PATRICIA RUCKER DE BASSI

PEGADAS DO PROGRAMADOR: CALCULANDO SUA CONTRIBUIÇÃO NO DESENVOLVIMENTO COLABORATIVO DE SOFTWARE

CURITIBA
2019

PATRICIA RUCKER DE BASSI

PEGADAS DO PROGRAMADOR: CALCULANDO SUA CONTRIBUIÇÃO NO DESENVOLVIMENTO COLABORATIVO DE SOFTWARE

Tese de doutorado apresentado ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Doutor em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Descoberta de Conhecimento e Aprendizagem de Máquina

Orientador: Prof. Dr. Emerson Cabrera Paraiso

CURITIBA
2019

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Luci Eduarda Wielganczuk – CRB 9/1118

B821p 2019

Bassi, Patricia Rucker
Pegadas do programador : calculando sua contribuição no desenvolvimento colaborativo de software / Patricia Rucker de Bassi ; orientador: Emerson Cabrera Paraiso. – 2019.
190 f. : il. ; 30 cm

Tese (doutorado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2019
Bibliografia: f. 167-179

1. Software – Desenvolvimento. 2. Programação (Computadores). I. Paraiso, Emerson Cabrera. II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD 22. ed. – 005.1

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

DECLARAÇÃO

Declaro para os devidos fins que a aluna PATRICIA RUCKER DE BASSI, defendeu sua tese de doutorado intitulada **"PEGADAS DO PROGRAMADOR: CALCULANDO SUA CONTRIBUIÇÃO NO DESENVOLVIMENTO COLABORATIVO DE SOFTWARE"**, na área de concentração Ciência da Computação, no dia 17 de abril de 2019, no qual foi aprovada.

Declaro ainda que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade, firmo a presente declaração.

Curitiba, 19 de agosto de 2019.

Programa de Pós-Graduação em Informática PPGIa PUCPR

Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática
Pontifícia Universidade Católica do Paraná

Rua Imaculada Conceição, 1155 Prado Velho CEP 80215 901 Curitiba Paraná Brasil
Tel.: (41) 3271 1669 Fax.: (41) 3271 2121 www.ppgia.pucpr.br

Dedicatórias

Ao Danilo e Bruno, meu porto seguro.

Aos meus pais, Mario e Guerrit.

Agradecimentos

À minha família, Danilo e Bruno, que sempre me apoiaram em toda a minha trajetória. E nestes últimos tempos, a paciência e colaboração de vocês foi muito importante para que eu tenha conseguido chegar até aqui. Este trabalho não seria possível sem o apoio e compreensão de vocês!

Aos meus pais, Mario e Guerrit, pela educação e princípios que me ensinaram e oportunidade de estudos que me proporcionaram. A eles, meu eterno agradecimento!

Aos amigos do PPGIA, pela constante troca, pela amizade e empatia, compartilhando momentos bons, mas também momentos difíceis, o que tornou a caminhada até aqui mais leve!

Aos professores de todos os níveis de ensino que transmitiram seu saber, sua experiência e me motivaram a ser um deles.

Ao meu orientador professor Emerson Cabrera Paraiso, por compartilhar experiência e conhecimento, pela paciência, orientação e indicações sempre relevantes.

Ao aluno de PIBIC do Curso de Sistemas de Informação da PUC-PR, Pedro Henrique Banali, e aos alunos do Curso de Sistemas de Informação, Alexandre Hauffe e Matheus Kevin Pelegrini, pela colaboração.

Aos colegas do Grupo de Pesquisa de Descoberta do Conhecimento e Aprendizagem de Máquina, Franciele Beal, Gregory Moro Puppi Wanderley e Matheus Camilo da Silva, pela colaboração e compartilhamento de conhecimento.

À CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil pelo apoio financeiro, Código de Financiamento 001.

E a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho.

Sumário

LISTA DE FIGURAS X

LISTA DE GRÁFICOS XII

LISTA DE QUADROS XIII

LISTA DE TABELAS XIV

LISTA DE SIGLAS XV

RESUMO xvii

ABSTRACT xviii

CAPÍTULO 1 19

INTRODUÇÃO 19

1.1. Motivação **24**

1.2. Objetivos **27**

1.3. Contribuições Científicas e Tecnológicas **28**

1.4. Hipóteses de Trabalho **30**

1.5. Organização do Documento **30**

CAPÍTULO 2 32

FUNDAMENTAÇÃO TEÓRICA 32

2.1. Trabalho Colaborativo Suportado por Computador **32**

2.2. Desenvolvimento Colaborativo de Software **36**
2.2.1. Colaborativo ou Cooperativo 38
2.2.2. Engenharia de Software Colaborativa 40
2.2.2. Colaboração no Desenvolvimento de Software 42

2.3. Sistemas de Controle de Versão **45**

2.4. Qualidade de Software **48**
2.4.1. Medidas e Métricas 48
2.4.2. Métricas de Qualidade de Software 49
2.4.3. Métricas de Código-fonte 51

2.5. Agrupamento das Métricas de Qualidade do Produto de Software 70
2.5.1 Condução da Revisão Sistemática de Literatura 71
2.5.2 Resultados e Discussões da Revisão Sistemática de Literatura 74

2.6. Recuperação da Informação 84
2.6.1. Processamento de Linguagem Natural 86
2.6.2. Análise de Sentimento 87

2.7. Considerações Finais 90

CAPÍTULO 3 91

ESTADO DA ARTE 91

3.1. Seleção de Artigos 91

3.2. Métricas de Software 93

3.3. Medidas de Colaboração 99

3.4. Considerações Finais 101

CAPÍTULO 4 103

PROCEDIMENTOS METODOLÓGICOS 103

4.1. Caracterização da Pesquisa 103

4.2 Estruturação desta Pesquisa 105
4.2.1 Ferramentas Utilizadas 107
4.2.2 Escopo da Pesquisa 108
4.2.3 Agrupamento das Métricas e as Influências 109

4.3. Experimentos Realizados 110
4.3.1. Experimento 1 para Montar a Base de Dados Histórica 111
4.3.2. Experimento 2 para Calcular o Grau de Contribuição do Desenvolvedor 111
4.3.3. Experimento 3 para Verificar as Mensagens dos *Commits* e *Issues* 112

4.4. Cenários de Análise 114

4.5. Considerações Finais 114

CAPÍTULO 5 115

CONTRIBUIÇÃO INDIVIDUAL DE UM DESENVOLVEDOR EM UM PROJETO COLABORATIVO DE DESENVOLVIMENTO DE SOFTWARE 115

5.1. Contribuição no Desenvolvimento Colaborativo de Software 116
5.1.1 Papéis dos Participantes de um SCV e a Contribuição 118

5.2. Cálculo do Grau de Contribuição do Desenvolvedor 124

5.3. Considerações Finais 131

CAPÍTULO 6 133

RESULTADOS EXPERIMENTAIS E DISCUSSÕES 133

6.1. Experimento 1 133
6.1.1 Coleta de Dados 133
6.1.2. Análise das Métricas dos *Commits* 135
6.1.3 Resultados e Discussões do Experimento 1 143

6.2. Experimento 2 144
6.2.1 Coleta de Dados 144
6.2.2. Cálculo do Grau de Contribuição do Desenvolvedor 146
6.2.3 Resultados e Discussões do Experimento 2 151

6.3. Experimento 3 153
6.3.1 Análise das Mensagens dos *Commits* e *Issues* 155
6.3.2 Coleta de dados 156
6.3.3 Resultados e Discussões do Experimento 3 157

6.4. Considerações Finais 159

CAPÍTULO 7 160

CONSIDERAÇÕES FINAIS 160

7.1. Relevância da Pesquisa 160

7.2 Contribuições da pesquisa 161

7.3 Limitações da pesquisa 162

7.4 Conclusões 163

7.5 Trabalhos futuros 165

REFERÊNCIAS 167

APÊNDICE A – GRÁFICOS DAS MÉTRICAS DO EXPERIMENTO 1 180

APÊNDICE B – PROTÓTIPO DA APLICAÇÃO PARA CÁLCULO DO GC[D] 187

Lista de Figuras

Figura 2.1:Relacionamento entre CSCW, HCI, CSCL e outras áreas da ciência 34
Figura 2.2*:* Sistemas de gestão de defeitos (bugs)..... 38
Figura 2.3: Sistemas de controle de versão 38
Figura 2.4: Linha do tempo do desenvolvimento colaborativo de software 40
Figura 2.5: Áreas da Computação relacionadas à CSE 41
Figura 2.6: Exemplo de revisão e ramificação 46
Figura 2.7: Exemplo de SCV centralizado 47
Figura 2.8: Exemplo de Aplicação da Métrica CCM 53
Figura 2.9: Classe C 56
Figura 2.10: Exemplo de classes com o mesmo valor de LCOM 58
Figura 2.11: Cálculo de LOC para a classe Multiplicacao.java 60
Figura 2.12: Cálculo de NOC para o projeto OperacoesAritmeticas 61
Figura 2.13: Cálculo do NOM para a classe Msg.java..... 61
Figura 2.14: Classe Multiplicacao.java com NAC = 2..... 62
Figura 2.15: Atributo estático PI e método estático sqrt() da classe Math.java . 63
Figura 2.16: Classe Multiplicacao.java com PAR = 2..... 64
Figura 2.17: Projeto OperacoesAritmeticas com NOP = 4..... 65
Figura 2.18: Exemplo de Acoplamento Aferente 66
Figura 2.19: Exemplo de Acoplamento Eferente..... 66
Figura 2.20: Exemplo de pacote instável..... 67
Figura 2.21: Exemplo de pacote estável..... 67
Figura 2.22: Gráfico Abstração X Instabilidade..... 68
Figura 2.23: Representação do processo de recuperação da informação 85
Figura 2.24: Transformação da sentença na estrutura sintática para a forma lógica..... 86
Figura 2.25: Exemplo de estrutura sintática da frase..... 87
Figura 4.1: Fluxo do método desta pesquisa 107
Figura 4.2: Fluxo dos experimentos desta pesquisa 110
Figura 4.3: Fluxo do experimento 1 para montar a base de dados histórica 111
Figura 4.4: Fluxo do experimento 2 para calcular o Grau de Contribuição do Desenvolvedor 112
Figura 4.5: Fluxo do experimento 3 para verificar as mensagens dos *commits* e *issues*.....113
Figura 5.1: Representação do Modelo 3Cs dos Sistemas Colaborativos utilizando Diagrama de Venn.....116
Figura 5.2: Representação do Modelo 3Cs incluindo a contribuição utilizando o Diagrama de Venn..... 122
Figura 6.1a: Split do código-fonte do *commit* 8 do Projeto Elastic/Search antes da contribuição do desenvolvedor 136
Figura 6.1b: Split do código-fonte do *commit* 8 do Projeto Elastic/Search após a contribuição do desenvolvedor 137
Figura 6.2a: Split de trecho do código-fonte do *commit* 10 do Projeto

Elastic/Search antes da contribuição do desenvolvedor 138
Figura 6.2b: Split de trecho do código-fonte do *commit* 10 do Projeto Elastic/Search após a contribuição realizada pelo desenvolvedor..... 139
Figura 6.3: Exemplo de *dashboard* da aplicação..... 149
Figura 6.4a: Exemplo de gráfico dos valores das métricas por artefato do projeto 150
Figura 6.4b: Exemplo de "recomendação" por métrica e artefato para o desenvolvedor ter condições de melhorar o código..... 151
Figura 6.5: Exemplo de uma mensagem de *commit*..... 154
Figura 6.6: Exemplo de um *issue* 154
Figura 6.7: Método do experimento de análise léxica das mensagens de *commit* e *issues*..... 156
Figura 6.8: Resultado da pesquisa no Dicionário Thesaurus e os pesos da polaridade das palavras de ordem..... 157
Figura A.1: Split de trecho do código do *commit* 8 do Projeto Elastic/Search 185
Figura B.1: Exemplo dos dados das métricas dos *commits* realizados referente ao Projeto Elastic/Search..... 188
Figura B.2: Exemplo dos dados das métricas dos *commits* no formato XML referente ao Projeto Elastic/Search..... 188
Figura B.3: Exemplo de valor do GC[d] e *dashboard* do nível de qualidade do código no Projeto Elastic/Search 189
Figura B.4: Exemplo de retorno do valor da métrica em cada artefato do código em forma de gráfico de barras do Projeto Elastic/Search..... 190
Figura B.5: Exemplo de retorno das "recomendações" com base nos valores das métricas de cada artefato do código do Projeto Elastic/Search 190

Lista de Gráficos

Gráfico 6.1: Aplicação da métrica WMC na classe NodeServlet do projeto Elastic/Search 136
Gráfico 6.2: Aplicação da métrica CCM ao método embedImages do projeto CSSEmbed 140
Gráfico 6.3: Aplicação da métrica CCM do método uri do projeto Elastic/Search 141
Gráfico 6.4: Aplicação das métricas NSF, NSM e LCOM* da classe CSSURLEmbedder do projeto CSSEmbed 142
Gráfico 6.5: Aplicação da métrica WMC na classe CSSURLEmbedder do projeto CSSEmbed 142
Gráfico 6.6: Aplicação das métricas Ce, Ca e RMI do pacote net.nczonline.web.cssembed do projeto CSSEmbed 143
Gráfico A.1: Métrica WMC da classe NodeServlet.ServletRestChannel do Projeto Elastic/Search 180
Gráfico A.2: Métrica WMC da classe ServletRestChannel do Projeto Elastic/Search 181
Gráfico A.3: Métrica WMC da classe NodeServlet do Projeto Elastic/Search 181
Gráfico A.4: Métrica CCM ou VG do método Uri do Projeto Elastic/Search . 182
Gráfico A.5: Métrica CCM ou VG do método NodeServlet.ServletRestChannel#sendResponse do Projeto Elastic/Search ... 182
Gráfico A.6: Métrica Ce do pacote org.elasticsearch.wares do Projeto Elastic/Search 183
Gráfico A.7: Métrica Ca do pacote org.elasticsearch.wares do Projeto Elastic/Search 184
Gráfico A.8: Métrica RMI ou I do pacote org.elasticsearch.wares do Projeto Elastic/Search 184
Gráfico A.9: Métricas coletadas do *commit* 8 do Projeto Elastic/Search 185
Gráfico A.10: Métricas coletadas do *commit* 18 do Projeto Elastic/Search 186

Lista de Quadros

Quadro 2.1: Intersecção de pares de métodos 57
Quadro 2.2: Termos e consultas da revisão realizada nas bases de conhecimento 71
Quadro 2.3: Critérios de exclusão aplicados 73
Quadro 2.4: Total de artigos classificados pela QP abordada..... 74
Quadro 3.1: Alguns termos e consulta da pesquisa realizada nas bases de conhecimento 92
Quadro 3.2: Total de artigos levantados pela pesquisa realizada 93
Quadro 3.3: Resumo dos artigos relacionados a Métricas de Software 94
Quadro 3.4: Medidas em ambientes de colaboração 100
Quadro 5.1: Definição de cooperar/cooperação e contribuir/contribuição.....117
Quadro 5.2: Grau de contribuição a partir do GC[d] calculado 129

Lista de Tabelas

Tabela 2.1: Escala da Complexidade Ciclomática..... 53
Tabela 2.2: WMC e Riscos Associados 54
Tabela 2.3: Lista das métricas e as faixas de avaliação de risco na qualidade do produto 76
Tabela 2.4: Resumo da influência das métricas na qualidade do código 81
Tabela 2.5: Compilado das métricas de código fonte, grau de risco de influência e item de qualidade do produto de software influenciado 82
Tabela 5.1: Pesos por grau de risco das métricas de código-fonte 126
Tabela 5.2: Recomendações retornadas para o desenvolvedor baseado no retorno não desejável/bom/ausente/baixo das métricas de código-fonte 129
Tabela 6.1: Base de dados dos experimentos 134
Tabela 6.2: Base de dados dos experimentos 145
Tabela 6.3: Cálculo do grau de contribuição de cada desenvolvedor do projeto Elastic/Search 147
Tabela 6.4: Cálculo do grau de contribuição de cada desenvolvedor do projeto CSSEmbed 147
Tabela 6.5: Cálculo do grau de contribuição de cada desenvolvedor do projeto Elastic/Hadoop 148
Tabela 6.6: *Issue* do *commit* 28 e suas métricas de código-fonte McCabe e WMC 158

Lista de Siglas

A – Abstração

APS – A*utomated Production Systems*

Ca - Acoplamento aferente

CBO – Acoplamento entre Classes

CCM – Complexidade Ciclomática de McCabe

Ce - Acoplamento eferente

CK – Métricas de Chidamber e Kemerer

CMMI – *Capability Maturity Model Integration*

CSCL – *Computer Supported Colaborative Learning*

CSCW – *Computer Supported Cooperative Work*

CSD – *Collaborative Software Development*

CSE – *Collaborative Software Engineering*

D – Distância normalizada da sequência principal

DIT – Profundidade da Árvore de Herança

Dn – Distância normalizada da sequência principal

EPL – *Eclipse Public License*

FOUT – *Fan Out*

GQM – *Goal, Question and Metrics*

HCI – *Human-Computer Interface*

I – Métrica de instabilidade

IDE – *Integrated Development Environment*

IEEE – *Institute of Electrical and Electronic Engineers*

ISO – *International Organization for Standardization*

LOCC – Linhas de Código por Classe

LCOM* – Falta de coesão em Método

LOC – Linhas de código

LLVM – *Low Level Virtual Machine*

MLOC – Linhas de código do método

MIT – *Massachusetts Institute of Technology*

MPS.Br – Melhoria do Processo de Software Brasileiro

NAC – Número de atributos por Classe

NBD – Profundidade de blocos aninhados

NOA – Número de atributos por classe

NOC – Número de filhos

NOI – Número de interfaces

NOM – Número de métodos

NOP – Número de parâmetros

NOP – Número de pacotes

NORM – Número de métodos sobrescritos

NSF – Número de atributos estáticos

NSM – Número de métodos estáticos

OO – Orientado a Objeto

OSS – *Open Source Software*

PHP – *Personal Home Page / Hypertext Preprocessor*

PLN – Processamento de Linguagem Natural

PMD – Analisador de código fonte

RFC – Resposta para uma Classe

RI – Recuperação da Informação

RMA – *Robert Martin Abstraction*

RMI – *Robert Martin Instability*

SCM – *Source Code Management*

SCV – Sistema de Controle de Versão

SD – *Software Development*

SGBD – Sistema Gerenciador de Banco de Dados

SIX - Índice de especialização

SLOC – *Source Line of Code*

SVM – *Support Vector Machines*

TLOC – Total de linhas de código

VID – *Version Identifier*

WMC – Métodos Ponderados por Classe

XML – *Extensible Markup Language*

RESUMO

O desenvolvimento de software pode ser considerado uma atividade colaborativa dependente de tecnologia e desempenhado por um grupo de pessoas. Os sistemas colaborativos, nos quais podem ser inserido o desenvolvimento colaborativo de software, contemplam três atividades: coordenação, comunicação e cooperação. Pesquisas mostram que poucos são os estudos que quantificam os benefícios da colaboração por serem de difícil mensuração. Esta pesquisa propõe uma atividade específica que se chama *contribuição*, que é o que se considera um dos passos para avaliar a cooperação. A contribuição pode ser rastreada, coletada e quantificada. Conhecendo-se o grau de contribuição do desenvolvedor, os gestores de projeto terão subsídios para compor de forma eficiente as equipes de desenvolvimento de software, além de permitir que os desenvolvedores busquem solucionar suas deficiências em relação ao desenvolvimento de código, resultando, teoricamente, em melhoria da qualidade do software. Esta tese propõe também o cálculo do grau de contribuição do desenvolvedor no desenvolvimento colaborativo de software utilizando métricas de qualidade do código-fonte. Os experimentos desenvolvidos para comprovar a hipótese proposta verificam que é possível recuperar os valores das métricas do código-fonte, a cada *commit* realizado pelo desenvolvedor, e mapear estes valores, identificando as variações entre um *commit* e outro. Mostram também que as variações das métricas entre um *commit* e outro, as combinações das métricas que influenciam os itens de manutenibilidade, reusabilidade, testabilidade e complexidade do software e o risco de os valores das métricas influenciarem nos índices de qualidade do software permitem o cálculo do grau de contribuição do desenvolvedor. E por fim, demostram que é possível recuperar das mensagens deixadas nos *commits* e *issues* possíveis ”ordens” para os desenvolvedores realizarem as tarefas. Os resultados dos experimentos validaram o cálculo do grau de contribuição do desenvolvedor e mostraram que as métricas de código-fonte têm condições de indicar o grau desta *contribuição,* em relação a qualidade do produto de software, com o *commit* que ele realizou no projeto.

Palavras-Chave: Contribuição em Desenvolvimento de Software, Desenvolvimento Colaborativo de Software, Métricas de Qualidade de Software.

ABSTRACT

Software development can be considered a collaborative activity dependent on technology and performed by a group of people. Collaborative systems, where collaborative software development can be classified, include three activities: coordination, communication and cooperation. Research shows that there are few studies quantifying the benefits of collaboration, since it is difficult to measure. This study proposes a specific activity, *The Contribution*, which is a step forward to evaluate cooperation. It can be tracked, collected and quantified. By knowing the developer's contribution, project managers can create software development teams efficiently, also allowing developers seek to solve their code development shortcomings, resulting in software quality improvement. This thesis also proposes an equation to calculate developer's contribution degree in collaborative software development using source code quality metrics. The experiments developed to prove the proposed thesis show that it is possible to retrieve source code metrics values, at each developer´s commit, and map these values, identifying variations between one commit and another. Experiments also show that metric´s variations between one commit and another, the combination of metrics that influence software maintainability, reusability, testability and complexity, and metric´s risk levels values showing probably influence on software quality indexes allow this research to calculate developer contribution degree. And finally, they show it is possible to retrieve from messages left in commits and issues possible "orders" that carry out developer´s tasks. These results validated calculation of developer´s contribution degree and proved that source code metrics were able to indicate how developer had contributed in software product quality by his commit accomplished in the project.

Key Words: Contribution in Software Development, Collaborative Software Development, Software Quality Metrics.

Capítulo 1

Introdução

Aprender é uma atividade decorrente da contínua busca pela adaptação ao meio ambiente físico e social, o que ocorre em todos os momentos de nossas vidas. Aprende-se muito com os outros, resolvendo problemas em conjunto, obtendo explicações sobre problemas já resolvidos, explicando nossas soluções, debatendo sobre vantagens e desvantagens de uma determinada escolha, fazendo e recebendo críticas, contestando-as, reconsiderando-as, construindo sínteses coletivas, dentre outras atividades em grupo (Fagundes *et al.*, 1999).

Os jovens da Geração Y (Geração Digital) e Geração Z (Geração da Internet) se acostumaram às novas tecnologias, foram moldados pelo ciberespaço e utilizam a internet para obter informações e realizar pesquisa desde a idade escolar. Estas gerações buscam informações incessantemente, não estão mais restritos a livros e revistas, e se acostumaram a investigar as informações antes de tomarem decisões sobre qualquer assunto. Estas gerações gostam de integrar vida doméstica à profissional, estão acostumados a conviver com a diversidade social e cultural, não possuem uma noção rígida de limite de tempo, espaço de lazer, trabalho e estudo. Rejeitam hierarquias e burocracias, gostam de jogar no trabalho, trabalhar em casa, horários flexíveis e remuneração baseada no desempenho (Tapscott e Williams, 2008).

A Web 2.0 ou Web Social permitiu o surgimento e popularização das mídias sociais como *blog, wiki, podcast* e redes sociais *online* nas quais o foco central é a comunicação entre pares, a troca de experiências, o compartilhamento e a construção coletiva. Além do uso para a socialização e a interação, as mídias sociais estão sendo

utilizadas no compartilhamento explícito de estruturas conceituais, o que tem promovido o surgimento de novos tipos de ambientes de trabalho colaborativo, como o *Knowledge Work Environment* (Antilla, 2008). Estes ambientes de trabalho colaborativo atendem as necessidades das gerações X e Y, que estão no mercado de trabalho, e permitem o compartilhamento de conhecimento.

Dentro deste contexto pode-se citar os Sistemas Colaborativos. Ellis e colaboradores (Ellis *et al.*, 1991) classificam em três dimensões os sistemas que dão suporte ao trabalho em grupo: comunicação, coordenação e colaboração. A comunicação é caracterizada pela troca de mensagens, pela argumentação e pela negociação entre pessoas; a coordenação é caracterizada pelo gerenciamento de pessoas, atividades e recursos; e a cooperação é caracterizada pela atuação conjunta no espaço compartilhado para a produção de objetos ou informações.

Para que um trabalho seja caracterizado como colaborativo, é preciso ocorrer comunicação, coordenação e cooperação, conforme representado no Modelo 3C (Pimentel *et al.*, 2006). No âmbito do desenvolvimento de software, o ambiente onde ocorre o desenvolvimento colaborativo conta com ferramentas automatizadas que auxiliam na organização e gestão do trabalho colaborativo, como os *groupwares*.

Os sistemas computacionais desenvolvidos atualmente são muito complexos para serem desenvolvidos por uma única pessoa. Estudos realizados por (Tapscott e Williams, 2008), (Campagnolo *et al.*, 2009) (Whitehead, 2010) e (Mohtashami *et al.*, 2011), mostram que o desenvolvimento de software pode ser considerado como uma atividade colaborativa, dependente de tecnologia e desempenhada por grupos de pessoas. Considera-se também que, durante o desenvolvimento de software, as atividades colaborativas são tão ou mais frequentes que as atividades individuais. Estas atividades normalmente envolvem pessoas com diferentes papéis, tais como: gerentes/coordenadores de projetos, analistas de negócio, arquitetos de software, analistas de sistemas, projetista de interface gráfica, desenvolvedores/codificadores, testadores, engenheiros de requisitos, entre outras áreas. Todos estes profissionais trabalham com diferentes tipos de artefatos em diferentes atividades, sendo a maioria delas atividades que requerem trabalho em equipe e colaboração para serem desenvolvidas.

O desenvolvimento de software pode então ser caracterizado como um ambiente no qual ocorre desenvolvimento colaborativo, onde tem-se bem definidas as atividades de coordenação de projeto, desempenhado por um dos integrantes da equipe, os

momentos bem definidos de troca de informações entre os integrantes da equipe de desenvolvimento caracterizando a comunicação, e a cooperação entre os integrantes, onde cada um desenvolve sua atividade para um propósito comum, o produto de software.

Segundo De Souza e colegas (De Souza *et al.*, 2012) os próprios engenheiros de software reconhecem que a atividade de desenvolvimento de software é uma atividade colaborativa. Em razão disto, profissionais e pesquisadores da área de desenvolvimento de software criaram diversas práticas e ferramentas que enfatizam a colaboração e a coordenação das atividades. Dentre as práticas colaborativas no desenvolvimento de software podem ser citados os processos de software envolvendo a coordenação da colaboração entre os vários profissionais com tarefas e papeis específicos que definem, por exemplo, a sequência das atividades, os modelos e produtos a serem entregues, entre outros; a programação em pares, que é uma prática proposta no método ágil *eXtreme Programming*; a construção colaborativa de modelos, na criação de diagramas de casos de uso, classes entre outros; construção colaborativa de código-fonte, entre outras práticas.

Dentre as ferramentas disponíveis para desenvolvimento colaborativo de software tem-se os sistemas de gestão de defeitos (*bugs*) como o Bugzilla (https://www.bugzilla.org), o JIRA (https://jira.atlassian.com); os sistemas comerciais para desenvolvimento colaborativo de software como o IBM *Rational Team Concert* (www.ibm.com/software/products/pt/rtc), o Microsoft *Visual Studio Team System* (https://www.visualstudio.com); e os sistemas de controle de versão de software, que controlam a evolução e integridade dos produtos de software por meio do controle e registro das mudanças, como o GitHub (www.github.com), GitLab (www.gitlab.com) e Bitbucket (www.bitbucket.org) que cumprem o papel de repositório de projetos de software. Nestes repositórios de projetos de software o usuário pode hospedar projetos comerciais e/ou *open source*, voltados para o desenvolvimento colaborativo de software.

Nestes ambientes é possível visualizar as atividades de coordenação, uma vez que cada projeto tem um responsável pela divulgação dos requisitos deste projeto, cronograma e inserção dos códigos desenvolvidos pelos integrantes da equipe no projeto. A comunicação entre os integrantes do grupo pode ser visualizada nos *chats*, mensagens e documentação internos ao projeto. Já a cooperação pode ser caracterizada pelas sugestões de metodologias de desenvolvimento dos requisitos e propostas de novos requisitos pelos integrantes da equipe de desenvolvimento. Estes repositórios

disponibilizam uma quantidade considerável de dados que podem gerar informações consistentes e conhecimento relevante. Estes dados podem ser o histórico de versões de um sistema, documentação deixada pelos desenvolvedores durante o projeto, requisitos de projeto solicitados aos desenvolvedores, entre outros. Basicamente o histórico dos projetos está armazenado nestas ferramentas de controle de versionamento.

Nesta pesquisa, está sendo considerado, conforme (Pimentel *et al.*, 2006), que um sistema colaborativo constitui um ciberespaço específico onde o usuário pode criar novas formas de trabalho e interação social. Conforme (Shah, 2010), grandes empresas e projetos complexos necessariamente envolvem muitas pessoas, que são levadas a colaborar para atingir seus objetivos. Assim, assume-se que um projeto de desenvolvimento de software pode ser considerado uma atividade colaborativa, desempenhada por grupos de pessoas e dependente de tecnologia (Thompson *et al.*, 2009), (Campagnolo *et al.*, 2009) e (Mohtashami *et al.*, 2011).

Pode-se dizer então, que o desenvolvimento colaborativo de software é condizente com as necessidades das novas gerações no sentido de que os jovens desejam colaborar, interagir e compartilhar, sem uma hierarquia rígida, com flexibilidade de horário e lugar, favorecendo a criação e a informalidade. Esta forma de trabalho mostra-se então promissora considerando as novas gerações Y e Z que estão no mercado de trabalho atualmente, e também na visão das organizações desenvolvedoras de software, as quais têm oportunidade de trabalhar com equipes de desenvolvimento geograficamente dispersas em um mesmo projeto de software. Este formato de indústria de software tem trazido benefícios competitivos de custo e prazo no desenvolvimento de software (Franzago *et al.*, 2018) e (Middleton *et al.*, 2018).

Considerando então o Modelo 3Cs apresentado e “mapeando-o” para o ambiente de desenvolvimento colaborativo de software, pode-se dizer que a coordenação é responsável pelo gerenciamento do projeto e da equipe de desenvolvimento. A comunicação pode ser dividida entre comunicação explícita, que seria a comunicação direta incluindo a troca de e-mails, reuniões formais de projeto e reuniões informais entre os membros do projeto, e a comunicação implícita ou indireta que é observada, por exemplo, nos comentários deixados pelos desenvolvedores dentro dos códigos fonte, nos *commits* realizados e mensagens trocadas dentro da ferramenta que gerencia o ambiente colaborativo. Já a cooperação pode ser identificada por sugestões, auxílio técnico e comentários gerais que ocorrem entre os membros do projeto, além da efetiva

contribuição individual de cada membro, desenvolvendo e realizando *commit* de código-fonte no software que está sendo desenvolvido.

Neste contexto, percebe-se o papel importante assumido pelos desenvolvedores: construir (escrever) o código-fonte da solução. Quando o desenvolvedor escreve o código e disponibiliza para o coordenador (gerente do projeto) inserir no projeto (*commit*) pode-se dizer que está havendo uma contribuição. Isto é, o indivíduo não está somente cooperando com sugestões para o bom desempenho do projeto, mas efetivamente contribuindo, por meio de seu esforço cognitivo, inteligência e trabalho mensurável, com o produto de software que está sendo desenvolvido. Considera-se, no contexto desta pesquisa, que a contribuição ocorre quando efetivamente é realizada uma atividade que altera a situação do software que, em função de sua característica de mudança no código do sistema, pode ser rastreada. Desta forma, pode-se considerar que a cooperação tem relação direta com a contribuição.

Uma vez definido este contexto colaborativo, formado por diferentes participantes, que contribuem de diferentes formas, podemos nos perguntar: como é possível medir, ou mesmo comprovar, que está havendo uma efetiva colaboração? Na comunidade de CSCW (*Computer Supported Cooperative Work* ou Trabalho Colaborativo Suportado por Computador), por exemplo, é senso comum entre (Thompson *et al.*, 2009), (Whitehead, 2010), (De Souza *et al.*, 2012), (Magdaleno, 2015) (Franzago *et al.*, 2018) que favorecer a colaboração traz benefícios para um projeto colaborativo.

A medida da colaboração, em senso mais amplo, não é trivial. Pesquisadores de diversas áreas tentaram medir a colaboração em sistemas de CSCW ((Mattsson, 2011), (Thompson *et al.*, 2009), (Baker *et al.*, 2002), (Cugini *et al.*, 1997)) e mesmo dentro de equipes de desenvolvimento de software ((Mohtashami *et al.*, 2011), (Schwind e Wegmann, 2008), (Cook e Churcher, 2004), (van der Hoek e Sarma, 2003), (Robillard e Robillard, 2000), (Johnson *et al.*, 1997)). Percebe-se que nestes projetos a medida é, na maioria dos casos, subjetiva. Trabalha-se com mensagens trocadas entre participantes, relatos de reuniões, entre outras formas de comunicação ((Mattsson, 2011), (Thompson *et al.*, 2009), (Baker *et al.*, 2002), (Cugini *et al.*, 1997)), e outros relacionam a coordenação ((Mohtashami *et al.*, 2011), (Schwind e Wegmann, 2008), (Cook e Churcher, 2004), (van der Hoek e Sarma, 2003), (Robillard e Robillard, 2000), (Johnson *et al.*, 1997)). Não há disponível na literatura, até onde vai nosso conhecimento, pela pesquisa aqui realizada, uma medida que seja prática (automática e não invasiva) capaz

de mensurar se há colaboração no contexto do desenvolvimento de software, e qual é o seu grau de intensidade.

No desenvolvimento de software, o coordenador do projeto colaborativo é o participante responsável, dentre outros, pelo respeito ao cronograma previamente estabelecido e pela aplicação das melhores práticas a fim de assegurar a qualidade do produto em desenvolvimento. Esta qualidade pode estar diretamente ligada ao grau de colaboração e comprometimento dos participantes da equipe de desenvolvimento. E ainda que não tenha sido estabelecida cientificamente esta relação, o bom senso predominante acredita que quanto maior a colaboração entre os participantes, melhores as chances de se chegar ao produto final mais rapidamente e em melhores condições de qualidade (Thompson *et al.*, 2009), (Whitehead, 2010) e (Magdaleno, 2015) (Franzago *et al.*, 2018).

