RepositoryMining”[19];
- Na linha 23, indica-se o diretório onde está o repositório Git;
- Na linha 24, indica-se “o que quer” minerar no repositório, no caso todos os *commits*.
- Nas linhas 25 e 26, inicializa-se o processo de mineração através do método “process(ASTVisitor visitor, CSVFile file)”[19], no qual passa-se como parâmetro a instância do “visitor” e o diretório do arquivo CSV para a escrita do histórico padrão que o Metric Miner “escreve”, porém

para esse estudo, esse arquivo CSV não será utilizado como base de dados, pois a base de dados terá um formato diferente do formato padrão;

- Na linha 30, é chamado o método "createHistoricBackup(String pathHistoric)", esse método foi implementado para criar a base de dados no formato desejado para esse estudo, passando todo o histórico de métodos coletado pelo "visitor", como será apresentado nas próximas figuras.

```
19  public class JavaParserVisitor implements CommitVisitor {
20
21      private static ProjectStats projectStats = new ProjectStats( nameProject: "ProjectName");
22      int countMethodNull = 0;
23      MethodVisitor visitor;
24
25      public void process(SCMRepository repo, Commit commit, PersistenceMechanism writer) {
26          try {
27              repo.getScm().checkout(commit.getHash());
28              List<RepositoryFile> files = repo.getScm().files();
29
30              for(RepositoryFile file : files) {
31                  if(!file.fileNameEndsWith( suffix: "java")) continue;
32                  File soFile = file.getFile();
33                   visitor = new MethodVisitor();
34
35                  new JDTRunner().visit(visitor, new ByteArrayInputStream(readFile(soFile).getBytes()));
36
37                  String fileName = soFile.getPath();
38                  String nameMethod = visitor.getMethodName();
39
40                  if(visitor.getMethodName() == null){
41                      countMethodNull ++;
42                      nameMethod = "invalid-name-method-" + String.valueOf(countMethodNull);
43                  }
44
45                  projectStats.addCommit(commit.getHash());
46                  projectStats.createMethodStats(commit.getHash(),
47                                                 fileName,
48                                                 nameMethod,
49                                                 visitor.getNumberOfStatements());
50
51              }
52          } finally { repo.getScm().reset(); }
```

Figura 4.1.2.c: Clase "*JavaParserVisitor*"

Como já foi mencionado, a classe "JavaParserVisitor" implementa a interface "CommitVisitor", que por sua vez, é a interface responsável por "visitar" cada *commit* do repositório através do método iterativo inicializado na linha 24.

- Na linha 26, o repositório é redirecionado (ou seja, o repositório regride para estado do "commit" da vez);
- Na linha 27, é recuperado todos os arquivos daquele "commit";
- Na linha 29 inicia-se uma iteração por todos os arquivos recuperados;
- Na linha 30, é ignorado todos os arquivos que não sejam ".java";
- Na linha 31, é criado a instância do arquivo "a ser visitado";
- Na linha 32, é criado a instância do "visitante de métodos",

através da classe "MethodVisitor", essa classe herda da classe "ASTVisitor" e será apresentada na próxima figura;

- Na linha 34, é criado uma thread[20], através da instância do "JDTRunner"[19], basicamente, essa instância é responsável por "converter" o arquivo em uma estrutura de árvore JDT, separado os métodos em cada nó da árvore, para que assim o "methodVisitor" possa percorrer cada método;
- Na linha 36 e 37, é capturado o nome do arquivo Java (classe ou interface) e o nome do método;
- Na linha 39 a 41, é verificado se a assinatura do método foi recuperada corretamente, pois como será apresentado posteriormente, houve alguns problemas em relação a isto. Caso não foi recuperada, então é atribuído uma assinatura padrão "invalid-name-mehod-x", cujo "x" é a quantidade de métodos com "nome inválidos", ou seja, métodos que sua assinatura não foram recuperadas.
- Nas linhas 44 e 45, a instância "projectStats" (declarada na linha 21) da classe "ProjectStats", salva em uma estrutura de dados todas as informações recuperadas, tais como: o *commit*, o nome da classe ou interface, a assinatura do método e a quantidade de *statements* de cada método, após toda a recuperação do histórico é escrito na base de dados. O fluxo envolvendo a classe "ProjectStats" será descrita na seção 4.3

.

```
8   public class MethodVisitor extends ASTVisitor {
9
10      private String methodName;
11      private int numberOfStatements = 0;
12
13      public boolean visit(MethodDeclaration node) {
14
15          String[] parameter = {"("};
16          if(node.parameters() != null){
17              node.parameters().forEach( n -> {
18                  parameter[0] +=  n.toString() + ",";
19                  });
20          }
21          if(parameter[0].length() > 1){
22              parameter[0] = parameter[0].substring(0, parameter[0].length() - 1);
23          }
24
25          parameter[0] += ")";
26          methodName = node.getName() + parameter[0];
27
28          if(node.getBody() != null){
29              if(node.getBody().statements() != null){
30                  numberOfStatements = node.getBody().statements().size();
31              }
32          }
33          if(numberOfStatements <= 0){
34              numberOfStatements = 0;
35          }
36
37          return super.visit(node);
38      }
39
```

Figura 4.1.2.d: Classe *"Method Visitor"*

Como mencionado anteriormente, a classe "MethodVisitor", é responsável por visitar cada método, como descrito abaixo:

- Na linha 13, houve a declaração do método: "visitor(MethodDeclaration node)"
- Nas linhas 15 até 25, foi feito a recuperação de cada parâmetro do método, para posteriormente unir ao nome do método (pois a AST trata cada parâmetro como um nó do método);
- Na linha 26, foi feito a concatenação entre o nome do método e os parâmetros, formando assim a assinatura do método sendo atribuído em "methodName";
- Nas linhas 28 até 35, foi feito a recuperação da quantidade de *statements,* desse método nesse *commit*. Checando antes se a quantidade de *statements* foi recuperado com sucesso, caso não seria substituído por zero *statements,* sendo atribuído em "numberOfStatements".

19. Código nativo do Metric Miner

20. Thread: https://pt.wikipedia.org/wiki/Thread_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o)

4.2 Visão Geral sobre o histórico de desenvolvimento dos softwares escolhidos

Essa subseção apresentará os softwares escolhidos para esse estudo e uma visão geral sobre o domínio e o histórico de desenvolvimento de cada software.

4.2.1 Domínio de *softwares* escolhidos.

Para a coleta sobre o histórico de métodos foram escolhidos 6 repositórios, com controle de versão através da tecnologia Git, esses repositórios pertencem ao *startup* Clip[32], uma marca da empresa privada e alagoana Meyer[33] (trabalha com venda e aluguel de impressoras multifuncionais), estes possuem até 3 anos de desenvolvimento, sendo os mais relevantes em uso, atualmente, são todos desenvolvidos em Java. Esses repositórios foram escolhidos, porque a autora desse estudo faz estágio nessa empresa e para esse estudo a proximidade com os desenvolvedores foi de suma importância para coleta de informações e por conseguinte apresentação de resultados para melhor análise de desenvolvimento dos softwares envolvidos. Portanto, os repositórios, ou seja, as aplicações escolhidas foram: Clip, Clip OCR, Clip Commons, Painel Admin, Meyer Control e Document Generator. Segue abaixo, as descrições de domínios de cada aplicação.

4.2.1.1 Clip:

É um *web service*[23] desenvolvido em Java com uma arquitetura REST[24], funciona como um GED[25] (Gerenciamento Eletrônico de Documentos ou Gestão Eletrônica de Documentos).

4.2.1.2 Clip OCR

Utilizando a tecnologia OCR[26] (Reconhecimento ótico de caracteres), é um serviço externo da aplicação Clip, utilizado realizar para extrair textos de arquivos em formatos ".pdf" [27] e ".jpeg" [28] para operações como leitura e busca de texto.

21.Amazon AWS: disponível em: https://aws.amazon.com/pt/

22. framework: disponível: https://pt.wikipedia.org/wiki/Framework

23.*web service*: disponível em: https://pt.wikipedia.org/wiki/Web_service

4.2.1.3 Clip Commons

É uma aplicação auxiliar para as demais aplicações da marca Clip, como o próprio nome diz, realiza operações podem ser reutilizadas, ou seja, são comuns, para outras aplicações, essas operações podem estar relacionadas criação e tratamento de banco de dados, ligação com serviços externos como Amazon AWS[21], framerwork[22] e etc.

4.2.1.4 Meyer Control

É um *web service* interno da empresa Meyer, utilizado para o gerenciamento de vendas e aluguéis de impressora, controle de suprimentos e contratos

4.2.1.5 Painel Admin

É um *web service* interno da Marca Clip, utilizado para controle de vendas do software Clip, logo realiza funções administrativas como criação, atualização de vendedores, criação e atualização de clientes destes vendedores, relatórios de vendas e produção.

4.2.1.6 Document Generator

É um *web service* auxiliar do projeto Painel Admin, responsável pela geração de relatórios de venda e produção.

\

---

24. REST: disponível em: https://pt.wikipedia.org/wiki/REST
25. GED: disponível em: https://pt.wikipedia.org/wiki/Gerenciamento_eletr%C3%B4nico_de_documentos
26. OCR: disponível em: https://pt.wikipedia.org/wiki/Reconhecimento_%C3%B3tico_de_caracteres
27. pdf: disponível em: https://pt.wikipedia.org/wiki/Portable_Document_Format
28. jpeg: https://pt.wikipedia.org/wiki/Joint_Photographic_Experts_Group

Segue abaixo o diagrama com o fluxo geral entre os softwares:

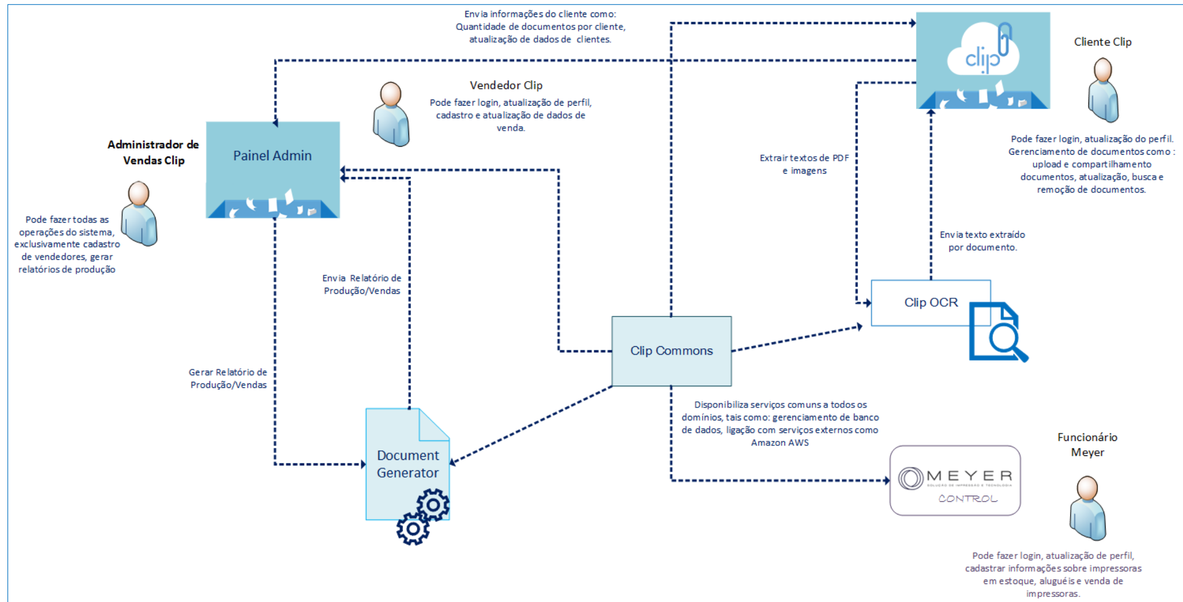

Figura 4.2.1: Fluxo projetos Clip

Dessa forma, iniciou-se o processo de coleta de histórico dos repositórios, que será apresentada na próxima seção, como visão geral da estrutura de cada repositório, foi coletada a quantidade de *commits* e número atual[29] de classes, essas métricas foram quantificadas através da ferramenta Understand (já mencionada anteriormente).

Tabela: Estrutura dos Projetos[29]

Nome Projeto	Número de Classes	Número de *Commits*
Clip	215	3492
Painel Admin	240	2038
Document Generator	13	73
Meyer Control	177	1782
Clip Commons	36	46
Clip OCR	37	72

29. Estes números, devem ser considerados a versão em produção, durante as datas 27/11/2016 até 23/12/2016.

Como é possível observar, os 3 maiores projetos (em quantidade de *commits* e classes), são justamente os 3 principais *web services*: Clip, Painel Admin, Meyer Control.

4.3 Base de dados com o histórico de métodos

Essa subseção apresentará o processo de construção da base de dados do histórico de métodos e uma visão geral do histórico obtido.

4.3.1 Processo de construção

Então iniciou-se o desenvolvimento do projeto *Open Source Extract Minerator*[30], que utilizando a API do *Metric Miner*, coleta e armazena o nome da classe, a assinatura do método, e a relação número de *statement* por *commit*, sendo armazenados todos os dados em uma base CSV[20], com a seguinte estrutura:

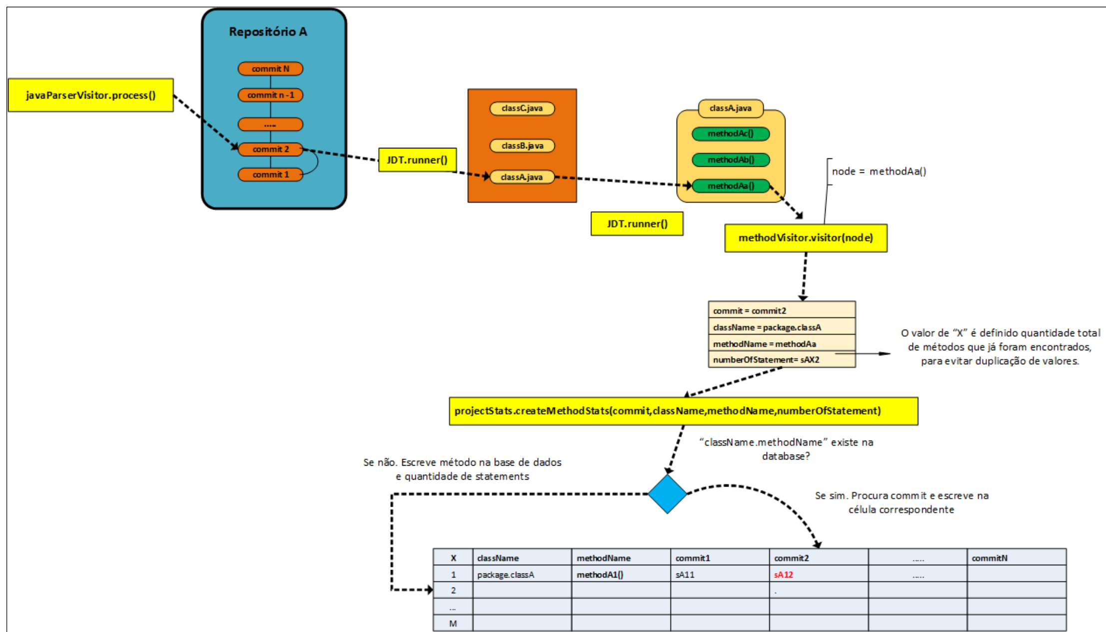

Figura 4.3.1: Processo de construção de histórico de métodos

O histórico de dados foi armazenado em arquivos de formato ".csv", por oferecer melhor visualização e manipulação de dados através de técnicas de mineração, para a integração com a ferramenta *Extract Minerator Tool* foi utilizada uma API do OpenCSV [20] que permite ter uma estrutura de dados semelhante a de uma tabela e assim fazer manipulação de dados, e só depois foi feita a escrita em arquivo, ganhando desta forma tempo e desempenho durante a coleta de dados.

4.3.2 Visão geral sobre o histórico de métodos obtido:

Durante a coleta de dados, foi observado que muitas instâncias da base de dados, tinham a coluna "methodName" vazia, pois o *Metric Miner* encontrou *commits* em que os arquivos estavam em conflito, não conseguindo assim obter a assinatura dos métodos, os gráficos abaixo apresentam, por projeto, o número métodos encontrados com assinatura e sem assinatura.

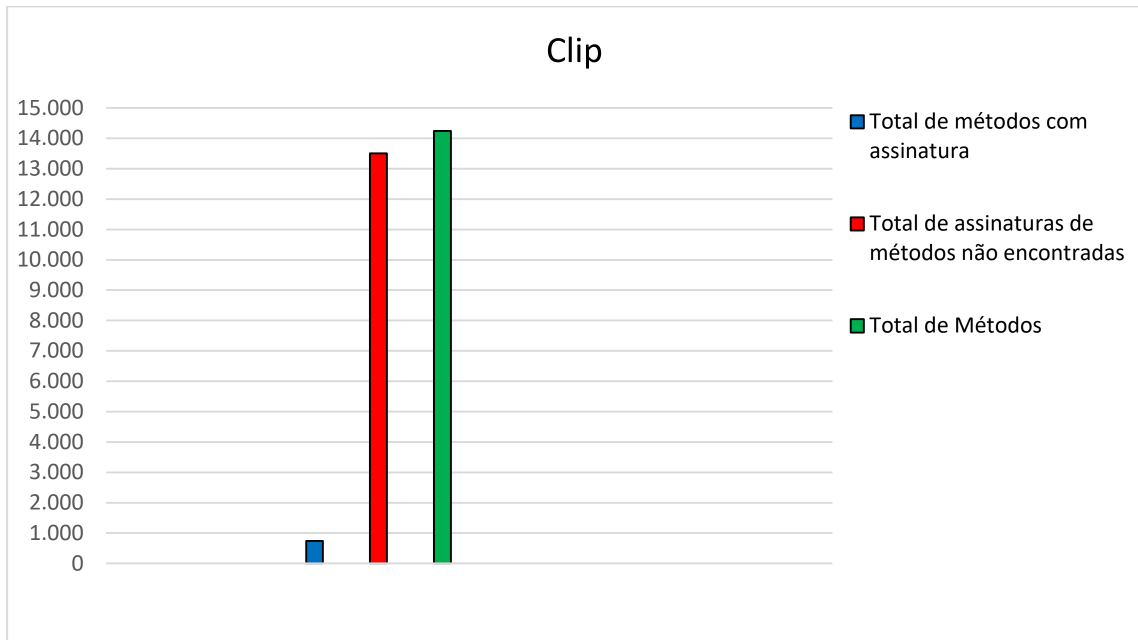

Figura 4.3.2.a: Total de métodos encontrados no projeto Clip

No projeto Clip, foram encontrados ao todo, 14.239 instâncias de métodos, porém 13.503 instâncias, não foi possível encontrar assinaturas e 736 métodos com assinaturas encontradas.

30. Extract Method Minerator: https://github.com/anacarlagb/extract-method-minerator.git

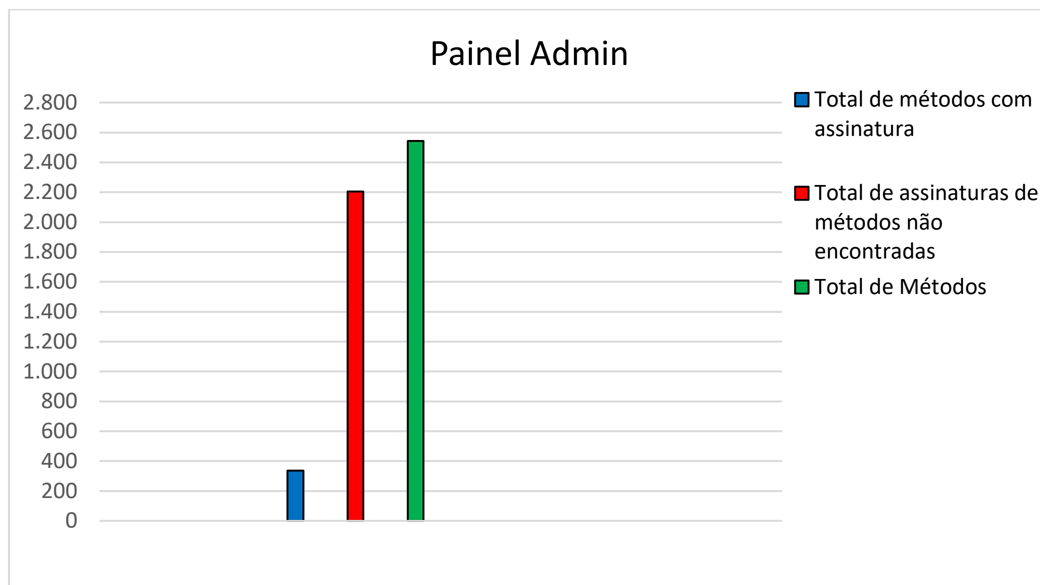

Figura 4.3.2.b: Total de métodos encontrados no projeto Painel Admin

---

No projeto Painel Admin, foi possível encontrar 2.543 instâncias de métodos, mas 2.206 são instâncias com assinaturas não encontradas e 337 com assinaturas encontradas.

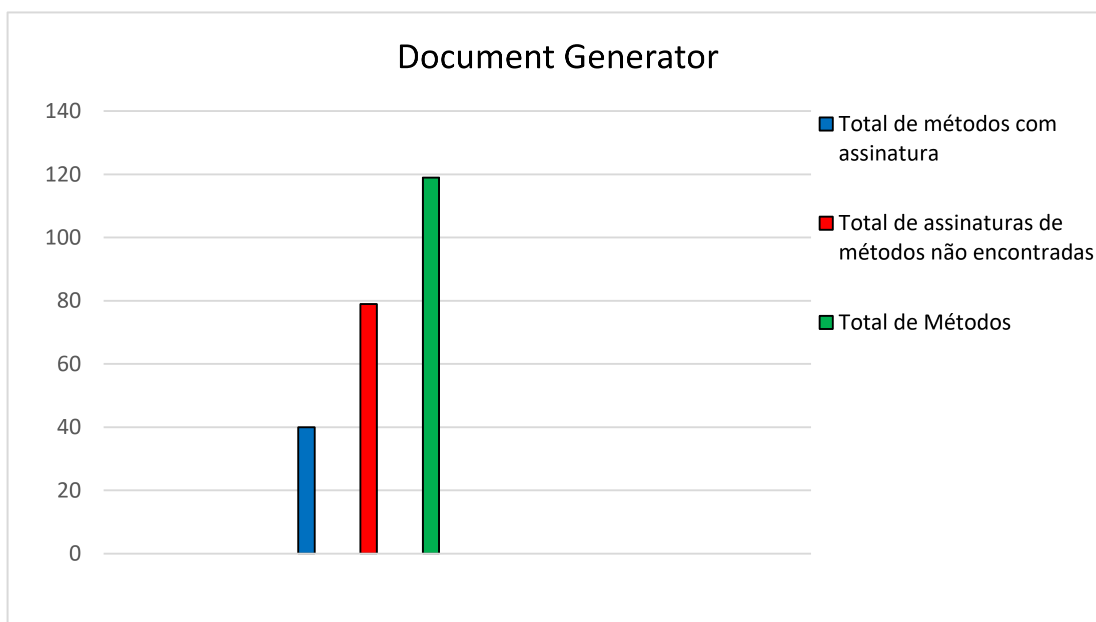

Figura 4.3.2.c: Total de métodos encontrados no projeto Document Generator

No projeto Document Generator foram encontrados 119 métodos, 79 métodos não foi possível encontrar assinatura e 40 métodos foram encontrados

com assinatura.