1.1. Motivação

Independentemente de todos os benefícios conhecidos, promover uma colaboração eficaz da equipe continua a ser um desafio. Na área de Engenharia de Software a colaboração é fundamental para as organizações lidarem com os desafios dos ambientes modernos de negócio, que envolvem flexibilidade, adaptação e estar aberta a inovação (Chesbrough, 2006). Segundo Thompson e colegas (2009), a medição de colaboração é um tema de investigação aberto na literatura. Magdaleno e colegas (2015) apontam que ainda existe espaço de pesquisa para melhorar a estratégia de medição visando obter formas refinadas para determinar o potencial de colaboração no desenvolvimento de software. Os pesquisadores ainda discutem quais as práticas, processos e ferramentas que são capazes de fomentar a colaboração e controlar a sua execução (Araujo e Borges, 2007) (Mistrik *et al.*, 2010). Nessa área de pesquisa, o foco nas pessoas - envolvendo seus talentos, habilidades e conhecimento - e a preocupação com a colaboração e comunicação aparecem de forma recorrente.

As atividades e as pessoas envolvidas no desenvolvimento de software buscam melhores formas de desenvolver softwares de qualidade. Assim, uma das motivações desta pesquisa é a de que sendo possível aferir o nível de colaboração em uma equipe, ações podem ser tomadas para intensificar o rendimento desta equipe, buscando maximizar a qualidade do software produzido. Porém, a literatura em relação ao desenvolvimento colaborativo de software mostra que as pesquisas até então realizadas não apresentaram uma efetiva medida de colaboração, ou seja, uma fórmula computável

que um gerente de projeto pudesse usar para aferir a colaboração de sua equipe. Esta situação se deve ao fato de que a colaboração é uma medida subjetiva e seu conceito ainda é controverso, uma vez que os conceitos de colaboração e cooperação se sobrepõem e necessitam serem mais bem definidos. Assim, no âmbito desta pesquisa, não se tentou medir a colaboração existente entre os membros de uma equipe de desenvolvimento de software. A proposta aqui é de uma medida para o grau de contribuição individual de cada participante, pois acredita-se que a contribuição é parte integrante da colaboração, sendo assim o primeiro passo para sua obtenção.

Medir a contribuição individual dos participantes de um projeto colaborativo não é trivial, uma vez que cada participante (que pode assumir diferentes papéis ao longo do projeto) evidencia sua contribuição no projeto de forma diferente. Dentre os diferentes papéis existentes, trabalhou-se nesta pesquisa com os desenvolvedores, por serem elementos chave no processo. Os desenvolvedores "expressam" sua contribuição ao projeto por meio de modificações no código-fonte. Nesta pesquisa, assume-se que é possível medir a contribuição individual dos desenvolvedores por meio de inspeções realizadas no código-fonte e em eventual documentação produzida.

O histórico da análise da contribuição individual para cada desenvolvedor do projeto pode vir a definir o perfil atual do desenvolvedor, indicando em qual atividade este pode desempenhar melhor suas habilidades nas diversas tarefas do desenvolvimento de software, em quais conteúdos necessita melhorar seus conhecimentos e até em uma proposta de combinação de diversos perfis de desenvolvedor que juntos melhorariam o desempenho da equipe de desenvolvimento de software como um todo. A partir deste conhecimento pode-se contribuir com os gerentes de projetos de desenvolvimento de software a conduzir, e melhor organizar, as equipes de desenvolvimento de software, levando-se em consideração o perfil de cada desenvolvedor. Também, os desenvolvedores podem visualizar qual a influência que sua interferência no código-fonte pode gerar na qualidade do produto final do projeto.

A indústria de software também pode se beneficiar da medida de contribuição individual do desenvolvedor, considerando que atualmente os setores de Recursos Humanos das organizações avaliam seus desenvolvedores para promoção ou aumento de salário, baseado em tempo de serviço, muitas vezes por não terem outra forma de medir e comparar o desempenho de seus colaboradores. Solla e colegas em (Solla *et al.*, 2011) citam que medir a produtividade de um desenvolvedor é uma atividade desafiadora, e

dentre as novas métricas citadas por eles constam a qualidade do código, definindo que um bom código é fácil de ser entendido e lido, além de ser possível de ser testado, ser suportável e ser mantido. Outra métrica citada por esses autores é o desempenho do desenvolvedor em trabalhos colaborativos, quando dois ou mais desenvolvedores trabalham juntos, interagindo entre eles para resolver problemas durante o desenvolvimento do software. Fazendo uso do grau de contribuição individual do desenvolvedor, proposto nesta pesquisa, os setores de Recursos Humanos podem desenvolver programas de progressão de carreira a seus colaboradores/desenvolvedores baseados em desempenho profissional calculado por métricas concisas.

Considerando os avanços dos projetos de software voltados para OSS (*Open Source Software*) instalados em ambientes de controle de versão como GitHub, GitLab e Bitbucked, trabalhos como (Pinto *et al.*, 2016) e (Middleton *et al.*, 2018) citam o problema dos desenvolvedores que pouco se envolvem com o projeto no qual contribuem com *commits* que abrangem novos requisitos e correção de erros. Este pouco envolvimento dos desenvolvedores, em alguns casos, pode degradar a qualidade do código-fonte em função de uma contribuição de baixa qualidade. O grau de contribuição individual do desenvolvedor, resultado desta pesquisa, pode sinalizar um problema. Esta sinalização pode auxiliar tanto o gerente do projeto, que pode vir a não aceitar o *commit* deste desenvolvedor no projeto, quanto o próprio desenvolvedor, que tendo conhecimento do problema gerado por sua ação no código, pode vir a corrigir a forma de programar a alteração realizada. Esta ação consciente do desenvolvedor pode permitir que seja mantida, ou até mesmo, melhorada a qualidade do produto após o *commit* realizado.

A dívida técnica (*technical debt*) é outro ponto crítico para a indústria de desenvolvimento de software, que aumenta o seu custo de desenvolvimento com o passar do tempo. O termo *technical debt* tem ganhado importância e vem sendo expandido e refinado. No início das pesquisas a dívida técnica tratava de assuntos internos de qualidade na programação e atualmente seus estudos abrangem todo o ciclo de vida do software. A partir desta evolução nas pesquisas, surgiram diversas abordagens de dívida técnica, como dívida técnica de modelagem de software, dívida técnica de sistemas produzidos automaticamente (APS – *automated production systems*), dívida técnica de código e arquitetura de software, entre outras.

A dívida técnica pode se tornar um problema sério no desenvolvimento de projetos de software se não for dada a devida atenção. Muitas vezes gerentes de projeto

precisam decidir sobre o direcionamento dos projetos sem conhecer a dívida técnica já existente deste projeto, e o resultado destas decisões podem incorrer em atrasos de cronograma e diminuição na qualidade do produto deste projeto. O cálculo do grau de contribuição individual do desenvolvedor, objeto desta pesquisa, pode ajudar no gerenciamento do nível de qualidade do produto de software. A cada *commit* do desenvolvedor, é possível verificar se o nível de qualidade do produto de software degradou, melhorou ou manteve-se, desta forma, contribuindo para que a dívida técnica referente ao código e arquitetura do software se mantenha, ou que seja de conhecimento e gerência do gerente de projetos, permitindo uma melhor tomada de decisão. Estudos de Fontana e colegas (Fontana, *et al.*, 2012) e Capitán e Vogel-Heuser (Capitán e Vogel-Heuser, 2017) mostram a influência e relação das métricas de código-fonte na qualidade do código que pode vir a influenciar na dívida técnica do projeto de software.

Também o próprio desenvolvedor terá interesse em conhecer seu desempenho e influência na construção do código. Estudos revelam que os desenvolvedores se preocupam com a influência dos códigos que constroem e buscam, em ambientes colaborativos e de plataforma livre, soluções e respostas para seus questionamentos (Umarji e Seaman, 2009) (Kononenko *et al.*, 2016) (Tahir *et al.*, 2018) (Sae-Lim, Hayashi e Saeki, 2018).

Também a pesquisa de Lavallée e Robillard (Lavallée e Robillard, 2012) mostra que dentre os impactos dos processos de melhoria no desenvolvimento de software ainda existem obstáculos e oportunidades para estudos relativos à qualidade no desenvolvimento e nos produtos de software. Esta situação, assim como as supracitadas, mostram a importância e necessidade desta pesquisa.

1.2. Objetivos

Esta pesquisa tem como objetivo propor um método não-invasivo para medir o grau de contribuição individual entre os desenvolvedores participantes de um projeto colaborativo de desenvolvimento de software.

Os objetivos específicos compreendem:

- Analisar, por meio de uma revisão sistemática de literatura, o impacto dos valores das métricas de código-fonte nos itens de qualidade do produto de software;

- Empregar as métricas de código-fonte, capazes de identificar alterações nos itens de qualidade do produto de software, no cálculo do grau de contribuição individual do desenvolvedor em um ambiente de desenvolvimento colaborativo de software;
- Desenvolver um método não invasivo de coleta das métricas a cada *commit* realizado pelo "contribuinte";
- Desenvolver um protótipo para validar o método proposto.

1.3. Contribuições Científicas e Tecnológicas

Uma das contribuições científicas deste trabalho é a definição da equação para o cálculo do grau de contribuição individual do desenvolvedor, e por consequência, o conceito de "Contribuição" no desenvolvimento colaborativo de software. A partir da análise dos ambientes de desenvolvimento colaborativo de software foi possível perceber a diferença de atividades existentes dentro do contexto de cooperação abordado pelo Modelo 3Cs e a percepção de uma atividade específica que efetivamente promove alteração no contexto do produto de software, o código-fonte. Esta atividade foi denominada e conceituada, neste trabalho, como "Contribuição". A aplicação do termo contribuição em pesquisas analisadas a partir da revisão sistemática de literatura permitiram embasamento para esta denominação. A partir daí, foi aplicado esse conceito com sua fórmula de cálculo, no desenvolvimento colaborativo de software no nível de código-fonte como ponto de partida para o entendimento e geração da métrica de grau de contribuição. Este conceito considera que a contribuição do desenvolvedor, quando realiza uma alteração no código-fonte, pode vir a aumentar, diminuir, ou não influenciar a qualidade do produto deste software no que tange à complexidade, reusabilidade, testabilidade e manutenibilidade.

Para que fosse possível o cálculo do grau de contribuição individual do desenvolvedor, foi realizada uma revisão sistemática de literatura abordando as métricas de qualidade de código-fonte e suas influências na qualidade do produto de software, mais especificamente da complexidade, reusabilidade, testabilidade e manutenibilidade do código-fonte gerado pelo desenvolvedor em um ambiente colaborativo de desenvolvimento de software. Os estudos mostraram que a qualidade do código-fonte em relação à complexidade, reusabilidade, testabilidade e manutenibilidade pode ser medida

a partir das métricas de código-fonte, porém, tais pesquisas são isoladas e validam um dos itens de qualidade. Desta forma outra contribuição científica é a identificação das métricas de qualidade de software capazes de sinalizar as variações (aumento, diminuição ou não influência) que as alterações realizadas pelo contribuinte no código-fonte exerceram sobre a qualidade do produto, tendo em vista a manutenibilidade, testabilidade, complexidade e reusabilidade do código de forma conjunta.

A efetividade dessa inovação foi validada por meio de experimentos, primeiramente mostrando que existe variação nas métricas de código-fonte entre os *commits* realizados em um mesmo código-fonte por diferentes desenvolvedores nos ambientes colaborativos de desenvolvimento de software. Para tanto foram tomados como base de dados para análise os projetos OSS (*Open Source Software*) disponíveis no GitHub. A partir desta constatação, foi realizada uma revisão sistemática de literatura para responder as questões referentes a quais métricas de código-fonte influenciam qual item de qualidade de produto de software considerando a complexidade, reusabilidade, testabilidade e manutenibilidade. Esta revisão sistemática também permitiu a definição de uma tabela de grau de risco de as métricas de código-fonte influenciarem nos índices de qualidade do produto de software.

Constatou-se que existe um fator de "contribuição" do desenvolvedor em relação à qualidade de produto do software, quando este realiza uma alteração no código-fonte. Considerando também que a "contribuição" pode aumentar, diminuir ou manter o nível de qualidade do produto do software a partir do grau de risco das métricas, definiu-se a equação para o cálculo de grau de contribuição individual do desenvolvedor. A equação para o cálculo do grau de contribuição individual do desenvolvedor em um ambiente colaborativo de desenvolvimento de software é mais uma contribuição científica desta pesquisa.

Com base nos dados de projetos do GitHub, foi realizado um experimento para calcular o grau de contribuição do desenvolvedor a cada *commit* realizado no código-fonte. Este cálculo é apresentado em formato de *dashboard* informando os itens de qualidade de produto verificados pelo cálculo do grau de contribuição do desenvolvedor. A partir do retorno do experimento e dos estudos durante a revisão sistemática foi possível fornecer ao desenvolvedor um retorno, como se fosse uma "recomendação", para que ele tenha condições de melhorar a qualidade do código-fonte com base nos dados das métricas calculadas e em qual artefato o problema está localizado. O desenvolvedor aceitando a "recomendação" e melhorando o código terá seu grau de contribuição

recalculado a partir do novo *commit* realizado e assim sucessivamente.

Outro experimento focou na análise dos comentários deixados pelos desenvolvedores e gerentes de projeto em forma de mensagens de *commits* e *issues* dentro do ambiente de desenvolvimento do projeto de software. Este experimento buscou verificar se é possível identificar indicações do gerente de projeto nestes comentários que venham a influenciar no modo como o desenvolvedor realizou a alteração/construção do código-fonte.

Por fim, a contribuição tecnológica deste trabalho está na disponibilização do método, da base de dados coletada e do algoritmo desenvolvido para confirmar as hipóteses descritas na sequência.

1.4. Hipóteses de Trabalho

Neste trabalho a hipótese principal a ser validada afirma que é *possível medir o grau de contribuição individual entre os participantes de um projeto colaborativo de desenvolvimento de software no nível do código-fonte.*

Esta hipótese básica demanda três hipóteses secundárias, a saber:

- *Existe "contribuição" entre os integrantes de uma equipe de desenvolvimento colaborativo de software.*
- *Dada uma coleção de métricas de qualidade de software, estas são capazes de expressar o grau de contribuição individual dos participantes do projeto de desenvolvimento colaborativo de software de qualidade no nível de código-fonte.*
- *Os comentários e mensagens deixados pelos coordenadores e desenvolvedores nas mensagens dos commits e issues do projeto podem vir a mostrar influência na contribuição individual do desenvolvedor.*

1.5. Organização do Documento

Este trabalho está organizado em sete capítulos. Este primeiro capítulo é composto pelas considerações iniciais, a motivação que levou à realização desta pesquisa, os objetivos, as contribuições e as hipóteses que se desejou comprovar.

O Capítulo 2 apresenta a fundamentação teórica do trabalho que aborda os itens necessários para o entendimento da pesquisa realizada. Primeiramente é definido CSCW

(*Computer-Supported Cooperative Work*) e Desenvolvimento Colaborativo de Software, na sequência os conceitos de Colaboração e Cooperação e dos Sistemas de Controle de Versão. São apresentados os conceitos de Qualidade de Software, Medidas e Métricas e Métricas de Código-fonte para balizar a qualidade do produto de software, além da revisão sistemática de literatura realizada para identificar as métricas de código-fonte que podem ser associadas aos itens de qualidade do produto de software e suas faixas de influência para serem aplicadas no cálculo do grau de contribuição do desenvolvedor, também são apresentados os conceitos relacionados à Recuperação da Informação, de forma mais específica a análise de sentimentos, para verificação das mensagens de *commits* e *issues* deixadas pelos desenvolvedores e gerentes de projeto dentro do projeto de software.

O Capítulo 3 traz os principais trabalhos relacionados com a pesquisa proposta e o conjunto de características utilizado para a seleção desses trabalhos. Esse capítulo foi dividido em duas seções. A primeira seção trata dos trabalhos relacionados que focam nas métricas de software. Na segunda, são apresentados trabalhos que se referem às medidas de colaboração.

O Capítulo 4 aborda o método de pesquisa aplicado, as ferramentas utilizadas na implementação do método, o ambiente e cenário, e os experimentos realizados.

O Capítulo 5 apresenta o conceito de "contribuição" e do cálculo do grau de contribuição individual de um desenvolvedor em um projeto colaborativo de desenvolvimento de software com base nas métricas de código-fonte e suas influências nos itens de qualidade do produto de software.

O Capítulo 6 abarca os resultados e discussões relativos aos experimentos realizados.

No Capítulo 7 estão as conclusões da pesquisa, focando em sua relevância e contribuições, além das limitações e dificuldades encontradas e são apresentadas propostas de trabalhos futuros.

Para complementar o material apresentado, no Apêndice A podem ser verificados gráficos plotados para analisar as variações das métricas de código-fonte nos diferentes artefatos do projeto analisado. Já o Apêndice B contempla o protótipo da aplicação desenvolvida para o cálculo do grau de contribuição do desenvolvedor.

Capítulo 2

Fundamentação Teórica

Neste capítulo apresentamos os conceitos relacionados ao método proposto nesta pesquisa, tais como o CSCW e desenvolvimento colaborativo de software, Colaboração e Cooperação. Os Sistemas de Controle de Versão utilizados para controle e organização quando se desenvolve software de forma colaborativa também são apresentados, uma vez que definem o ambiente onde esta pesquisa foi realizada.

Na quarta seção são apresentados os conceitos de Qualidade de Software e Medidas e Métricas que permitem entender as formas de medição da qualidade do código-fonte de um projeto. Os conceitos referentes à Recuperação da Informação, relacionados à análise de sentimentos, aplicados na análise das mensagens dos *commits* e *issues* postados no projeto de software, também são apresentados neste capítulo.

2.1. Trabalho Colaborativo Suportado por Computador

Um grupo de trabalho pode ser visto como uma reunião de pessoas que interagem para compartilhar informações e tomada de decisão, com o objetivo de auxiliar o desempenho de cada membro em sua área de responsabilidade (Robbins e Finley, 2000). Com o passar do tempo a computação contribuiu de forma significativa para trabalhos colaborativos em grupo, independente da distância geográfica. Neste sentido, os grupos não interagem apenas localmente (face-a-face), surgindo as equipes virtuais, que são equipes que usam a tecnologia da informação para juntar seus membros dispersos, para que possam atingir seus objetivos comuns. Nessas equipes as pessoas interagem de forma o*nline*, colaborando entre si, utilizando meios de comunicação, como redes de computadores, videoconferências, correio eletrônico, “blogs”, sistemas de mensagens

instantâneas, redes sociais, entre outros. Neste contexto surge uma nova área de estudos chamada CSCW (*Computer-Supported Cooperative Work*), em português, Trabalho Cooperativo Suportado por Computador.

O termo CSCW foi cunhado em 1984 por Iren Greif do MIT (Massachusetts Institute of Technology) e Paul Cashman da *Digital Equipment Corporation* durante um workshop que visava entender como as pessoas trabalhavam (Grundin, 1994), e foi redefinido posteriormente por Ellis e colegas como "*sistema baseado em computador para dar suporte a grupos de pessoas engajadas numa tarefa comum (ou objetivo) e que provê uma interface para um ambiente compartilhado*" (Ellis *et al.*, 1991).

A área de CSCW é uma ciência multidisciplinar que estuda as formas de trabalho em grupo auxiliadas por tecnologia e comunicação, e pode ser dividida em duas áreas principais: CS e CW. No início havia uma confusão entre as áreas de CSCW e HCI (*Human-Computer Interaction*), em português Interação Humano-Computador, em 1980 definiu-se que HCI diz respeito ao suporte a indivíduos no contexto computacional e CSCW diz respeito a facilitar o trabalho entre grupos e entre indivíduos em um mesmo local e/ou distribuído, processo explicado por Ellis (Ellis *et al.*, 1991). Ainda neste contexto, surgiu em 1986 o termo CSCL (*Computer-Supported Collaborative Learning*) que se origina do CSCW e pode ser definido como a área das ciências da aprendizagem que estuda como as pessoas podem aprender em grupo com o auxílio do computador. Assim como o CSCW, o CSCL acredita que os sistemas computacionais podem suportar e facilitar o trabalho em grupo, mesmo não sendo face-a-face (Stahl *et al.*, 2006). Na Figura 2.1 é possível visualizar a interação entre as áreas de CSCW, HCI e CSCL, além de outras áreas de estudo da Ciência da Computação e da Ciência Cognitiva.

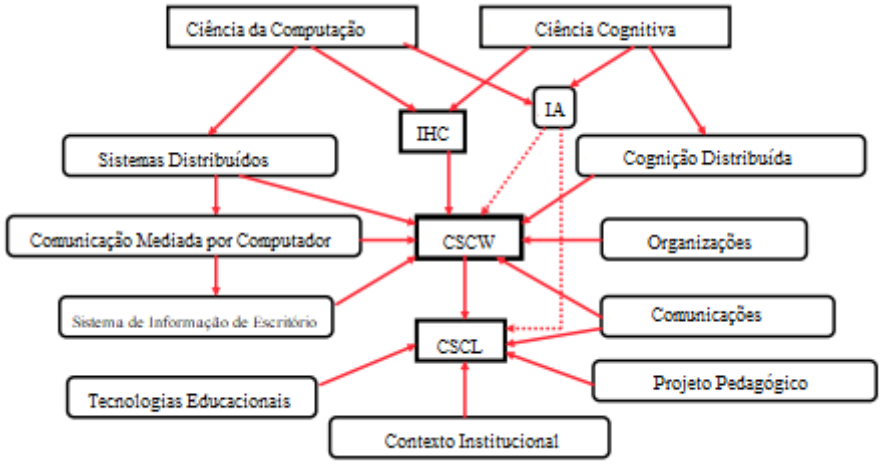


Figura 2.1:Relacionamento entre CSCW, HCI, CSCL e outras áreas da ciência
Fonte: Adaptado de (Morch, 2012)

O CSCW é um produto de um movimento social particular baseado em computador e não uma simples classe de tecnologia. Os pesquisadores da área de CSCW defendem que esta tecnologia frequentemente sugere meios de transformar a forma das pessoas trabalharem (Kling, 1991).

Segundo Moeckel e Forcellini (Moeckel e Forcellini, 2007) a ideia do conceito de colaboração em CSCW é de adicionar esforços, competências e habilidades, focando em um determinado objetivo, que pode ser a inovação tecnológica ou o desenvolvimento de sistemas de software de qualidade. Em alguns casos, os projetos tornam-se possíveis em função da efetiva colaboração entre pessoas interessadas que podem ser internas ou externas à organização.

Neste contexto, CSCW envolve estudos abordando o uso de tecnologias computacionais para suportar a colaboração entre membros de um projeto, distribuído ou não, de forma a criar um ambiente favorável à melhoria da qualidade, resolução de conflitos e redução de tempo e dinheiro no desenvolvimento de novos produtos. Estas ações são possíveis uma vez que o CSCW provê condições favoráveis para compartilhamento de informação entre os componentes da equipe de projeto distribuído, permitindo melhorar a eficácia das decisões coletivas, facilitando o uso, compartilhamento e retorno das informações. Assim, estimula a interação e reduz problemas como atividades desorganizadas, dominação entre os membros da equipe, pressão entre os membros e inibição.

É importante salientar a diferença entre o conceito de CSCW e *Groupware.* O termo *Groupware* é utilizado para descrever as ferramentas de CSCW que suportam o trabalho em grupo. Segundo Robbins e Finley (Robbins e Finley, 2000), *Groupware* é um software distribuído e interativo utilizado em redes de computadores voltado para o trabalho em grupo. Estas ferramentas permitem o compartilhamento de ideias com eficiência e acurácia entre os membros da equipe, simplificando processos e permitindo desenvolvimento paralelo de tarefas e aumentando o compartilhamento de conhecimento e experiência entre os membros da equipe de trabalho. Podem ser citados como exemplos de ferramentas de *Groupware* os correios eletrônicos (e-mails), sistemas de mensagens instantâneas (*instant message*), grupos de discussão, os *blogs*, as ferramentas de escrita colaborativas, salas de aula virtuais, *workflow*, entre outros.

Ellis e colegas (Ellis *et al.*, 1991) classificam em três dimensões os sistemas que dão suporte ao trabalho em grupo: comunicação, coordenação e colaboração. A comunicação é caracterizada pela troca de mensagens, pela argumentação e pela negociação entre pessoas; a coordenação é caracterizada pelo gerenciamento de pessoas, atividades e recursos; e a cooperação é caracterizada pela atuação conjunta no espaço compartilhado para a produção de objetos ou informações. Para que um trabalho seja caracterizado como colaboração, é preciso ocorrer comunicação, coordenação e cooperação, conforme representado no Modelo 3C (Pimentel *et al.*, 2006).

Trazendo este conceito para o momento atual percebe-se que grandes empresas e projetos complexos necessariamente envolvem muitas pessoas, que são levadas a colaborar para atingir seus objetivos. Um dos motivos para colaborar é a ocorrência da diversidade de opiniões em um grupo, possibilitando a análise de questões sob diferentes pontos de vista, e que potencialmente resulta em uma avaliação melhor. Já um requisito fundamental para a comunicação é o estabelecimento de uma linguagem ou protocolo compartilhado, de forma que as partes consigam se entender. Também é necessário haver certo nível de conhecimento compartilhado, de forma que o significado da comunicação seja compreendido e não apenas os sinais. Este conhecimento é chamado de senso comum *(Common Ground*) (Shah, 2010). Importante considerar também que a organização do grupo envolve a definição de papéis. Dentro de um grupo, participantes assumem diferentes papéis de forma permanente ou temporária. A distribuição de papéis assegura que as diversas funções do grupo estão previstas e cobertas. Cada papel está associado a um conjunto de funções e responsabilidades.

No contexto da área de Engenharia de Software, pode-se dizer que o software é produzido de forma colaborativa, com a participação de diversos especialistas (gerentes, analistas, desenvolvedores, testadores, entre outros). Estes especialistas trabalham em equipes e sofrem com organização e a multiplicação de documentos que são gerados durante as atividades, geralmente escritos de forma colaborativa. Por fim, o objetivo de um trabalho em grupo é produzir algum produto. É preciso um espaço compartilhado, ainda que virtual, para que todos possam trabalhar juntos. A atividade e as tarefas conjuntas, o espaço e os recursos disponíveis são importantes para definir os sistemas colaborativos (Shah, 2010).

Conforme será apresentado a seguir, as pesquisas que envolvem CSCW contribuem para dar suporte ao desenvolvimento colaborativo de software ((Mistrik *et al.*, 2010), (Cook e Churcher, 2005), (Robillard e Robillard, 2000)).

2.2. Desenvolvimento Colaborativo de Software

O desenvolvimento de software passou a ser uma atividade essencial, uma vez que as organizações dependem cada vez mais desta ferramenta para gerir seus negócios. Considerando que o desenvolvimento de software é uma atividade colaborativa, os membros da equipe de desenvolvimento de software precisam coordenar suas atividades, planejar novas ações, tomar decisões, realizar as atividades previstas e também se comunicar para desenvolver um software (Whitehead, 2007).

O coordenador de um projeto colaborativo de desenvolvimento de software é um participante responsável, dentre outros, pelo respeito ao cronograma previamente estabelecido e pela aplicação das melhores práticas a fim de assegurar a qualidade do produto. A qualidade pode estar diretamente ligada ao grau de colaboração e comprometimento dos participantes da equipe de desenvolvimento. Ainda que não tenha sido estabelecida cientificamente esta relação, o bom senso predominante acredita que quanto maior a colaboração entre os participantes melhores as chances de se chegar ao produto final mais rapidamente e em melhores condições de qualidade.

O desenvolvimento de software é uma atividade complexa, constituída de diversas etapas (análise, especificação, projeto, implementação, teste, entre outras), que também precisa ser executada com a utilização de um conjunto de ferramentas adequadas. Geralmente são ferramentas específicas para o desenvolvimento de software (uma IDE,

por exemplo), mas também ferramentas de uso geral como as de gestão de projetos e de trabalho colaborativo (Campagnolo *et al.*, 2009).

As ferramentas tradicionais de software usualmente proveem características limitadas de suporte à colaboração. Este conjunto de ferramentas precisa de mecanismos que suportem a cooperação, comunicação e coordenação, as três atividades que compõem os sistemas colaborativos. Visando o desenvolvimento colaborativo de software, dentro da cooperação pode-se citar o compartilhamento de espaço de trabalho, repositórios de dados, suporte para *merging* e *differencing*, configuração, teste, projeto, gerenciamento de processos. Considerando a comunicação é possível citar e-mails, mensagens, anotações, vídeo/áudio. Na coordenação considera-se travamento para acesso compartilhado, controle de versão, *hand-over*, auditoria. Diversos estudos de equipes, processos, ferramentas, projetos de mundo real têm mostrado a necessidade de utilizar de forma apropriada processos, gerenciamento de projetos, técnicas e seleção de ferramentas de forma que permitam a colaboração efetiva e eficiente ((Sarma *et al.*, 2010), (Damian *et al.*, 2010)).

Segundo De Souza e colegas (De Souza *et al.*, 2012) os próprios engenheiros de software reconhecem que a atividade de desenvolvimento de software é uma atividade colaborativa. Em razão disto profissionais e pesquisadores da área de desenvolvimento de software criaram diversas práticas e ferramentas que enfatizam a colaboração e a coordenação das atividades.

Dentre as práticas colaborativas no desenvolvimento de software podem ser citados os processos de software envolvendo a coordenação da colaboração entre os vários profissionais com tarefas e papéis específicos. Definem, por exemplo, a sequência das atividades, os modelos e produtos a serem entregues, entre outros. A programação em pares, que é uma prática proposta no método ágil *eXtreme Programming*, no qual uma dupla de programadores utiliza um único computador e um programador júnior codifica enquanto um programador sênior acompanha a codificação, esta prática foca nas estratégias buscando código de qualidade e evolução da equipe. A construção colaborativa de modelos, na criação de diagramas de casos de uso, classes entre outros. A construção colaborativa de código-fonte, entre outras práticas.

Dentre as ferramentas disponíveis para desenvolvimento colaborativo de software tem-se os sistemas de gestão de defeitos (*bugs*) (Figura 2.2) que utilizam técnicas como revisão por pares, teste unitário, teste de usuário, entre outros, e como exemplo, podem

ser citados o Bugzilla (https://www.bugzilla.org), o JIRA (https://jira.atlassian.com), entre outros; os sistemas comerciais para desenvolvimento colaborativo de software como o IBM *Rational Team Concert* (www.ibm.com/software/products/pt/rtc), o Microsoft Visual Studio Team System (https://www.visualstudio.com), entre outros; e os sistemas de controle de versão de software, que controlam a evolução e integridade dos produtos de software por meio do controle e registro das mudanças (Figura 2.3), como o GitHub (www.github.com), GitLab (www.gitlab.com) e Bitbucket (www.bitbucket.org), que cumprem o papel de repositório de projetos de software.

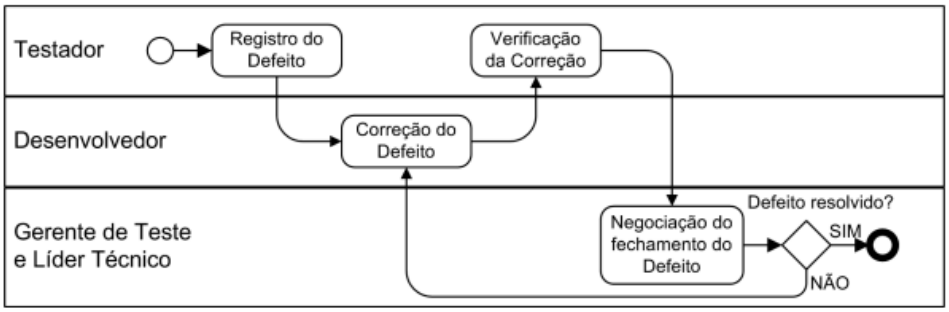


Figura 2.2*:* Sistemas de gestão de defeitos (bugs)

Fonte: Extraído de (De Souza *et al.*, 2012)

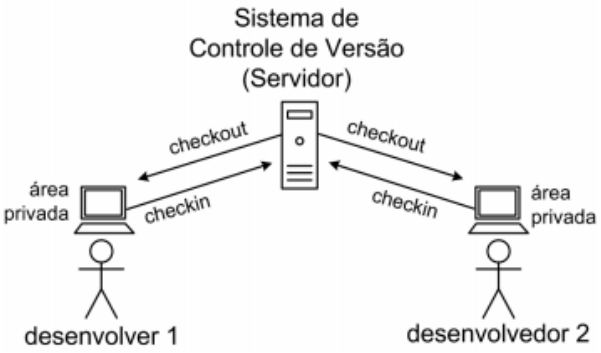


Figura 2.3: Sistemas de controle de versão

Fonte: Extraído de (De Souza *et al.*, 2012)

2.2.1. Colaborativo ou Cooperativo

Conforme descrito, o trabalho colaborativo é a essência do desenvolvimento de software moderno. Em um ambiente colaborativo, os desenvolvedores trabalham em

equipe, contribuindo para a realização das tarefas e dividindo responsabilidades comuns. Com a colaboração, o conhecimento do trabalho é compartilhado e passa a ser de propriedade geral a todos os envolvidos na equipe. Além disso, as responsabilidades e os riscos também são partilhados, e as falhas são mais rapidamente controladas e corrigidas.

Existe uma discussão quanto à aplicação da palavra "colaboração" no contexto da computação (Altmann e Pomberger, 1999). Graves (Graves, 1994) e Smith (Smith, 1996) acreditam que o termo "cooperação" seja mais abrangente, com distinções hierárquicas entre os participantes. Por outro lado, na "colaboração" existe um objetivo comum entre as pessoas que trabalham em conjunto, sem hierarquia rígida (Nitzke *et al.*, 1999).

Roschelle e Teasley (Roschelle e Teasley, 1995) também notaram uma distinção entre os termos. Para esses autores "cooperação" abrange divisão do trabalho entre os participantes, sendo que cada pessoa é responsável por esta parte da solução do problema. Já o termo "colaboração" é definido como sendo o engajamento mútuo entre os participantes em um esforço coordenado para resolver o problema juntos. Ainda, segundo Roschelle e Teasley (1995), a principal diferença entre a cooperação e a colaboração é que na cooperação as tarefas são divididas (hierarquicamente) em subtarefas independentes, e na colaboração os processos podem ser divididos em camadas entrelaçadas que são realizadas por todos os participantes envolvidos. Winer e Ray (Winer e Ray, 1994) fornecem uma definição de colaboração como sendo uma relação mutuamente benéfica e bem definida celebrada entre duas ou mais organizações com a finalidade de alcançar resultados que, em conjunto, serão mais propensos a serem alcançados do que sozinhos.