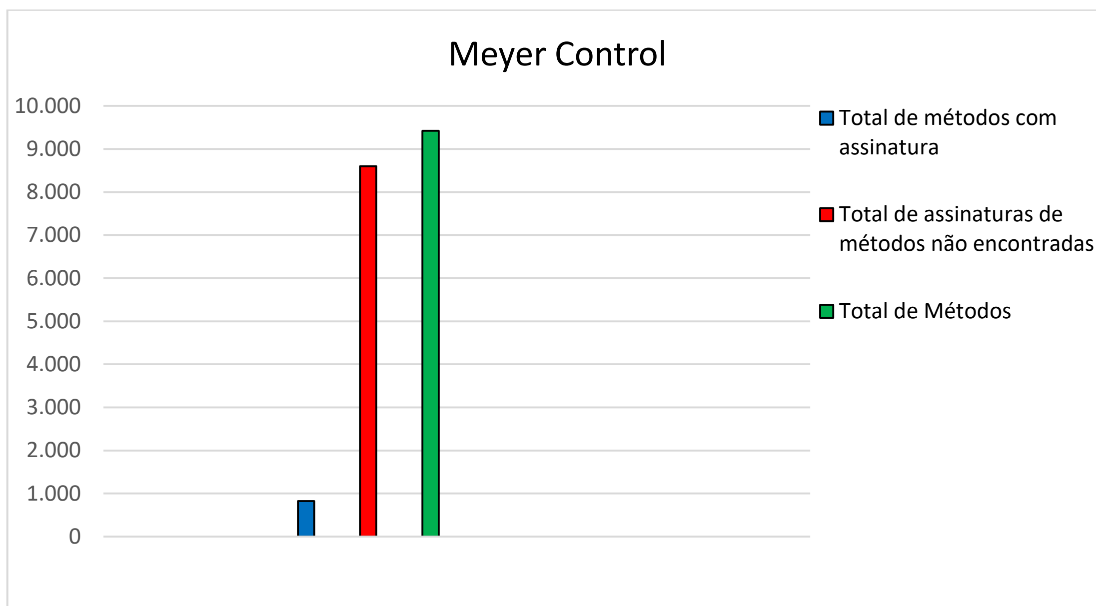

Figura 4.3.2.d: Total de métodos encontrados no projeto Meyer Control

No projeto Meyer Control, foram encontradas 8.601 instâncias sem assinaturas, 823 instâncias com assinaturas, totalizando 9.424 instâncias.

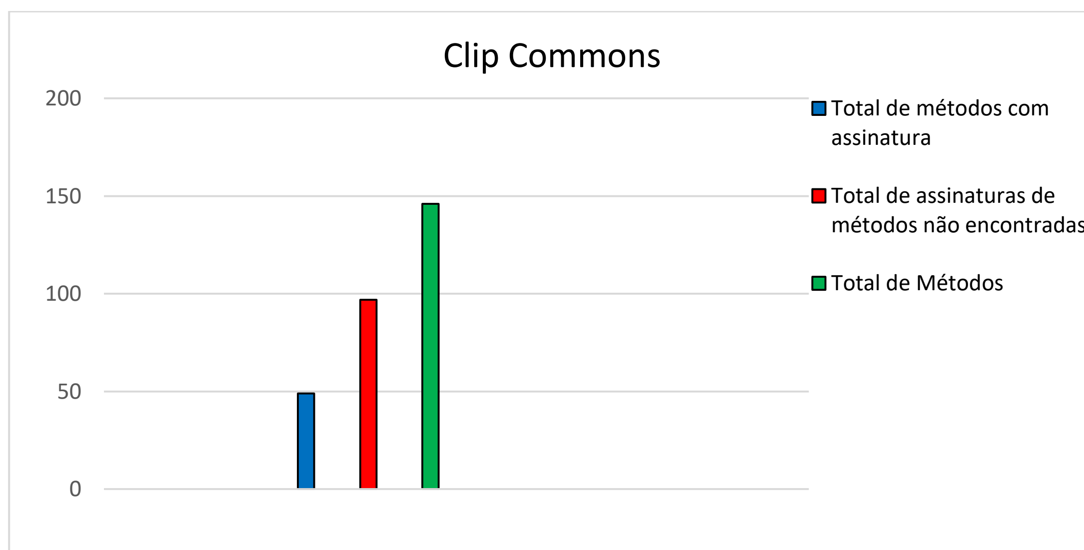

Figura 4.3.2.e: Total de métodos encontrados no projeto Clip Commons

No projeto Clip Commons, foi possível encontrar 146 instâncias de métodos, mas 97 são instâncias com assinaturas não encontradas e 49 com assinaturas encontradas.

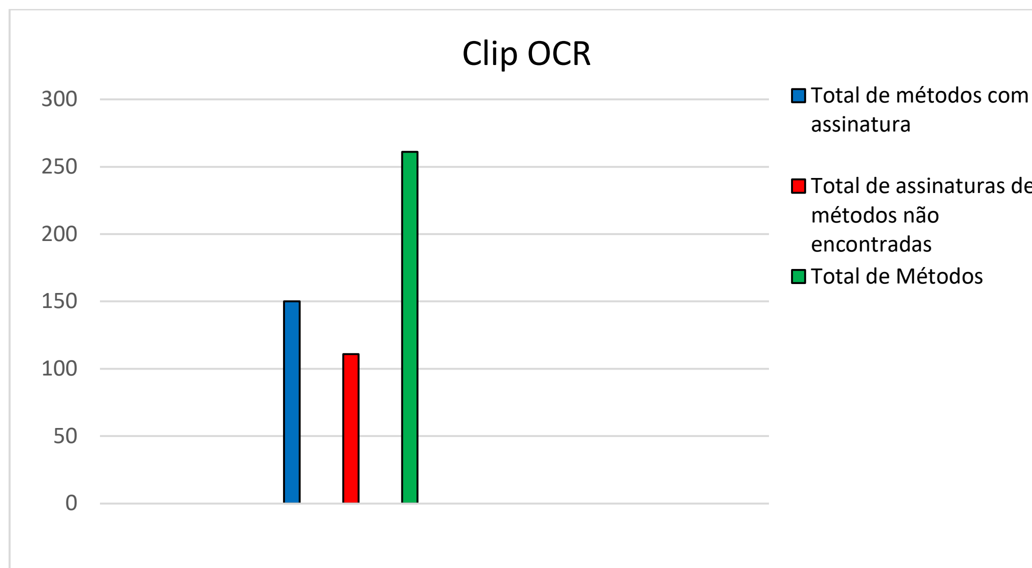

Figura 4.3.2.f: Total de métodos encontrados no projeto Clip OCR

No projeto Clip OCR, foram encontrados ao todo, 261 instâncias de métodos, porém 111 instâncias, não foi possível encontrar assinaturas e 150 métodos com assinaturas encontradas. Para este estudo, só será analisado as instâncias que foram encontradas as assinaturas dos métodos.

4.3.2.1 Histórico de métodos renomeados

Após a coleta de dados, foi iniciada análise do histórico dos métodos, a fim de encontrar diminuição de *statements* entre os *commits*, para facilitar a busca e para melhor visualização do histórico de cada método, foi criado um script em uma linguagem de programação chamada R[31] (linguagem muito utilizada em análises de probabilidade e estatísticas) para a plotagem de gráficos do histórico de métodos. Segue abaixo algumas amostras dos gráficos criados:

31. R: Disponível em: https://www.r-project.org/

32. Clip, disponível em http://www.goclip.com.br/

33. Meyer, disponível em http://www.meyerr.com.br/

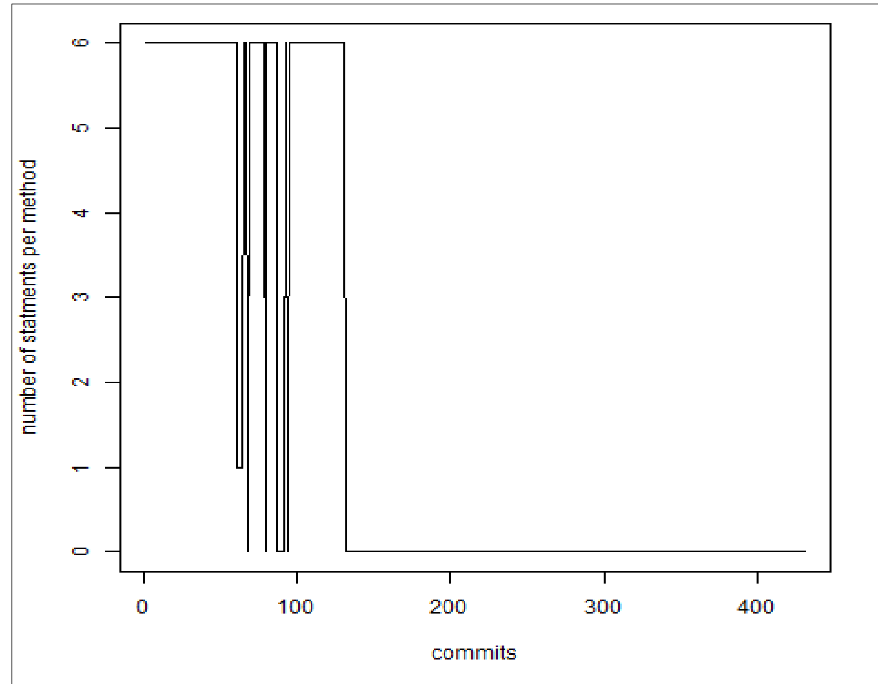

Figura 4.3.2.1a: Histórico do método 1*

* assinaturado método: startConnection(String url,String userName,String password,String driverName

---

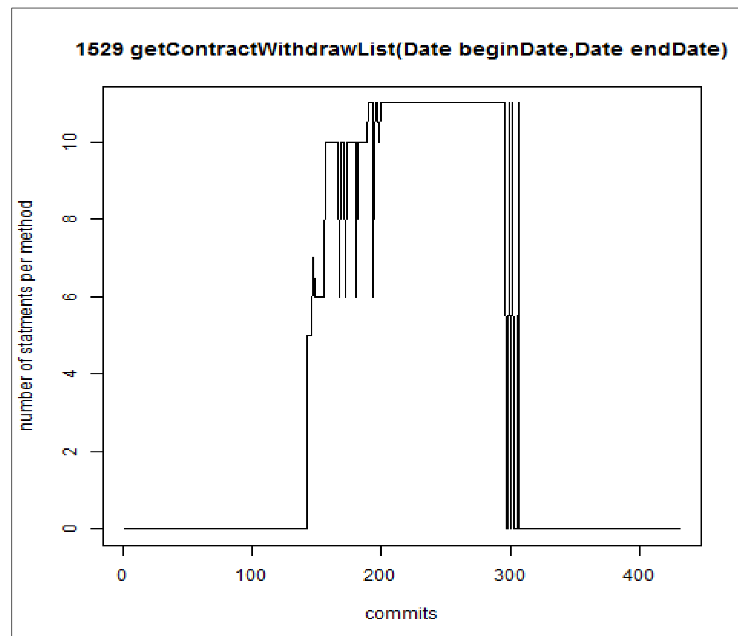

Figura 4.3.2.1.b: Assinatura do método 2

Após a análise de todos os gráficos, percebemos que aproximadamente 90% dos gráficos estavam em algum ponto indo constantemente para zero *statement*, foram levantadas duas possíveis causas para estas quedas:

1º - Os métodos podem ter sido removidos;

2º - Os métodos podem ter sido renomeados e, portanto, a continuação do seu histórico esteja como se fosse outro método.

4.3.2.1.1 Técnicas e ferramentas utilizadas para a recuperação de histórico de métodos renomeados

Dessa forma, a fim de obter um histórico mais completo, foi iniciado a busca por mecanismos para unir o histórico de métodos renomeados e os métodos possivelmente removidos, realizar o estudo da diminuição de *statements* durante o "tempo de vida" desses métodos. Sobre os métodos possivelmente renomeados, foi adotado a abordagem de utilizar uma ferramenta que detectasse métodos que em foram aplicados o padrão de refatoramento *Rename Method* [1], padrão utilizado na literatura para detectar renomeações de métodos, depois que fosse encontrado essas renomeações, seria criado uma ferramenta para dado o primeiro histórico de métodos, a ferramenta poderia buscar através dos resultados de renomeações, o histórico dos métodos renomeados e unir ao histórico do método antes da renomeação.

4.3.2.1.1.1 Refactoring Miner

Para esta união de históricos é necessário coletar as seguintes informações dos casos de renomeações:

- *Commit* de renomeação;
- Nome da classe a qual o método pertence;
- Assinatura do método antes da renomeação;
- Assinatura do método depois da renomeação.

Após investigações na literatura sobre ferramentas de detecção de *Rename Method,* foi observado que em [6], foi utilizada uma ferramenta chamada *Refactoring Miner*, que apresentou alto nível de precisão e nas versões mais recentes detecta *Rename Method*, através dos resultados é possível aplicar técnicas de mineração para obter todos as informações

necessárias descritas acima. Na seção 5.3 será comentada mais sobre as técnicas para a detecção de refatoramento utilizada por essa ferramenta.

. Entretanto, o *Refactoring Miner* não disponibiliza uma forma de salvar essas informações em uma base de dados, pois isso se fez necessário desenvolver um módulo que utilizando a API do *Refactoring Miner* conseguisse escrever todos os resultados coletados em um arquivo de texto. Após o desenvolvimento do módulo, foi possível obter resultados em arquivos de texto com o seguinte formato:

```
Commit:a37d205596787f804998dbcb4218d4fded70e95a
Rename Method public removeGeneralPermissions(permission Permission) : void


renamed to public removePermissions(permission Permission) : void


in class br.com.goclip.model.user.UserGroup
```

Figura 4.3.2.1.1.1.a: Fragmento do resultado do *Refactoring Minner*

Segue abaixo, o gráfico com o total de renomeações encontrados por projeto.

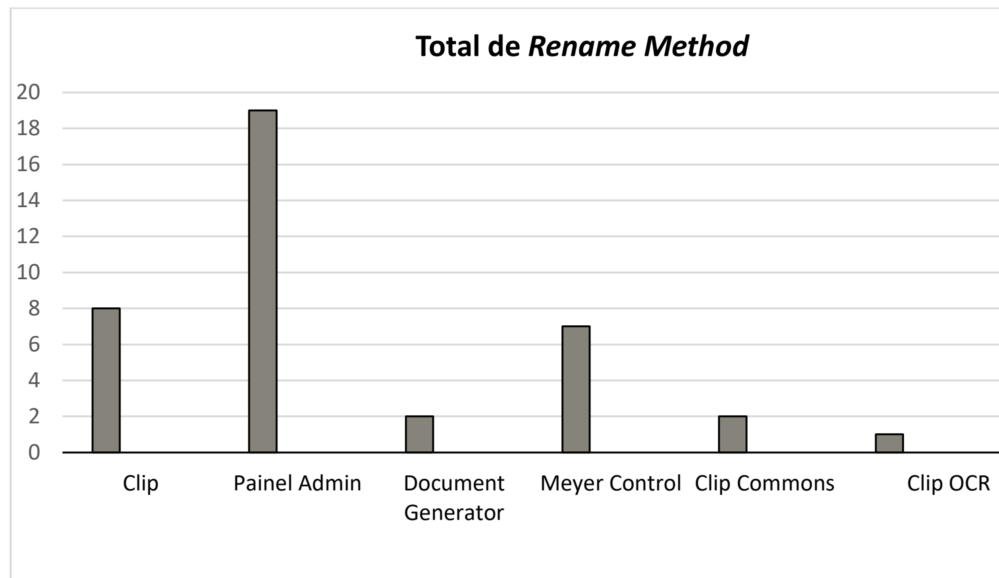

Figura 4.3.2.1.1.1.b: Total de *Rename Method* detectados pelo *Refactoring Miner* por projeto.

Após a coleta de *Rename Method's* foi desenvolvido uma ferramenta para através dos resultados de renomeações unir os históricos dos métodos antes da renomeação e depois da renomeação, esse mecanismo será descrito na próxima seção.

4.3.2.1.1.2 Rename Minerator Tool

A ferramenta responsável pela união do histórico dos métodos foi chamada de ***Rename Minerator Tool***, utilizando o processo descrito abaixo:

```
HistoricProject  main()
13   * Created by ana.carlagb on 21/09/16.
14   */
15  public class HistoricProject {
16
17      private List<Method> methods = new ArrayList<>();
18      private HistoricWriter minner = new HistoricWriter();
19      private String historic = "path/antigo_historico_metodos_sem_uniao_renames.csv";
20
21
22      public void populateMethodHistoric(BufferedReader br) throws IOException{
23          minner.minerate(br);
24          minner.getMethodHistoric().stream().forEach( methodHistoric -> {
25                  Method method = new Method();
26                  method.setHistoric(methodHistoric);
27                  methods.add(method);
28              }
29          );
30
31          minner.writeHistoric( project: this, historic,  newHistoricURL: "path/novo_historico_metodos_com_uniao_renames.csv");
32
33      }
34
35      public static void main(String[] args) {
36
37          HistoricProject project = new HistoricProject();
38          BufferedReader br = null;
39          try {
40              //arquivo que foi salvo os resultados do Refactoring Miner
41              br = new BufferedReader(new FileReader( fileName: "path/historico_rename_method.txt"));
42              project.populateMethodHistoric(br);
43          } catch (IOException e) {
44              e.printStackTrace();
45          }
46
```

Platform and Plugin Updates
IntelliJ IDEA is ready to update.

Figura 4.3.2.1.1.2.a: Classe "*HistoricProject*"

A classe acima "HistoricProject" é a classe principal da ferramenta, segue o fluxo:

- Na linha 42, inicializou a ferramenta, passando como parâmetro o diretório do arquivo gerado pelo *Refactoring Miner* com o resultado de renomeações detectadas;
- Na linha 22, é declarado o método "populateMethodHistóric(BufferedHeader br)", esse método é responsável pelo fluxo principal de mineração de dados do arquivo do *Refactoring Miner* para gerar a nova base de dados.
- Na linha 23, o "minner", objeto responsável pela mineração de dados, chama o método "minerate(BufferedHeader br)" esse método é responsável por ler o arquivo e coletar informações como assinatura de método, nome da classe, *commit* que houve a renomeação e armazenar em uma estrutura de dados. Também é feito a construção do histórico de renomeções, ou seja, de método "A" foi renomeado para método "B", então o histórico de statements vai ser concatenado ao histórico de "A", a partir do commit de renomeação e a linha de B será apagada da base de dados.

- Nas linhas 24 até a linha 29, esse histórico foi transferido para a modelagem de dados utilizada por “*Extract Minerator Tool*”, para facilitar a concatenação de histórico.
- Na linha 31, foi chamado o método responsável por escrever o novo histórico, para isto foi preciso passar como parâmetros, o histórico antigo através do atributo “historic”, a estrutura de dados construída a partir dos resultados de renomeações, informações coletadas pela instância do “HistoricProject” (“this”), o último parâmetro é o diretório do novo histórico a ser gerado.

Criando então uma nova base de dados com métodos que possivelmente nunca foram renomeados e/ou foram removidos com métodos que possivelmente foram renomeados. Após a geração da nova base de dados, foi observado que as bases estavam com a mesma quantidade de métodos das bases antigas para todos os projetos, ou seja, não houve união de histórico, pois as assinaturas dos métodos depois da renomeação não foram encontradas na base de dados antiga, devido aos problemas citados anteriormente de muitas assinaturas de métodos não terem sido encontradas pelo *Metric Miner*. Dessa forma, o total de métodos válidos para a análise de diminuição de *statements* foram os seguintes:

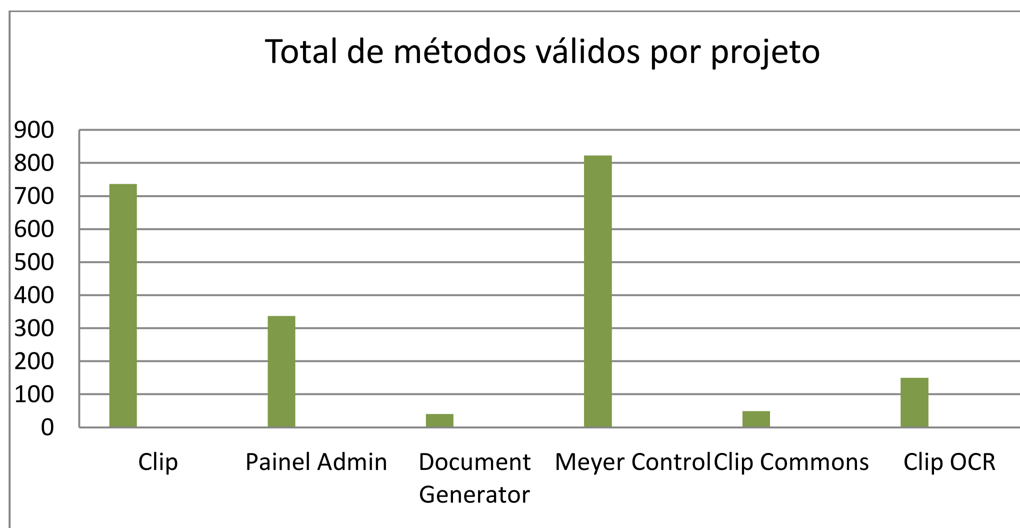

Figura 4.3.2.1.1.2.b: Total de métodos válidos por projeto para análise de *statements*

Ou seja, a quantidade de métodos por projeto foram: 736 (Clip), 337 (Painel Admin), 40 (Document Generator), 823(Meyer Control), 49 (Clip Commons), 150 (Clip OCR).

5. Materiais e métodos utilizados para o histórico de diminuição de *statements* em métodos

Essa seção apresentará técnicas e ferramentas utilizadas para coletar o histórico de diminuição de statements em métodos e uma visão geral sobre processo de investigação sobre as causas dessas diminuições de statements encontradas.

5.1 Técnica utilizada para detectar diminuição de *statements* em métodos

O próximo passo, foi através do histórico de cada método coletar os *commits* que houveram queda de *statement*. Para isto, foi desenvolvida uma aplicação simples que analisa a quantidade de *statement* por commit de cada método e compara se esse *statement* é menor que o *statement* do commit anterior, ignorando todos os *statements* que são iguais a zero. Salvando assim as chamadas "queda de statements" no seguinte formato:

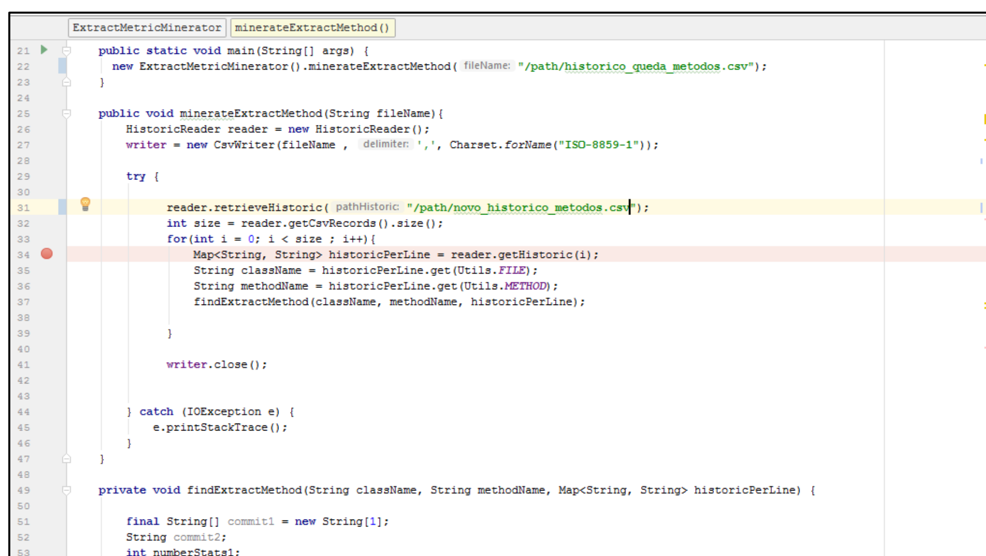

Figura 5.1.a: Método "minerateExtactMethod"

A classe “ExtractMetricMinerator”, é responsável pela inicialização da aplicação na linha 22, através do método cujo passa-se como parâmetro diretório da base de dados que será gerado, esse método é:

“minerateExtractMethod(String fileName)”

- Na linha 31, é chamado um método “retrieveHistoric(String pathHistoric),” que é responsável por recuperar a base de dados com o histórico de métodos analisados por projeto. Essa recuperação é feita para estrutura de dados.
- Na linha 32, é atribuído a quantidade de métodos da base de dados.
- Nas linhas 33 até 39, é feito uma iteração na estrutura de dados que está o histórico de métodos, cujo em cada método é procurado uma diminuição de *statements* entre *commits*, método chamado na linha 37, que será apresentado na próxima figura.

```
53      private void findExtractMethod(String className, String methodName, Map<String, String> historicPerLine) {
54          int numberStats1;
55          int numberStats2;
56
57          List<String> headerList = new ArrayList<>(historicPerLine.keySet());
58          List<String> bodyList = new ArrayList<>(historicPerLine.values());
59
60          for (int i = 0; i < bodyList.size(); i++) {
61              String value1 = bodyList.get(i);
62
63              if(value1 != null && !value1.isEmpty()){
64                  if(i + 1 < bodyList.size()) {
65                      String value2 = bodyList.get(i + 1);
66                      if (StringUtils.isNumericSpace(value1) && StringUtils.isNumericSpace(value2)) {
67                          numberStats1 = Integer.valueOf(value1.replaceAll("\\s+",""));
68                          numberStats2 = Integer.valueOf(value2.replaceAll("\\s+",""));
69
70                          if (numberStats1 > numberStats2 && numberStats2 > 0 ) {
71                              if (i < headerList.size() && i + 1 < headerList.size()) {
72                                  try {
73                                      writer.write(className);
74                                      writer.write(methodName);
75
76                                      writer.write(headerList.get(i));
77                                      writer.write(bodyList.get(i));
78
79                                      writer.write(headerList.get(i + 1));
80                                      writer.write(bodyList.get(i + 1));
81
82                                      writer.endRecord();
83
84                                  } catch (IOException e) {
85                                      e.printStackTrace();
86                                  }
```

Figura 5.1.b: Método “findExtactMethod”

O método acima, é responsável por procurar a diminuição de *statements* entre os commits do método definido no parâmetro “methodName”, no parâmetro “historicPerLine” está todo o histórico *statement* por *commit* de cada método.