Quando se verifica a definição de CSCW, esta leva o termo cooperação em sua sigla, porém, nitidamente em sua definição a ideia central é a colaboração, e a cooperação é uma das atividades realizadas dentro do ambiente - comunicação, coordenação e cooperação – segundo Ellis (Ellis *et al.*, 1991). Pode-se considerar então que no CSCW, assim como no CSE (*Collaborative Software Engineering)*, em português Engenharia de Software Colaborativa, os processos podem ser divididos em camadas entrelaçadas que são realizadas por todos os participantes envolvidos (Roschelle e Teasley, 1995). Da mesma forma que pode ser considerada uma relação mutuamente benéfica e bem definida (análise, arquitetura, programação, entre outras) celebrada entre duas ou mais organizações (indivíduos ou equipes, no caso) com a finalidade de alcançar resultados que, em conjunto, serão mais propensos a serem alcançados do que sozinhos.

Ainda neste contexto a cooperação, como parte das atividades necessárias para que exista um sistema colaborativo suportado por CSCW e no CSE, é visivelmente definida como uma divisão do trabalho entre os participantes, onde cada pessoa é responsável por uma parte da solução do problema (Roschelle e Teaslley, 1995). Segundo Pimentel (Pimentel *et al.*, 2006), a cooperação é caracterizada pela atuação conjunta no espaço compartilhado para a produção de objetos ou informações no CSCW, e os diagramas e códigos-fonte no CSE, que, de forma geral, caracterizam a parte da solução do problema.

A partir dos estudos realizados foi possível montar uma linha do tempo dos conceitos envolvendo o desenvolvimento colaborativo de software, apresentada na Figura 2.4.

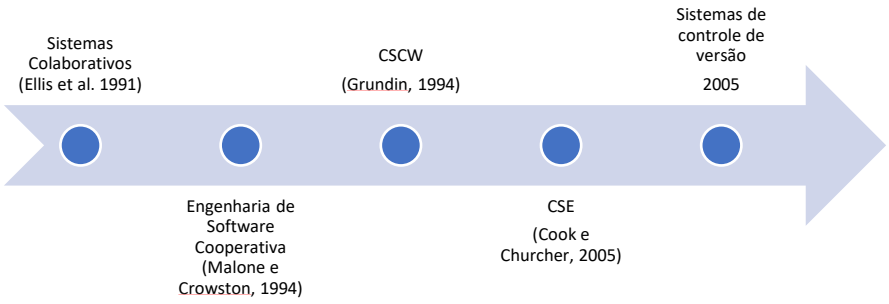


Figura 2.4: Linha do tempo do desenvolvimento colaborativo de software
Fonte: a própria autora.

Nesta pesquisa está sendo utilizado o termo colaborativo, desta forma iremos nos referenciar a Engenharia de Software Colaborativa e Desenvolvimento Colaborativo de Software. Além disto, para se chegar ao objetivo proposto, os dois termos comentados aqui não se encaixam na métrica de grau de contribuição que se almeja calcular. Desta forma, torna-se necessária a definição do termo "Contribuição" que será apresentada no Capítulo 5.

2.2.2. Engenharia de Software Colaborativa

Segundo Cook e Churcher (Cook e Churcher, 2004), a engenharia de software é inevitavelmente colaborativa e a área de CSE (*Collaborative Software Engineering)* analisa formas baseadas em computador para apoiar os programadores e ferramentas de comunicação, gestão de artefatos e coordenação das tarefas. A área de CSE é oriunda da

interseção de seis áreas da Computação – sistemas distribuídos, visualização de software, sistemas de *groupware*, interação humano-computador, processos de engenharia de software e gerenciamento de configurações, conforme pode ser visto na Figura 2.5.

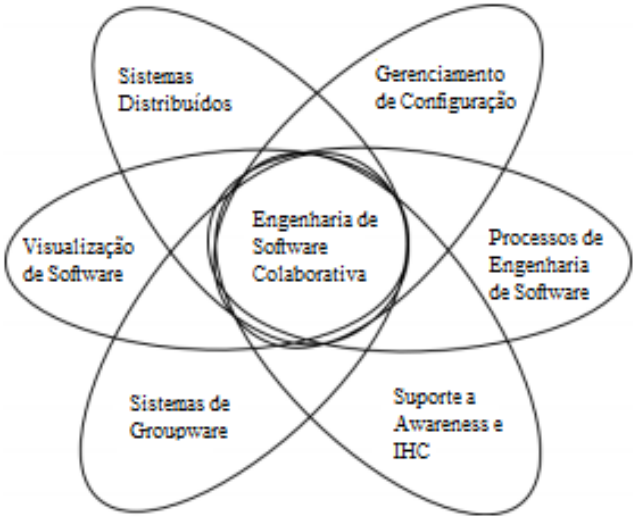


Figura 2.5: Áreas da Computação relacionadas à CSE
Fonte: Adaptado de (Cook e Churcher, 2005)

Dentre as ferramentas estudadas pela CSE, as de inspeção, *groupware* e SCM (*Source Code Configuration Management*) interessam para esta pesquisa. As ferramentas de inspeção no campo da CSE suportam basicamente duas funções: permite aos usuários do grupo, inspecionar de forma colaborativa o código e o projeto, ou a usuários individuais inspecionarem o código e o projeto que foi desenvolvido de forma colaborativa. As ferramentas de inspeção são diferentes das de gerenciamento uma vez que o foco é a inspeção e investigação dos artefatos de engenharia de software visando benefícios para futuro desenvolvimento e refinamento. Já as ferramentas de gerenciamento estão preocupadas com a coordenação do grupo e controle dos artefatos. (Cook e Churcher, 2005)

As tecnologias de *Groupware* permitem que as interfaces de usuário de ferramentas simples possam ser replicadas e compartilhadas em tempo real por múltiplos usuários. Esta tecnologia pode ser aplicada diretamente no sentido de auxiliar as ferramentas de desenvolvimento multiusuário de engenharia de software ((Bannon e Schmidt, 1991), (Grundin, 1994)).

Os sistemas SCM são fundamentais para a engenharia de software uma vez que permitem o controle de versão, ramificação e gestão de artefatos no intuito de aliviar a carga dos desenvolvedores nos esforços de controlar as múltiplas versões dos produtos de software. Mesmo os projetos desenvolvidos por um único usuário podem se beneficiar de um controlador de versões, pois incluem a habilidade de retornar versões anteriores. Os SCM permitem manter os arquivos de código coordenados permitindo *check-ins* regulares e *builds* de projeto, atividades que organizam os esforços de codificação. Desta forma o SCM aborda o fato de que diversas pessoas podem trabalhar em um mesmo código base. Para facilitar este controle dois esquemas são tipicamente empregados: travamento de acesso ao arquivo ou cópia do arquivo e posterior *merge.* Os SCM são sistemas fundamentais para a maioria das ferramentas de desenvolvimento de software, tanto colaborativo como para usuário único (Chu-Carroll e Sprenkle, 2000). Os sistemas de controle de versões são aplicações de SCM.

À medida que a engenharia de software se torna mais colaborativa a visualização das informações torna-se importante às ferramentas de desenvolvimento. Os resultados destas informações são apresentados aos engenheiros de forma a serem úteis e minimizando a sobrecarga da informação, o campo de métricas de software e visualização concentra-se na análise e extração de métricas úteis dos projetos de software. A geração de métricas por si só pode ser útil no auxílio e monitoramento das atividades dos usuários e progresso da equipe em um ambiente colaborativo. Neste sentido, com a dimensão da atividade individual do usuário junto à equipe e a habilidade das ferramentas CSE de capturar em tempo real a atividade dos programadores é possível ter um bom grau de acurácia e um bom nível de granularidade das medidas coletadas, o que aumenta o potencial de geração de novas métricas, tipos de visualizações e animações dos resultados obtidos (Cook e Churcher, 2005).

2.2.2. Colaboração no Desenvolvimento de Software

Conforme (Robinson e Sharp, 2010) e (Richardson *et al.*, 2010), observa-se que várias tendências em engenharia de software têm aumentado os desafios em torno da colaboração em desenvolvimento de software. Atualmente, é possível verificar que o desenvolvimento ágil trabalha com evolução rápida dos requisitos, arquiteturas emergentes e com isto demandam documentação, estratégias de organização de equipe e gerenciamento de projetos e de comunicação diferentes. Também as organizações que desenvolvem software de forma virtual com equipes distribuídas em diversos países, as

diversas empresas que compõem uma mesma organização e equipes que trabalham de forma distribuída exigem maior suporte para o compartilhamento de conhecimento, coordenação e cooperação entre os membros do grupo. Nestes casos é comum a comunicação ficar comprometida em função de fuso horário, diferente cultura e diferenças de idioma. Os projetos que utilizam software aberto apresentam desafios semelhantes, uma vez que são caracterizados por uma gama ampla de participantes, organizações e contribuições vindas de equipes e indivíduos com motivações diferentes. Outra tendência de desenvolvimento de software a ser considerada é o desenvolvimento global de software que acontece em equipes alocadas em diversos países com diferentes idiomas, diferentes culturas, utilizando diferentes ferramentas e, em alguns casos, diferentes processos de software e plataformas (Richardson *et al.*, 2010).

A colaboração no contexto de engenharia de software, segundo Murta e colegas (Murta *et al.*, 2010) envolve diferentes aspectos, como comunicação implícita e comunicação explícita entre os desenvolvedores, a divulgação das ações dos outros desenvolvedores, a coordenação das tarefas de desenvolvimento procurando evitar o retrabalho e buscando atingir os objetivos do projeto, manter uma memória compartilhada com históricos anteriores em relação a ações realizadas durante o desenvolvimento, e disponibilizar um espaço compartilhado onde o trabalho realizado por um desenvolvedor esteja disponível para os demais.

A transferência de conhecimento é um processo de comunicação que exige uma interação rigorosa e uma troca de informação eficiente. Considerando o desenvolvimento local de software, esta troca já é difícil de quantificar em relação ao tipo e a quantidade de conhecimento trocado entre os desenvolvedores. Considerando ainda o desenvolvimento remoto de software, onde estas trocas ocorrem por meio de uma infraestrutura tecnológica, torna-se necessário que este conhecimento trocado seja explícito e é preciso identificar meios eficientes de tornar este processo de troca de conhecimento explícito dinâmico e o mais abrangente possível (Mistrik *et al.*, 2010).

Robillard e Robillard (2000) citam que o trabalho colaborativo é estudado na área de engenharia de software sob diversos pontos de vista. Nakakoji e colegas (Nakakoji *et al.*, 2010) citam que se percebe nas equipes de desenvolvimento de software que a colaboração ocorre com ou sem comunicação explícita. Por um lado, os desenvolvedores se envolvem em colaboração por meio de artefatos sem comunicação explícita, como escrevendo comentários nos códigos que serão lidos por outros. Por outro lado, a

comunicação explícita torna-se necessária quando os desenvolvedores devem pedir informações aos seus colegas, que de outra forma não seria possível obtê-las. Estudos têm demonstrado que tanto as equipes de desenvolvimento de software local quanto distribuída se comunicam para adquirir informações necessárias ao desenvolvimento.

Robillard e Robillard apresentam uma classificação do trabalho colaborativo no desenvolvimento de software. Eles definem quatro tipos de trabalho colaborativo derivados de medições empíricas das atividades desenvolvidas durante o desenvolvimento de software: o trabalho colaborativo obrigatório, que são as reuniões formais agendadas; o trabalho colaborativo por chamada, que acontece quando os membros da equipe convocam uma reunião para resolver um problema, e que normalmente é de ordem técnica; o trabalho colaborativo *ad hoc*, quando os membros da equipe trabalham na mesma tarefa e ao mesmo tempo; e o trabalho individual que ocorre quando um membro da equipe trabalha por conta própria em uma tarefa relacionada com o projeto.

Dentre estes quatro tipos de colaboração indicados por Robillard e Robillard, o trabalho colaborativo *ad hoc* é o tipo de atividade que descreve as atividades de colaboração observadas quando as pessoas estão trabalhando ao mesmo tempo, na mesma sala e em tarefas relacionadas. A troca de informações neste caso caracteriza-se por ser informal e esporádica. A maior parte do tempo um companheiro de equipe interrompe seu trabalho para responder a uma pergunta de outro colega de equipe e observa-se também que algumas pessoas utilizam o e-mail para fazer perguntas e/ou dar respostas mesmo que estejam sentados na mesa ao lado. Desta forma, segundo Robillard e Robillard, medir a colaboração significativa nas atividades *ad hoc* é difícil, uma vez que existem poucas informações sobre o conteúdo e a utilidade destes intercâmbios. No entanto, eles são responsáveis por mais de um terço do tempo de trabalho, acontece entre dois companheiros de equipe e frequentemente precedem uma longa sessão de trabalho individual.

Robillard e Robilard concordam que as pessoas são itens importantes a serem avaliados no desenvolvimento de software, no entanto, pouco se sabe sobre o mecanismo e o valor da interação entre elas durante o desenvolvimento de um projeto de software.

Conforme observado nos estudos realizados, o desenvolvimento colaborativo de software considerando sua abrangência - tamanho da equipe do projeto, localização, muitas vezes distribuída, e tamanho do projeto em si - dependem de um sistema de

controle e organização robusto e eficiente para que seja possível a coordenação. Os Sistemas de Controle de Versão (SCV), do inglês *Version Control System*, também conhecido como SCM (*Source Code Management*), apresentam um ambiente adequado e controlável para o desenvolvimento colaborativo de software disponibilizando ferramentas de coordenação, comunicação e cooperação.

2.3. Sistemas de Controle de Versão

A natureza descentralizada e colaborativa dos projetos de software em organizações gerou a necessidade do uso de repositórios e ferramentas de gestão e acompanhamento de tarefas para facilitar o planejamento, coordenação, cooperação e comunicação entre os desenvolvedores. Este fato é característico em ambientes de desenvolvimento *open source* e ágil. O uso destas ferramentas entre a comunidade de desenvolvedores agiliza o processo de detecção de falhas e a elaboração de possíveis soluções, melhorando a qualidade do software (Raymond, 1999).

Um SCV tem a finalidade de gerenciar diferentes versões de um documento permitindo uma melhor organização, controle e acompanhamento do projeto que está sendo desenvolvido de forma colaborativa. Dentro deste contexto, o ambiente SCV permite acompanhamento de histórico de desenvolvimento e desenvolvimento paralelo, e estes recursos são relevantes em um ambiente de desenvolvimento colaborativo.

No SCV os arquivos do projeto ficam armazenados em um repositório, um servidor, onde o histórico de versões é salvo. Durante o desenvolvimento do software os desenvolvedores acessam e resgatam a última versão disponível e fazem uma cópia local, nesta cópia eles podem trabalhar e alterar. Cada alteração realizada pelo desenvolvedor é enviada ao servidor e é atualizada a versão do software, assim como dos demais desenvolvedores da equipe. Para evitar problemas de mais de um desenvolvedor editar um mesmo arquivo o SCV trabalha com ferramentas que mesclam o código, isto é, avisa o desenvolvedor caso o arquivo que ele tenha trabalhado tenha sido atualizado. O SCV envia as novas informações adicionadas pelo outro desenvolvedor e irá mesclar as diferentes versões atualizando a versão em uso. Os SCV proporcionam ferramentas que disponibilizam informações como, onde foram realizadas as alterações, trechos de código incluídos e removidos e casos de conflito.

Segundo (Estublier *et al.*, 2005), apesar de terem sido desenvolvidas várias ferramentas de controle de versões, existem alguns termos que são comuns a todas essas

ferramentas, pois são atividades que qualquer SCV proporciona:

- item de configuração: representa cada um dos elementos de informação que são criados, ou que são necessários, durante o desenvolvimento de um produto de software, que devem ser identificados de maneira única e sua evolução deve ser possível de rastreamento;

- repositório: local de armazenamento de todas as versões dos arquivos;

- versão: representa o estado de um item de configuração que está sendo modificado. Toda versão deve possuir um identificador único, ou VID (*Version IDentifier*);

- revisão: resultado da correção de defeitos ou implementação de uma nova funcionalidade. As revisões evoluem sequencialmente, e a maior substitui a anterior. Na Figura 2.6, 2 é uma revisão de 1;

- ramo: versão paralela ou alternativa. Os ramos não substituem as versões anteriores e são usados concorrentemente em configurações alternativas. Na Figura 2.6, 3.1 é uma ramificação de 3; e 3.2 é uma revisão de 3.1;

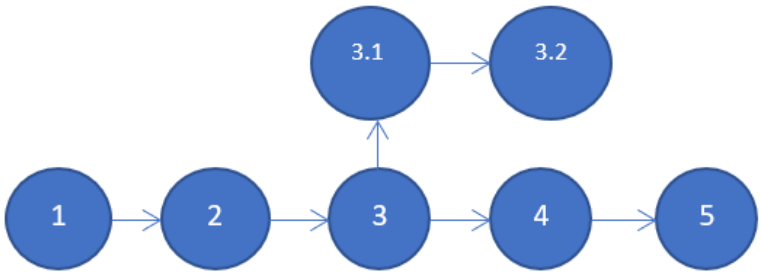


Figura 2.6: Exemplo de revisão e ramificação
Fonte: Adaptado de (Estublier *et al.*, 2002).

- espaço de trabalho: espaço temporário para manter uma cópia local da versão a ser modificada. Isola as alterações feitas por um desenvolvedor de outras alterações paralelas, tornando essa versão privada;

- *check out* (clone): ato de criar uma cópia de trabalho local do repositório;

- *update*: ato de enviar as modificações contidas no repositório para a área de trabalho;

- *commit*: ato de criar um artefato no repositório pela primeira vez ou criar uma

nova versão do artefato quando este passar por uma modificação;

- *merge*: é a mescla entre versões diferentes, objetivando gerar uma única versão que agregue todas as alterações realizadas. Para a realização da mescla são utilizados algoritmos que diferenciam os itens de configuração em questão, o que caracteriza o conceito de Delta, que é a diferença entre o código anterior e o novo código, isto é, o que foi incluído e excluído pelo desenvolvedor no *commit*, gerando a nova versão por meio do *merge* deste novo *commit*;

- *changeset*: coleção atômica de alterações realizadas nos arquivos do repositório.

A Figura 2.7 apresenta exemplo de uma arquitetura de SCV, na qual é possível visualizar três desenvolvedores realizando *commit* no código-fonte que está depositado no repositório central.

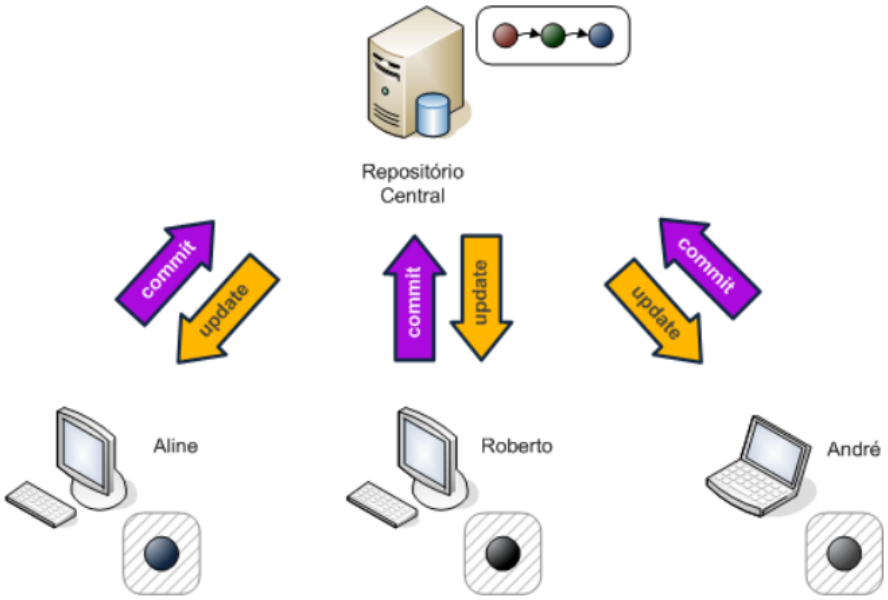


Figura 2.7: Exemplo de SCV centralizado

Fonte: Extraído de (Dias, 2009).

O fato de o desenvolvimento colaborativo de software permitir que desenvolvedores de diversas regiões, com diferentes culturas e perfis tenham possibilidade de realizar alterações no código-fonte de um projeto de software, sucinta a necessidade de se validar a qualidade do produto deste software. A verificação de itens de qualidade do produto de software pode ser realizada a partir de métricas, como as da seção 2.4.

2.4. Qualidade de Software

Os estudos realizados ((Rahman e Devanbu, 2013), (Munson e Elbaum, 2011), (Kitchenham, 2010), (Matsumoto *et al.* 2010), (Honglei *et al.* 2009), (Fenton e Neil, 2000)) mostram que no contexto de engenharia de software, as medidas de qualidade de software medem o quão bem o software foi projetado e o quanto ele atende às conformidades do projeto. A qualidade de software é geralmente descrita em relação a sua adequação a finalidade. Desta forma ela depende de um número de fatores e pode ser avaliada por diferentes pontos de vista, como a visão do usuário, a visão do desenvolvimento, a visão do produto e a visão baseada em valor. A visão do usuário avalia o produto de software de acordo com a necessidade do usuário; a visão do desenvolvimento visa os aspectos de produção do produto de software, isto é, focada na conformidade com o processo o que implica em um produto melhor, por exemplo, utilizando os modelos propostos pela ISO 9126, ISO 25000, MPS.Br (Melhoria do Processo de Software Brasileiro) e o CMMI (*Capability Matury Model Integration*; a visão do produto busca as características internas do produto e suas funcionalidades, a ideia é que o controle dos indicadores internos de qualidade do produto irá influenciar positivamente na qualidade final externa do produto de software, isto é a qualidade de uso; e a visão baseada em valor que é guiada pela existência de diversos departamentos na organização e cada um tem um diferente ponto de vista em relação ao produto (setor de marketing tem uma visão mais de usuário e o setor técnico tem uma visão de produto, por exemplo). Ao final os diferentes pontos de vista acabam se complementando na busca pela qualidade final do produto de software. Os níveis de qualidade de software podem ser medidos. Torna-se importante então, diferenciar o conceito de medidas e métricas.

2.4.1. Medidas e Métricas

Cugini e colegas em (Cugini *et al.*, 1997) consideram que as medidas são indicadores de sistemas, usuários e grupos de desenvolvimento que podem ser observados de forma isolada ou coletivamente enquanto a tarefa está sendo realizada. Estas medidas coletadas podem ser de tempo, distância, ou outro evento que possa ser contabilizado. A medição pode ser realizada de forma direta e coletada de forma automática.

As métricas, segundo (Cugini *et al.*, 1997) são derivadas da interpretação de uma ou mais medidas e podem ser uma combinação da interpretação de métricas e mais

medidas. A eficiência, por exemplo, é uma métrica derivada da interpretação das medidas de tempo, desempenho do usuário e uso da ferramenta.

É possível dizer então que a medida é um valor que pode ser observado, enquanto a métrica associa significado a estes valores por meio do julgamento humano (Cugini *et al.* 1997).

Boas métricas devem permitir o desenvolvimento de modelos que sejam eficientes na predição do espectro de processos ou produtos. Desta forma, uma métrica considerada ótima deve ser: (i) simples, isto é, precisamente definida de forma que esteja claro como a métrica é avaliada; (ii) objetiva, na medida do possível, não permitindo margem de interpretação; (iii) facilmente obtida, por exemplo, a um custo razoável; (iv) válida, devendo medir o que é destinado a ser medido; (v) e robusta, isto é, relativamente insensível a alterações insignificantes do processo ou produto (Cugini *et al.* 1997).

2.4.2. Métricas de Qualidade de Software

As métricas de software proporcionam medição não só do produto de software como de todo o processo de produção de software, sendo de grande importância em todo o seu ciclo de vida. Estas métricas fornecem padrões para o desenvolvimento de software que envolvem os requisitos de software, projetos, programas e testes. O desenvolvimento de software rápido e em grande escala aumentou a complexidade do produto, o que torna a qualidade difícil de controlar. Para que se tenha um controle bem-sucedido desta qualidade tornou-se necessário o uso de métricas de software. Os preceitos de métricas de software são coerentes, compreensíveis e bem estabelecidos, e muitas métricas relacionadas à qualidade do produto foram desenvolvidas e são largamente utilizadas (Rawat *et al.*, 2012).

Baseado na revisão de Honglei *et al.* (2009), existem 3 tipos de métricas de software: métricas de processo, métricas de projeto e métricas de produto. As métricas de processo focam no processo de desenvolvimento de software, principalmente nos aspectos ligados à duração do processo, custo incorrido e no tipo de metodologia utilizada. Estas métricas podem ser usadas para melhorar o desenvolvimento e manutenção de software e abrangem, por exemplo, a eficácia de remoção de defeitos durante o desenvolvimento, o padrão de defeitos durante os testes, e o tempo de resposta do processo de correção.

As métricas de projeto são usadas para monitorar a situação e o status do projeto. Estas métricas previnem quanto a potenciais problemas ou riscos, calibrando o projeto e

ajudando a aperfeiçoar o plano de desenvolvimento de software, uma vez que descrevem as características e execução do projeto. Elas incluem o número de desenvolvedores do software, o perfil dos desenvolvedores ao longo do ciclo de vida do software, custo, cronograma e produtividade (Honglei *et al.*, 2009).

Já as métricas de produto descrevem os atributos do produto de software em qualquer fase do seu desenvolvimento. Estas métricas podem medir o tamanho do programa, a complexidade de design do software, desempenho, portabilidade, facilidade de manutenção, e a escala do produto. Podendo ser utilizadas para prever a qualidade do produto durante o desenvolvimento do software ou do produto final (Honglei *et al.*, 2009).

A manutenibilidade de software é definida como "a facilidade com a qual um sistema de software ou componente pode ser modificado tendo em vista a correção de falhas, melhoria de desempenho e outros atributos, ou de adaptação a um ambiente alterado" (IEEE Std, 1990). A manutenibilidade é considerada um importante atributo de qualidade do produto de software e está associado ao processo de manutenção, que tem sido reconhecido por representar uma grande parte do custo no ciclo de vida de desenvolvimento do software. Por consequência, a manutenção de um sistema de software pode ter impacto significativo nos custos do software, o que significa que é importante ter um eficiente sistema de previsão e validação da manutenibilidade do software e assim gerenciar efetivamente estes custos. As pesquisas relacionadas à previsão de manutenibilidade de software incluem o uso de fatores mensuráveis que tenham efetiva influência na atividade de manutenção de software (Bhatt *et al.*, 2006).

A testabilidade denota a habilidade de um sistema no sentido de ser testado, baseado no padrão IEEE, "o grau em que um sistema ou componente facilita o estabelecimento de critérios de teste e da realização dos testes para determinar se estes critérios foram cumpridos" (IEEE Std, 1990). O teste de software é o processo de executar o programa com a intenção de encontrar erros e atualmente é uma das técnicas mais utilizadas para avaliar a validade da implementação do sistema. Porém, este processo é custoso e, por esta razão, ser capaz de produzir sistemas fáceis de serem testados tem sido uma preocupação importante durante o ciclo de vida de desenvolvimento do software, assim como para organizar os ensaios de teste do software. No entanto, este conceito não é fácil de ser captado e formalizado, uma vez que muitos fatores podem influenciar a capacidade de teste. Diversas métricas têm sido propostas para prever a capacidade de teste especialmente no nível de código-fonte, uma vez que ajudam os testadores a detectar

partes do sistema que possivelmente sejam difíceis de testar, permitindo que se organize e planeje previamente o trabalho de teste (Bruntink e van Deursen, 2004).

Reusabilidade é o grau no qual um componente de software pode ser reutilizado, o que reduz o custo de desenvolvimento do software em função de que permite menor escrita de código e mais modularização. Um bom processo de reuso de software aumenta a produtividade, qualidade e confiabilidade, diminuindo os custos e tempo de desenvolvimento do software. As métricas podem ser utilizadas para determinar fatores de qualidade que afetam a reusabilidade do código, uma vez que um componente tem certas características que tendem a facilitar ou não seu reuso (Washizaki *et al.*, 2004).

A complexidade de um código afeta diretamente o seu entendimento, o que está diretamente ligado à compreensão do programa, que é um processo cognitivo e está relacionado com o esforço mental necessário pelo usuário (neste caso, desenvolvedores, testadores e pessoal da manutenção) que lida com o código e precisa entendê-lo para realizar suas atividades. Este esforço está diretamente ligado a relativa dificuldade, tempo e esforço necessário para compreender o software (Misra e Akman, 2008).

Os itens de qualidade do produto de software citados podem ser medidos utilizando-se as métricas de código-fonte.

2.4.3. Métricas de Código-fonte

A qualidade do produto de software pode ser medida por meio de medidas e métricas do código-fonte. Estas medidas procuram gerenciar e reduzir a complexidade das estruturas, melhorar a manutenibilidade e o desenvolvimento dos códigos-fonte e, consequentemente, o próprio software (Yu e Zhou, 2010).

As métricas de código-fonte mais conhecidas são: métrica da Complexidade Ciclomática de McCabe (CCM), (Mccabe, 1976); métricas para software orientado a objetos de Chidamber e Kemerer (CK) (Chidamber e Kemerer, 1991); métricas de tamanho, tais como: Número de Linhas de Código (LOC, do inglês *Lines of Code*) (Yu e Zhou, 2010), Número de Classes (NOC, do inglês *Number of Classes*) (Oliveira *et al.*, 2008), Número de Métodos de Classes (NOM) (Henderson-Sellers, 1996) e Número de Atributos por Classe (NAC, do inglês *Number of Attributes per Class*) (Henderson-Sellers, 1996), Número de Atributos Estáticos (NSF, do inglês *Number of Static Attributes*) (Oliveira *et al.*, 2008), Número de Interfaces (NOI, do inglês *Number of Interfaces*) (Oliveira *et al.*, 2008) e Número de Pacotes (NOP, do inglês Number of Packages) (Horstman, 2004); métricas de Acoplamento de Robert C. Martin (Martin,

1994); outras métricas de herança, como: Métodos Sobrescritos (NORM, do inglês *Number of Overridden Methods*) (Schroeder, 1999) e Índice de Especialização (SIX, do inglês *Specialization Index*) (Henderson-Sellers, 1996); e outras métricas de métodos, como: Profundidade de Blocos Aninhados (NBD, do inglês *Nested Block Depth*) (Oliveira *et al.*, 2008), Número de Parâmetros (PAR, do inglês *Number of Parameters*) (Oliveira *et al*, 2008), Número de Métodos Estáticos (NSM, do inglês *Number of Static Methods*) (Oliveira *et al.*, 2008). A seguir, estas métricas serão descritas.

2.4.3.1. Complexidade Ciclomática de McCabe

Thomas J. McCabe em 1976 propôs a medida de Complexidade Ciclomática de McCabe (CCM ou VG). CCM é a medida da quantidade de caminhos lógicos, linearmente independentes, de um código-fonte (McCabe, 1976). O resultado desta medida pode ser utilizado para indicar a quantidade máxima de testes que devem ser executados para garantir que todos os comandos foram executados ao menos uma vez.

Esta medida está relacionada com o grau de dificuldade de entendimento, com a testabilidade e a manutenibilidade de um código-fonte. Um método com CCM alta significa que terá mais caminhos lógicos a serem testados e terá também uma manutenção mais difícil (Silva *et al.*, 2012).

Na CCM a estrutura de programa é representada por um grafo *V(G)*, com *n* vértices e com *e* arestas. A fórmula para calcular a CCM é dada na Equação 2.1:

$$V(G) = e - n + 2 \qquad (2.1)$$

A Figura 2.8 apresenta um exemplo de aplicação da métrica CCM em um pequeno trecho de programa escrito em linguagem de programação Java. Nela pode-se visualizar também o grafo que representa a estrutura deste programa.

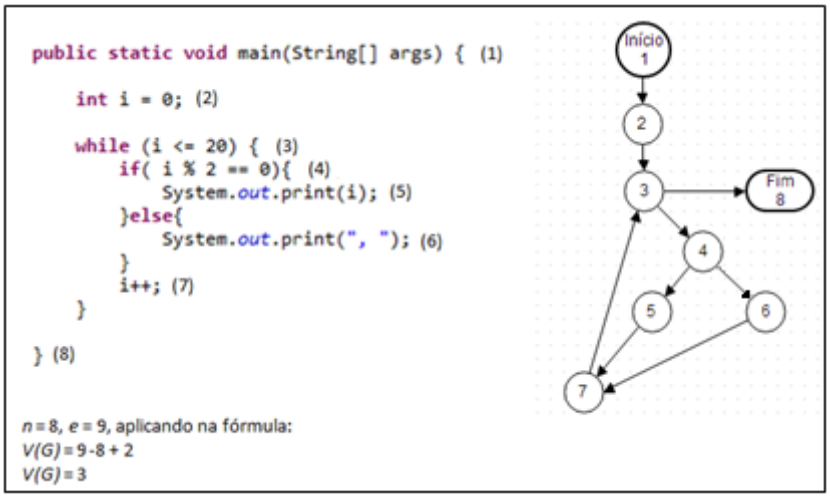


Figura 2.8: Exemplo de Aplicação da Métrica CCM
Fonte: Adaptado de (Beal, 2013).

Analisando a Figura 2.8 pode-se ver uma estrutura de código-fonte representada num grafo com 8 nós e 9 arestas. No código-fonte, cada linha contém o número do nó correspondente do grafo, entre parênteses. Aplicando a fórmula, obtém-se o resultado igual a 3.

A medida de complexidade ciclomática pode ser utilizada como um indicador tanto no desenvolvimento quanto na manutenção, que pode ser utilizado no monitoramento e acompanhamento dos riscos conforme mudanças e alterações vão sendo implementadas no código-fonte (Anderson, 2004).

Em Anderson (2004) pode ser vista uma escala com faixas de valores para a CCM (ou VG), o nível da complexidade e os riscos associados. A Tabela 2.1 apresenta esta escala.

Tabela 2.1: Escala da Complexidade Ciclomática

Complexidade Ciclomática (CCM)	Complexidade do Método	Avaliação do Risco
01-10	Baixa	Baixo risco
11-20	Moderada	Risco moderado
21-50	Elevada	Risco elevado
Maior que 50	Alta	Impossível de testar, risco muito alto

Fonte: Adaptado de (Anderson, 2004).

As estruturas (ou métodos) que apresentam um valor de CCM entre 01 e 10, são consideradas estruturas com complexidade baixa e com baixo risco associado. As estruturas com CCM entre 11 e 20 são consideradas estruturas com complexidade

moderada com riscos moderados associados. Estruturas com CCM entre 21 e 50 são estruturas com complexidade elevada e os riscos são elevados. Por fim, as estruturas com CCM maiores que 50 possuem complexidade alta e os riscos são muito altos, ou seja, impossível testar todos os caminhos linearmente independentes.