- Na linha 54, é declarado o atributo “numberStats1” que armazena a quantidade de *statement* do *commit* anterior.

- Na linha 55, é declarado o atributo “numberStats2” que armazena a quantidade de *statement* do próximo *commit*.
- Na linha 57, é recuperado a lista de todos os *commits*;
- Na linha 58, é recuperado a lista da quantidade de *statement;*
- Nas linhas 61 até 68, atribui-se os valores dos statements as variáveis locais “numberStats1” e “numberStats2”;
- Na linha 70, é comparado se o *statement* do *commit* corrente na iteração é maior que o *statement* do próximo *commit.* Caso sim, nas próximas linhas é gravado a chamada “queda de de *statements*” na base de dados o seguinte formato.

Tabela 3: Modelo – Histórico de diminuição de statements por método

Index Method	**className**	**methodName**	**commitBefore**	**staementBefore**	**commitAfter**	**statementAfter**
1	classA	methodAa	$commit_m$	x_{1m}	$commit_{m+1}$	$x_{1(m+1)}$
...	....	....	....	....	....	....
n	classY	methodbY	$commit_z$	x_{nz}	$commit_{z+1}$	$x_{n(m+1)}$

5.2 Base de dados com o histórico de diminuição de *statements* em métodos

5.2.1 Estrutura da Base de dados

Ou seja, cada campo representa:

- className: o nome da classe a qual pertence o método que houve a diminuição de *statement;*
- methodName: a assinatura do método que houve a diminuição de *statement;*
- commitBefore: o commit antes da a diminuição de *statement;*
- statementBefore: a quantidade de *statement* no *commit* antes da diminuição.
- commitAfter: o commit que houve a diminuição de *statement;*
- statementAfter: a quantidade de *statement* no *commit* da diminuição.

5.2.2 Visão geral sobre o histórico de diminuição de statements obtido

Ao todo, foram encontradas 156 quedas ou diminuição de statments, sendo 100 (Clip), 8 (Painel Admin), 4 (Document Generator), 28 (Meyer Control), 0 (Clip Commons) e 16 (Clip OCR), como mostra o gráfico a seguir.

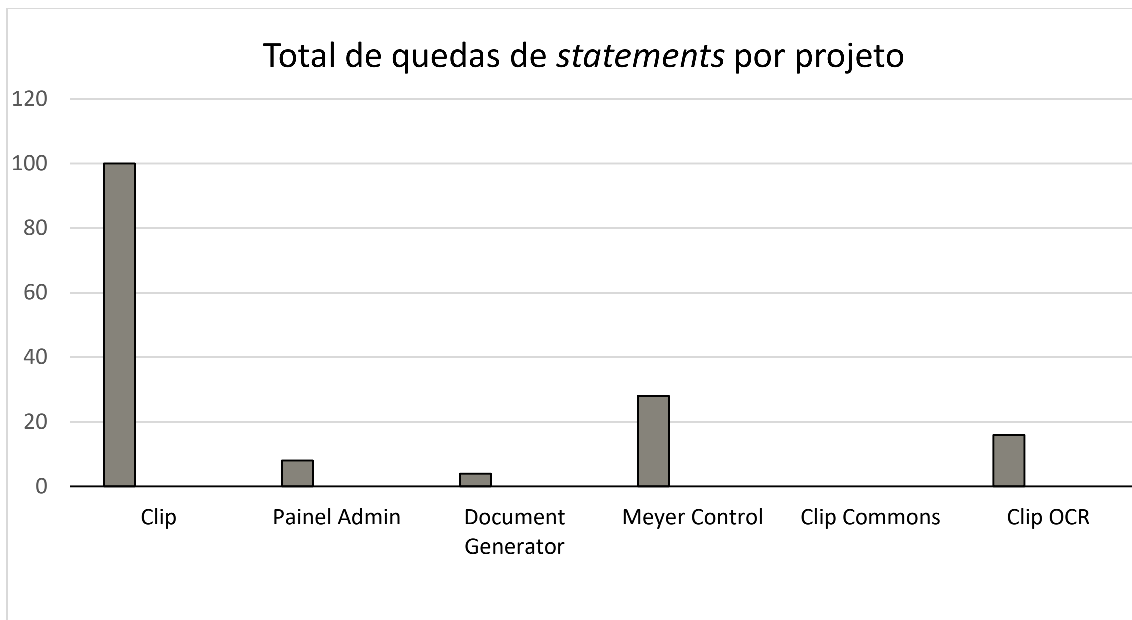

Figura 5.2.2: Total de quedas de *statements* por projeto

5.3 Investigação das causas da diminuição de *statements*

Essa seção apresentará o estudo comparativo entre a detecção de ferramenta e de desenvolvedores que avaliaram as causas da diminuição de *statements*.

5.3.1 Visão geral sobre o processo de investigação

Após a coleta da queda de *statements*, foram realizados 2 tipos de verificações em todas as instâncias das quedas de *statements* encontradas, a fim de responder as seguintes perguntas:

- A queda de statements foi provocada por algum refatoramento?
- Se sim, então que refatoramento foi aplicado?
- Senão, então o que causou a dimunuição do *statements*?

5.3.1.1 Processo de Detecção

Para responder estas questões, foi realizado uma análise de dados com desenvolvedor com experiência de nível intermediário, com experiência em análise de *bad smells* e padrões de refatoramento.

Esta análise foi realizada através de 2 passos:

- **Verificação automática**: foi utilizada a ferramenta *Refactoring Miner* para detectar possíveis refatoramentos nesses métodos que sofreram diminuição de *statements.*

- **Verificação manual:** os desenvolvedores analisaram todos os métodos, como estavam no *commit* antes e *commit* depois e observar o que mudou de um *commit* para o outro e responder as questões acima.

Para coletar essas informações analisadas, foi necessário criar uma base de dados para cada projeto analisado, todas as bases tinham a mesma estrutura, descrita abaixo:

Tabela 4: Estrutura da base de dados

Coluna	**Valor**
className	O nome da classe que houve a queda de *statements*
methodName	O nome do método que houve a queda de *statements*
commitBefore	O commit antes da queda de *statements*
statementBefore	A quantidade de *statement* no *commit* antes da queda de *statements.*
commitAfter	O commit que houve a diminuição de *statement.*
statementAfter	A quantidade de *statement* no *commit* da queda de *statements*.
Tool	Indica se a ferramenta *Refactoring Miner* detectou padrões de refatoramento no método analisado com os valores: 1 - Caso sim 0 - Caso não
Refactoring	Indica se a detecção manual encontrou padrões de refatoramento no método analisado com os valores: 1 - Caso sim 0 - Caso não
RefactoringType_Tool	Indica quais padrões de refatoramento foram detectados pelo *Refactoring Miner* no método analisado
RefactoringType	Indica quais padrões de refatoramento foram detectados pela detecção manual no método analisado

5.3.1.1.3 Detecção Automática

A ferramenta utilizada para a detecção de refatoramento foi o *Refactoring Miner,* pois de acordo com [6] apresentou maior nível de precisão na detecção e também esta ferramenta será utilizada em trabalhos futuro. Essa ferramenta descrita em [6], implementa uma versão do algoritmo UMLDiff para análise de Orientação a Objeto. Este algoritmo é usado para inferir o conjunto de classes, método e campos adicionados, excluídos ou movidos entre *commits* de código, após a execução deste algoritmo, um conjunto de regras é usado para identificar diferentes tipos de refatorações. As regras utilizadas para a detecção de refatoramento estão descritas em [12].

O *Refactoring Minner* , é desenvolvido em Java e está disponibilizado para uso em IDE, no caso o Eclipse IDE e para uso em linha de comando. Na seção 7.2 é descrito os tipos de refatoramento que essa ferramenta detecta. mas para esse estudo considerado a detecção de *Move Method, Rename Method* e *Extract Method*. Foi utilizado a versão para Eclipse IDE, em que foi preciso implementar 2 classes que pudessem ser específicas para coletar resultados para esse estudo, descrita abaixo:

Refactoring Miner:

- ***ExtractMethodPerCommitModule***: que avalia padrões de refatoramento gerados por *commit*, no caso foi utilizado o valor do *commitAfter.*
- ***ExtractMethodMineratorModule*:** detecta padrões de refatoramento em todos os *commits* de uma *branch*, nesse caso foi considerado a *branch* principal, a *master*.

5.3.1.1.4 Detecção Manual

Para a verificação manual, foi utilizado comandos Git, foram sugeridos os seguintes comandos:

$ **git show commit:path/className**: mostra como está o arquivo (a classe) naquele commit, sendo necessário realizar a análise no ***commitBefore*** e ***commitAfter*.**

$ **git diff commitAfter commitBefore -- path/className:** mostra as mudanças do arquivo (a classe) entre os dois *commits*;

$ git diff commitAter commitBefore: mostra todas as mudanças ocorridas no projeto entre os dois *commits*.

6. Resultados e discussões

Após as verificações os desenvolvedores relataram que há mudanças repetidas entre os commits, ou seja, quando os desenvolvedores compararam o que mudou do *commitBefore* para o *commitAfter*, observaram que a mesma queda de statements estavam relatadas em outros *commits*, essas repetições são normais, pois é comum durante o desenvolvimento fazer *merge* (mesclar) *commits,* dessa forma é como se a mesma mudança tivesse acontecido em vários *commits*, mas na verdade só aconteceu no primeiro e o relato de mudança foi propagado para os demais.

Portanto, retirando as instâncias repetidas e considerando apenas a primeira ocorrência de cada queda de *statement*, o gráfico abaixo apresenta o número de quedas analisadas e a quantidade de métodos em que essas quedas ocorreram, observando que houve métodos que ocorreram mais de uma queda.

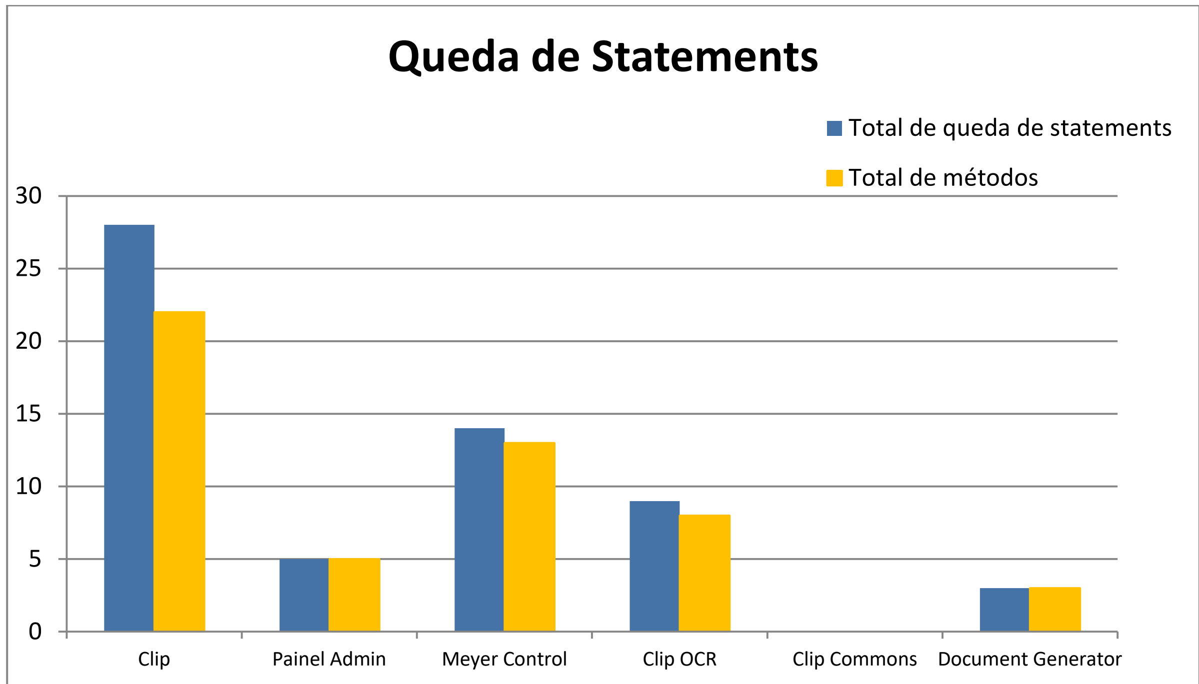

Figura 6.a: Total de quedas de *statements* e métodos por projeto

Por conseguinte, segue a análise das respostas obtidas através das verificações:

- A queda de statements foi provocada por algum refatoramento?

Para responder esta pergunta foi realizado um estudo comparativo entre os resultados da ferramenta *Refactoring Miner* sobre a detecção de algum padrão de refatoramento e a resposta dos desenvolvedores se aquela queda de *statement* ocorreu de fato um refatoramento. Então foi levado em consideração as seguintes combinações de respostas:

Tabela 5: Modelo de combinações de respostas

Desenvolvedor Versus Refactoring Miner

	***False Positive* (FP)**	***True Positive* (TP)**	***False Negative* (FN)**	***True Negative* (TN)**
Refactoring Miner	detectou	Detectou	não detectou	não detectou
Desenvolvedor	não detectou	Detectou	detectou	não detectou

Segue abaixo, as respostas obtidas por projeto.

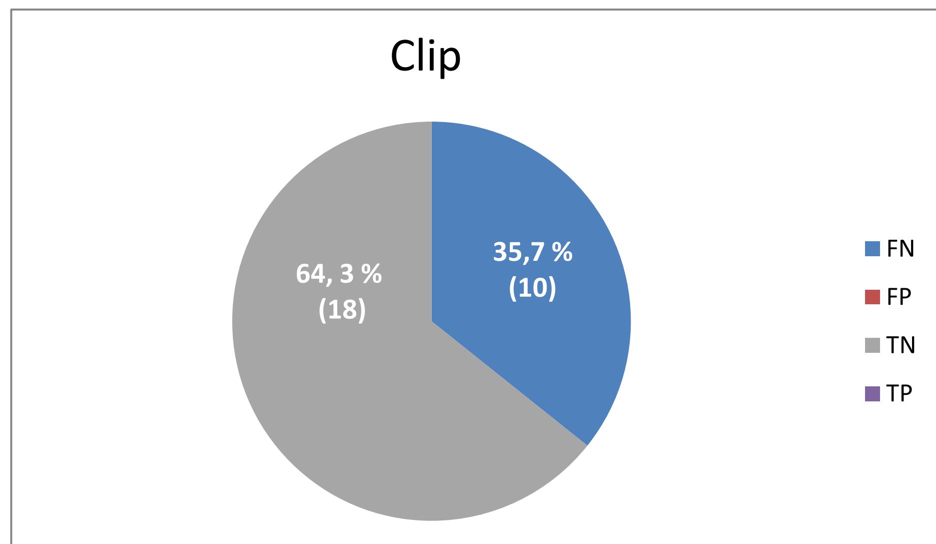

Figura 6.b:Total de refatoramentos encontrados – Projeto Clip

O gráfico acima apresenta que mais de 50% dos métodos analisados não foram aplicados nenhuma técnica de refatoramento, porém 40,9% dos métodos ,de acordo com os desenvolvedores , receberam alguma técnica de refatoramento, mas a ferramenta *Refactoring Miner* não detectou nenhum desses casos.

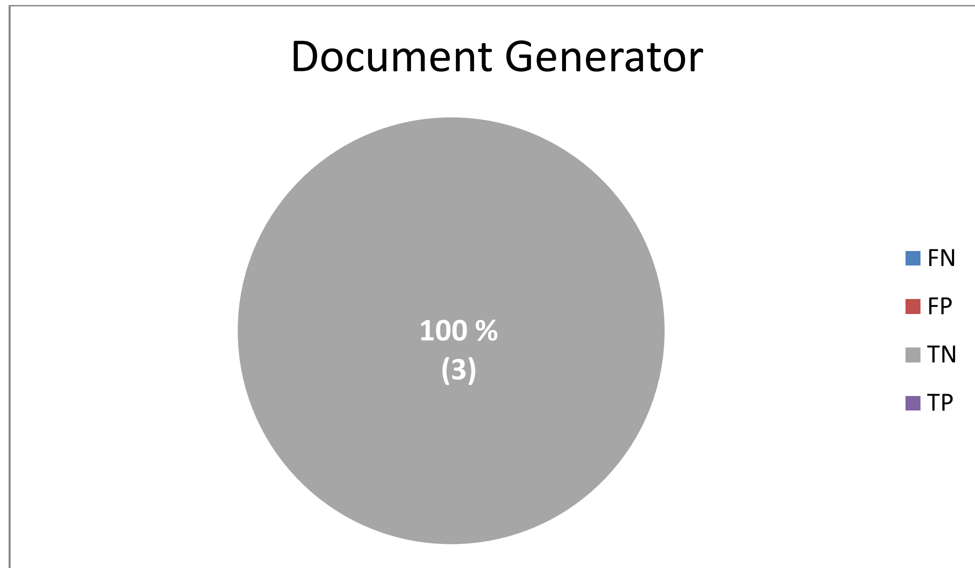

Figura 6.c:Total de refatoramentos encontrados – Projeto Document Generator

O *Document Generator* é o projeto que tem a menor quantidade de instâncias analisadas e nenhuma dessas instâncias foram detectadas técnicas de refatoramento, nem pela ferramenta e nem pelos desenvolvedores.

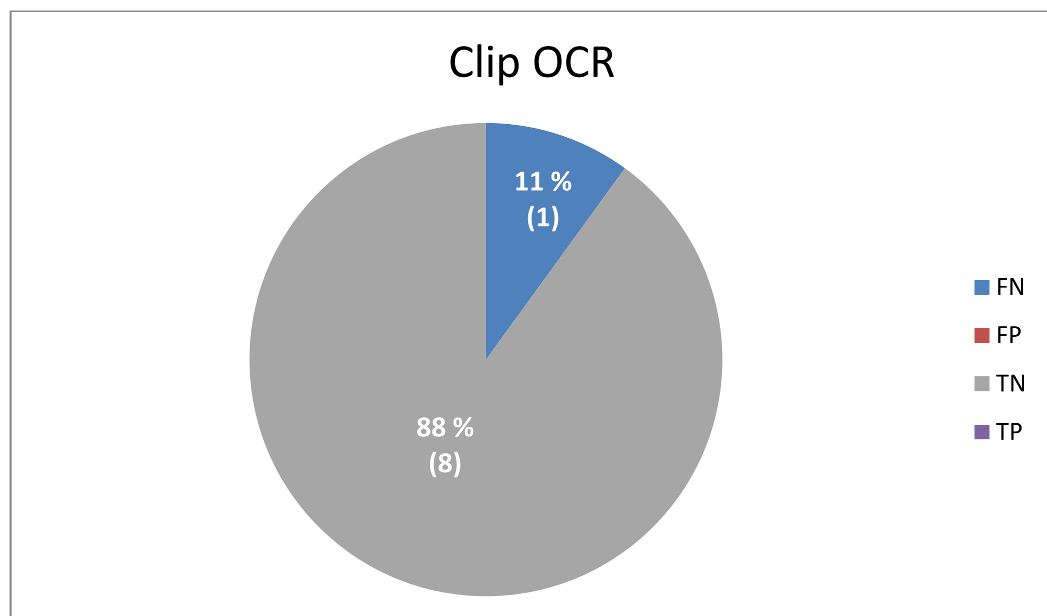

Figura 6.d:Total de refatoramentos encontrados – Projeto Clip OCR

O Clip OCR, projeto com 9 métodos analisados, sendo que, de acordo com os resultados obtidos, 7 métodos não sofreram refatoramento e 2 métodos foram refatorados de acordo com os desenvolvedores, mas a ferramenta não detectou estes *refactorings*.

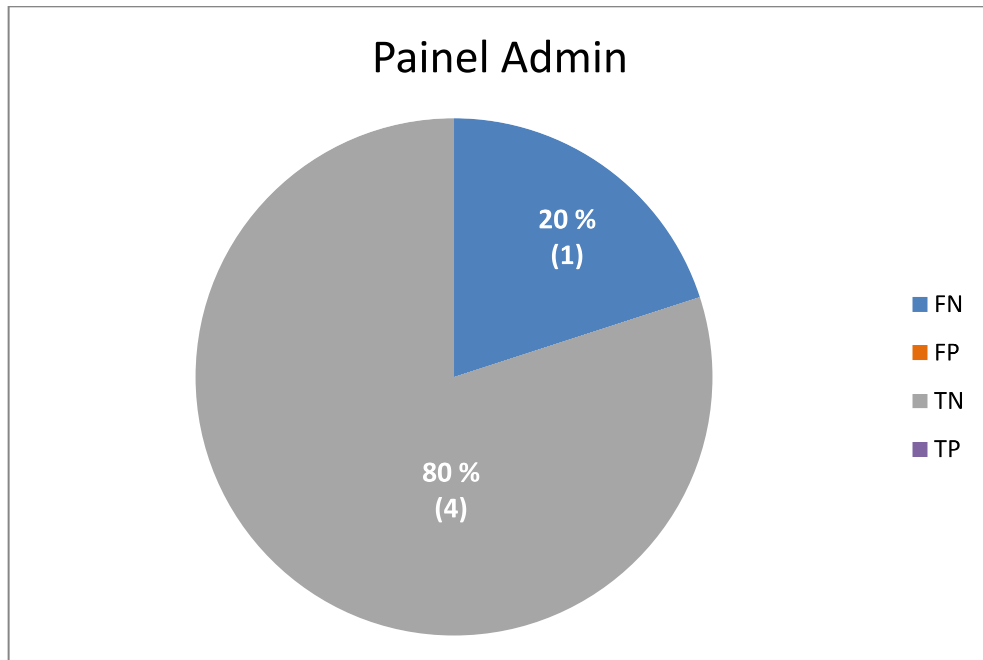

Figura 6.e:Total de refatoramentos encontrados – Projeto Painel Admin

Sobre o projeto Painel Admin, dos 5 métodos analisados, 4 métodos não foram aplicados nenhum refatoramento, porém um método, os desenvolvedores encontraram a um refatoramento que a ferramenta não detectou.

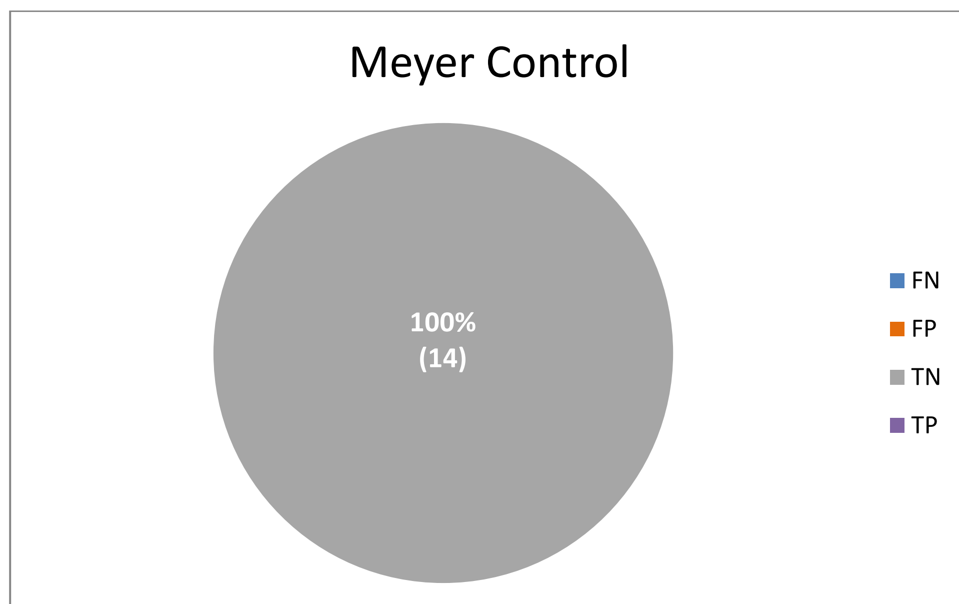

Figura 6.f: Total de refatoramentos encontrados – Projeto Meyer Control

De acordo com os resultados obtidos do projeto Meyer Control, nenhum dos

métodos foram aplicados nenhuma técnica de refatoramento.