2.4.3.2. Métricas de Chidamber e Kemerer (CK)

É um conjunto com seis métricas para softwares orientados a objetos proposta em 1991 por Shyam R. Chidamber e Chris F. Kemerer (1991). São elas: Métodos Ponderados por Classe, Profundidade da Árvore de Herança, Número de Filhos, Acoplamento entre Classes, Falta de Coesão em Métodos e Resposta para uma Classe.

A métrica Métodos Ponderados por Classe (WMC), do inglês *Weighted Methods per Class* mede a complexidade de uma classe levando em consideração o somatório das complexidades de cada método de uma classe. A complexidade dos métodos pode ser calculada utilizando CCM, ou pode ser assumido um valor único, no caso 1, para cada método. Neste último caso, o WMC de uma classe seria calculado pelo número de métodos de uma classe. A Equação 2.2 apresenta a fórmula para calcular WMC. Onde, *n* = número de métodos e *Ci* = complexidade dos métodos.

$$WMC = \sum_{i=1}^{n} Ci \qquad (2.2)$$

Uma classe com um grande número de métodos é mais complexa para ser mantida. Um número grande de métodos em uma classe indica que ela é utilizada em poucas situações e que dificilmente será reutilizada em outros projetos. Em alguns casos, este tipo de classe se faz necessário, porém, o ideal é projetar classes mais reutilizáveis. O número de métodos também influencia na herança de classes, pois, é o número de métodos que os filhos (subclasses) irão herdar.

A Tabela 2.2 apresenta uma escala de valores para a WMC e riscos associados. Esta tabela é uma adaptação de Olague *et al.* (2006) que utilizou estes valores em seu trabalho sobre manutenibilidade e degradação de software orientado a objetos.

Tabela 2.2: WMC e Riscos Associados

CK WMC Limiar (x)	Complexidade da Classe	Risco Associado
$1 \leq x \leq 20$	Boa	Baixo risco
$20 \leq x \leq 100$	Moderada	Risco moderadamente elevado
$x > 100$	Alta	Risco alto, motivo para investigação.

Fonte: Adaptado de (Olague *et al.*,2006).

A métrica Profundidade da Árvore de Herança (DIT), do inglês *Depth of Inheritance Tree*, mede a quantidade de classes ancestrais que antecedem uma classe numa árvore de herança. Esta medida pode ser utilizada para indicar o nível de abstração utilizado na modelagem de um sistema. Uma classe com DIT alto significa que ela herda muitos métodos de superclasses o que a torna mais complexa para ser mantida. Esse resultado pode ser interpretado como um problema de abstração ou, como um indicador de reuso (Henderson-Sellers, 1996). A Equação 2.3 apresenta a fórmula para calcular DIT. Onde, *n* = quantidade de classes ancestrais.

$$DIT = n \qquad (2.3)$$

O Número de Filhos (NSC), do inglês *Number Of Children*, é a medida do número de subclasses imediatas de uma classe em uma árvore de herança. Uma classe com muitos filhos (subclasses) é um indicador de reuso, mas, pode representar também problemas de abstração (Rosenberg *et al.*, 1999). A Equação 2.4 apresenta a fórmula para calcular NSC. Onde, *n* = número de subclasses imediatas.

$$NSC = n \qquad (2.4)$$

O Acoplamento entre Classes (CBO), do inglês *Coupling Between Object*, mede o grau de dependência entre classes de software orientado a objetos. A medida de acoplamento diz quanto uma classe depende de outras classes. Quando uma classe depende de muitas outras classes, tem-se um forte acoplamento. Quando uma classe depende de poucas classes é dito fraco acoplamento. As duas medidas, fraco e forte acoplamento, indicam o grau de manutenibilidade e reusabilidade de uma classe. Forte acoplamento indica que se a classe sofrer alterações, todas as dependentes também deverão ser modificadas dificultando a manutenibilidade e a reutilização desta classe em outros projetos. O fraco acoplamento é o ideal, diminui a complexidade do software tornando a manutenção mais simples e possibilitando o reuso de classes (Li, 2008). A Equação 2.5 apresenta a fórmula da medida CBO. Onde, *g* indica quanto a classe depende de outras classes.

$$CBO = g \qquad (2.5)$$

A métrica Falta de Coesão em Métodos **(**LCOM), do inglês *Lack of Cohesion in Methods*, mede a coesão entre os métodos de uma classe. Para Chidamber e Kemerer (1991), LCOM é dado pelo número de pares de métodos com similaridade igual à zero (*P*), menos o número de pares de métodos cuja similaridade é diferente de zero (*Q*) (Equação 2.6). Onde, *P* é o número de pares de métodos que não compartilham atributos; *Q* é o número de pares de métodos que compartilham atributos.

$$LCOM = P - Q, \quad quando\ P > Q, \quad senão\ LCOM = \qquad (2.6)$$

Para um melhor entendimento da métrica LCOM, nos próximos parágrafos será apresentado um exemplo adaptado de Chidamber e Kemerer (1991).

Considere uma classe C (Figura 2.9) com os métodos M1, M2 e M3 e com os atributos A1, A2, A3, A4 e A5. O método M2 utiliza os atributos A1 e A2 e, o método M3 utiliza os atributos A4 e A5. O relacionamento entre um método e os atributos utilizados por ele é representado por um conjunto Ii. Neste caso, os conjuntos são: I1 = {A1, A2, A3}, I2 = {A1, A2} e I3 = {A4, A5}. Onde, I1 é o conjunto dos atributos utilizados pelo M1, I2 o conjunto de atributos utilizados M2 e I3 o conjunto dos atributos usados por M3.

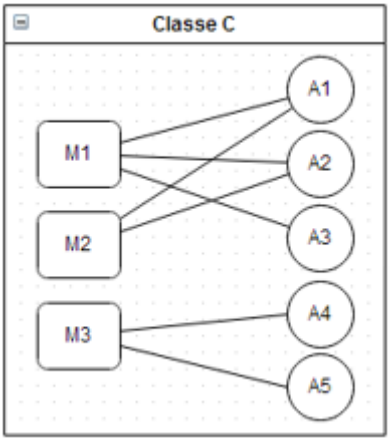


Figura 2.9: Classe C

Fonte: Adaptado de (Beal, 2013).

Para encontrar os pares similares (*P*) e os pares não similares (*Q*) da classe C da Figura 2.9, faz-se a intersecção dos pares dos métodos conforme o Quadro 2.1.

Quadro 2.1: Intersecção de pares de métodos

$M1 \cap M2 = \{A1, A2\} \neq \emptyset$
$M1 \cap M3 = \emptyset$
$M2 \cap M3 = \emptyset$

Após obter as intersecções, pode-se calcular *P* e *Q*. Para calcular *P*, conta-se o número de intersecções de pares de métodos que resultaram um conjunto vazio (Ø), neste caso o resultado é igual a 2. Para calcular *Q*, conta-se o número de intersecções diferentes de vazio, neste caso, o resultado é igual a 1.

Com *P* e *Q* calculados, pode-se aplicar a Equação 2.6 para encontrar a LCOM. Aplicando *P* e *Q* deste exemplo, obtém LCOM = 1.

De acordo com Horstmann (2004), uma classe coesa representa apenas um conceito. Caso isso não ocorra, a classe deve ser dividida em classes menores. Um valor de LCOM alto indica baixa coesão e que a classe deve ser dividida em duas ou mais. Classes que executam mais de uma função são mais complexas e difíceis de serem compreendidas e testadas.

A métrica LCOM passou por revisões para corrigir algumas falhas, uma das revisões é a de Henderson-Sellers (1996) chamada de LCOM*. A nova revisão foi criada porque, segundo o autor, existiam muitos casos de classes diferentes onde o valor da LCOM era o mesmo, mas, não representavam a mesma coesão. Um exemplo pode ser visto na Figura 2.10, onde duas classes (a) e (b) apresentam o mesmo valor de LCOM, porém, a classe (a) tem baixa coesão e a classe (b) tem alta coesão.

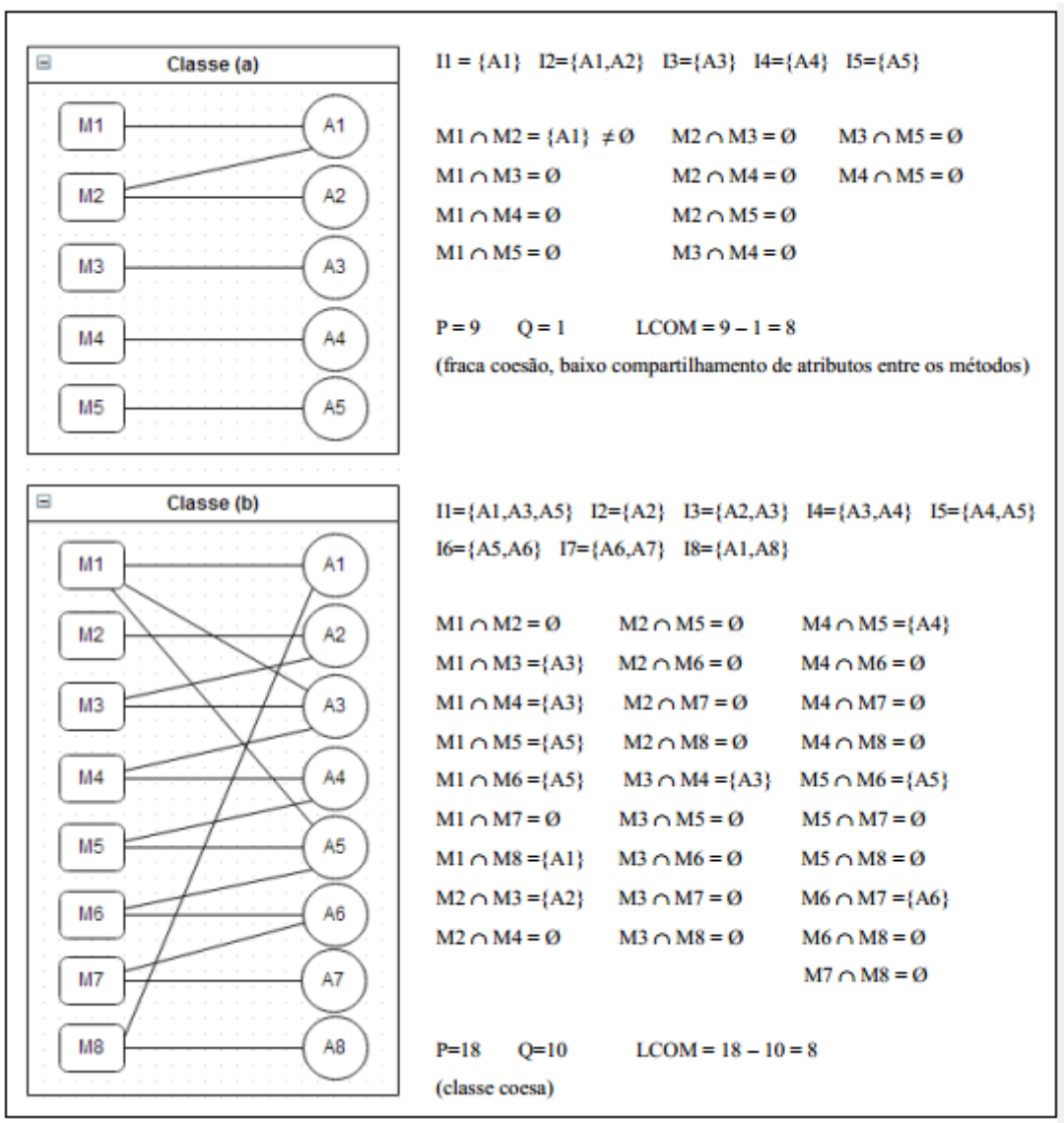


Figura 2.10: Exemplo de classes com o mesmo valor de LCOM
Fonte: Adaptado de (Henderson-Sellers, 1996).

Outro problema levantado por Henderson-Sellers (1996) é que nem sempre um valor de LCOM igual à zero é um indicador de boa coesão. Desta forma o autor propôs a LCOM*, onde uma classe é considerada coesa quando todos os seus métodos acessam todos os atributos, resultando em LCOM igual à zero. Valores baixos, próximos de zero, são indicativos de coesão. Valores próximos de 1 indicam falta de coesão e, valores maiores que 1 indicam problemas sérios de coesão, concluindo que certamente a classe deverá ser subdivida em outras classes.

A LCOM* é calculada da seguinte forma: dada uma classe C, com um conjunto de métodos {Mi} (M1, M2, ... Mn) que acessam um conjunto de atributos {Ai} (A1, A2,

... An), calcula-se o número de métodos que acessam cada atributo, representado por m (Ai), na sequência faz-se a média de m (Ai) para todos os atributos e subtrai-se o número de métodos (m), o resultado é dividido por (1-m). A fórmula para calcular a LCOM* é dada pela Equação 2.6a de Henderson-Sellers (1996).

$$LCOM* = \frac{(\frac{1}{a}\sum_{i=1}^{a} m(Ai)) - m}{1-m} \qquad (2.6a)$$

Resposta para uma Classe (RFC), do inglês, *Response for a Class*, é a soma dos métodos que podem ser invocados em resposta a uma mensagem recebida por um objeto de uma classe. Neste caso, são considerados os métodos da classe e todos os outros métodos que são chamados pelos métodos da própria classe. Quanto maior for o número de métodos que podem ser chamados em resposta a uma mensagem, maior é a complexidade da classe. Quanto maior for o número de métodos que podem ser chamados de fora da classe, maior é a dificuldade para compreendê-la (Chidamber e Kemerer, 1991). A Equação 2.7 exibe a fórmula para calcular RFC. Onde, *m* = número de métodos da classe e *r* = número de métodos chamados pelos métodos da classe.

$$RFC = m + r \qquad (2.7)$$

2.4.3.3. Métricas de Tamanho

A métrica Número de Linhas de Códigos calcula o número de linhas de código. Esta medida contabiliza linhas referentes a cabeçalhos, declarações e comandos em um código-fonte, excluindo-se linhas de comentários e linhas em branco (Henderson-Sellers 1996). Ela é também conhecida como LOC (*Lines of Code*) ou SLOC (*Source Lines of Code*) (Henderson-Sellers, 1996).

É uma medida de tamanho que pode ser utilizada para avaliar a complexidade. Obter a medida LOC é importante porque o aumento do número de linhas de código aumenta a complexidade do software.

A LOC pode ser calculada em nível de métodos (MLOC, do inglês *Method Lines of Code*), ou em nível de unidade de código (TLOC, do inglês *Total Lines of Code*).

A fórmula para calcular LOC é apresentada na Equação 2.8. Onde, *n* = número de linhas de código.

$$LOC = n \qquad (2.8)$$

A Figura 2.11 apresenta o cálculo de LOC para a classe *Multiplicacao.java*.

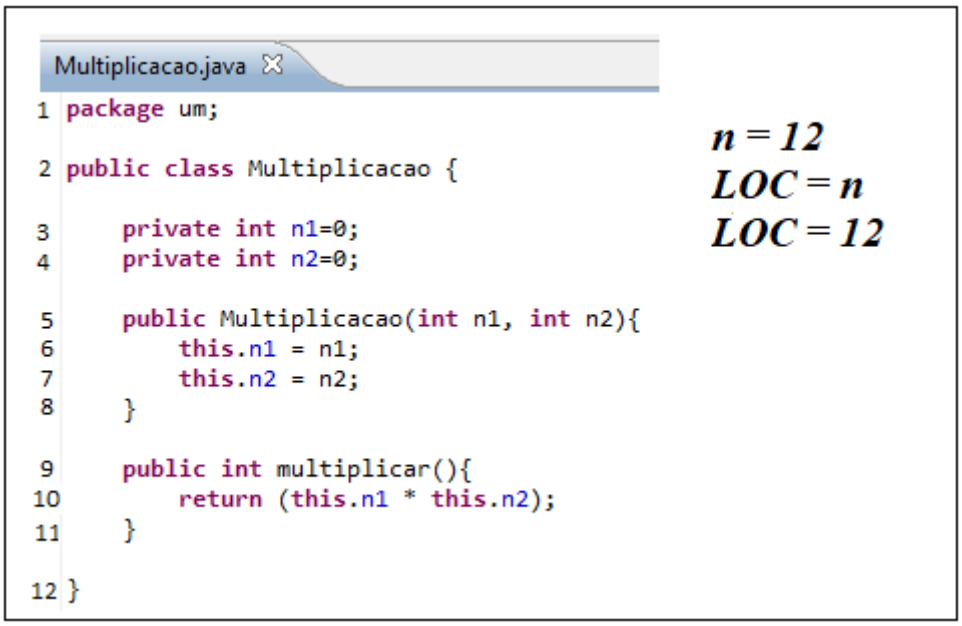


Figura 2.11: Cálculo de LOC para a classe Multiplicacao.java

Fonte: Adaptado de (Beal, 2013).

A métrica Número de Classes (NOC), do inglês *Number of Classes*, é a contagem das classes de um projeto (Equação 2.9). Onde, *c* = número de classes.

$$NOC = c \qquad (2.9)$$

A Figura 2.12 apresenta um projeto chamado *OperacoesAritmeticas*, com 4 pacotes e 5 classes.

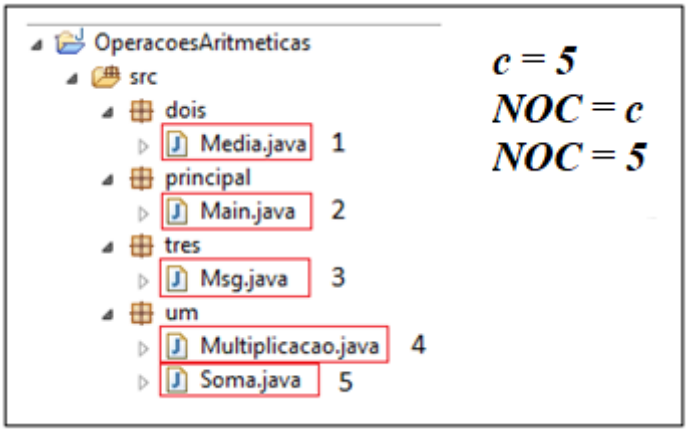


Figura 2.12: Cálculo de NOC para o projeto OperacoesAritmeticas
Fonte: Adaptado de (Beal, 2013).

O Número de Métodos de Classes (NOM), do inglês *Number of Methods*, refere-se a contagem simples do número de métodos (inclusive métodos de instâncias privadas e métodos herdados) de uma classe (Equação 2.10). Classes com um número elevado de métodos tendem a serem mais específicas, consequentemente menos reutilizáveis. Onde, *n* = número de métodos da classe.

$$NOM = n \qquad (2.10)$$

A Figura 2.13 apresenta o cálculo do NOM para a classe *Msg.java*

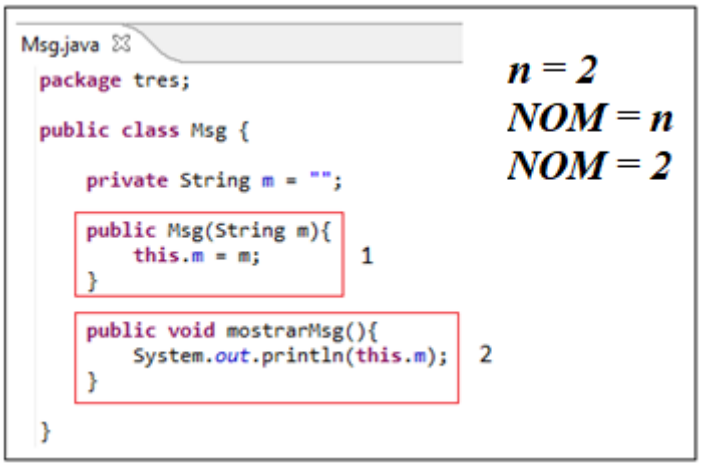


Figura 2.13: Cálculo do NOM para a classe Msg.java
Fonte: Adaptado de (Beal, 2013).

A métrica Número de Atributos por Classe (NAC), do inglês *Number of Attributes per Class*, refere-se a contagem de atributos (variáveis de instância) de uma classe (Equação 2.11). Um número grande de atributos pode dificultar a compreensão e a modificação de uma classe. Onde, *n* = número de atributos da classe.

$$NAC = n \qquad (2.11)$$

A Figura 2.14 apresenta a classe *Multiplicacao.jav*a com valor de NAC = 2.

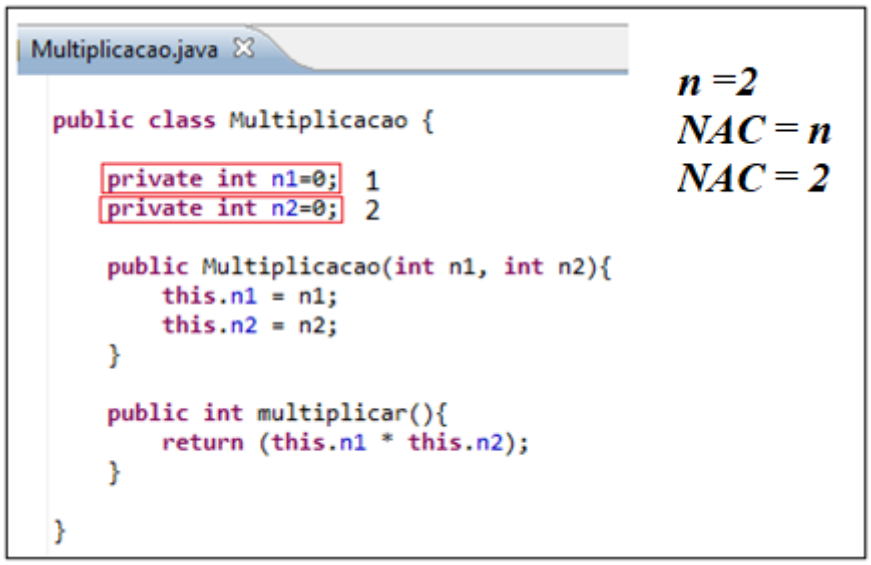


Figura 2.14: Classe Multiplicacao.java com NAC = 2

Fonte: Adaptado de (Beal, 2013).

Esta métrica também é referenciada como Número de Campos (NOF – *number of fields*), nesta pesquisa iremos referenciar como NOF.

A métrica Número de Atributos Estáticos (NSF), do inglês *Number of Static Fields*, calcula a quantidade de atributos estáticos de uma classe. Os campos estáticos podem dificultar o entendimento e a manutenção de uma classe quando se está trabalhando em um ambiente de programação Orientada a Objetos (OO) (Horstman, 2004). A Equação 2.12 representa a métrica NSF. Onde, *ns* é o número de atributos estáticos da classe.

$$NSF = ns \qquad (2.12)$$

A métrica Número de Métodos Estáticos (NSM), do inglês *Number of Static Methods*, calcula a quantidade de métodos estáticos de uma classe (Oliveira *et al.*, 2008). A Equação 2.13 representa a métrica NSM. Onde, *sm* é o número de métodos estáticos.

$$NSM = sm \qquad (2.13)$$

Os métodos e atributos estáticos podem ser chamados diretamente sem necessidade de criar um objeto da classe. A Figura 2.15 apresenta a classe *Circunferência.java* que chama o atributo estático *PI* da classe *Math* e o método estático *sqrt()*, também da classe *Math*, para fazer a raiz quadrada da variável de instância *raio*.

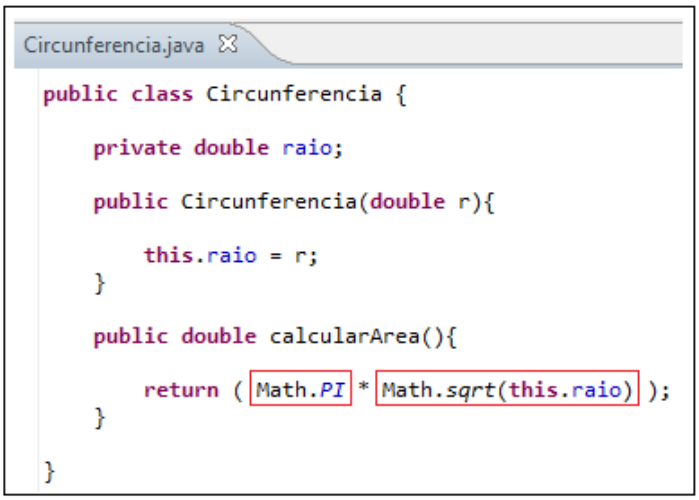


Figura 2.15: Atributo estático PI e método estático sqrt() da classe Math.java
Fonte: Adaptado de (Beal, 2013).

Apesar de eles melhorarem o desempenho de um sistema, eles devem ser usados para fins específicos. Métodos estáticos em excesso podem indicar programação procedural e não orientado a objeto. Campos estáticos dificultam o entendimento e a manutenção do software (Horstman, 2004).

A métrica Número de Parâmetros (PAR), do inglês *Number of Parameters*, calcula a quantidade de parâmetros passados em um método (Oliveira *et al.* 2008). Uma classe com um número muito grande de parâmetros pode indicar que a classe executa mais de uma função e precisa ser dividida em subclasses. A PAR está representada na Equação 2.14. Onde, *np* é o número de parâmetros.

$$PAR = np \quad (2.14)$$

A Figura 2.16 apresenta a classe *Multiplicacao.java* com valor de PAR = 2.

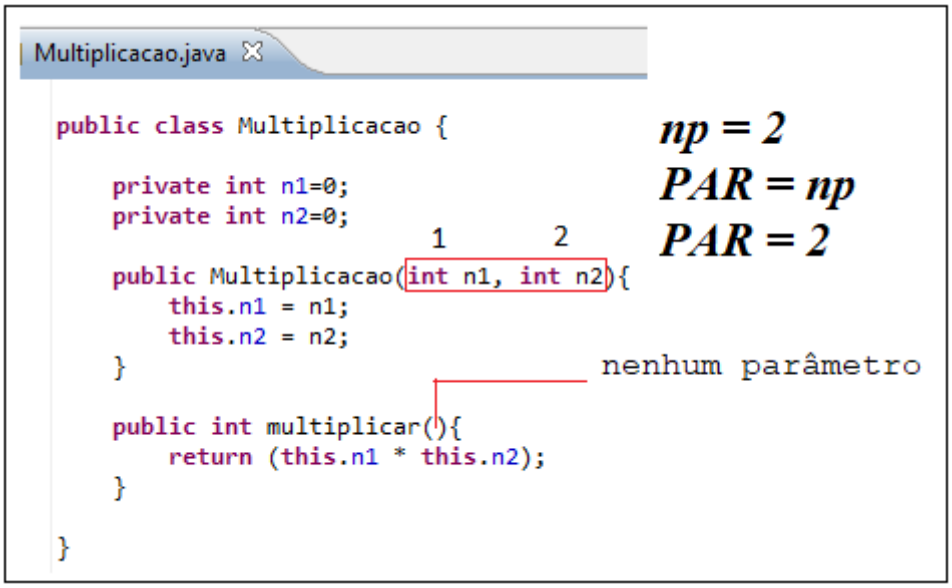


Figura 2.16: Classe Multiplicacao.java com PAR = 2
Fonte: Adaptado de (Beal, 2013).

O Número de Interfaces (NOI), do inglês *Number of Interfaces*, refere-se à quantidade de interfaces. As interfaces promovem o reuso, pois declaram um conjunto de métodos e suas assinaturas, sem implementações, deixando a cargo das classes que as implementam definir as implementações dos métodos. Elas também reduzem o acoplamento entre as classes (Horstman, 2004). A métrica NOI está representada na Equação 2.15. Onde, *i* = número de interfaces.

$$NOI = i \quad (2.15)$$

A métrica Número de Pacotes (NOP), do inglês *Number of Packages*, é a contagem de pacotes definidos em um projeto. Os pacotes estruturam e organizam classes relacionadas, separando classes específicas de um programa de classes utilitárias que podem ser compartilhadas com outros programas. Os pacotes promovem o reuso e facilitam a manutenção (Horstman, 2004). A Equação 2.16 representa a fórmula para calcular uma NOP. Onde, *p* = número de pacotes.

$$NOP = p \tag{2.16}$$

A Figura 2.17 apresenta o projeto *OperacoesAritmetica* com NOP = 4 (pacotes um, dois, três e principal).

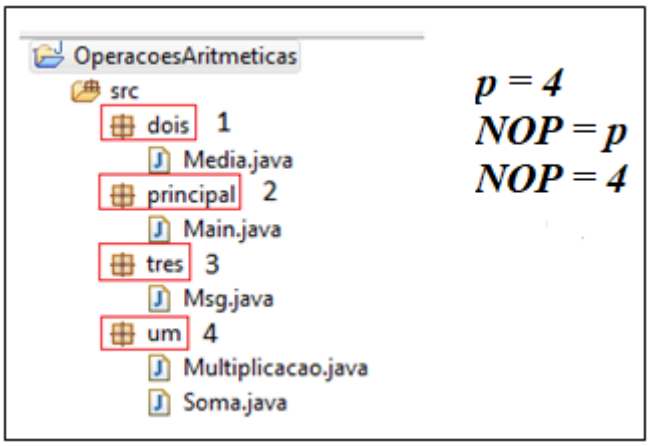


Figura 2.17: Projeto OperacoesAritmeticas com NOP = 4
Fonte:Adaptado de (Beal, 2013).

2.4.3.4. Métricas de Acoplamento de Martin

Martin (1994) definiu um conjunto de métricas para acoplamento em nível de pacotes. Estas métricas medem a dependência dos pacotes e fornecem resultados que podem ser utilizados como indicativos de reusabilidade e manutenibilidade.

Um software que apresenta seus pacotes fortemente acoplados (ou dependentes) é mais difícil de ser mantido, modificado e reutilizado. Quando uma classe de um pacote precisa ser alterada, todas as classes de outros pacotes que são dependentes a ela precisam ser alteradas, desencadeando uma modificação em cascata, o que dificulta muito a modificação e a manutenção. Além disso, a dependência das classes de diferentes pacotes dificulta o reuso. O custo para separar as classes de interesse para reuso das classes dependentes pode ser muito alto, inviabilizando a reutilização (Martin, 1994).

As métricas propostas por Martin (1994) são: Acoplamento Aferente, Acoplamento Eferente, Instabilidade, Abstração e Distância Normalizada. Na sequência elas serão descritas.

O Acoplamento Aferente (*Ca*), do inglês *Afferent Coupling*, é o número de classes de outros pacotes que dependem de classes de um pacote considerado (Equação 2.17).

Onde, *n* é o número de classes de outros pacotes que dependem de classes de um pacote considerado.

$$Ca = n \qquad (2.17)$$

A Figura 2.18 apresenta o pacote A com as classes C1 e C2, o pacote B com a classe C3 e o pacote C com a classe C4. As classes C3 e C4 são dependentes das classes C1 e C2 do pacote A.

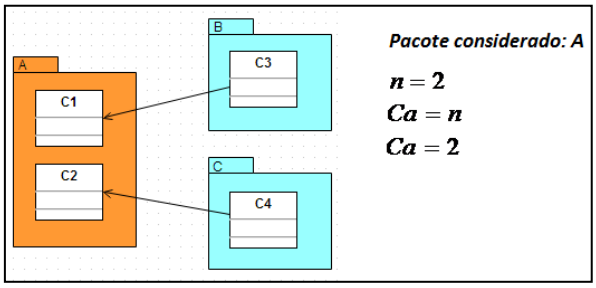


Figura 2.18: Exemplo de Acoplamento Aferente

Fonte: Adaptado de (Beal, 2013).

O Acoplamento Eferente (*Ce*), do inglês *Efferent Coupling*, é o número de classes de um pacote que dependem de classes de outros pacotes (Equação 2.18). Onde, *n* é o número de classes de um pacote que dependem de classes de outros pacotes.

$$Ce = n \qquad (2.18)$$

A Figura 2.19 apresenta o pacote D com as classes C5 e C6, o pacote E com a classe C7 e o pacote F com a classe C8. As classes C5 e C6 do pacote A dependem das classes C7 e C8 dos pacotes E e F, respectivamente.

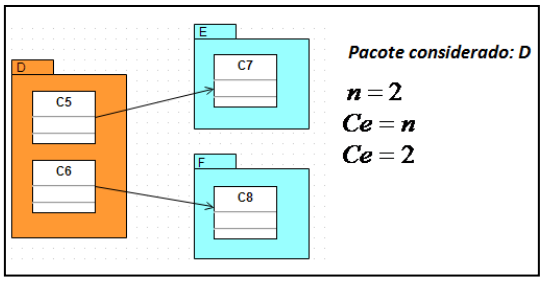


Figura 2.19: Exemplo de Acoplamento Eferente

Fonte: Adaptado de (Beal, 2013).

A *Ca* e a *Ce* quando calculadas a nível de classe, são conhecidas como *Fan-in* e *Fan-out* (ou CBO), respectivamente (Sato *et al.* 2007).

A métrica Instabilidade (*I ou RMI*), do inglês *Instability*, tem como resultado valores entre o intervalo [0,1]. Valores próximos de zero, indicam estabilidade e valores próximos de um, instabilidade. O valor 0 é a máxima estabilidade da classe e o valor 1, a máxima instabilidade. A Equação 2.19 apresenta como calcular *I*.

$$I = \frac{Ce}{(Ca+Ce)} \qquad (2.19)$$

Um pacote é instável quando ele depende de muitos outros pacotes. Quanto menor a dependência, mais estável ele é, indicando condições de reuso. Na Figura 2.20 tem-se um exemplo de pacote instável, o pacote A. Ele tem dependência com os pacotes B e C. O seu valor de *I* é igual a 1.

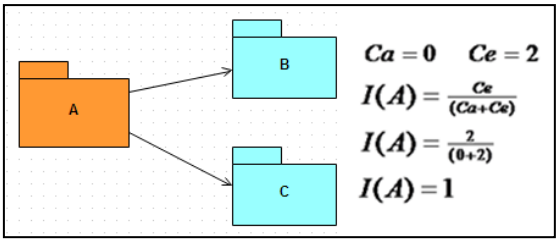


Figura 2.20: Exemplo de pacote instável

Fonte: Adaptado de (Beal, 2013).

Na Figura 2.21, tem-se um exemplo de pacote estável, o pacote D. Ele não possui dependência e seu valor de *I* = 0. Os pacotes E e F são pacotes que dependem do pacote D.

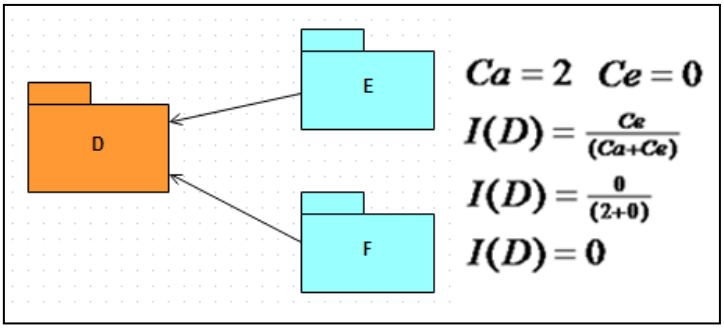


Figura 2.21: Exemplo de pacote estável

Fonte: Adaptado de (Beal, 2013)

A métrica Abstração (*A ou RMA*), do inglês *Abstractness*, mede o número de classes abstratas e interfaces, dividido pelo número total de tipos em um pacote. Esta medida resulta em um valor entre o intervalo [0,1], onde um valor próximo de zero indica um pacote concreto, ou seja, um pacote onde a maioria das classes é concreta. Valor de *A* próximo de 1, indica pacote abstrato, formado na sua maior parte de classes abstratas. *A* = 0, indica pacote totalmente concreto e *A* = 1 indica pacote totalmente abstrato. A Equação 2.20 apresenta a fórmula para calcular *A*. Onde, *Na* é o número de classes abstratas e *Nc* é o número de classes concretas do pacote.

$$A = \frac{Na}{Nc} \qquad (2.20)$$

Um pacote concreto inviabiliza a sua extensão porque suas classes não são abstratas. Ao mesmo tempo em que um pacote concreto e instável dificulta a manutenção. Pacotes abstratos são ideais porque suas classes abstratas podem ser estendidas sem necessidade de modificação (Martin, 1994).

Outra métrica proposta por Martin (1994) é a Distância Normalizada a partir de Sequência Principal (*Dn*), do inglês *Normalized Distance*. Esta métrica está relacionada com a Abstração e a Instabilidade. Martin (1994) utiliza um gráfico para demonstrar esta relação (Figura 2.22). O eixo X representa a Instabilidade e o eixo Y a Abstração, ambos com valores variando entre o intervalo [0,1]; com uma reta perpendicular traçada pelas coordenadas (0,1) e (1,0) que é a chamada Sequência Principal. As coordenadas (0,1) e (1,0) representam os tipos de pacotes ideais: *I*=0 e *A*=1 (pacote estável e abstrato) e *I* = 1 e *A* = 0 (instável e concreto).

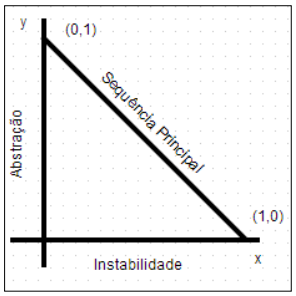


Figura 2.22: Gráfico Abstração X Instabilidade

Fonte: Adaptado de (Martin, 1994).

A Sequência Principal representa os valores de equilíbrio entre a Instabilidade e a Abstração. Os pacotes que se encontram na Sequência Principal apresentam um número equilibrado de classes abstratas e concretas em relação às dependências Aferentes e Eferentes. A localização ideal de um pacote na Sequência Principal é uma das extremidades da Sequência (Martin, 1994).

A Distância Normalizada a partir de Sequência Principal *(Dn)* é calculada pela Equação 2.21. Um valor de *Dn* igual a zero indica que o pacote está sobre a Sequência Principal. Um valor de *Dn* igual a um indica que o pacote está distante da Sequência Principal. Onde, *A* é o valor de Abstração e *I* o valor de Instabilidade.

$$Dn = |A + I - 1| \qquad (2.21)$$

2.4.3.5. Outras Métricas de Herança

A métrica Métodos Sobrescritos (NORM), do inglês *Number of Overridden Methods*, mede o número de métodos sobrescritos por uma subclasse. Um valor alto de NORM indica que a subclasse sobrescreveu o comportamento da superclasse, o que pode indicar que elas não possuem comportamentos comuns. O ideal nestes casos, é que a subclasse crie novos métodos em vez de sobrescrever os métodos da superclasse (Schroeder, 1999). A Equação 2.22 apresenta o cálculo de NORM. Onde, *mo* é o número métodos sobrescritos.

$$NORM = mo \qquad (2.22)$$

A métrica Índice de Especialização (SIX), do inglês *Specialization Index*, calcula o índice de especialização de cada subclasse. Primeiramente é calculado o NORM da subclasse que é multiplicado pelo seu DIT, na sequência o resultado da multiplicação é dividido pelo número de métodos da subclasse (NOM) (Henderson-Sellers 1996). A fórmula para calcular SIX é apresentada na Equação 2.23.

$$SIX = \frac{NORM * DIT}{NOM} \qquad (2.23)$$

Um número alto de SIX pode indicar problemas de abstração, subclasses de uma hierarquia estão reescrevendo muitos métodos das superclasses.

2.4.3.6. Outra Métrica de Complexidade

Outra métrica que pode ser aplicada para avaliar a complexidade do código-fonte é a métrica Profundidade de Blocos Aninhados (NBD), do inglês *Nested Block Depth*. Esta métrica calcula a profundidade de blocos de instruções aninhados. Esta medida é utilizada para avaliar a complexidade do código. Um valor alto de NBD indica código complexo (Oliveira *et al.* 2008). A NBD está representa na Equação 2.24. Onde, *bd* é a profundidade de blocos aninhados.

$$NBD = bd \qquad (2.24)$$

Com o objetivo de buscar o grau de contribuição de cada participante do projeto colaborativo de software focando no código-fonte do sistema esta pesquisa leva em consideração as métricas de código-fonte que têm o potencial de identificar variações na manutenibilidade, testabilidade, reusabilidade e complexidade do produto de software. A partir da coleta e análise dos *commits* realizados pelo desenvolvedor é possível conhecer os dados relativos às métricas de código-fonte relevantes de cada *commit.* Estas métricas agrupadas retornam a informação necessária para calcular o grau de contribuição individual do desenvolvedor, objeto deste trabalho. Os estudos que levaram ao agrupamento das métricas de código-fonte serão relatados a seguir.

2.5. Agrupamento das Métricas de Qualidade do Produto de Software

Esta fase do trabalho apresenta a revisão sistemática da literatura realizada com o objetivo de compreender melhor a influência das métricas de código-fonte nos itens de qualidade do produto de software com relação a complexidade, reusabilidade, testabilidade e manutenibilidade. A revisão também teve como objetivo identificar lacunas de pesquisa e contribuiu significativamente para a motivação desta tese. A revisão iniciou com uma base de 2.134 artigos e foi concluída com a análise detalhada de 130 artigos.

O Capítulo 4 (Procedimentos Metodológicos) detalha todas as etapas de pesquisa realizadas ao longo deste trabalho, e é importante ressaltar que esta revisão sistemática foi a etapa que possibilitou entender melhor o objeto de estudo e a construção da equação do cálculo do grau de contribuição do desenvolvedor, que é um dos objetivos desta tese.

2.5.1 Condução da Revisão Sistemática de Literatura

O método de pesquisa utilizado foi a revisão sistemas de literatura com objetivo de identificar, avaliar e interpretar as pesquisas disponíveis e relevantes para uma questão em particular (Kitchenham e Charters, 2007). Para a condução desta revisão foi realizado um planejamento da revisão, identificação da pesquisa, seleção dos estudos e classificação dos resultados obtidos.

Na fase de planejamento da revisão foi especificado o protocolo com os processos e os métodos para a aplicação da revisão sistemática da literatura. Neste momento foi definido o objetivo da revisão, as questões de pesquisa, as principais fontes primárias de estudo e os critérios para inclusão e exclusão dos artigos. O principal objetivo deste estudo foi identificar em quais itens de qualidade de software os valores das métricas de código-fonte podem influenciar.

Uma revisão sistemática de literatura visa encontrar o máximo possível de estudos primários relacionados à questão de pesquisa utilizando uma estratégia de busca imparcial. De acordo com esta premissa, foram utilizadas palavras chaves que pudessem identificar o máximo possível de trabalhos relevantes O Quadro 2.2 apresenta os termos e as consultas de pesquisa que foram aplicadas na busca por trabalhos realizados dentro do escopo desta revisão sistemática de literatura. Como todas as bases utilizadas são internacionais, a linguagem utilizada para efetivar a busca foi o inglês.

Quadro 2.2: Termos e consultas da revisão realizada nas bases de conhecimento

Termos de pesquisa	Consultas
Code Metrics *Code Quality Metrics* *Source-code Metrics* *Source-code Quality Metrics* *Software Quality Metrics* *Testability* *Maintainability* *Reusability* *Complexity*	*Code OR Source-code AND "quality metrics"* *Code OR "Source Code" AND "quality metrics"* *"Software Quality Metrics"* *Code OR Source-code AND "quality metrics" AND Testability OR Maintainability OR Reusability OR Complexity* *Code OR "Source Code" AND "quality metrics" AND Testability OR Maintainability OR Reusability OR Complexity* *"Software Quality Metrics" AND Testability OR Maintainability OR Reusability OR Complexity*

As bases científicas digitais onde as buscas foram realizadas consideraram os

estudos de Kitchenham e Charters (Kitchenham e Charters, 2007) e de Brereton e colegas (Brereton *et al.*, 2007). Dentre as diversas bases digitais disponibilizadas, as buscas foram realizadas nas bases científicas digitais: ScienceDirect, SpringLink, IEEE e ACM Digital, nas quais foram filtradas publicações do tipo *journal* e conferência. A pesquisa foi realizada em 2016 e 2017 e não houve delimitação de período, ou seja, publicações de qualquer ano foram consideradas. As buscas nas bases de conhecimento utilizando os termos de pesquisa foram aplicadas tanto no título quanto no *abstract* dos trabalhos. Desta forma foi obtida uma significativa base de 2.134 trabalhos.

Na sequência foi avaliado se os 2.134 estudos primários identificados no estágio anterior forneciam uma evidência direta sobre a questão de pesquisa. Critérios de exclusão foram definidos no protocolo para a seleção adequada dos estudos conforme descrito na sequência.

O primeiro critério de exclusão foi a identificação dos artigos de interesse com títulos duplicados. Neste momento foram excluídos 8 artigos. Na sequência foi verificado se, claramente, o objeto de pesquisa (métricas de código-fonte ou itens de qualidade de software) não era discutido no artigo. Para conduzir esta etapa foi realizada a leitura do *abstract* dos 2.126. Em caso de dúvida, o artigo foi mantido para a avaliação na próxima etapa. Esta etapa foi a que mais excluiu artigos da base de estudos primários, foram excluídos 1.811 artigos.

A terceira etapa teve como critério de exclusão a verificação se, claramente, não é discutida a aplicação das métricas de código-fonte nos itens de qualidade de software. Desta forma, com os 315 artigos restantes, foram realizadas buscas nos artigos utilizando as palavras-chave *code metrics*, *quality metrics*, *testability*, *maintainability*, *reusabilility* ou *complexity*, procurando evidências da aplicação destes temas. Esta estratégia foi utilizada pelo fato de que estes termos são os mais relacionados ao objetivo da revisão. A identificação de uma das palavras-chave selecionava o artigo para ser mantido na base, eventualmente foi realizada a leitura da introdução e conclusão do artigo. Em caso de dúvida, o artigo foi mantido para uma nova avaliação. Nesta etapa foram eliminados 119 artigos.

Finalmente, os 196 artigos restantes da etapa anterior foram lidos na íntegra para se certificar da aplicação das métricas de código-fonte nos itens de qualidade de software. Os 66 artigos nos quais não houve aplicação da relação entre as métricas de código-fonte e os itens de qualidade de software foram excluídos da base de artigos. O Quadro 2.3

mostra um resumo dos critérios de exclusão aplicados em cada uma das etapas das avaliações dos artigos selecionados na base inicial.

Quadro 2.3: Critérios de exclusão aplicados

Etapa	Critérios de exclusão	Artigos excluídos
1ª	Artigos com títulos duplicados	8
2ª	Claramente, os temas, métricas de código-fonte ou itens de qualidade de software, não são discutidos no artigo	1.811
3ª	Claramente, não é discutida a aplicação das métricas de código-fonte nos itens de qualidade de software	119
4ª	Não houve aplicação da relação entre as métricas de código-fonte e os itens de qualidade de software	66

Após a execução das etapas aplicando os critérios de exclusão resultaram 130 artigos. Estes artigos foram lidos, analisados e classificados de acordo com o tema abordado buscando responder às questões de pesquisa (QP) relevantes para nortear a construção da equação do cálculo do grau de contribuição do desenvolvedor, que é um dos objetivos desta tese. As questões de pesquisa propostas foram:

QP1: Existe relação entre as métricas de código-fonte e os itens de qualidade de software? – o objetivo desta questão de pesquisa foi identificar artigos envolvendo a relação entre as métricas de código-fonte calculadas e os itens de qualidade de software, focando na testabilidade, manutenibilidade, reusabilidade e complexidade;

QP2: Quais métricas de código-fonte podem indicar uma influência na testabilidade/manutenibilidade/reusabilidade/complexidade do software? - esta questão de pesquisa teve como objetivo artigos abordando as métricas de código-fonte e qual item(s) de qualidade de software pode(m) ser influenciado(s) pela respectiva métrica;

QP3: Quais os valores das métricas de código-fonte que indicam uma boa/ruim influência na testabilidade/manutenbilidade/reusabilidade/complexidade do software? – nesta questão de pesquisa buscou-se identificar trabalhos que estudaram os valores das métricas de código-fonte e qual o grau de risco destes valores influenciarem um determinado item de qualidade de software.

O Quadro 2.4 mostra a classificação dos artigos selecionados conforme as QP apresentadas. No caso da QP3, a classificação separou por item de qualidade de software influenciado, no caso, testabilidade, manutenibilidade, reusabilidade e complexidade.

Quadro 2.4: Total de artigos classificados pela QP abordada

Questão de Pesquisa (QP)		Total de artigos
QP1: Existe relação entre as métricas de código-fonte e os itens de qualidade de software?		49
QP2: Quais métricas de código-fonte podem indicar uma influência na testabilidade/manutenibilidade/reusabilidade/complexidade do software?		26
QP3: Quais os valores das métricas de código-fonte que indicam grau de risco bom/ruim em relação a influência na testabilidade/manutenbilidade/reusabilidade/complexidade do software?		
	Testabilidade	7
	Manutenibilidade	16
	Reusabilidade	13
	Complexidade	19
TOTAL		**130**

A classificação apresentada no Quadro 2.4 foi proposta com o objetivo de prover respostas mais claras às questões de pesquisa estabelecidas. A base de trabalhos analisada permitiu a construção das relações entre as métricas de código-fonte e os itens de qualidade do produto de software utilizadas nesta pesquisa e possibilitou a identificação dos conceitos que foram aplicados na equação do cálculo do grau de contribuição do desenvolvedor. Os resultados e discussões resultantes desta revisão sistemática de literatura seguem na sequência.

2.5.2 Resultados e Discussões da Revisão Sistemática de Literatura

Em um primeiro momento obteve-se as faixas de grau de risco de cada métrica de código-fonte aplicadas a programação Orientada a Objetos quando está sendo analisada a qualidade do produto de software. Os valores de algumas destas faixas foram originalmente descritos pelos próprios idealizadores das métricas, como (McCabe, 1976), (Chidamber e Kemerer, 1991) (Henderson-Sellers, 1996) e (Martin, R. C., 1994). Com a disseminação destes conceitos e disponibilização de aplicativos que facilitaram a recuperação destas informações, novos estudos foram propostos e as faixas de avaliação de risco na qualidade do produto do software sofreram alterações e foram atualizadas, pode-se até dizer que de acordo com os novos contextos de programação, como pode ser percebido nos trabalhos de (Horstmann, 2002), (Anderson, 2004), (Olague *et al.*, 2006), (Oliveira *et al.*, 2008), (Li, 2008) e (Filó *et al.*, 2015).

Nesta pesquisa são utilizadas as seguintes métricas para calcular o grau de contribuição do desenvolvedor: a Complexidade Ciclomática de McCabe, as métricas de

Chidamber e Kemerer para software orientado a objetos, as métricas de tamanho e as métricas de acoplamento de Martin. A seleção deste conjunto de métricas (CCM, CK e de tamanho) se deu em razão das pesquisas realizadas em relação à aplicação das métricas visando verificar a qualidade do código-fonte com respeito à manutenibilidade, testabilidade, entendimento (complexidade) e reusabilidade.

Segundo Anderson (2004), a CCM ou VG é uma das métricas de complexidade mais bem aceitas e utilizadas pela comunidade de desenvolvimento de software. As métricas de Chidamber e Kemerer, de acordo com Soliman *et al.* (2010) formam um dos conjuntos mais importantes de métricas orientadas a objetos, sendo também o conjunto mais amplamente referenciado. Segundo Plosch *et al.* (2010) as métricas de tamanho, CCM e CK são utilizadas em diversos trabalhos para monitorar a qualidade do produto de software em desenvolvimento, estabelecendo limiares para indicar se medidas de interesses foram violadas, ou não.

Também baseado nos estudos realizados por Riaz e colegas (Riaz *et al.*, 2009), verificou-se que as métricas de qualidade do produto de software focadas no código-fonte relacionadas a tamanho, complexidade e acoplamento foram as que retornaram previsões bem-sucedidas em relação a manutenibilidade de software. Estudo realizado por Bruntink e van Deursen (Bruntink e van Deursen, 2004) mostram a influência das métricas MLOC, Número de Campos, NOM, WMC, RFC, no que tange a testabilidade. Com relação à reusabilidade do código, estudo de Washizaki e colegas (Washizaki *et al.*, 2004), apesar de focarem nas métricas utilizadas na análise de teste caixa preta, mostram que WMC, DIT, CBO e LCOM* podem ser utilizadas como medidas de reusabilidade das classes em OO. Misra e Akman (Misra e Akman, 2008) através de estudos identificam que as métricas WMC, DIT, NOC e RFC têm condições de identificar índices que detectam o grau de complexidade do código analisado, no caso o grau de entendimento do código pelo usuário. Os estudos de (Anderson, 2004), (Olague *et al.*, 2006), (Henderson-Sellers, 1996), (Horstmann, 2002), (Li, 2008), (Martin, 2002), (Schroeder, 1999) e (Oliveira *et al.*, 2008) também relatam que as métricas de código-fonte podem indicar situações de aumento ou diminuição da qualidade do código relacionado com a manutenibilidade, a testabilidade, reutilizabilidade e complexidade.

Estudos mais recentes ((Yu e Zhou, 2010) (Karus e Dumas, 2012) (Ferreira *et al.*, 2012) (Rawat *et al.*, 2012) (Silva *et al.*, 2012) (Gómez *et al.*, 2015) (Taibi, 2013) (Drouin *et al.*, 2013) (Singh, 2013) (Wallace e Sheetz, 2014) (Finlay *et al.*, 2014) (Mehdi *et al.*,

2014) (Goyal *et al.*, 2014) (Filó *et al.*, 2015) mostram a aplicabilidade de métricas de código-fonte na busca pela validação da qualidade do produto de software. Também a pesquisa de Varela e colegas (Varela *et al.*, 2017) validou as escolhas em relação as métricas de código-fonte utilizadas nesta pesquisa, assim como a ferramenta de coleta destas métricas.

Os trabalhos citados foram analisados, comparados, combinados e a partir deles foi possível indicar as influências das métricas de código-fonte nos itens de qualidade de produto de software conforme relatado na sequência. Na Tabela 2.3 é possível verificar um resumo do grau de risco de a alteração realizada no código-fonte influenciar a qualidade do produto. A descrição de cada métrica da lista, bem como as faixas de avaliação de risco em relação à influência na qualidade do produto, foi definida a partir da revisão sistemática de literatura realizada.

Tabela 2.3: Lista das métricas e as faixas de avaliação de risco na qualidade do produto

Métrica de código-fonte	Grau do Risco
CCM ou VG – Complexidade Ciclomática de McCabe	Baixo – 1<= VG <= 10 Moderado – 11 <= VG <= 20 Alto – 21 >= VG <= 50 Muito Alto – VG > 50 (Anderson, 2004)
WMC – Métodos Ponderados por Classe	Baixo – 1 <= WMC <= 20 Moderado – 21 <= WMC <= 100 Alto – WMC > 100 (Olague *et al.*, 2006)
DIT – Profundidade da Árvore de Herança	Ausente – DIT = 0 Desejável – 1 <= DIT <= 5 Profundo – DIT > 5 (Henderson-Sellers, 1996)
CBO – Acoplamento entre Classes	Fraco – 00 <= CBO <= 05 Forte – CBO > 05 (Li, 2008)
RFC – Resposta para uma Classe	Bom – 0 <= RFC >= 50 Ruim – RFC > 50 (Chidamber e Kemerer, 1991)
NOM – Número de métodos	Desejável – 1 <= NOM <= 20 Aceitável – 21 <= NOM <= 40 Alto – NOM > 40
NSC – Número de filhos	Bom – NSC <=1 Regular – 1 < NSC <=3 Ruim – NSC > 3 (Filó *et al.*, 2015)

(continua)

Métrica de código-fonte	Grau do Risco
LCOM* - Falta de coesão em Métodos	Bom – 0 <= LCOM* <= 0,5 Regular – 0,5 < LCOM * <= 1 Ruim – LCOM * > 1 (Horstmann, 2004) (Henderson-Sellers, 1996)
MLOC - Número de linhas de código do método	Bom – MLOC <=10 Regular – 10 < MLOC <= 30 Ruim – MLOC > 30 (Filó *et al.*, 2015)
NBD - Profundidade de blocos aninhados	Bom – NBD <=1 Regular – 1 < NBD <=3 Ruim – NBD > 3 (Filó *et al.*, 2015)
NORM - Número de métodos sobrescritos	Bom – NORM <=2 Regular – 2 < NORM <=4 Ruim – NORM > 4 (Filó *et al.*, 2015)
SIX - Índice de especialização	Bom – SIX <= 0,019 Regular – 0,019 < SIX <= 1,333 Ruim – SIX > 1,333 (Filó *et al.*, 2015)
NOF - Número de campos	Bom – NOF <=3 Regular – 3 < NOF <=8 Ruim – NOF > 8 (Filó *et al.*, 2015)
NSM - Número de métodos estáticos	Bom – NSM <=1 Regular – 1 < NSM <= 3 Ruim – NSM > 3 (Filó *et al.*, 2015)
PAR - Número de parâmetros	Bom – PAR <=2 Regular – 2 < PAR <= 4 Ruim – PAR > 4 (Filó *et al.*, 2015)
NOC - Número de classes	Bom – NOC <= 7 Regular – 11 < NOC <= 28 Ruim – NOC > 28 (Filó *et al.*, 2015)
Ca - Acoplamento aferente	Bom – Ca <=7 Regular – 7 < Ca <= 39 Ruim – Ca >39 (Filó *et al.*, 2015)
Ce - Acoplamento eferente	Bom – Ce <=6 Regular – 6 < Ce <= 16 Ruim – Ce > 16 (Filó *et al.*, 2015)
I ou RMI - Instabilidade	Bom/estável – $I = 0$ Ruim/instável – $I = 1$ (Martin, R. C., 1994)
A ou RMA - Abstração	Bom/abstrata – A = 1 Ruim/concreta – A = 0 (Martin, R. C., 1994)

A partir dos estudos realizados durante esta revisão sistemática de literatura propõe-se nesta pesquisa o agrupamento das métricas de acordo com a influência destas nos índices de qualidade do produto de software relacionados à manutenibilidade, reusabilidade, testabilidade e complexidade do código.

A Complexidade Ciclomática de McCabe (CCM ou VG) é utilizada para medir a complexidade de um método e, um baixo valor de CCM do método geralmente é melhor já que representa uma boa qualidade do código. A CCM não pode ser utilizada como medida de complexidade de uma classe em função da herança, mas a combinação do valor de CCM individual dos métodos pode retornar a CCM de uma classe. A CCM está diretamente relacionada ao item de qualidade do produto de software que diz respeito aos atributos de complexidade.

A métrica WMC de uma classe é a soma da complexidade de seus métodos. O número de métodos e a complexidade de cada um é um indicador de quanto tempo e esforço será necessário para desenvolver e manter a classe. Uma classe com um baixo valor de WMC normalmente aponta para um alto polimorfismo. Uma classe com um alto valor de WMC indica que ela é complexa, de aplicação específica, e desta forma é difícil de ser reutilizada e mantida. Esta métrica avalia o entendimento, complexidade e reuso do código, desta forma um aumento no WMC indica um aumento na complexidade e, consequentemente um decréscimo no índice de qualidade do produto do software.

A profundidade de uma classe em relação a hierarquia de herança (DIT) indica o maior caminho até a raiz da árvore de herança da classe e é medida pelo número de ancestrais da classe. Quanto mais profunda for a classe na hierarquia, maior o número de métodos herdados, tornado mais difícil de identificar seu comportamento. Quanto maior a profundidade, maior a complexidade do projeto, já que mais métodos e classes estão envolvidos. Desta forma a métrica DIT afeta diretamente a qualidade do produto de software já que aumenta o reuso, mas também aumenta a complexidade do código.

A métrica de CBO de uma classe é a soma de todas as classes com as quais determinada classe tem acoplamento. Para um projeto eficiente, o acoplamento deve ser baixo, já que um alto acoplamento é ruim para um projeto modular e não permite o reuso. Uma classe com baixo valor de acoplamento será mais independente e com facilidade de reuso em outras aplicações. Um alto acoplamento aumenta a complexidade do software e os módulos tornam-se mais difíceis de serem entendidos, alterados ou corrigidos,

quando estão fortemente inter-relacionados com outros módulos. O CBO avalia a eficiência, reusabilidade e complexidade do software, e desta forma, afeta diretamente a qualidade do produto do software.

A métrica RFC de uma classe mede o número de métodos que são invocados em resposta a uma mensagem recebida por um objeto de uma classe. Esta métrica mostra o quanto uma classe se comunica com outras classes. Quanto maior o número de métodos invocados por uma classe através das mensagens, maior será a complexidade desta classe. Desta forma, se um grande número de métodos é invocado em uma resposta a uma mensagem, o teste e *debbug* desta classe torna-se complicado pois necessita de um alto nível de entendimento por parte do testador. É possível então dizer que, se o valor de RFC aumentar, o esforço necessário para testar também aumenta, já que a sequência necessária de testes aumenta. Esta métrica avalia a testabilidade do código-fonte, desta forma afeta diretamente a qualidade do produto de software.

O número de filhos (NSC) mede a quantidade de descendentes diretos de cada classe. As classes com alto valor de NSC são consideradas difíceis de modificar e usualmente necessitam de mais testes em função dos efeitos das alterações realizadas em todos os seus filhos. Elas também são consideradas mais complexas já que uma classe com um alto número de filhos deve prover serviços em um grande número de contextos e por isto precisam ser mais flexíveis. Por outro lado, quanto maior o número de filhos maior a reusabilidade, já que a herança é uma forma de reuso. Quando uma classe tem um alto número de filhos será necessário mais teste para cada método, aumentando o tempo de testes do software. A métrica NSC avalia eficiência, reausabilidade e testabilidade e, portanto, afeta a qualidade do produto de software.

A coesão indica a grau de relação entre os métodos dentro de uma classe. Um bom projeto maximiza a coesão e promove o encapsulamento. Uma alta coesão significa que as classes não dividem seus atributos com outras classes e podem ser facilmente reutilizadas. Desta forma uma alta coesão promove a reusabilidade e diminui a complexidade. A métrica LCOM* mede o grau de falta de coesão entre os métodos, um aumento no valor da LCOM* diminui a qualidade do software. Seu valor afeta diretamente a reusabilidade, eficiência e complexidade do código-fonte e consequentemente, o nível de qualidade de produto do software.

O número de linhas de código representa o tamanho do software. Esta medida é utilizada para avaliar o grau de entendimento do software, já que quanto maior o tamanho

do código-fonte menor o grau de entendimento dele. Códigos muito grande podem também colocar em risco seus atributos de reusabilidade. A métrica MLOC retorna o número de linhas de código do método, assim pode-se concluir que a MLOC afeta diretamente a qualidade de produto do software. Porém é possível identificar que naturalmente o tamanho do software aumenta durante seu ciclo de vida, e isto claramente indica que o grau de entendimento do software diminui durante seu ciclo de vida e desta forma a qualidade também diminui. Aliado ao aumento do tamanho do software durante seu ciclo de vida, o software também tende a ficar mais complexo em função de novas versões e sua manutenção tende a se tornar uma tarefa mais difícil. Estes fatores colaboram para a necessidade de uma medição contínua da qualidade do produto do software. Esta medição pode ser estimulada pelo cálculo do grau de contribuição do desenvolvedor a cada *commit* que ele realiza no software, auxiliando os gerentes de projeto na busca de manter o nível de qualidade da aplicação dentro dos limites desejáveis pela organização, ou com uma mínima variação.

Novamente com base nos trabalhos pesquisados, analisados e associados durante a revisão sistemática de literatura, conforme a Tabela 2.3, foi possível categorizar as 20 métricas de código-fonte em quatro categorias diferentes. O primeiro grupo de categoria aborda as Métricas de Complexidade, considerando que artefatos mais complexos em OO como código, classes e métodos podem tornar o código difícil de entender, manter e testar ((McCabe, 1976), (Chidamber e Kemerer, 1991), (Henderson-Sellers, 1996), (Martin, 2002) e (Filó *et al.*, 2015)). As métricas de complexidade ciclomática de McCabe, métodos ponderados por classe, falta de coesão no método e profundidade de blocos aninhados podem indicar situação de complexidade levando os usuários a dispenderem mais tempo e esforço cognitivo para entenderem o programa ((Anderson, 2004), (Olague *et al.*, 2006), (Oliveira *et al.*, 2008), (Horstmann, 2002) e (Filó *et al.*, 2015)).

A segunda categoria, Métricas de Herança, reúne as métricas de profundidade da árvore de herança, número de filhos, número de métodos sobrescritos e índice de especialização. Eles formam a categoria de herança, uma vez que podem indicar problemas de abstração e manutenção de software ((Chidamber e Kemerer, 1991), (Schroeder, 1999), (Henderson-Sellers, 1996) e (Filó *et al.*, 2015)).

Algumas métricas de tamanho, como número de linhas de código do método, número de atributos por classe, o número de atributo estático, número de métodos estáticos, número de parâmetro, número de interface e número de pacotes pode indicar

situações na complexidade, manutenção e reutilização do código ou item ((Henderson-Sellers, 1996), (Harrison *et al.*, 1997) e (Filó *et al.* ,2015)). Estas métricas foram classificadas na categoria de Métricas de Tamanho.

Finalmente, um forte acoplamento no software torna o código mais difícil de manter, modificar e reutilizar, e também pode comprometer a abstração do código e sua estabilidade. Os índices de acoplamentos aferentes e eferentes, a instabilidade, abstração e distância normalizada da sequência principal são métricas que podem indicar situações de acoplamento entre códigos e itens ((Martin, 2002) e (Filó *et al.*, 2015)). Desta forma, estes índices foram categorizados como Métricas de Acoplamento.

A Tabela 2.4 apresenta um resumo destes estudos, agrupando essas métricas com base em sua influência sobre a qualidade do código.

Tabela 2.4: Resumo da influência das métricas na qualidade do código

Grupo	Influência da métrica na qualidade do código	Métrica
Métricas de Complexidade	Quanto mais complexo o código/classe/método mais difícil de ser entendido, mantido e testado.	Complexidade ciclomática de McCabe - CCM ou VG
		Métodos ponderados por classe - WMC
		Falta de coesão em métodos- LCOM*
		Profundidade de blocos aninhados – NBD
		Resposta para uma Classe – RFC
Métricas de Herança	Pode indicar problemas de abstração e manutenibilidade do código.	Profundidade da árvore de herança - DIT
		Número de filhos – NSC
		Número de métodos sobrescritos - NORM
		Índice de especialização - SIX
Métricas de Tamanho	Pode indicar problemas de entendimento, manutenibilidade e reusabilidade do código.	Linhas de código do método - MLOC
		Número de campos - NOF
		Número de métodos estáticos - NSM
		Número de parâmetros - PAR
		Número de classes - NOC
		Número de métodos – NOM
Métricas de Acoplamento	Forte acoplamento no software torna o código difícil de manter, modificar e reutilizar. Pode também comprometer a estabilidade e abstração do código.	Acoplamento aferente - Ca
		Acoplamento eferente - Ce
		Acoplamento entre objetos das classes – CBO
		Métrica de instabilidade – I ou RMI
		Abstração – A ou RMA

Fonte: a própria autora.

A partir desta revisão sistema de literatura foi possível compilar relevâncias na

indicação das métricas de código-fonte indicando a influência nos itens da qualidade do produto de software de manutenibilidade ((Riaz, *et al.*, 2009) e (Dash *et al.*, 2012)), testabilidade (Almugrin *et al.*, 2016), reusabilidade (Mijaé e Stapié, 2015) e complexidade do código (Honglei *et al.*, 2009). A partir destas pesquisas foi possível identificar quais itens de qualidade do produto de software a métrica de qualidade de código poderia influenciar. A Tabela 2.5 apresenta um compilado destes estudos.