Como descrito acima, a segunda pergunta desse estudo, pretende avaliar quais foram as mudanças que causaram a queda de *statement,* como complemento da primeira pergunta foi coletado tanto do *Refactoring Miner* quanto dos desenvolvedores quais foram estas mudanças, porém o que foi observado nas respostas dos desenvolvedores é que os método que foram considerados refatorados, de acordo com a primeira pergunta, na verdade esses refatoramentos foram causados por mudanças externas, ou sejam outros métodos chamados dentro do método avaliado, que foram refatorados. Para tratar esses casos, então as respostas foram organizadas em 3 grupos: mudanças internas, refatoramentos internos e externos. Segue abaixo a descrição de cada grupo:

- **Mudança interna**: foram considerados para este grupo adição ou remoção de *statemtents* (tais como blocos de repetição, condicional, variáveis e demais), linhas comentadas ou chamadas de métodos que foram removidas.
- **Refatoramento interno**: quando a técnica foi aplicada dentro do método, como por exemplo:
  - ❖ Extract Method: se uma parte do código do método foi retirado para outro método.
  - ❖ Move Method: se o método foi movido para outra classe.
  - ❖ Rename Method: se a assinatura do método sofreu alteração.

- **Refatoramento externo**: quando a técnica foi aplicada em método chamado pelo método analisado.

Os gráficos abaixo, apresentarão as respostas obtidas separadas por grupo.

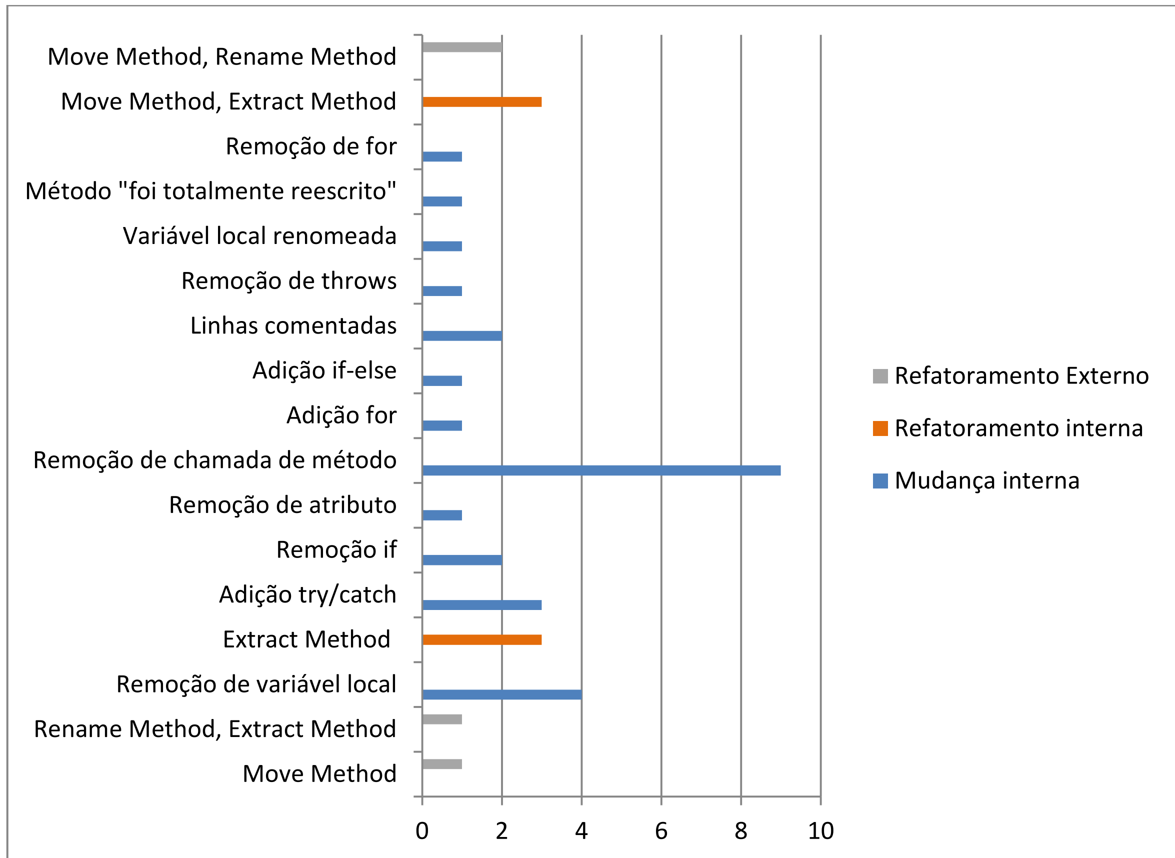

Figura 6.g: Causas da diminuição de *statements* nos métodos analisados do Projeto Clip

No projeto Clip, foram analisados 22 métodos, como apresenta o gráfico acima foram encontradas 12 causas para a queda de *statements*, sendo que há métodos que houve mais de uma destas causas. Os desenvolvedores encontraram 5 refatoramentos internos , todos *Extract Method,* os desenvolvedores responderam que esses *Extract's* ocorreram na maioria das vezes eram variáveis atribuídas dentro do método e foram extraídas para outros métodos, 9 refatoramento externos, de acordo com os desenvolvedores o *Move Method* estavam relacionados a métodos que estavam como estáticos e foram movidos para outra classe, deixando de ser estáticos, o *Rename Method* estavam mais relacionados a métodos externos que sofreram mudanças na assinatura do método, ou seja que foram adicionados parâmetros.

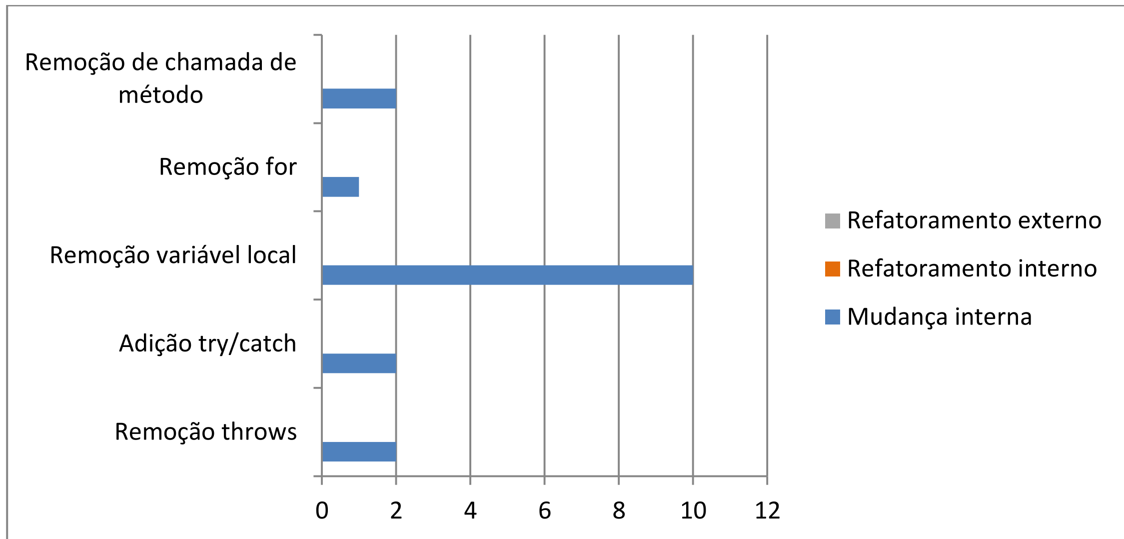

Figura 6.h: Causas da diminuição de *statements* nos métodos analisados do Projeto Meyer Control

De acordo com o gráfico do projeto Meyer Control, a maior parte das quedas foram causadas por remoção de variáveis locais, não foi encontrada nenhuma refatoramento interno e nem externo.

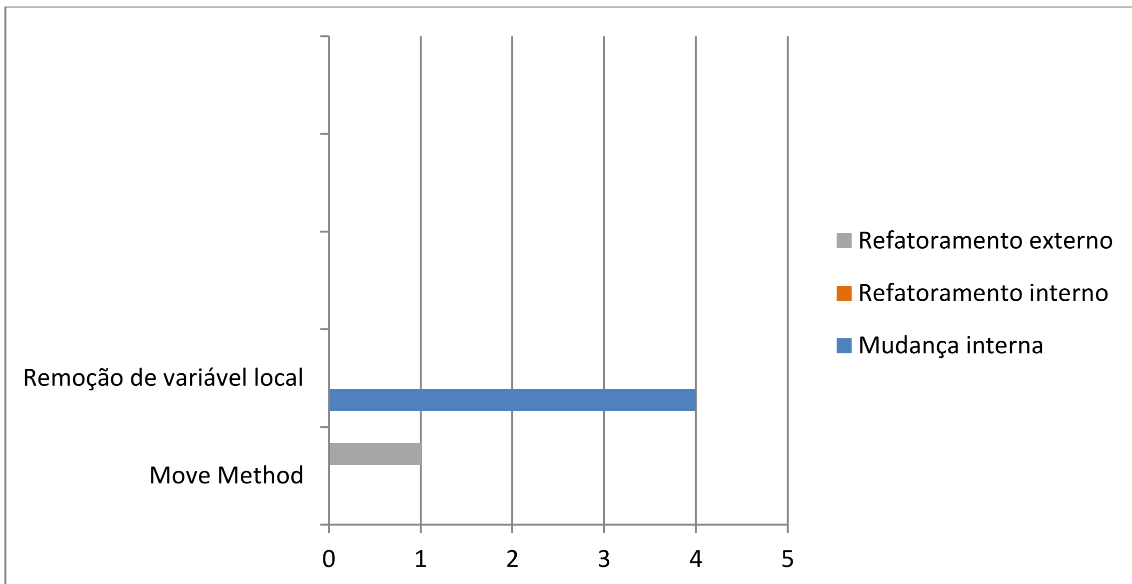

Figura 6.i: Causas da diminuição de *statements* nos métodos analisados do Projeto Painel Admin.

O projeto Painel Admin, semelhante ao Meyer Control, também teve mais mudanças relacionadas a variáveis locais e um método chamado dentro do método analisado foi movido para outra classe.

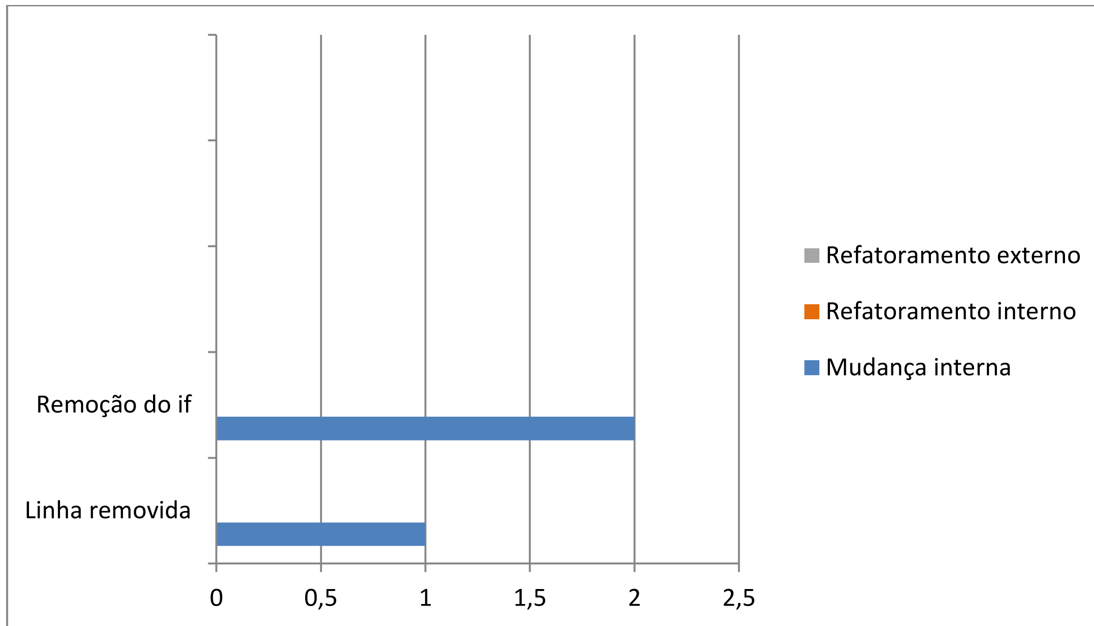

Figura 6.j: Causas da diminuição de *statements* nos métodos analisados do Projeto Document Generator.

No Document Generator foram removida 2 estruturas condicionais (if).