Tabela 2.5: Compilado das métricas de código fonte, grau de risco de influência e item de qualidade do produto de software influenciado

Métrica de código-fonte	Grau do Risco	Item de qualidade do produto de software influenciado
Complexidade Ciclomática de McCabe - CCM ou VG	Baixo – 1<= VG <= 10 Moderado – 11 <= VG <= 20 Alto – 21 >= VG <= 50 Muito Alto – VG > 50 (Anderson, 2004)	Complexidade
Métodos Ponderados por Classe - WMC	Baixo – 1 <= WMC <= 20 Moderado – 21 <= WMC <= 100 Alto – WMC > 100 (Olague *et al.*, 2006)	Complexidade Testabilidade Reusabilidade
Profundidade da Árvore de Herança - DIT	Ausente – DIT = 0 Desejável – 1 <= DIT <= 5 Profundo – DIT > 5 (Henderson-Sellers, 1996)	Complexidade
Acoplamento entre Classes - CBO	Fraco – 00 <= CBO <= 05 Forte – CBO > 05 (Li, 2008)	Reusabilidade Manutenibilidade
Resposta para uma Classe - RFC	Bom – 0 <= RFC >= 50 Ruim – RFC > 50 (Chidamber e Kemerer, 1991)	Complexidade Testabilidade
Número de métodos - NOM	Desejável – 1 <= NOM <= 20 Aceitável – 21 <= NOM <= 40 Alto – NOM > 40	Reusabilidade Testabilidade
Número de filhos - NSC	Bom – NSC <=1 Regular – 1 < NSC <=3 Ruim – NSC > 3 (Filó *et al.*, 2015)	Reusabilidade Manutenibilidade
Falta de coesão em Métodos - LCOM*	Bom – 0 <= LCOM* <= 0,5 Regular – 0,5 < LCOM * <= 1 Ruim – LCOM * > 1 (Horstmann, 2004) (Henderson-Sellers, 1996)	Testabilidade Reusabilidade
Número de linhas de código do método - MLOC	Bom – MLOC <=10 Regular – 10 < MLOC <= 30 Ruim – MLOC > 30 (Filó *et al.*, 2015)	Complexidade

(continua)

Métrica de código-fonte	Grau do Risco	Item de qualidade do produto de software influenciado
Profundidade de blocos aninhados – NBD	Bom – NBD <=1 Regular – 1 < NBD <=3 Ruim – NBD > 3 (Filó *et al.*, 2015)	Reusabilidade
Número de métodos sobrescritos - NORM	Bom – NORM <=2 Regular – 2 < NORM <=4 Ruim – NORM > 4 (Filó *et al.*, 2015)	Complexidade
Índice de especialização - SIX	Bom – SIX <= 0,019 Regular – 0,019 < SIX <= 1,333 Ruim – SIX > 1,333 (Filó *et al.*, 2015)	Manutenibilidade Complexidade
Número de campos - NOF	Bom – NOF <=3 Regular – 3 < NOF <=8 Ruim – NOF > 8 (Filó *et al.*, 2015)	Complexidade Manutenibilidade
Número de métodos estáticos – NSM	Bom – NSM <=1 Regular – 1 < NSM <= 3 Ruim – NSM > 3 (Filó *et al.*, 2015)	Complexidade Manutenibilidade
Número de parâmetros - PAR	Bom – PAR <=2 Regular – 2 < PAR <= 4 Ruim – PAR > 4 (Filó *et al.*, 2015)	Reusabilidade
Número de classes - NOC	Bom – NOC <= 7 Regular – 8 < NOC <= 28 Ruim – NOC > 28 (Filó *et al.*, 2015)	Complexidade Manutenibilidade
Acoplamento aferente - Ca	Bom – Ca <=7 Regular – 7 < Ca <= 39 Ruim – Ca >39 (Filó *et al.*, 2015)	Reusabilidade Manutenibilidade
Acoplamento eferente - Ce	Bom – Ce <=6 Regular – 6 < Ce <= 16 Ruim – Ce > 16 (Filó *et al.*, 2015)	Reusabilidade Manutenibilidade
Instabilidade – I ou RMI	Bom/estável – $I = 0$ Ruim/instável – $I = 1$ (Martin, 1994)	Reusabilidade Manutenibilidade
Abstração – A ou RMA	Bom/abstrata – A = 1 Ruim/concreta – A = 0 (Martin, 1994)	Reusabilidade Manutenibilidade

Fonte: a própria autora.

A partir dos dados disponíveis dos projetos de software armazenados nos SCV, além do código-fonte alterado ou desenvolvido pelos desenvolvedores da equipe de desenvolvimento colaborativo de software, também foram analisadas as mensagens

deixadas pelos desenvolvedores e gerentes de projeto nos *commits* e *issues* dos projetos de software. Para realizar a análise nestes textos através da Recuperação da Informação foi utilizada a análise de sentimentos, cujos conceitos seguem a seguir.

2.6. Recuperação da Informação

O termo Recuperação da Informação (RI), do inglês *Information Retrieval*, foi criado por Calvin Mooers em torno de 1950 e designa uma área multidisciplinar da Ciência da Computação envolvendo psicologia cognitiva, arquitetura da informação, linguística, semiótica, ciência da informação, ciência da computação, biblioteconomia e estatística. Esta área da computação trata do armazenamento de documentos e a recuperação automática de informações. A RI, dentre as diversas definições do termo, busca o tratamento da informação envolvendo a catalogação, categorização ou indexação e classificação da informação, particularmente textual, demandada pelo usuário (Russell e Norvig, 1995). Na abordagem da RI, o usuário de um sistema de recuperação de informação está interessando em recuperar informação sobre um determinado assunto e não em recuperar dados que satisfazem sua expressão de busca. Esta característica é o que diferencia os sistemas de recuperação dos Sistemas Gerenciadores de Banco de Dados (SGBD).

O processo de RI consiste em identificar, no conjunto de dados (*corpus*) de um sistema, quais dados atendem à necessidade de informação do usuário. Desta forma os sistemas de RI tratam com objetos linguísticos (textos) e herdam toda a problemática inerente ao tratamento da linguagem natural. Para tanto os sistemas de RI precisam representar o conteúdo do *corpus* e apresentá-los ao usuário de maneira que ele entenda e satisfaça sua necessidade de informação. Para ser eficaz na tarefa de satisfazer a necessidade de informação do usuário os sistemas de RI ordenam os documentos de acordo com seu grau de relevância baseado no contexto do usuário. A noção de relevância é um conceito fundamental em RI e é um componente importante para calcular a classificação dos documentos que venha a gerar o conjunto de respostas para o usuário.

O principal objetivo da RI é recuperar o maior número possível de documentos relevantes e o menor número possível de documentos não relevantes. Para facilitar o processo de RI é importante adotar um vocabulário padronizado, chamado vocabulário controlado, que pode utilizar linguagem natural ou artificial para representar o conteúdo do corpus (Chowdhury, 2010). Na Figura 2.23 está representado o processo de

recuperação da informação.

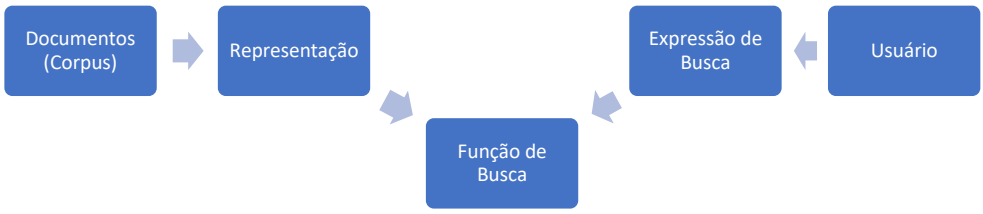


Figura 2.23: Representação do processo de recuperação da informação
Fonte: a própria autora.

Os passos que envolvem o sistema de RI podem ser definidos como: operação de consulta que envolve a especificação do conjunto de termos associados ou não por operadores booleanos que representam a solicitação do usuário; operação de indexação que envolve a criação de estruturas de dados associados aos documentos do *corpus* e; a pesquisa e ordenação que envolve o processo de recuperação de documentos de acordo com a solicitação do usuário e sua ordenação por meio de um grau de similaridade entre o documento e a consulta. A classificação pode ser calculada utilizando modelos para representar os documentos e a consulta ao usuário. E a eficiência de um sistema de RI está diretamente ligada ao modelo ele utiliza. Um modelo, por sua vez, influencia diretamente no modo de operação do sistema. Os modelos clássicos são: modelo booleano, modelo vetorial e probabilístico e o modelo semântico. Apesar de alguns destes modelos terem sido criados nos anos 60 e 70 e aperfeiçoados nos anos 80 as suas ideias principais ainda estão presentes na maioria dos sistemas de recuperação atuais. Outros modelos resumem propostas mais recentes que utilizam derivados da inteligência artificial e representam alternativas promissoras para estudos (Russell e Norvig, 1995).

Quando um sistema de RI utiliza linguagem natural é necessário aplicar métodos do chamado Processamento da Linguagem Natural (PLN) com o objetivo de alcançar maior precisão da informação recuperada. As pesquisas têm evoluído neste campo uma vez que é difícil encontrar a informação relevante, principalmente pela quantidade de informação disponível. Os pesquisadores têm procurado abordagens alternativas para solucionar este problema. Além da aplicação de métodos estatísticos, o processamento da linguagem natural, com motivação linguística, é uma dessas alternativas (Chowdhury,

2010).

2.6.1. Processamento de Linguagem Natural

O processamento de linguagem natural trata computacionalmente os diversos aspectos da comunicação humana, como sons, palavras, sentenças e discursos, considerando formatos e referências, estruturas e significados, contextos e usos. Desta forma o PLN é um conjunto de técnicas computacionais para a análise de textos em um ou mais níveis linguísticos, com o propósito de simular o processamento humano da língua (Russell e Norvig, 1995).

O estudo do PLN parte do princípio que a representação do significado da sentença, independente de contexto, é obtida fazendo uso de uma forma lógica. A forma lógica codifica os possíveis sentidos de cada palavra e identifica os relacionamentos semânticos entre palavras e frases. Uma vez que os relacionamentos semânticos são determinados, alguns sentidos para as palavras tornam-se inviáveis e, assim, podem ser desconsiderados (Allen, 1995) e (Franconi, 2001). Na Figura 2.24 está representada a transformação de uma sentença na estrutura sintática para a forma lógica.

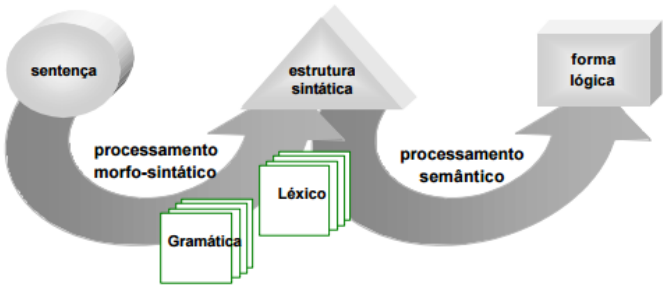


Figura 2.24: Transformação da sentença na estrutura sintática para a forma lógica
Fonte: Adaptado de (Allen, 1995) e (Franconi, 2001).

A estrutura sintática de uma sentença é obtida através do processamento morfológico, sendo a representação desta estrutura regida por leis gramaticais, definidas em uma gramática (Figura 2.25). Outras informações necessárias à esta etapa, como as categorias morfológicas das palavras, são encontradas em um léxico. Já o mapeamento da estrutura sintática da sentença em sua forma lógica é realizado pelo processamento semântico e, nele, o léxico também atua, com informações sobre o significado dos itens lexicais. Desta forma a gramática e o léxico são recursos necessários para a transformação da sentença em forma lógica (Russell e Norvig, 1995).

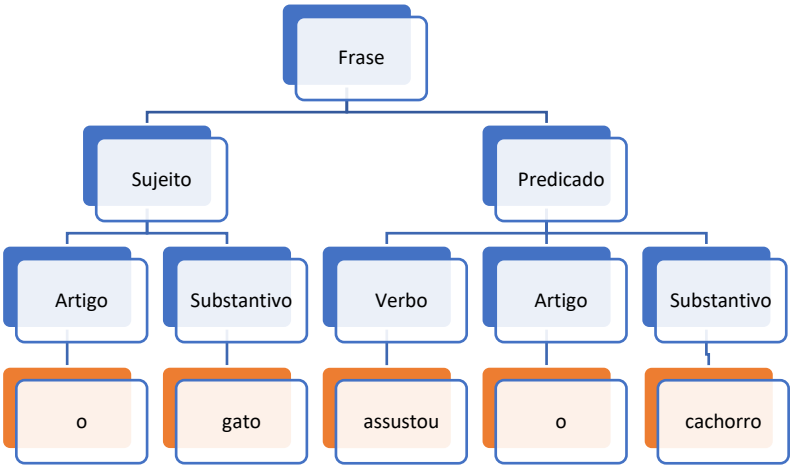


Figura 2.25: Exemplo de estrutura sintática da frase

Fonte: a própria autora.

O PLN não se caracteriza como um modelo de RI, uma vez que não propõe uma estrutura para a representação dos documentos e não formaliza de forma explícita uma função de busca. Porém, é por meio do PLN que a RI se aproxima do arsenal metodológico, a Inteligência Artificial, e viabiliza soluções para alguns de seus problemas. O PLN aplicado às expressões de busca de um sistema de RI é importante para interpretar a necessidade de informação dos usuários. Porém, o número de palavras reduzido das expressões de busca que são utilizadas pelos usuários dificulta esta tarefa, não permitindo uma interpretação adequada das expressões ((Allen, 1995) (Franconi, 2001)).

A atuação mais importante da PLN na RI está na interpretação do conteúdo dos documentos, a fim de gerar uma representação adequada destes, porém o uso do PLN não elimina a necessidade da utilização de métodos estatísticos e pode ser visto como uma ferramenta complementar a estes métodos. Assim, na maioria das vezes as técnicas de PLN são utilizadas na melhoria do desempenho de algumas tarefas da RI tradicional, como a indexação automática ((Allen, 1995) (Franconi, 2001)).

Uma forma de processamento de linguagem natural, análise de texto e linguística computacional que pode ser aplicada é a análise de sentimento ou análise de opinião comentada a seguir.

2.6.2. Análise de Sentimento

A análise de sentimentos (*sentiment analisys*) também tratada como análise de

opinião (*opinion mining*) é um campo da área de mineração de dados. A análise de sentimentos é o processo de atribuição de um valor de humor quantitativo (positivo, negativo ou neutro) a um fragmento de texto. Esta área de pesquisa tem sido praticada em vários domínios, como por exemplo, estudos de análise de sentimentos para revisões de filmes, análises de produtos, notícias e *blogs*. Além de estar sendo aplicada em tarefas de engenharia de software, como avaliar revisões de aplicativos ou analisar as emoções do desenvolvedor em mensagens de *commits* ((Lin *et al.*, 2018) (Jongeling *et al.*, 2014)).

A pesquisa sobre análise de sentimento tem dois focos: identificar se uma determinada entidade textual é subjetiva ou objetiva e identificar polaridade (positiva/ neutra/negativa) de textos subjetivos. No domínio da análise de sentimento, os textos pertencem às classes positivas ou negativas. Podendo haver classes binárias ou com valores múltiplos, como positivo, negativo e neutro (ou irrelevante). A extração de sentimento abrange diferentes técnicas como: análise lexical, análise baseada em aprendizagem de máquina e análise híbrida ou combinada (Pang e Lee, 2008).

A técnica de análise lexical utiliza um dicionário que consiste em léxicos pré-marcados. O texto de entrada é convertido em *tokens*. Para cada *token* encontrado é realizada uma correspondência no dicionário léxico. Se houver uma correspondência no conjunto do léxico marcado como positivo, a pontuação será adicionada ao total de pontuação para o texto de entrada. Caso contrário, a pontuação será diminuída ou a palavra será marcada como negativa. Esta é uma técnica simples e que, nas pesquisas, mostrou-se eficiente. Porém, tem uma limitação, seu desempenho degrada drasticamente com o crescimento exponencial do tamanho do dicionário utilizado para comparação (Sadegh *et al.*, 2012).

As técnicas de análise baseadas em aprendizagem de máquina, na análise de sentimento, empregam a variante de aprendizagem supervisionada. Esta técnica compreende as seguintes etapas: coleta de dados, pré-processamento, dados de treinamento, classificação e resultados de plotagem. Para a aplicação desta técnica, nos dados de treinamento, é fornecida uma coleção de *corpora* com *tags*. É apresentado ao classificador uma série de vetores de características dos dados. Na sequência, um modelo é criado com base no conjunto de dados de treinamento, que então é empregado sobre o novo texto para ser classificado. Nessa técnica, a seleção apropriada dos vetores de características é que irá garantir a acurácia do classificador. Existe uma variedade de características, como: o número de palavras positivas, o número de palavras negativas, o

comprimento do documento, o algoritmo SVM (*Support Vector Machines*) e o algoritmo Naïve Bayes, por exemplo. Esta abordagem possuí limitações, mas supera a limitação da abordagem lexical da degradação do desempenho, e apresenta um bom funcionamento mesmo quando o tamanho do dicionário cresce exponencialmente (Shaikh e Deshpande, 2016).

Com os avanços na análise de sentimentos, as pesquisas começaram a explorar a possibilidade de uma abordagem híbrida que combina a precisão de uma técnica de aprendizado de máquina e a velocidade da técnica lexical. Existem alguns tipos diferentes de combinação destas duas técnicas que conseguiram obter bons resultados de desempenho na determinação da análise de sentimento.

Técnica de análise lexical

A técnica de análise lexical utiliza um dicionário de palavras positivas e negativas, como por exemplo, amor e ódio, e conta quantas vezes elas ocorrem. Modificações desta abordagem incluem a identificação de termos negadores, palavras que melhoram o sentimento em outras palavras (como “realmente amar”, “odiar”) e estruturar sentenças em geral. Uma abordagem mais sofisticada busca identificar recursos de texto que poderiam ser subjetivos em alguns contextos e, em seguida, usar informações contextuais para decidir se elas são subjetivas em cada novo contexto (Pang e Lee, 2008).

Existem 3 métodos para construir um léxico de sentimento: construção manual, métodos baseados em *corpus* e métodos baseados em dicionário. Como esta pesquisa não requer grande *corpora* ou mecanismos com capacidades especiais e com base na complexidade e aplicação (Pang e Lee, 2008), estudos apontaram que os métodos baseados em dicionário se encaixam no âmbito deste trabalho.

Algumas ferramentas de técnicas de análise lexical especializada foram estudadas, como SentiWordNet (www.sentiwordnet.isti.cnr.it), SentiStrength (www.sentistrength.wlv.ac.uk), SenticNet (www.sentic.net/projects), Índice de Felicidade ((Dodds *et al.*, 2011), (Mitchell *et al.*, 2013)). Como resultado, para analisar a presença de palavras específicas nas mensagens dos *commits* e *issues*, os estudos apontaram para o uso do SentiStrength. Que é uma ferramenta de extração de sentimentos lexical especializada em lidar com textos curtos e de baixa qualidade, como é o caso das mensagens dos *commits* e *issues*.

2.7. Considerações Finais

Neste capítulo foram apresentados conceitos importantes e necessários que nortearam este trabalho, como o ambiente colaborativo de desenvolvimento de software, qualidade de software e as métricas de código-fonte disponíveis na literatura. A partir da revisão sistema de literatura apresentada, os trabalhos selecionados foram analisados, comparados e agrupados de forma a construir o conhecimento aplicado nesta pesquisa para a construção da equação do cálculo do grau de contribuição do desenvolvedor participante no desenvolvimento colaborativo de software.

Considerando as métricas de qualidade de software disponíveis na literatura, esta pesquisa utiliza as métricas de produto voltadas para o código-fonte. De forma mais específica, são utilizadas as métricas que influenciem na manutenibilidade, testabilidade, reusabilidade e entendimento do código (complexidade). Isto porque uma das medidas de qualidade do produto de software está ligada à sua facilidade de manutenção, teste, reuso e entendimento do código desenvolvido.

As variações nos índices de qualidade do produto de software é que nortearam o cálculo do grau de contribuição do participante no desenvolvimento colaborativo de software. Indicando se houve aumento, redução ou manteve-se os índices de qualidade do software ao longo do ciclo de desenvolvimento do projeto. Estas métricas e suas relações com a qualidade do produto de software foram estudadas e analisada e esta relação está detalhada no Capítulo 5.

Também o conceito de Recuperação da Informação foi tratado neste capítulo e de forma específica a Análise de Sentimentos que foi aplicada na verificação das mensagens dos *commits* e *issues*. Para fins desta pesquisa, analisou-se manualmente uma pequena amostra de mensagem de *commits* e *issues* postados pelos desenvolvedores e gerentes de projeto nos projetos de software recuperados do GitHub. Percebeu-se que estas mensagens eram geralmente curtas e escritas em linguagem natural informal. Considerando que esta pesquisa teve o interesse de identificar palavras específicas em mensagens de *commit* e *issues*, a técnica de análise lexical foi escolhida para ser estudada e aplicada.

Na sequência serão apresentados os trabalhos relacionados a esta pesquisa.

Capítulo 3

Estado da Arte

Considerando o escopo desta pesquisa e os estudos realizados no Capítulo 2, algumas pesquisas relacionadas aos temas que envolvem desenvolvimento colaborativo de software, métricas de software e medidas de colaboração foram avaliadas, com objetivo de melhor situar a investigação científica em relação aos trabalhos já realizados na área. Nesta análise foram considerados os trabalhos que pudessem contribuir para a definição do cálculo do grau de contribuição do desenvolvedor, analisando critérios como a atualidade, a relevância e o alinhamento com os objetivos da pesquisa.

Visando conhecer o estado atual dos temas relevantes para esta pesquisa, foram buscados trabalhos que realizaram revisões sistemáticas sobre estes temas por considerar que estas disponibilizam informações relevantes e que podem contribuir para o entendimento mais acurado sobre os temas.

Em um primeiro momento é apresentada a forma como os artigos foram selecionados nas bases de dados eletrônicas disponíveis. Os trabalhos estudados foram agrupados em duas seções: métricas de software e medidas de colaboração. A partir destes estudos foi possível verificar lacunas de pesquisa na área e trabalhos relacionados com o objeto desta tese, norteando a pesquisa realizada.

3.1. Seleção de Artigos

A fim de encontrar os artigos que deram suporte para elaboração desta pesquisa, algumas bases de artigos foram consultadas. Estas bases são apresentadas nos trabalhos de Kitchenham e Charters (Kitchenham e Charters, 2007) e de Brereton e colegas (Brereton *et al.,* 2007). O primeiro trabalho identificou nove bases eletrônicas de

pesquisa, e o segundo trabalho citado, catalogou oito bases. Justifica-se a utilização dos trabalhos de (Kitchenham E Charters, 2007) e (Brereton *et al.*, 2007) por serem os trabalhos adotados pela comunidade de engenharia de software como referências para realização de revisões sistemáticas. Ressalta-se que a utilização destas fontes serviu para determinar, além do encaminhamento de uma revisão sistemática de literatura, quais bases de conhecimento eletrônicas seriam utilizadas para a busca de artigos relevantes para os objetivos desta pesquisa. Utilizando as bases científicas digitas: ScienceDirect, SpringLink, IEEE e ACM Digital, foram pesquisados artigos de trabalhos que tratassem de temas envolvendo colaboração, medidas e métricas de colaboração e desenvolvimento colaborativo de software.

O Quadro 3.1 apresenta alguns dos termos e as consultas de pesquisa que foram utilizadas. Da mesma forma que a revisão sistemática apresentada no Capítulo 2, como todas as bases usadas são internacionais, a linguagem utilizada para efetivar a busca foi o inglês.

Quadro 3.1: Alguns termos e consulta da pesquisa realizada nas bases de conhecimento

Termos de pesquisa	Consultas
Computer Supported Cooperative Work (CSCW) *Groupware* *Collaborative Software Development (CSD)* *Collaborative Software Engineering* *Cooperative Software Development* *Cooperative Software Engineering* *Metrics in CSCW* *Metrics in CSD* *Metrics in Collaboration*	*Collaboration AND "software development"* *Collaboration OR Cooperation AND "software development"* *Metric OR Measure AND "software development"*

Ainda seguindo as etapas da revisão de literatura apresentada no Capítulo 2, aqui também as buscas nas bases de conhecimento utilizando os termos de pesquisa foram realizadas tanto no título quanto no *abstract* dos trabalhos. Desta forma foi obtida uma significativa base de trabalhos. O próximo passo, após a identificação preliminar dos artigos de interesse, foi a eliminação dos títulos duplicados e também a exclusão dos artigos que não tratassem dos temas de interesse deste trabalho. A totalização dos artigos levantados por este método é apresentada no Quadro 3.2.

Quadro 3.2: Total de artigos levantados pela pesquisa realizada

Termo agrupador	Total de artigos
Collaborative software engineering	48
Collaborative metrics and measures	30
Software metrics	28
Software development teams	15
Open source collaborative development	15
Software quality and measures	14
Mining software repositories	11
Groupware	3
Cooperative work	3
TOTAL	167

As análises das seções seguintes foram realizadas por meio das referências obtidas utilizando o processo citado além de verificação de material relevante nas referências dos artigos estudados.

3.2. Métricas de Software

O estudo e aplicação de métricas de software remontam a meados da década de 1960, quando as métricas primitivas de linhas de código (LOC) eram utilizadas como a base para a medição da produtividade (LOC desenvolvida por mês) e qualidade (defeitos por KloC) no desenvolvimento de software. Conforme Fenton e Neil (Fenton e Neil, 2000), em 1971, Akiyama propôs o uso de métricas para predição de qualidade de software propondo um modelo baseado em regressão para o módulo de densidade de defeitos (número de defeitos por linha de código) onde a linha de código era utilizada como um indicador bruto da complexidade. Este foi um início na tentativa de extrair uma medida objetiva de qualidade de software por meio da análise de parâmetros observados no sistema.

Com a crescente adoção das linguagens de programação OO nos anos noventa, as pesquisas voltadas para métricas de software evoluíram significativamente. Chidamber e Kemerer argumentam que a orientação a objetos, tida como um avanço importante no desenvolvimento de software, e suas práticas, exigem medidas para adotar este paradigma de programação de forma adequada nas organizações. Este fato, juntamente com críticas às métricas já existentes resultou no desenvolvimento do conjunto de métricas de Chidamber e Kemerer (CK) (Chidamber e Kemerer, 1994). Na sequência vieram os estudos realizados por Henderson-Sellers (Henderson-Sellers, 1996), Harrison e colegas (Harrison *et al.*, 1997) e Martin (Martin, 2002). Assim um número considerável de

métricas foi proposto para medir vários atributos de sistemas de software OO tais como, métricas de tamanho, de herança, de complexidade, de coesão e acoplamento (Henderson-Sellers, 1996).

Estes estudos mostram o desenvolvimento de métricas de software que atuam de forma essencial no entendimento e controle do processo de engenharia de software, além de poderem ser utilizadas para analisar a evolução da qualidade dos sistemas (Lee *et al.*, 2007). E, segundo Mens e Demeyer (2001), as métricas possuem um número de características suficientemente interessantes capazes de comprovar e suportar a evolução do desenvolvimento do software. Para comprovar estes fatos, estudos têm sido conduzidos para investigar a relação entre as métricas OO e os atributos de qualidade do produto de software. E os resultados têm mostrado que as métricas de software se mostram úteis na avaliação e predição da qualidade do produto de software e suporte às demais atividades da engenharia de software (Basili, 1992). Entretanto, com a crescente complexidade e tamanho dos sistemas OO, os estudos realizados nesta revisão de literatura mostraram que seria mais apropriado utilizar estas várias métricas de forma simultânea, ao invés de independente.

No Quadro 3.3 tem-se um resumo dos artigos relacionados estudados e a contribuição com vistas ao objetivo desta pesquisa.

Quadro 3.3: Resumo dos artigos relacionados a Métricas de Software

Autor	Publicação	Objetivo	Contribuição
Mens e Demeyer, 2001	"*Future Trends in Software Evolution Metrics", 4th International Workshop on Principles of Software Evolution (IWPSE). 2001.*	Visão geral de como as métricas retornam subsídios para analisar a evolução do software. Analisa as métricas de software antes da ocorrência da evolução (preditiva) e depois da evolução (retrospectiva).	As métricas podem ser utilizadas para medir se a qualidade do software melhorou ou degradou entre duas versões.
Dagpinar e Jahnke, 2003	*"Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison", 10th Working Conference on Reverse Engineering (WCRE). 2003.*	Investiga o significado de várias métricas buscando predizer manutenibilidade de software.	Categorizou as métricas em 4 grupos: tamanho, herança, coesão e acoplamento.

(continua)

Autor	Publicação	Objetivo	Contribuição
Washizaki *et al.*, 2004	*"A Metrics Suite for Measuring Reusability of Software Components", 9th International Software Metrics Symposium. IEEE, 2004.*	Utiliza combinação de métricas para avaliar a reusabilidade.	Apesar de focarem nas métricas utilizadas na análise de teste caixa preta, mostram que WMC, DIT, CBO e LCOM podem ser utilizadas como medidas de reusabilidade das classes em OO.
Buntink e van Deursen, 2004	*"Predicting Class Testability using Object-Oriented Metrics", 4th IEEE International Workshop on Source Code Analysis and Manipulation, 2004.*	Utiliza combinação de métricas para avaliar a testabilidade.	Mostram a influência das métricas Linhas de Código por Classe - LOCC, Número de Campos - NOF, Número de Métodos - NOM, WMC, RFC (*response for class*), FOUT (*fan out*) no que tange a testabilidade.
Ambu *et al.*, 2006	*"Studying the evolution of quality metrics in an agile/distributed project", Extreme Programming and Agile Processes in Software Engineering, 2006.*	Seus estudos focaram na evolução de métricas de qualidade em projetos de metodologia ágil e distribuídos.	Coletaram métricas de processo e de produto (métricas de qualidade de CK). Investigaram como a distribuição das equipes de desenvolvimento impacta na qualidade dos códigos fonte.
Lee *et al.*, 2007	*"Metrics and evolution in open source software", 7th International Conference on Quality Software (QSIC), 2007.*	Visão geral da evolução das métricas de software em aplicações *open source*, buscando provar que as métricas de software podem ser utilizadas para predizer a qualidade ao longo da evolução do sistema.	O trabalho explora a evolução de sistemas *open source* em termos de tamanho, acoplamento e coesão e discute as mudanças da qualidade baseado nas leis de evolução de Lehman's (Lehman *et al.*, 1997).
Jemakovics *et al.*, 2007	*"Visual Identification of Software Evolution Patterns",9th International Workshop on Principles of Software Evolution (IWPSE) and 6th ESEC/FSE Joint Meeting, 2007*	Propuseram uma abordagem de identificação visual da evolução dos padrões de evolução do software relacionada a requisitos. Permite que os gerentes de projeto tenham a evolução do desenvolvimento do sistema sob controle.	Mostra a evolução do sistema junto à implementação de seus requisitos. Utilizaram as métricas de complexidade, acoplamento e as de coesão definidas por CK.
Misra e Akman, 2008	*"Weighted Class Complexity: a Measure of Complexity for Object Oriented System" Journal of Information Science and Engineering, 2008.*	Utiliza combinação de métricas para avaliar a complexidade.	Identificam que as métricas WMC, DIT, NOC e RFC têm condições de identificar índices que detectam o grau de complexidade do código analisado, no caso o grau de entendimento do código pelo desenvolvedor.

(continua)

Autor	Publicação	Objetivo	Contribuição
Kalliamvakou *et al.*, 2009	*"Measuring Developer Contribution from Software Repository Data", MCIS 4th Mediterranean Conference on Information Systems, 2009*	Utiliza a plataforma Alitheia e seus *plugins* para calcular as métricas básicas como linhas de código e participação na troca de mensagens.	Define conceito de contribuição e fator da contribuição utilizando métrica de linhas de código, dados que envolvam *bugs* e pesos.
Riaz *et al.*, 2009	*"A Systematic Review of Software Maintainability Prediction and Metrics" 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, 2009.*	Utiliza combinação de métricas para avaliar a Manutenibilidade.	As métricas de qualidade de software focadas no código-fonte relacionadas a tamanho, complexidade e acoplamento foram as que retornaram previsões bem-sucedidas em relação a manutenibilidade de software.
Xie *et al.*, 2009	*"Towards a better understanding of software evolution: An Empirical study on open source software", International Conference on Software Maintenance (ICSM), 2009.*	Analisa a evolução de 7 programas *open source*, investigando e validando algumas leis da evolução de Lehman (Lehman *et al.*, 1997).	Mostrou similaridades nos padrões de evolução dos programas estudados. Utilizou métricas de código-fonte e informações de projeto e defeitos para analisar o crescimento e alterações no software e avaliar a qualidade do software.
Yu *et al.*, 2011	*"Using Bug Reports as a Software Quality Measure", 16th International Conference on Information Quality (ICIQ), 2011.*	Estudam a possibilidade de utilizar o número de bugs para medir a qualidade do software.	Utilizam métodos estatísticos para analisar a correlação entre o número de bugs e as alterações no software.
Karus e Dumas, 2012	"*Code churn estimation using organizational and code metrics: an experimental comparison". Journal of Information* and *Software Technology, 2012*.	Utilizou medida de Churn.	Medida de Churn, que mede a variação, com base na soma do número de linhas de código adicionados, modificados e apagados, feita em um componente ao longo de um período, aplicada ao código-fonte.
Singh, Gagandeep, 2013	*"Metrics for Measuring the Quality of Object-Oriented Software. ACM SIGSOFT Software Engineering Notes, 2013.*	Apresenta nove métricas de qualidade de software OO e suas influências na qualidade do software.	Aplica as métricas de qualidade de software OO em uma aplicação JFreeChart e demonstra a validade da influência das variações nos valores das métricas na qualidade do software durante o ciclo de vida do software.

(continua)

Autor	Publicação	Objetivo	Contribuição
Drouin *et al.*, 2013	*"Analyzing software quality evolution using metrics: an empirical study on open source software". Journal of Software, 2013.*	Analisa de forma empírica a qualidade da evolução dos sistemas utilizando métricas.	Propõem a métrica Qi (*quality assurance indicator*) que captura de forma integrada atributos de OO e métricas de design OO, provando que a métrica Qi reflete a evolução da qualidade nos sistemas estudados.
Chauhan, R. Singh, R. Saraswat, A. Joya, A. H. Gunjan, V. K, 2014	*"Estimation of Software Quality using Object Oriented Design Metrics", International Journal of Innovative Research in Computer and Communication Engineering, 2014*	Estuda e Analisa diversas métricas de design OO e mostra a utilidade destas métricas para determinar a qualidade do software desenvolvido.	As métricas de design OO utilizadas são: Mood Metrics, CK Metrics, QMood Metrics.
Finlay *et al.*, 2014	*"Data stream mining for predicting software build outcomes using source code metrics". Journal of Information and Software Technology, 2014.*	Utiliza métricas de código-fonte e técnicas de mineração de dados para prever sucesso/fracasso na construção de software.	Apresenta os resultados de uma tentativa sistemática para prever o sucesso e/ou fracasso na construção de um produto de software utilizando métricas de código-fonte. Para atingir este objetivo, eles usaram uma técnica de mineração de fluxo de dados, o algoritmo Hoeffding Tree.
Foucault *et al.*, 2014	*"Code ownership in open-source software". EASE, 2014.*	Utiliza métricas de propriedade de Bird. O estudo replicou métricas de propriedade de Bird *et al.* em projetos de software livre.	Essas métricas são conhecidas por serem bons indicadores de qualidade e podem ser usadas para organizar equipes de desenvolvimento de software. Eles observaram que as métricas de código clássicas são melhores indicadores de qualidade.
Perkusich *et al.*, 2015	*"A Bayesian network approach to assist on the interpretation of software metrics". SAC 2015, 2015*	Utiliza Redes Bayesianas.	Propuseram um método para a construção de Redes Bayesianas para auxiliar na interpretação de métricas de software considerando a influência de fatores subjetivos.