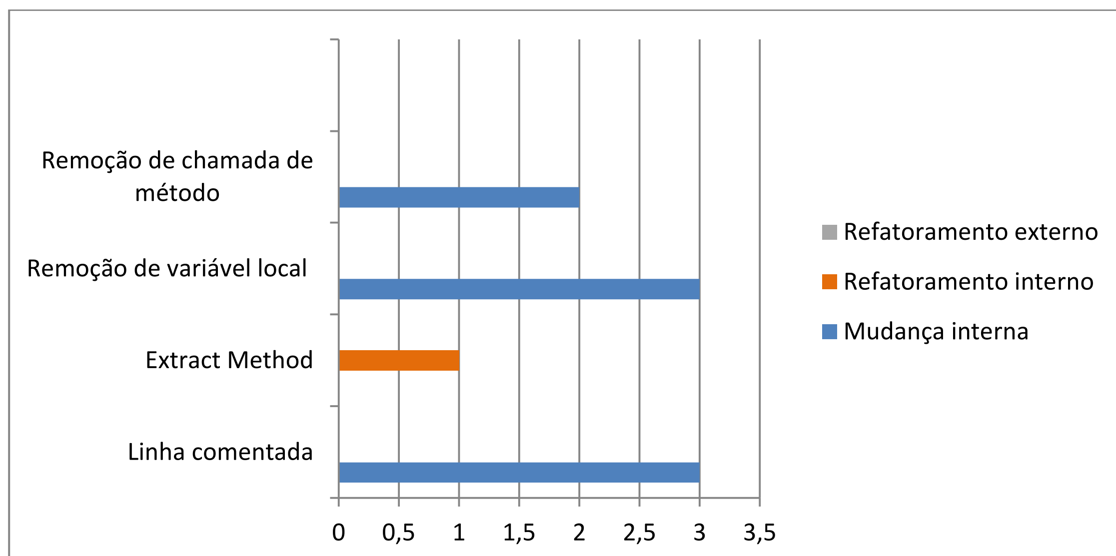

Figura 6.k: Causas da diminuição de *statements* nos métodos analisados do Projeto Clip OCR.

O Clip OCR, dos 9 métodos analisados, apenas 1 foi encontrado *Extract Method,* o restante das quedas foram causadas por mudanças internas.

7. Trabalhos Relacionados

7.1 "Do they Really Smell Bad? A Study on Developes' Perception of Bad Code Smells"

No artigo [19], desenvolvido por [Palomba et. al. 2014], este artigo apresenta um estudo empírico com o objetivo de analisar até que ponto desenvolvedores percebem *bad smells* como problemas de implementação e/ou design. Para esse estudo foram utilizados 12 tipos de *smells* para ser detectado em 3 projetos Java *Open Source* – ArgoUML, Eclipse e JEdit. Para a detecção foram envolvidos 10 desenvolvedores dos projetos envolvidos, 24 "*outsiders*", dos quais 9 são desenvolvedores industrial e 15 são estudantes do mestrado.

Os 12 *Bad Smells* foram: *Class Data Should Be Private, Complex Class, Feature Envy, God Class, Inappropriate Intimacy, Lazy Class, Long Method, Long Parameter List, Middle Man, Refused Bequest, Spaghetti Code, Speculattive Generallity*. Esses *bad smells* são problemas relacionados a complexidade, tamanho e design.

Foram apresentados a cada desenvolvedor partes de código que poderiam ser *bad smells,* esses trechos de código foram identificados por um estudante de mestrado, baseado em resultados de uma ferramenta de detecção de *Bad Smells* chamada DECOR. Dessa forma, foram questionados aos desenvolvedores a seguinte pergunta: "Em sua opinião, este componente de código expõe algum problema de desing e/ou implementação? " Se a resposta for "sim", os desenvolvedores eram solicitados para "Em sua opinião, por favor, explique, quais são os problemas que afetam esse componente de código. " Com essas perguntas, os autores comparam se os desenvolvedores perceberam a presenças de *bad smells* e mais ainda se os identificaram.

Os desenvolvedores tiveram que responder essa pergunta para 20 componentes de ArgoUML, 22 do Eclipse e 18 do JEdit. Em suma, os resultados apresentaram que cerca de 70% dos componentes foram identificados como *bad smells* pelos desenvolvedores, sendo que, curiosamente, os problemas relacionados a tamanho, complexidade foram mais notados por desenvolvedores profissionais, outra conclusão relevante, é que *bad smells Class Data Should Be Private, Middle Manm Long Parameter List, Lazy Class e Inappropriate Intimacy* foram apenas considerados como problemas simples de design e não como *bad smells.*

7.2 “Why we refactor? Confession of Github Contributor”

Esse artigo [6], criado por [Silva, Tsantalis, Valente et. al. 2016] apresenta resultados de um experimento realizado durante 61 dias, que teve como objetivo detectar ocorrências de refatoramento em projetos *Open Source* e posteriormente entrar em contato com os desenvolvedores que aplicaram os possíveis refatoramentos a fim de coletar informações sobre as motivações que levaram a esses refatoramentos, caso sejam de fato refatoramento.

Foram selecionados 748 repositórios, incluindo projetos conhecidos como JetBrain/Intelijj Community, Apache/Cassandra. Foram catalogados 12 tipos de refatoramento: *Extract Method, Move Class, Move Attribute, Move Method, Inline Method, Rename Package, Extract Superclass, Pull Up Method, Pull Up Attribute, Extract Interface, Push Down Attribute, Push Down Method.*

A ferramenta utilizada para detectar os refatoramentos foi *Refactoring Minner*, por ser uma ferramenta que pode ser independente de IDE, ao total foram encontradas 463 instâncias de refatoramento entre 124 projetos diferentes e entre 222 *commits*. O refatoramento que foi mais detectado foi *Extract Method's,* com 118 ocorrências. Após a fase de detecção, iniciou-se o processo de contato com os desenvolvedores, foram enviados 465 *e-mails*, porém 195 foram respondidos. Pelas respostas coletadas, o estudo apresenta que as principais causas de refatoramentos foram a evolução de requisitos e não tanto a correção de *smells*, sobre o *Extract Method* apresentou 11 motivações diferentes, sendo apenas 2 motivações relacionadas a *smells* (remover duplicação de código e decomposição de método). Também foi evidenciado, que de fato, desenvolvedores utilizam IDE's como ferramenta de refatoramento durante o desenvolvimento, como IntelliJ IDEA.

8. Considerações Finais

Contudo, foi possível avaliar que os processos de mineração de dados de repositórios l ainda trazem muitas dificuldades, apesar de ferramentas automatizadas para coleta de dados, mesmo assim durante o processo de mineração de informações foram encontrados bastante erros que dificultaram a coleta de uma amostra maior de métodos para o estudo, aproximadamente 90% dos históricos dos projetos não foram analisados devido a problemas encontrados durante a mineração.

Primeiro, foi o *Metric Miner*, que não conseguiu encontrar mais de 75% dos métodos dos projetos escolhidos, isso acarretou em que não obtivermos o histórico completo da maioria dos métodos, uma vez que foram detectados que há métodos que foram renomeados durante o desenvolvimento dos softwares. Porém mesmo, com a pequena amostra de métodos coletados, foi desenvolvida uma ferramenta para recuperar o histórico de métodos renomeados, a fim de obter um histórico mais completo possível. Porém essa tentativa não obteve sucesso, uma vez que os históricos dos métodos renomeados não foram encontrados pelo *Metric Miner.*

A partir da amostra obtida, foi possível encontrar queda de *statements,* a fim de responder se queda de *statements* foi provocada por algum refatoramento, foram realizadas análise automática e a análise manual, de acordo com os resultados apresentados neste estudo, conclui-se que há casos de uso reais em que queda de *statements* podem ser provocados por refatoramento, que podem ser internos ou externos, ou seja dentro do próprio método ou fora do método, também foi apresentado que a diminuição de *statements* também podem estar relacionados não somente ao *Extract Method*, mas também a outros padrões de refatoramento como *Move Method* e *Rename Method,* essas quedas também podem indicar mudanças no comportamento do método, como foi possível encontrar que, em alguns casos, a diminuição de *statements* foram provocadas por remoção de estruturas condicionais ou estruturas de repetição, outro ponto observado é que muitos desses métodos são métodos de testes, o que dificultou nossa conclusão de resultados, tomando como aprendizado para que em trabalhos futuros, a camada de testes possa ser retirada da análise.

Sobre as análises comparativas entre desenvolvedor e ferramenta apresentaram que, em alguns casos, o conceito de refatoramento está ainda muito subjetivo entre desenvolvedor e ferramenta de detecção, o estudo apresentou casos que desenvolvedores consideraram *Extract Method's* que a ferramenta não detectou. Pelos resultados coletados, foi possível

observar que casos considerados refatoramento, por desenvolvedores, foram provocados por refatoramento em outros métodos envolvidos ou em outros trechos de código, o que demonstra que desenvolvedores quando procuram trechos refatorados busca ter um olhar macro de toda a aplicação, diferentemente, da ferramenta de detecção que limita-se somente a estrutura do método.

8.1 Trabalhos Futuros

Para o melhoramento dessa análise, pretende-se:

- Replicar esse estudo em repositórios *Open Source,* a fim de aumentar a coleta de histórico de métodos, para melhorar a análise;
- Ignorar, durante a análise, classes de testes do repositório, a fim de coletar mudanças nas classes relacionadas ao design da aplicação;
- Entrar em contato com os desenvolvedores que realizaram as mudanças no código e questionar as causas da diminuição de *statements*.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] FOWLER, M. Refactoring - Improving the Design of Existing. 2002

[2] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. "*A Case Study in Refactoring a Legacy Component for Reuse in a Product Line*". In Proceedings of the International Conference on Software Maintenance, pages 369-378, 2005.

[3] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. "*Does Refactoring Improve Reusability?*" In Proceedings of the Inter- national Conference on Software Reuse, pages 287-297, 2006.

[4] T. Mens, and T. Tourwe. "*A Survey of Software Refactoring*". IEEE Trans. Softw. Eng., 2004.

[5] *Version Control*, disponível em: https://en.wikipedia.org/wiki/Version_control

Último acesso em: 26 Jan 2017.

[6] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente, *"Why We Refactor? Confessions of GitHub Contributors", 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (FSE'2016), Seattle, WA, USA, November 13-18, 2016.

[7] Metric Miner, disponível em: <http://www.metricminer.org.br/>
Último acesso em 23 Dez 2016.

[8]*Statement*,disponível <https://en.wikipedia.org/wiki/Statement_(computer_science)>
Último acesso em 27 Nov 2016

[9] Y. Aiko, M. Leon, *"Do Developers Care about Code Smells? An Exploratory Survey*", WCRE 2013, IEEE , Koblenz, Germany.

[10] J. R. Foster. "*Cost Factors in Software Maintenance*". PhD thesis, University of Durham, 1993.

[11] H. Nafiseh, "*Which Factors Affect Software Projects Maintenance Cost More?*" . In

Acta Informatica Medica, pags 63-66, 2013.

[12] T, Nikolaos, G. Victor, S. Eleni, H. Abram. *"A Multidimensional Empirical Study on Refactoring Activity"* CASCON, 2013.

[13] S. Yonghee, M. Andrew, W. Laurie, O. Jason. *"Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities"*
National Science Foundation Grant. 2009.

[14]C. Jun-Ru, "*Introduction Git*", disponível em:
<http://www.ncyu.edu.tw/files/site_content/cc/1010531git.pdf>
Último acesso em 26 Jan 2017.

[15] Git, disponível em: <https://en.wikipedia.org/wiki/Git>.
Último acesso em 26 Jan 2017.

[16] JDT, disponível em: <http://www.eclipse.org/jdt/>
Último acesso em 03 Jan 2017.

[17] *Program Representations*, disponível em:
<http://courses.cs.vt.edu/cs5704/spring16/handouts/5704-7-ProgramRepresentations.pdf>
Último acesso em 03 Jan 2017.

[18]*Abstract Sintaxe Tree*, disponível em
< https://en.wikipedia.org/wiki/Abstract_syntax_tree>
Último acesso em 03 Jan 2017.

[19] P. Fabio, P. Massimiliano Di, O. Rocco, L. Andrea De, *"Do they Really Smell Bad? A Study on Developes' Perception of Bad Code Smells"*, IEEE International Conference on Software Maintenance and Evolution, 2014.

[20] CSV, disponível em <http://opencsv.sourceforge.net/>
Último acesso em 03 Jan 2017.