(continua)

Autor	Publicação	Objetivo	Contribuição
Warnars, H. L. H. S. *et al*., 2017	*"Object oriented Metrics to measure the quality of software upon PHP source code with PHP_depend study case request online system application", International Conference on Applied Communication Technologies (ComCom), 2017*	Verifica a qualidade do código em relação a manutenibilidade, entendimento, reuso, entre outros, utilizando métricas de código-fonte.	Aplica as métricas de código-fonte em aplicação de PHP.
Vytovtov, Petr. Markov, Evgeny, 2017.	*"Source code quality classification based on software metrics". 20th Conference of Open Innovations Association (FRUCT), 2017*	Utiliza o compilador LLVM. As métricas utilizadas são as de complexidade de Halstead, complexidade ciclomática e métricas de código de baixo nível.	Propuseram uma biblioteca para o compilador LLVM que avalia a qualidade do código-fonte durante a compilação e o programador recebe informações sobre a qualidade do código-fonte e os valores das métricas de software são utilizados para avaliar a qualidade do código.
Behnamghader, P. Alfayez, R. Srisopha, K. Boehm, B. 2017	*"Towards better understanding of software quality evolution through commit-impact analysis", International Conference on Software Quality, Reliability and Security (QRS), 2017*	Analisa a qualidade do código-fonte a cada *commit* realizado identificando quanto as alterações realizadas no código impactam na qualidade do software.	Utiliza métricas de tamanho, complexidade e dados de *code smells* do SonarQube e dados de qualidade de código do PMD. Também verifica métricas voltadas a segurança, ao todo são 9 métricas.
Savchenko, D. Hynninen, T. Taipale, O. 2018	*"Code quality measurement: case study", International Convention on Information and Communication Tecnhology, Eletronics and Microeletronics (MIPRO), IEEE, 2018*	Projetaram e analisaram a arquitetura de uma coleção de métricas que verificam qualidade de projetos de software.	Analisa e visualiza somente o item de qualidade referente a manutenibilidade do projeto de software.

Os estudos analisados mostraram que as métricas de código-fonte têm a capacidade de responder sobre a existência ou não de variações na evolução do desenvolvimento da qualidade do produto de software do sistema. Os estudos também mostraram que as métricas de código-fonte conseguem responder sobre as variações nos níveis de manutenibilidade, usabilidade, testabilidade e entendimento do software (complexidade), porém são estudos isolados. De uma forma geral, os estudos encontrados não pretendem integrar estas respostas, aplicando-as no sentido de retornar informação para o desenvolvedor buscando que este entenda o que ocorreu com a qualidade do

produto de software. Esta pesquisa entende que, na medida em que o desenvolvedor conhecer a influência que a alteração que ele realizou no código-fonte teve na qualidade do produto de software, ele buscará melhorar o código em nível de qualidade de software e consequentemente melhorar o sistema como um todo.

O estudo de Kalliamvakou e colegas (Kalliamvakou *et al.*, 2009) é o que mais se aproxima deste trabalho, uma vez que utiliza o conceito de contribuição e análise de métricas de código-fonte com base em repositório de projetos. Porém, os autores deixam claro que não utilizam somente o código-fonte como base para o cálculo da contribuição, eles utilizam a medida de linhas de código, mas escalonam-na de acordo com o resultado da combinação de outras ações do desenvolvedor no projeto, como participação em fóruns e lista de e-mails, detecção e solução de defeitos, atualização de páginas de *wiki*, entre outros. Estas ações também recebem pesos, uma vez que os autores consideram que as ações não têm o mesmo grau de importância na evolução do projeto. Os autores utilizam a combinação do impacto positivo ou negativo do desenvolvedor nestas atividades para calcular o fator de contribuição que é acrescido à medida de linhas de código gerada pelo desenvolvedor. O estudo de Kalliamvakou e colegas (2009) utiliza o conceito de cooperação para medir a contribuição do desenvolvedor.

3.3. Medidas de Colaboração

A colaboração é fundamental para as organizações lidarem com os desafios dos ambientes modernos de negócio, que envolvem flexibilidade, adaptação e estar aberta à inovação (Chesbrough, 2006). Independentemente de todos os benefícios conhecidos, promover uma colaboração eficaz da equipe no desenvolvimento colaborativo de software continua a ser um desafio. As ferramentas tradicionais de software usualmente proveem características limitadas de suporte à colaboração. Este conjunto de ferramentas precisa de mecanismos que suportem a colaboração, a comunicação e a coordenação. Dentro da colaboração pode-se citar o compartilhamento de espaço de trabalho, repositórios de dados, suporte para *merging* e *differencing*, configuração, teste, projeto, gerenciamento de processos. Considerando a comunicação é possível citar e-mails, mensagens, anotações, vídeo/áudio. Na coordenação considera-se travamento, controle de versão, *hand-over*, auditoria. Diversos estudos de equipes, processos, ferramentas, projetos de mundo real têm mostrado a necessidade de se utilizar de forma apropriada processos, gerenciamento de

projetos, técnicas e seleção de ferramentas que permitam a colaboração efetiva e eficiente (Mistrik *et al.*, 2010).

Assim, o projeto colaborativo de desenvolvimento de software, abordado pela área de Engenharia de Software Colaborativa (*Collaborative Sofware Engineering* - CSE), é um tema para o qual vários pesquisadores têm contribuído visando seu aprimoramento. Buscando auxiliar as organizações a entender melhor o contexto de colaboração e na criação de equipes efetivas e complementares de desenvolvimento, além de processos, técnicas e ferramentas. Os pesquisadores ainda discutem quais as práticas, processos e ferramentas que são capazes de fomentar a colaboração e controlar a sua execução (Araujo e Borges, 2007) (Mistrik *et al.*, 2010).

Assim as plataformas colaborativas tomam forma em diversos trabalhos de pesquisadores que contribuem para o aprimoramento da colaboração no desenvolvimento de software. Whitehead (2007) observou que não há estudos que quantifiquem os benefícios do uso das ferramentas de colaboração e propõe que o desenvolvimento de melhores métodos para avaliar o impacto de ferramentas de colaboração iria impulsionar a investigação nesta área, aumentando a confiança em resultados positivos, e permitindo convencer as equipes a adotar novas tecnologias. Segundo Thompson e colegas (2009) a medição de colaboração ainda é um tema de investigação aberto na literatura. E Magdaleno e colegas (2015) aponta que ainda existe espaço de pesquisa para melhorar a estratégia de medição visando obter formas refinadas para determinar o potencial de colaboração no desenvolvimento de software.

O Quadro 3.4 apresenta os estudos realizados no que tange a medidas em ambientes de colaboração, voltados ou não para a Engenharia de Software.

Quadro 3.4: Medidas em ambientes de colaboração

Autor	Domínio	Ferramenta	Itens que mede	Técnica da medição	Invasiva (sim/não)
Johnson, P. *et al.*, 1997	*Software Development* - SD	Leap	Esforço, Defeitos, Tamanho	Hackystat	não
Robillard, P. e Robillard, M., 2000	*Software Development* - SD	logbook	Comunicação, Colaboração	Coleta manual de log diário em um logbook	sim
Van der Hoeck e Sarma, 2003	*Software Development* - SD *Awareness*	Palantir		Grau da alteração	não
Cook, C. e Churcher, N., 2005	*Collaborative Software Engineering* - CSE	Caise	Avaliação heurística Análise e visualização de log	Heurística	sim

(continua)

Autor	Domínio	Ferramenta	Itens que mede	Técnica da medição	Invasiva (sim/não)
Thompson, A.M. Perry, J.L. e Miller, T.K., 2007	Atividade colaborativa	Questionário	Governança, administração, Autonomia, Mutualidade, Veracidade	Structural model	sim
Schwind, M. e Wegmann, C., 2008	*Software Development* - SD	SVNMAT	Software, Comunicação, Versões	Medida de distância	não
Mohtashami, M. Marlowe, T.J. e Ku, C. S., 2011	*Collaborative Software Development* - CSD	Comparativo	Baseado no CMMI	Nivel de maturidade CMMI	sim
Claret, M. D., 2012	Processos de negócios	Desempenho	Dados de execução do processo Mídias sociais E-mails	ColabMM GQM (goals, questions and metrics)	sim
Wanderley, G. M. P. Abel, M. Barthès, J. Paraiso, E. C., 2016	*Collaborative Software Development* - CSD	ACE4SD	Métricas de código-fonte Sistemas multi-agentes Recomendação	Coleta de dados de código-fonte	não
Middleton *et al.*, 2018	*Collaborative Software Development* - CSD	Desempenho	Comentários em discussões (*issues)* *Pull requests* realizados	Coleta de dados de projetos do GitHub	não

Os estudos analisados em relação às medidas de colaboração mostraram o grau de dificuldade de encontrar esta medida em função de que esta atividade envolve coordenação, comunicação e cooperação, que são atividades cognitivas e de difícil medição. Os estudos também mostram que as medições existentes são em sua maioria voltadas para a atividade de comunicação explícita que disponibiliza indícios físicos para análise como, e-mails, mensagens, *chats* e anotações em reuniões. Para medir as demais atividades foi preciso o uso de questionários, observações, análise de *logs*, entre outros, o que tornou algumas verificações invasivas.

3.4. Considerações Finais

Buscou-se mostrar neste capítulo o levantamento, por meio de uma revisão sistemática de literatura, de diversas pesquisas referentes aos temas que abrangem o trabalho proposto. Estas pesquisas foram principalmente aquelas sobre os objetos de interesse para o trabalho aqui proposto, a saber: desenvolvimento colaborativo de software, métricas de software e medidas de colaboração. Estes temas foram pesquisados

e apresentados nos diversos tópicos que compõe este capítulo, sendo mostrados trabalhos atuais de cada área relacionada.

Considerando que os estudos realizados em relação à medição da colaboração mostraram a dificuldade em obter esta informação, optou-se neste trabalho por uma linha intermediária na qual é possível buscar dados concretos, como por exemplo, o código-fonte.

A partir deste contexto foi possível rastrear as “pegadas do programador” durante o desenvolvimento do software, coletando variações nas métricas de código-fonte capazes de serem aplicadas no cálculo do grau de contribuição do participante do desenvolvimento colaborativo de software no nível de qualidade do produto de software. Os procedimentos metodológicos para o cálculo do grau de contribuição do desenvolvedor, proposto neste trabalho, estão descritos no Capítulo 4.

Capítulo 4

Procedimentos metodológicos

Neste capítulo são apresentados a abordagem metodológica da pesquisa, sua estruturação, as ferramentas utilizadas na implementação do método, o processo de coleta de dados para a avaliação das métricas de qualidade e os experimentos realizados para validar os pesos e dados necessários para a proposta da equação do cálculo do grau de contribuição do desenvolvedor. Para tanto, são abordadas algumas referências para estabelecer os conceitos necessários para a condução da pesquisa e avaliação dos resultados.

4.1. Caracterização da Pesquisa

Segundo Marconi e Lakatos (Marconi e Lakatos, 2007), para que haja produção de conhecimento científico é necessária uma sistematização de conhecimento aliada a um conjunto de proposições sobre os fenômenos que se pretende estudar, buscando um objetivo limitado e possível de ser submetido à verificação. De acordo com Galliano (Galliano, 1979), "*o método científico é um instrumento utilizado pela ciência na sondagem da realidade, mas um instrumento formado por um conjunto de procedimentos, mediante os quais os problemas científicos são formulados e as hipóteses cientificas são examinadas*". O conceito de método científico se aplica a diversos ramos de estudo, mesmo os não considerado ciência, porém, é possível afirmar que não há ciência sem o emprego de métodos científicos.

O método quantitativo é utilizado no desenvolvimento de pesquisas descritivas na qual se procura descobrir e classificar a relação entre variáveis, assim como na investigação de causalidade entre os fenômenos: causa e efeito. Este método representa, em linhas gerais, uma forma de garantir a precisão dos resultados evitando distorções de análise e interpretação. O método quantitativo caracteriza-se pelo emprego da quantificação tanto nas modalidades de coleta de informações, quanto no tratamento delas, utilizando técnicas estatísticas, das mais simples, como frequência, percentual, média, às mais complexas, como coeficiente de correlação, análise de regressão, entre outras (Oliveira, 2001) e (Gil, 2002). Considerando o que foi descrito, em relação à abordagem, esta pesquisa pode ser caracterizada como uma **pesquisa quantitativa,** já que traduz em números as informações referentes à qualidade do código-fonte do desenvolvedor, permitindo classificá-los e analisá-los.

De acordo com Contandriopoulos (Contandriopoulos, 1994) é possível classificar as várias abordagens de uma pesquisa nas seguintes categorias estratégicas: a pesquisa experimental, a pesquisa sintética, a pesquisa de desenvolvimento e a pesquisa de simulação. Considerando de forma conceitual as categorias citadas, esta pesquisa pode estar classificada dentro de uma **pesquisa de desenvolvimento**, conforme (Contandriopoulos, 1994) comenta "*a pesquisa de desenvolvimento é a estratégia de pesquisa que visa, utilizando de maneira sistemática os conhecimentos existentes, elaborar uma nova intervenção ou melhorar consideravelmente uma intervenção existente, ou ainda, elaborar um instrumento, um dispositivo ou método de medição*". Já que o objetivo desta pesquisa é, a partir das métricas de código-fonte propor uma nova métrica, o grau de contribuição do desenvolvedor, buscando melhorar a qualidade de produto do software, além de buscar uma aplicação mais prática das métricas de código-fonte, que seja de interesse do desenvolvedor e do gerente de projetos.

Esta pesquisa também possui características exploratórias, pois pretende, além de proporcionar familiaridade com o problema, buscar suporte às questões de hipótese da pesquisa. Segundo (Gil, 2002) uma pesquisa exploratória "*tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses. Pode-se dizer que estas pesquisas têm como objetivo principal o aprimoramento de ideias ou a descoberta de intuições. Seu planejamento é, portanto, bastante flexível, de modo que possibilite a consideração dos mais variados aspectos relativos ao fato estudado*". Então, para que os objetivos desta pesquisa pudessem ser

alcançados foi realizada também uma **pesquisa exploratória**, por meio de uma revisão sistemática de literatura. Kitchenham (Kitchenham, 2007), considera a revisão sistemática de literatura um tipo de estudo secundário, que visa identificar, avaliar e interpretar todos os resultados relevantes a um determinado tópico de pesquisa, fenômeno de interesse ou questão de pesquisa. Os resultados obtidos por diversos estudos primários correlatos atuam como fonte de informação a ser investigada por estudos secundários. A precisão e a confiabilidade proporcionadas pelos estudos secundários contribuem para a melhoria e para o direcionamento de novos tópicos de pesquisa, a serem investigados por estudos primários.

Assim, a revisão sistemática de literatura foi aplicada neste trabalho, de forma a elucidar o campo de estudos, identificar lacunas para a contribuição científica e dar suporte à definição dos objetivos e hipóteses da pesquisa. De forma específica, visando proporcionar maior familiaridade e entendimento do estado da arte em relação ao comportamento dos desenvolvedores em ambientes colaborativos de desenvolvimento de software, permitindo a construção da hipótese do conceito de contribuição, e a partir daí a proposta do cálculo do grau de contribuição do desenvolvedor, de forma não invasiva, mediante o uso das métricas de código-fonte com vistas a tornar este conhecimento mais explícito.

4.2 Estruturação desta Pesquisa

Esta pesquisa combinou diversos procedimentos técnicos (Gil, 2002) envolvendo uma análise nos ambientes de SCV com o objetivo de pesquisar as atividades desempenhadas pelos desenvolvedores em ambientes colaborativos de desenvolvimento de software; uma revisão sistemática de literatura com o objetivo de pesquisar a influência das métricas de código-fonte na qualidade do produto de software; definição dos métodos de extração das métricas de código-fonte para análise e do método de cálculo do grau de contribuição do desenvolvedor; experimentos práticos buscando associar a hipótese de pesquisa proposta em bases de dados reais primeiro para justificá-la e depois para validá-la. Os dados utilizados nos experimentos foram obtidos a partir de repositório de dados de projetos de software reais.

Foi realizada uma revisão sistemática das métricas de código fonte disponíveis na literatura e aplicáveis no ambiente colaborativo de desenvolvimento de software. A partir da seleção das 20 métricas, conforme citado no Capítulo 2 - Seção 2.5, foi dada a

continuidade ao estudo das métricas verificando a influência dos itens calculados nestas métricas nos itens de qualidade do produto de software, no caso, complexidade, manutenibilidade, reusabilidade e testabilidade.

Com a revisão de literatura focada na busca dos itens de qualidade do produto de software, que são influenciados pelos valores das métricas, também foi possível identificar na literatura o grau de risco desta influência. Isto é, a partir das pesquisas foi possível indicar a partir do resultado das métricas o grau de risco de o código escrito/alterado pelo desenvolvedor influenciar de forma positiva, negativa ou não alterar o nível de complexidade, manutenibilidade, reusabilidade e testabilidade do código que foi realizado *commit* no projeto.

Os experimentos para o cálculo do grau de contribuição do desenvolvedor foram implementados na linguagem Java, utilizando a plataforma de desenvolvimento Eclipse. No primeiro experimento, para obter os dados necessários para a análise, optou-se por um repositório de projetos *open source*. Um repositório de projetos *open source* tem a característica de hospedar projetos desenvolvidos de forma colaborativa, além de manter as características de um SCV oferecendo os dados necessários para montar a base de dados necessária para esta pesquisa. Os dados coletados para a montagem da base de dados histórica para a validação do trabalho foram recuperados do repositório de projetos GitHub. Os experimentos foram compilados no ambiente de desenvolvimento Eclipse e para a recuperação das métricas dos códigos fonte foi utilizado o P*lugin* Metrics também disponível no Eclipse.

Baseado nos objetivos traçados para esta pesquisa a Figura 4.1 apresenta o fluxo do método da pesquisa.

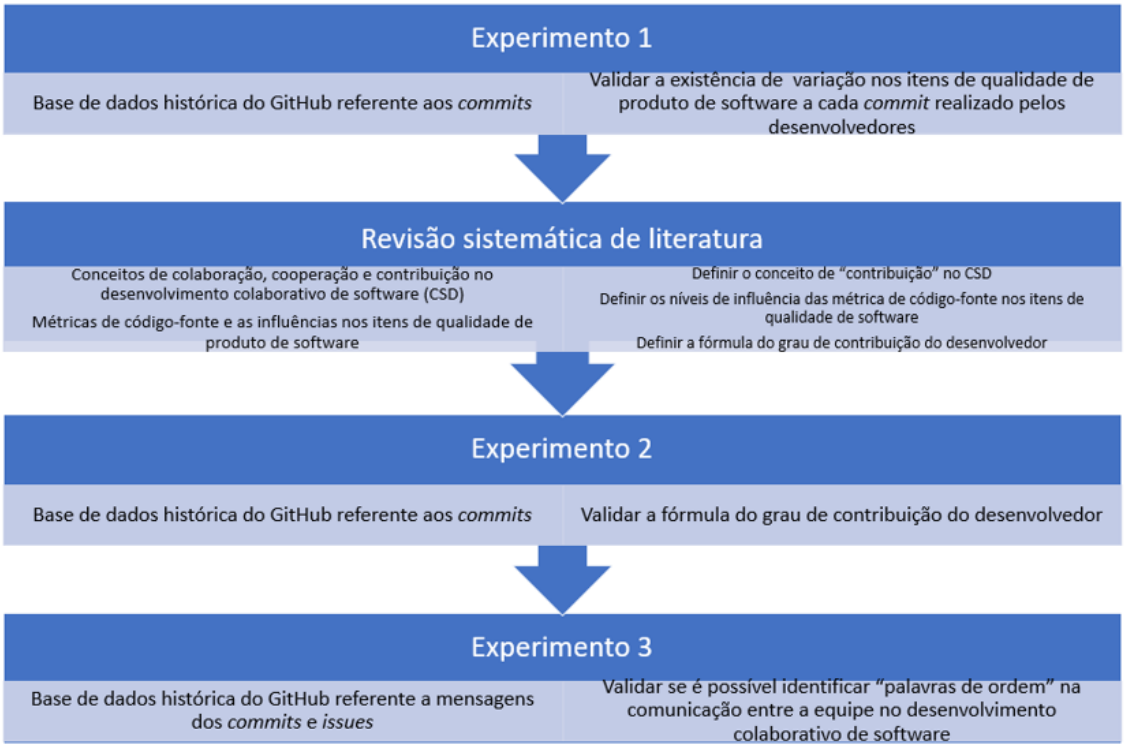


Figura 4.1: Fluxo do método desta pesquisa

4.2.1 Ferramentas Utilizadas

O GitHub (www.github.com) é um serviço de hospedagem distribuído desenvolvido em Ruby on Rails para projetos que utilizam o sistema de controle de versão. O GitHub é utilizado como um repositório *online* de códigos-fonte para projetos de código aberto. Neste ambiente é possível criar projetos, seguir outros desenvolvedores, baixar e/ou modificar projetos, receber atualizações de modificações de projetos, entre outros. Além de disponibilizar informações sobre os *commits* dos projetos que o utilizam, forma uma rede social que possibilita que outras pessoas acompanhem o desenvolvimento de um projeto, dispõe de recursos para visualizar gráficos de quantos *commits* um membro do projeto realizou ou está realizando, entre outras atividades. Este ambiente estimula os projetos *open source* e em função de seu lado social auxilia seus usuários a descobrirem novos projetos e a receber ajuda em projetos particulares, o que caracteriza a atividade cooperativa do desenvolvimento colaborativo de software.

O GitHub considera que os projetos de desenvolvimento de software muitas vezes têm uma comunidade em torno deles, formado por outros usuários desenvolvendo diferentes funções, que podem ser formais ou informais. As funções de usuário

consideradas na comunidade do GitHub são: o O*wner* que é o usuário ou organização que criou o projeto e tem o projeto em sua conta; os M*antainers and Collaborators* que são os usuários que trabalham no projeto e determinam a direção do projeto, muitas vezes o proprietário e o mantenedor são os mesmos, e têm acesso de escrita ao repositório; os *Contributors* que é todo aquele que teve um *commit* inserido em um projeto e; os C*ommunity Members* caracterizados pelos usuários que utilizam com frequência a aplicação e se preocupam com o projeto, eles são ativos em discussões visando novas atividades e solicitação de *requests*. Em função de estar inserido no contexto de desenvolvimento colaborativos de software, da credibilidade do uso do ambiente, do tamanho do repositório e da disponibilidade de acesso, o GitHub foi escolhido para a aplicação de coleta de dados históricos para validação desta pesquisa.

O ambiente Eclipse (https://eclipse.org/) foi escolhido em função de possuir uma arquitetura flexível que possibilita a sua extensão com o uso de *plugins*, permitindo adicionar funcionalidades ao ambiente. Por meio destes *plugins*, várias funcionalidades são providas, tais como, edição de código fonte, depuração e ferramentas necessárias ao desenvolvimento de software. No caso desta pesquisa foi utilizado o *Plugin* Metrics. O Plugin Metrics (http://sourceforge.net/projects/metrics/files/) é um projeto sob licença EPL (*Eclipse Public License* - http://opensource.org/licenses/EPL-1.0) para o IDE Eclipse. Este *Plugin* calcula métricas de código-fonte Java, fornecendo o valor da medida, a média e o desvio padrão por recursos (projeto/classe/método).

A outra ferramenta utilizada, neste caso para analisar as mensagens dos *commits* e *issues* postados pelo desenvolvedor e gerente de projetos no projeto de software, foi o SentiStrenght. O SentiStrenght é uma ferramenta de análise de sentimentos que avalia o sentimento de uma sentença observando as palavras que compõem a frase. Isto é, atribui pontuações positivas/negativas às palavras e depois resume essas pontuações para obter um sentimento geral para a frase. Esta ferramenta pode ter o dicionário léxico personalizado, podendo retornar pontuação positiva ou negativa em termos de domínio específico (www.sentistrength.wlv.ac.uk). Em função desta particularidade a ferramenta SentiStrenght foi utilizada para análise das mensagens de *commit* e *issue* postadas pelos desenvolvedores e gerentes de projeto nos projetos de software.

4.2.2 Escopo da Pesquisa

Para realizar os experimentos da pesquisa o método proposto foi aplicado primeiramente em bases de dados históricas de projetos *open source* hospedados no

repositório de projetos GitHub. Foram utilizados projetos desenvolvidos em Java com equipes de desenvolvedores entre 6 a 8 integrantes e/ou com um número considerável de *commits*. O número de desenvolvedores no projeto foi importante para ter um efetivo desenvolvimento colaborativo de software e o número de *commits* para que fosse possível identificar influências dos *commits* na qualidade do produto de software que estava sendo desenvolvido. Os primeiros projetos selecionados também tiveram a premissa de estarem finalizados.

O segundo experimento realizado também utilizou a base de dados histórica de projetos do GitHub, calculou o grau de contribuição do desenvolvedor, que indica se sua atuação em forma de *commit* do código-fonte influenciou de forma positiva, negativa ou não alterou o nível de qualidade do produto do software do projeto. Este experimento também mostra os resultados dos valores das métricas em formato de *dashboard* e retorna ao desenvolvedor "dicas" de como melhorar o código baseado nos resultados das métricas de código-fonte do *commit* realizado. Desta forma, o desenvolvedor tem a possibilidade de melhorar os níveis de qualidade do produto do software e terá um novo cálculo do seu grau de contribuição.

A proposta dos experimentos é ser não-invasiva, assim o desenvolvedor programa o código como está acostumado a fazer em seu ambiente de produção sem precisar preencher formulários e/ou se portar de maneira diferente do seu usual. Somente quando for realizado o *commit* no projeto é que o experimento informa ao desenvolvedor o grau de influência de seu trabalho no nível de qualidade do produto do software.

É necessário que o desenvolvimento colaborativo seja realizado em uma plataforma de controle de versões, um SCV (Sistema de Controle de Versão), utilizando o ambiente de desenvolvimento Eclipse com o *Plugin* Metrics para o retorno das métricas de código-fonte.

4.2.3 Agrupamento das Métricas e as Influências

Conforme apresentado no Capítulo 2, a partir das variações nas métricas de qualidade do produto de software detectadas no código-fonte realizado pelo contribuinte (Tabela 2.3) e agrupamento conforme os índices de qualidade (Tabela 2.4), foi possível indicar quais quesitos da qualidade do produto de software foram violados - manutenibilidade, testabilidade, reusabilidade ou entendimento do código (complexidade) – e em qual grau (Tabela 2.5). O compilado destes valores norteou o cálculo do grau de contribuição do desenvolvedor, conforme será visto no Capítulo 5.

Também a partir destas pesquisas foi possível mapear recomendações para que o desenvolvedor tenha condições de melhorar a qualidade do seu código e, por consequência, a do projeto como um todo (conforme será apresentado no Capítulo 5). O agrupamento das métricas, suas influências e as recomendações para o desenvolvedor demandaram considerável pesquisa para identificá-los, e foram validados nos cenários de base de dados históricos.

4.3. Experimentos Realizados

Nesta seção são apresentados os principais experimentos realizados para validar o cálculo do grau de contribuição do desenvolvedor. Primeiramente, foi realizado um experimento utilizando os dados de uma base de dados histórica e, em um segundo momento, foi realizado o experimento para o cálculo do grau de contribuição do desenvolvedor. Por último, foi realizada a análise das mensagens dos *commits* e *issues* deixados pelos desenvolvedores e gerentes de projeto dos projetos verificados, em relação à existência de traços de influência do gerente de projetos na alteração realizada pelo desenvolvedor. A Figura 4.2 apresenta o fluxo dos experimentos desta pesquisa.

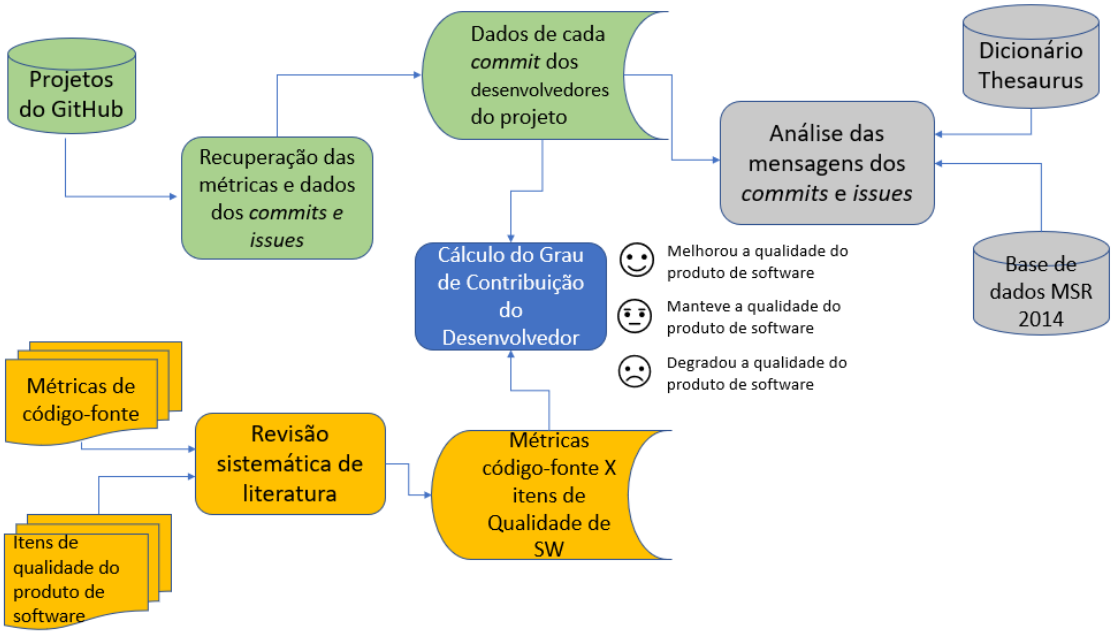


Figura 4.2: Fluxo dos experimentos desta pesquisa

Fonte: a própria autora.

4.3.1. Experimento 1 para Montar a Base de Dados Histórica

A proposta deste experimento buscou validar a primeira hipótese secundária desta pesquisa: e*xiste "contribuição" entre os integrantes de uma equipe de desenvolvimento colaborativo de software*. Trabalhando com projetos desenvolvidos de forma colaborativa, este experimento capturou as métricas do código a cada *commit* realizado nos projetos selecionados, armazenando os dados em relação à linha do tempo dos *commits* realizados pelos desenvolvedores envolvidos no projeto.

Para cada *commit* do contribuinte foram computadas as 20 métricas de qualidade, conforme as Tabelas 2.3 e 2.4, as quais foram coletadas dos projetos selecionados do GitHub considerando o cenário da pesquisa. Os resultados dos valores das métricas foram analisados e foram plotados gráficos monstrando os resultados do experimento. A Figura 4.3 apresenta o fluxo deste experimento.

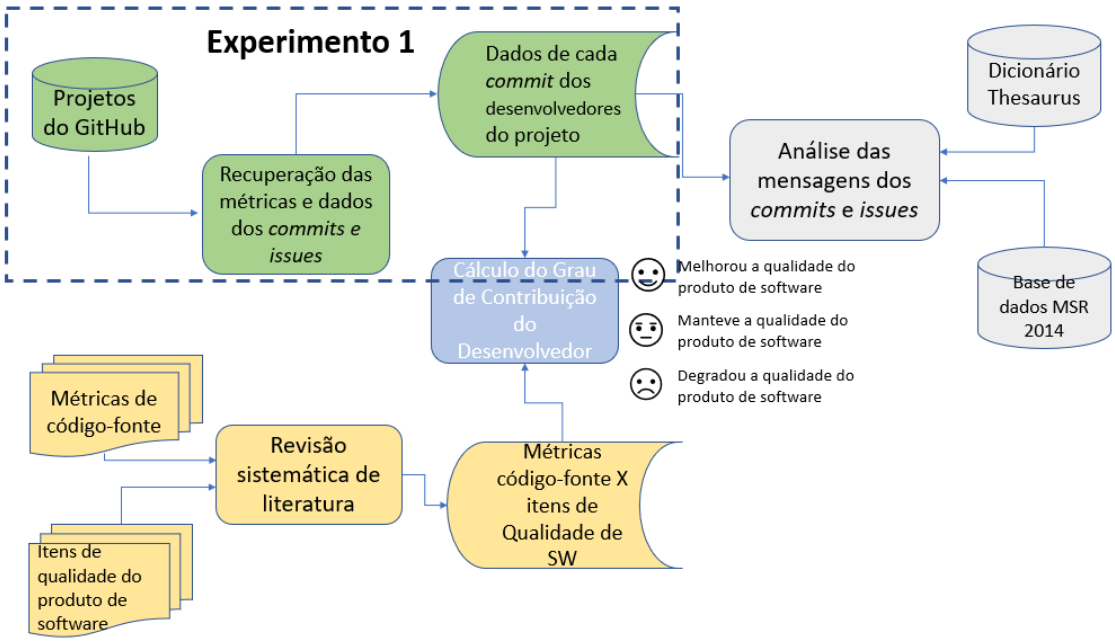


Figura 4.3: Fluxo do experimento 1 para montar a base de dados histórica

Fonte: a própria autora.

4.3.2. Experimento 2 para Calcular o Grau de Contribuição do Desenvolvedor

Este experimento teve como objetivo responder a segunda hipótese secundária desta pesquisa: *dada uma coleção de métricas de qualidade de software, estas são*

capazes de expressar o grau de contribuição individual dos participantes do projeto de desenvolvimento colaborativo de software de qualidade no nível de código-fonte.

A revisão sistemática de literatura descrita no Capítulo 2 forneceu subsídios para o cálculo do grau de contribuição do desenvolvedor, conforme será apresentado no Capítulo 5. De posse dos dados coletados conforme o experimento 1, referentes a cada *commit* no projeto, foi feito o cálculo do grau de contribuição do desenvolvedor, conforme a equação que está proposta no Capítulo 5. Para tanto, o experimento recupera os valores das métricas do código-fonte a cada *commit* realizado pelo desenvolvedor. Aplica a equação para o cálculo do grau de contribuição do desenvolvedor, e retorna, em forma de *dashboard*, os dados referentes aos itens de qualidade do produto de software, além do valor referente ao grau de contribuição do desenvolvedor após o *commit* realizado. A Figura 4.4 apresenta o fluxo deste experimento.

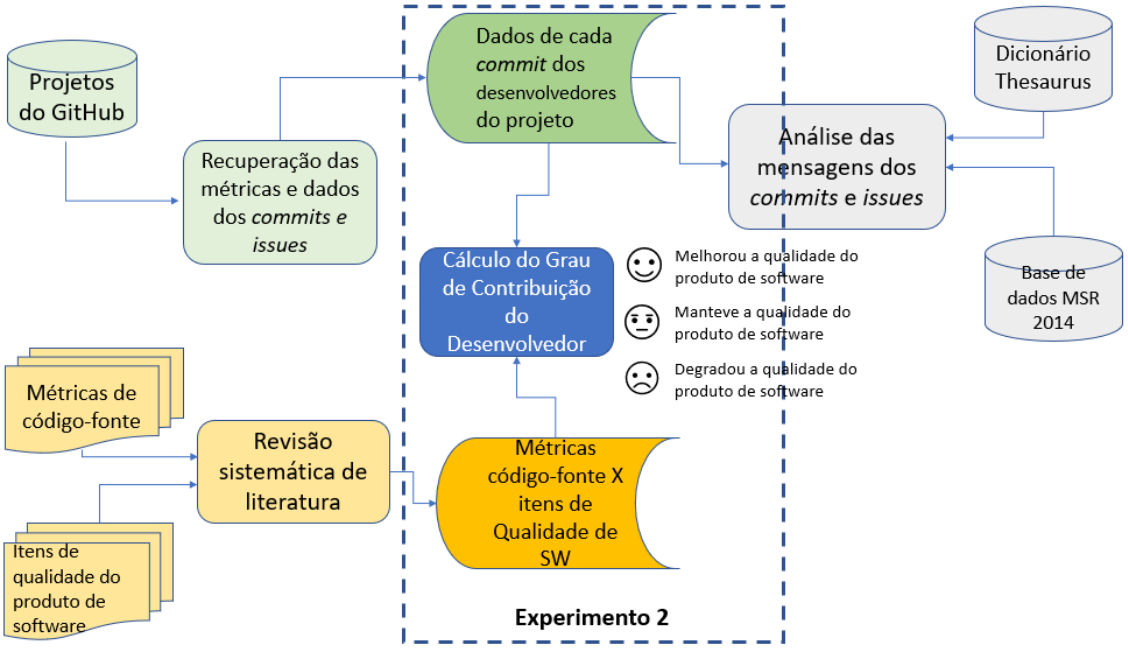


Figura 4.4: Fluxo do experimento 2 para calcular o Grau de Contribuição do Desenvolvedor

Fonte: a própria autora

4.3.3. Experimento 3 para Verificar as Mensagens dos *Commits* e *Issues*

Outro item no qual percebeu-se uma necessidade de análise específica está relacionado à alteração realizada pelo desenvolvedor no código: esta seguiu uma

solicitação do gerente do projeto ou até mesmo decorre de um requisito de projeto. Nestes casos o desenvolvedor pode ter realizado a alteração de forma a degradar o nível de qualidade do produto de software, não por sua vontade ou desconhecimento, mas por influência externa.

O estudo aplicado neste experimento foi buscar nas mensagens enviadas pelo desenvolvedor a cada *commit* realizado e os *issues* do projeto palavras que pudessem identificar se houve ou não alguma ordem ou determinação do gerente do projeto definindo o formato da alteração que deveria ser realizada pelo desenvolvedor. Para o desenvolvimento deste experimento foi utilizado o conceito de recuperação da informação e de forma específica a análise de sentimentos, descritos no Capítulo 2.

Segue na Figura 4.5 o fluxo aplicado no experimento para verificar as mensagens dos *commits* e *issues* realizados pelos desenvolvedores e gerentes de projetos.

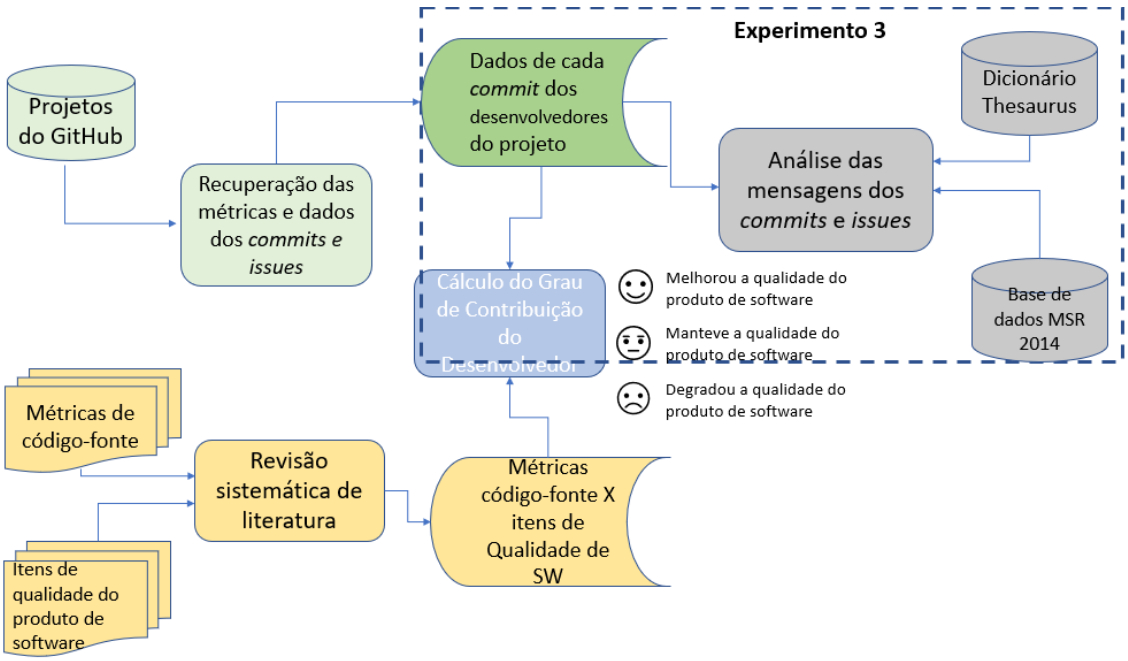


Figura 4.5: Fluxo do experimento 3 para verificar as mensagens dos *commits* e *issues*
Fonte: a própria autora.

Este experimento teve como objetivo responder a hipótese da pesquisa: *os comentários e mensagens deixados pelos coordenadores e desenvolvedores nas mensagens dos* commits *e* issues *do projeto podem mostrar influência na contribuição individual do desenvolvedor.*

4.4. Cenários de Análise

Foram traçados alguns cenários de análise baseados nos dados disponibilizados e analisados pelos experimentos realizados em dados históricos. Estes cenários de análise foram validados pelos experimentos realizados e nos dados constantes na Equação 5.1 (Capítulo 5) do cálculo do grau de contribuição individual do desenvolvedor (GC[d]).

Seguem os cenários:

1. Linhas de código alteradas e valores das métricas mantidos – cenário bom e atendido pela equação 5.1 do GC[d] proposta.
2. Linhas de código alteradas e valores das métricas melhorados – cenário ótimo e atendido pela equação 5.1 do GC[d] proposta.
3. Linhas de código alteradas e valores das métricas degradadas – cenário ruim e exige retorno do experimento informando em qual artefato o desenvolvedor pode melhorar seu desempenho. Cenário atendido pela equação 5.1 do GC[d] proposta.
4. Inserção de *commit* que tem métricas que melhoraram e ao mesmo tempo métricas que pioraram. Cenário atendido pela equação 5.1 do GC[d] proposta.

4.5. Considerações Finais

Neste capítulo foram elucidados os procedimentos metodológicos aplicados na pesquisa, foram citadas as ferramentas utilizadas na implementação do método, foram relatados também os experimentos realizados a fim de comprovar as hipóteses do trabalho, assim como os cenários de análise. Os cenários apresentados são resultantes de análise dos gráficos plotados a partir dos experimentos com dados históricos e retorno de revisões referentes à submissão de artigos desta pesquisa para eventos e congressos internacionais.

No Capítulo 5 é apresentada a equação do cálculo do grau de contribuição do desenvolvedor (GC[d]) e os detalhes dos experimentos realizados para coleta dos dados e aplicação da equação conforme o método de pesquisa proposto.

Capítulo 5

Contribuição Individual de um Desenvolvedor em um Projeto Colaborativo de Desenvolvimento de Software

Marlowe e colegas (Marlowe *et al.*, 2011) citam que um dos temas nos processos técnicos e de negócios é a avaliação e controle de qualidade. Para estes pesquisadores existem três objetivos que motivam esta avaliação: compreender melhor o processo para promover mais pesquisas na área; otimizar o processo colaborativo; e avaliar a colaboração em um determinado nicho. Quando existem indicadores de qualidade e métricas para a avaliação estas podem ser utilizadas para decidir quanto à continuidade do processo colaborativo, avaliar o estado atual do processo colaborativo no nicho específico, e avaliar o sucesso do processo colaborativo.

Neste capítulo é apresentado o conceito de Contribuição, um conceito concreto proposto e que permitirá o rastreamento, coleta e análise dos dados necessários para o cálculo do grau de contribuição do desenvolvedor. Também será descrita uma proposta de organização das métricas de qualidade de software de modo a facilitar o mapeamento da influência destas métricas nos índices de qualidade de software.

O conceito de Contribuição e o arranjo das métricas de qualidade do código são os fundamentos para os pressupostos do método do cálculo de grau de contribuição do desenvolvedor, também apresentado aqui.

5.1. Contribuição no Desenvolvimento Colaborativo de Software

No Capítulo 2 foram definidos os conceitos de Colaboração e Cooperação no âmbito do desenvolvimento colaborativo de software. A colaboração pode ser considerada uma relação mutuamente benéfica e bem definida (análise, arquitetura, programação, entre outras) celebrada entre duas ou mais organizações (indivíduos ou equipes, no caso) com a finalidade de alcançar resultados que, em conjunto, serão mais propensos a serem alcançados do que sozinhos. E a cooperação pode ser considerada como uma divisão do trabalho entre os participantes na qual cada pessoa é responsável por uma parte da solução do problema (Roschelle e Teaslley, 1995). É possível verificar que a colaboração envolve a cooperação. Porém, nenhum dos dois termos é capaz de expressar a atividade que se objetiva neste trabalho: calcular o grau de contribuição do participante no desenvolvimento colaborativo de software.

Figura 5.1: Representação do Modelo 3Cs dos Sistemas Colaborativos utilizando Diagrama de Venn

A Figura 5.1 apresenta o Modelo 3Cs que ilustra o conceito de sistemas colaborativos, onde o CSCW e o CSE estão inseridos. Transportando este conceito para o desenvolvimento colaborativo de software é possível visualizar nos ambientes de desenvolvimento colaborativo de software, as atividades de coordenação - caracterizada pelo gerenciamento de pessoas, atividades e recursos (Ellis *et al.*, 1991) - uma vez que cada projeto tem um responsável pela divulgação dos requisitos do projeto, cronograma e inserção dos códigos desenvolvidos pelos integrantes da equipe no projeto; a comunicação – caracterizada pela troca de mensagens, pela argumentação e pela

negociação entre pessoas (Ellis *et al.*, 1991) - entre os integrantes do grupo pode ser realizada através de *chats*, troca de mensagens e e-mails; e a cooperação – caracterizada pela atuação conjunta no espaço compartilhado para a produção de objetos ou informações (Ellis *et al.*, 1991) - que pode ser caracterizada pelas sugestões de metodologias de desenvolvimento dos requisitos, propostas de novos requisitos pelos integrantes da equipe de desenvolvimento, além da documentação do projeto para identificar/justificar ações do coordenador/desenvolvedor do projeto. Porém, no decorrer dos estudos realizados, percebeu-se tarefas distintas que envolvem a atividade de cooperação no desenvolvimento colaborativo de software.

A cooperação envolve sugestões, auxílio técnico e comentários em geral, mas também envolve a contribuição específica, isto é, efetivamente é realizada uma atividade pelo desenvolvedor que altera a situação do sistema. Evidenciando-se esta atividade no contexto da cooperação e identificando-se que esta atividade pode ser rastreada e, por conseguinte medida e validada, tornou-se possível isolar esta atividade cooperativa.

Segundo o Dicionário Michaelis (Moderno Dicionário Português Michaelis) cooperação e contribuição são definidas conforme o Quadro 5.1:

Quadro 5.1: Definição de cooperar/cooperação e contribuir/contribuição

Cooperar	**Contribuir**
(*lat cooperari*) *vti* 1 Agir ou trabalhar junto com outro ou outros para um fim comum; colaborar: *Lá todos cooperam na assistência social.* 2 Agir conjuntamente para produzir um efeito; contribuir: *As nossas boas obras cooperam para a nossa futura felicidade*.	(*lat contribuere*) *vti* 1 Concorrer para uma despesa comum: *O Estado contribuiu com dois terços do capital. Contribuíam todos para um benefício comum.* *vti* 2 Cooperar: *Era seu propósito contribuir com uma tese para a reforma da Constituição. Contribuí para isso mais do que todos. vti* 3 Ter parte em um resultado: *Essa gramática contribuirá a aplainar a aspereza do ensino do vernáculo entre nós. Essas forças vêm contribuir para a vitória de uma velha aspiração. vti* e *vint* 4 Pagar contribuição, ter parte em uma despesa comum: *Alguns contribuíram com dinheiro. Outros não puderam contribuir. vtd* 5 Pagar como contribuinte.
Cooperação	**Contribuição**
sf (*cooperar+ção*) 1 Ato de cooperar; colaboração; prestação de auxílio para um fim comum; solidariedade. 2 Organização da vida econômica, baseada no princípio de "fazer retornar o lucro" ao consumidor.	*sf* (*lat contributione*) 1 Ato de contribuir. 2 Quantia com que cada um entra para uma despesa comum. 3 Imposto, tributo. 4 Subsídio de caráter moral, social, literário ou científico, para alguma obra útil. *C. de guerra:* indenização que um país vencido ou cidade conquistada paga ao vencedor. *C. de registro:* a que se paga pela transmissão de

	bens ou de direitos; direitos de transmissão. *C. de sangue:* a obrigação do serviço militar. *C. direta:* imposto lançado nominalmente sobre o contribuinte. *C. indireta:* imposto lançado sobre os objetos de consumo. *C. industrial:* imposto lançado pelos poderes públicos sobre qualquer indústria, profissão, arte ou ofício.

Um sistema de controle de versão (SCV) tem por finalidade gerenciar versões de um documento possibilitando organização, controle e acompanhamento das diferentes versões dos projetos desenvolvidos de forma colaborativa. Durante o desenvolvimento colaborativo do software os desenvolvedores recebem instrução do coordenador do projeto por meio de atividades postadas no SCV. Nestas atividades estão descritos os requisitos e as alterações solicitadas. Estas atividades podem estar endereçadas ou não para um determinado desenvolvedor.

De posse destas informações o desenvolvedor verifica o código-fonte e busca o código que precisa ser alterado/incrementado. Havendo dúvidas o desenvolvedor pode utilizar o ambiente de mensagens associado ao requisito para questionar o coordenador, ou para postar dados referente a documentação do requisito. Após a programação e teste da atividade solicitada o desenvolvedor disponibiliza o *commit* atualizado e solicita ao coordenador do projeto que realize o *merge* do *commit*. O coordenador por sua vez, realiza os devidos testes na nova versão do *commit* e, estando tudo correto, realiza o *merge* do novo *commit* no projeto. Este processo ocorre dentro da ferramenta de SCV.

5.1.1 Papéis dos Participantes de um SCV e a Contribuição

Tomando como base SCV conhecidos no mercado, como o GitHub, o GitLab e o BitBucket, é possível visualizar atividades que o usuário tem permissão para realizar dentro do projeto do qual faz parte. Estas atividades estão relacionadas ao seu nível de permissão de atuação nas atividades do projeto. Na sequência estão relatadas estas atividades permitindo que posteriormente se identifique e defina a atividade de *contribuição*.

O GitHub possui dois tipos de usuários: o proprietário (*owner*) de repositório e os colaboradores (*collaborators*). As ações que o colaborador pode realizar no repositório são as de baixar modificações do repositório (*pull*), enviar notificações para o repositório (*push*), aplicar *labels* e *milestones*, controlar notificações de problemas (*issues*), gerenciar comentários em *commits*, *pull requests* e *issues*, aceitar ou recusar *merge* de requisições

de modificações (*pull request*), enviar requisições de modificações para repositórios originados de *forks* (*pull request*), criar e editar páginas de Wiki, criar e editar *releases*, desligar-se de um repositório colaborativo. O *owner*, além das ações do colaborador, citadas anteriormente, também pode excluir repositórios, alterar a visibilidade do projeto (público ou privado) e convidar novos colaboradores.

O GitLab é um gerenciador de repositório de software também baseado em Git e de software livre, cuja primeira versão foi disponibilizada em 2011. O diferencial do GitLab é que permite que os desenvolvedores armazenem o código em seus próprios servidores, ao invés de servidores de terceiros. Com a aquisição do GitHub pela Microsoft em junho/2018, diversos usuários migraram seus projetos para outros sistemas, como o GitLab.

Em relação aos privilégios de uso, o GitLab permite cinco tipos diferentes: convidado (*guest*), relator (*reporter*), desenvolvedor (*developer*), líder (*master*) e o proprietário (*owner*) do repositório. As ações que o *guest* pode realizar são as de criar notificações de problemas (*issues*), deixar comentários, visualizar a lista de *builds*, ver os *logs* de *builds*, procurar e baixar artefatos de *builds*. *O*s *reporter*s, além das ações do *guest*, podem também baixar códigos do projeto (*pull*), baixar o projeto, criar fragmentos de código (*snippets*), gerenciar as notificações de problemas (*issues*), gerenciar *labels*, visualizar estado dos *commits*, visualizar registros e o ambiente do projeto.

Os *developers*, além das ações do *guest* e do *reporte*r podem também gerenciar e aceitar requisições de *merge*, criar novas requisições de *merge*, criar novas *branches*, enviar modificações para *branches* não protegidas (*push* e *force-push*), remover *branches* não protegidas, adicionar mensagens de monitoramento em *commits* (*tags*), escrever na página da Wiki, cancelar e testar novamente *builds* de código, criar ou atualizar estados de *commits*, atualizar e remover registros e criar novos ambientes.

O nível *master*, além de todas as ações já citadas, tem permissão também para criar novos conjuntos de tarefas (*milestones*), adicionar novos membros no time, enviar modificações para *branches* protegidas (*push*), habilitar ou desabilitar envio de modificações de desenvolvedores à *branches* protegidas, editar ou remover notificações de *commits* (*tags*), editar informações do projeto, adicionar chaves de *deploy* no projeto, gerenciar páginas do projeto, gerenciar domínios das páginas e certificados, excluir ambientes. Por último, o *owner* do projeto, além de todas as ações dos níveis anteriores, pode também alterar a visibilidade do projeto, transferir projeto para outro *namespace*,

excluir o projeto, excluir páginas do projeto, remover *branches* protegidas e enviar modificações forçadamente para *branches* protegidas (*force-push*).

Comparando com o GitHub, os níveis de *guest* e *reporter* têm permissão de visualização e comentários, o *developer* recebe tarefas e realiza as atividades no código-fonte do projeto, o *master* tem o papel de gerente do projeto e o *owner* tem o mesmo papel do *owner* no GitHub.

O Bitbucket é um sistema de repositório e controle de versão baseado na web para código-fonte e desenvolvimento de projetos que utilizam o modelo Mercurial ou Git de sistemas de controle de versão, oferecendo a plataforma para uso comercial ou livre. O Bitbucket trabalha com ferramentas de desenvolvimento (Jira software) e conceitos no modelo ágil, desta forma as atividades gerenciadas pelo aplicativo utilizam estes recursos. No ambiente colaborativo do Bitbucket as permissões do projeto é que caracterizam as ações que os usuários deste ambiente colaborativo de desenvolvimento de software podem realizar no projeto. Um esquema de permissão é um conjunto de atribuições de usuário/grupo/função às permissões que o usuário tem de realizar determinadas atividades no projeto.

Basicamente, no ambiente colaborativo do BitBucket os usuários podem trabalhar em nível de projeto, pendências, acompanhamento das atividades do projeto e acompanhamento do cronograma das pendências. Desta forma as permissões estão disponíveis no nível de projeto, utilizando o conceito de *sprints*; pendências, que são as atividades que o usuário desenvolvedor realiza no código-fonte do projeto; gerenciar usuários votantes e seguidores do projeto; gerenciar os comentários relativos às pendências do projeto; gerenciar os arquivos anexados às pendências; gerenciar o rastreamento de tempo das pendências. No contexto de projeto o usuário tem permissão para realizar as atividades de administrar um projeto utilizando o ambiente Jira, procurar um projeto, gerenciar *sprints* dentro do projeto e no ambiente Jira, visualizar ferramentas de desenvolvimento do ambiente Jira, visualizar o fluxo de trabalho do projeto por meio de suas pendências.

Dentro do contexto de pendências os usuários com tal permissão realizam as atividades de atribuir pendências aos usuários com permissão de terem pendências atribuídas a ele, fechar pendências, criar pendências, excluir pendências, editar pendências, vincular pendências, modificar o relator de uma pendência, mover pendências entre projetos ou fluxo de trabalho, resolver pendências, agendar pendências,

configurar segurança das pendências e alterar *status* de uma pendência. As permissões de votantes e seguidores abrangem o gerenciamento da lista de usuários seguidores de uma pendência e visualizar a lista de usuários votantes e seguidores de uma pendência. Em relação aos comentários, as permissões envolvem adicionar, editar e/ou excluir todos os comentários de uma pendência e/ou somente os de seu usuário específico. As permissões de anexo envolvem criar, editar e/ou excluir todos os anexos de uma pendência e/ou somente os de seu usuário específico. A partir das permissões o gerente de projeto pode determinar as ações de cada usuário envolvido no ambiente colaborativo de desenvolvimento de software de forma individual, sem rótulos pré-definidos como ocorre no GitHub e GitLab, por exemplo.

O usuário em um projeto de desenvolvimento colaborativo de software pode realizar um "*fork*", que é quando ele faz uma cópia de um grupo de arquivos de código-fonte de um repositório, para realizar (quando o usuário for colaborador) ou propor alterações, analisar o código ou fazer reuso dele em outros projetos. O usuário pode ser um "collaborator" (ou colaborador) que tem permissão para realizar diversas ações nos arquivos dentro de um repositório, incluindo editá-lo. E também pode ser somente um "watcher" que caracteriza o usuário que solicita permissão para ser notificado das atividades que estão sendo realizadas no repositório. Este papel permite que o usuário acompanhe as atividades que estão sendo realizadas para verificar as alterações e até sugerir alterações e melhorias no projeto, mas sem atuar sobre elas.

Os SCV podem ser públicos ou privados, mesmo os públicos permitem que se criem contas privadas. As contas privadas abrangem um maior controle de permissões de ações e divisão de atividades dentro dos projetos, além de juntar múltiplos projetos e colaboradores ou times em um só espaço de colaboração.

A partir dos estudos das atividades realizadas pelos usuários colaboradores nos ambientes de desenvolvimento colaborativo de software conseguiu-se perceber uma atividade, bem específica, realizada pelos integrantes da equipe de desenvolvimento que é efetivamente a atividade que envolve a construção do código-fonte da aplicação. No caso do ambiente GitHub, esta atividade é desenvolvida pelo usuário colaborador, no ambiente GitLab o usuário desenvolvedor é quem tem esta prerrogativa e, em um nível mais moderado, o reporter. Já no ambiente do Bitbucket os usuários com permissão para resolver pendência, alterar status da pendência e fechar pendência são os que realizam esta atividade de desenvolver o código-fonte da aplicação. As demais atividades

realizadas por usuários nos ambientes colaborativos de desenvolvimento de software podem ser caracterizadas como de gerenciamento, observação e controle, que dentro do contexto dos Modelo 3Cs envolvem a coordenação, no caso do gerenciamento, e a cooperação no caso da observação e controle. Sendo que a comunicação ocorre dentro da ferramenta de SCV nas mensagens trocadas entre os usuários do ambiente colaborativo. Quando o desenvolvedor escreve o código e disponibiliza para o coordenador inserir no projeto (*commit*) pode-se dizer que está havendo uma contribuição. Isto é, o indivíduo não está somente cooperando com sugestões para o bom desempenho do projeto, mas efetivamente contribuindo, através de seu esforço cognitivo, inteligência e trabalho mensurável, com o produto de software que está sendo desenvolvido.

A partir destas definições, e considerando o conceito de sistemas colaborativos no ambiente de desenvolvimento colaborativo de software, percebe-se que a cooperação abrange uma contribuição e pode-se dizer que a contribuição indica uma tarefa efetiva de ação e alteração do resultado que se deseja atingir de forma cooperativa. Partindo-se deste princípio torna-se possível identificar as ações realizadas pelo programador no código-fonte, que acabam gerando as "pegadas do programador" e com isto é possível identificá-las, medi-las, validá-las e compará-las para identificar o grau de contribuição do programador no sistema que está sendo desenvolvido de forma colaborativa. Assim pode-se considerar que a cooperação tem relação direta com a contribuição, como ilustra a Figura 5.2.

Figura 5.2: Representação do Modelo 3Cs incluindo a contribuição utilizando o Diagrama de Venn

O conceito de contribuição no desenvolvimento de software está normalmente relacionado à produtividade, participação ou atividade, estes conceitos utilizam a definição de unidade de trabalho no contexto de engenharia de software. No início, os estudos consideravam unidade de trabalho as linhas de código (LOC) (Sackmann *et al.*, 1968), que continua sendo considerada uma medida de tamanho de software. As análises de ponto por função, por exemplo, consideram a métrica de LOC como unidade de trabalho para estimativa de software. Com o advento da programação orientada a objetos, classes, métodos e componentes também são considerados unidades de trabalho em engenharia de software (Card e Scalzo, 1999).

A partir das revisões sistemáticas realizadas para esta pesquisa foi possível perceber que alguns pesquisadores utilizam o termo "contribuição" de uma forma diferenciada dos termos colaboração e cooperação, conforme estamos aplicando nesta pesquisa. Kalliamvakou e colegas (Kalliamvakou *et al.*, 2009) verificaram que os desenvolvedores de software além de participar da escrita do código-fonte, participam também de discussões, submetem comentários e ideias na lista de e-mails e solucionam defeitos de software. Desta forma, (Kalliamvakou *et al.*, 2009) consideram em seus estudos que os repositórios de projetos, arquivos de lista de e-mails e base de dados de defeitos, entre outros, são a base de dados para discutir participação e atividade em um projeto de software. Comparando essa abordagem com os conceitos das atividades do desenvolvimento colaborativo de software, pode-se dizer que ela abrange o conceito de cooperação.

Middleton e colegas (Middleton *et al.*, 2018) procuram entender de que forma os desenvolvedores e colaboradores são aceitos nos times de desenvolvimento colaborativo de software OSS, e constataram que na maioria das vezes esta escolha é baseada na quantidade e nível de "contribuição" destes desenvolvedores voluntários. Estes dados ficam armazenados no histórico do desenvolvedor dentro da plataforma de desenvolvimento colaborativo de software e as atividades que Middleton e colegas consideraram na pesquisam foram discussões textuais em *issues* e *pull requests* e contribuição nos códigos-fonte dos projetos. Mostrando que pesquisadores efetivamente utilizam os termos contribuição, colaboração e cooperação de formas distintas na classificação das atividades realizadas no desenvolvimento colaborativo de software.

Também é possível visualizar as diferenças entre colaboração, cooperação (definidos no Capítulo 2) e contribuição nas definições do Cambridge Dictionary

(*Cambridge English Dictionary*): *"Collaboration – the situation of two or more people working together to create or achieve the same thing; Cooperation – the activity of working together with someone or doing what they ask you to do; Contribution – something that you contribute or do to help produce or achieve something together with other people, or to help make something successful."*

No contexto desta pesquisa, que está voltada para o cálculo da contribuição do desenvolvedor considerando os índices de qualidade do produto de software, utilizou-se o conceito de contribuição voltado para unidade de trabalho, considerando para isto as linhas de código, classes, métodos e componentes produzidos pelo desenvolvedor. Conceituamos então:

A contribuição é a participação mensurável de um contribuinte no processo de desenvolvimento em um projeto colaborativo de desenvolvimento de software.

O contribuinte é o participante cuja contribuição pode ser mensurada.

A partir deste conceito é possível propor um método para calcular a contribuição do participante no desenvolvimento colaborativo de software. Uma vez que é possível rastrear, calcular e analisar tais unidades de trabalho e inferir qual influência esta unidade de trabalho terá nas métricas de qualidade do software produzido/alterado pelo desenvolvedor. Na sequência, é apresentado estudo das métricas de qualidade de software voltado para o cálculo do grau de contribuição do desenvolvedor.

5.2. Cálculo do Grau de Contribuição do Desenvolvedor

Observando-se que os colaboradores em um projeto de software, desenvolvido em um ambiente colaborativo, atuam de diversas formas - um é o gerente daquele projeto, alguns colaboram com sugestões de alterações e novas funcionalidades e outros efetivamente alteram/constroem o código-fonte do projeto - percebeu-se a diferença nas atividades de colaboração e contribuição. Este fato gerou a busca por uma forma de identificar esta contribuição individual do desenvolvedor no desenvolvimento colaborativo de software.

Porém, com o foco de se trabalhar de forma não-invasiva, o desenvolvedor deve realizar suas atividades sem interferências, de forma natural. As pesquisas então se

voltaram para o código-fonte e às informações possíveis de serem retiradas do código-fonte que permitissem identificar a contribuição do desenvolvedor. A partir de experimento realizado foi identificada a possibilidade de recuperar a variação das métricas a cada *commit* realizado pelo desenvolvedor. Assim, as métricas de código-fonte foram selecionadas para buscar interpretar a contribuição do desenvolvedor. A partir desta seleção os estudos caminharam para a influência dos valores destas métricas nos itens de qualidade do produto de software. Buscou-se então uma forma de calcular o quanto o desenvolvedor contribuiu com o projeto, não só realizando a alteração solicitada, mas de forma que esta alteração tenha sido realizada dentro de um nível considerado razoável de qualidade do produto de software. Foi necessário buscar na literatura quais eram os níveis considerados razoáveis de qualidade do produto de software. Os estudos indicaram que as métricas de código-fonte se relacionavam com um ou outro item de qualidade do produto de software, como complexidade, reusabilidade, testabilidade e manutenibilidade, mas de forma individualizada.

A partir desta constatação, foram realizadas análises e agrupamentos para relacionar as métricas de código-fonte com os itens de qualidade do produto de software, resultando nas tabelas apresentadas na Seção 2.5 do Capítulo 2. Estas combinações foram testadas e aplicadas culminando na equação para o cálculo do grau de contribuição do desenvolvedor em um ambiente colaborativo de desenvolvimento de software.

O cálculo utilizado para definir o grau de contribuição do desenvolvedor envolveu a variação dos valores das métricas de qualidade calculados a cada *commit* realizado pelo contribuinte agrupada de acordo com a influência da métrica de qualidade do código-fonte no item de qualidade do produto de software (Tabela 2.4). Foi tomado como base de cálculo o compilado das faixas de influência da métrica na qualidade do código-fonte (Tabela 2.5) sobreposto a variação dos valores das métricas de qualidade do código-fonte.

Os valores das métricas retornadas a cada *commit* realizado pelo desenvolvedor são comparados com a tabela de grau de risco desta alteração influenciar na qualidade do produto de software (Tabela 2.5). Baseado neste grau de risco foram atribuídos pesos relacionados ao risco de influência da métrica na qualidade do produto de software. Estes pesos variam de 1 até 4, dependendo do número de níveis validados para cada métrica. Por exemplo, no caso do Acoplamento Aferente (Ca), temos três níveis de risco de influência bom, regular e ruim. Os pesos ficam distribuídos de forma que o item de maior risco tenha um peso maior. Indicando que, como existe um alto risco desta alteração

influenciar de forma negativa na qualidade do produto de software, o peso deste valor na fórmula é maior. Neste exemplo, o valor de Ca <= 7 terá peso 1 no cálculo do grau de contribuição do desenvolvedor, não interferindo no seu valor; Ca entre 7 e 39 terá peso 2; e Ca > 39 terá peso 3 no cálculo do grau de contribuição do desenvolvedor, retornando valor que identifica o grau de risco da programação realizada pelo desenvolvedor no código-fonte em relação à qualidade do produto do software. Segue na Tabela 5.1 o compilado dos pesos aplicados no cálculo do grau de contribuição do desenvolvedor.

Tabela 5.1: Pesos por grau de risco das métricas de código-fonte

Métrica de código-fonte	Grau do Risco (níveis)	Pesos por grau de risco (níveis)
Complexidade Ciclomática de McCabe - CCM ou VG	Baixo – 1<= VG <= 10 Moderado – 11 <= VG <= 20 Alto – 21 >= VG <= 50 Muito Alto – VG > 50	Baixo – 1 Moderado – 2 Alto – 3 Muito Alto - 4
Métodos Ponderados por Classe-WMC	Baixo – 1 <= WMC <= 20 Moderado – 21 <= WMC <= 100 Alto – WMC > 100	Baixo – 1 Moderado – 2 Alto – 3
Profundidade da Árvore de Herança - DIT	Ausente – DIT = 0 Desejável – 1 <= DIT <= 5 Profundo – DIT > 5	Ausente – 1 Desejável – 2 Profundo – 3
Acoplamento entre Classes - CBO	Fraco – 00 <= CBO <= 05 Forte – CBO > 05	Fraco – 1 Forte – 2
Resposta para uma Classe - RFC	Bom – 0 <= RFC >= 50 Ruim – RFC > 50	Bom – 1 Ruim – 2
Número de métodos - NOM	Desejável – 1 <= NOM <= 20 Aceitável – 21 <= NOM <= 40 Alto – NOM > 40	Desejável – 1 Aceitável – 2 Alto – 3
Número de filhos - NSC	Bom – NSC <=1 Regular – 1 < NSC <=3 Ruim – NSC > 3	Bom – 1 Regular – 2 Ruim – 3
Falta de coesão em Métodos - LCOM*	Bom – 0 <= LCOM* <= 0,5 Regular – 0,5 < LCOM * <= 1 Ruim – LCOM * > 1	Bom – 1 Regular – 2 Ruim – 3
Número de linhas de código do método - MLOC	Bom – MLOC <=10 Regular – 10 < MLOC <= 30 Ruim – MLOC > 30	Bom – 1 Regular – 2 Ruim – 3
Profundidade de blocos aninhados – NBD	Bom – NBD <=1 Regular – 1 < NBD <=3 Ruim – NBD > 3	Bom – 1 Regular – 2 Ruim – 3

(continua)

Métrica de código-fonte	Grau do Risco (níveis)	Pesos por grau de risco (níveis)
Número de métodos sobrescritos - NORM	Bom – NORM <=2 Regular – 2 < NORM <=4 Ruim – NORM > 4	Bom – 1 Regular – 2 Ruim – 3
Índice de especialização - SIX	Bom – SIX <= 0,019 Regular – 0,019 < SIX <= 1,333 Ruim – SIX > 1,333	Bom – 1 Regular – 2 Ruim – 3
Número de campos - NOF	Bom – NOF <=3 Regular – 3 < NOF <=8 Ruim – NOF > 8	Bom – 1 Regular – 2 Ruim – 3
Número de métodos estáticos – NSM	Bom – NSM <=1 Regular – 1 < NSM <= 3 Ruim – NSM > 3	Bom – 1 Regular – 2 Ruim – 3
Número de parâmetros - PAR	Bom – PAR <=2 Regular – 2 < PAR <= 4 Ruim – PAR > 4	Bom – 1 Regular – 2 Ruim – 3
Número de classes - NOC	Bom – NOC <= 7 Regular – 7 < NOC <= 28 Ruim – NOC > 28	Bom – 1 Regular – 2 Ruim – 3
Acoplamento aferente - Ca	Bom – Ca <=7 Regular – 7 < Ca <= 39 Ruim – Ca >39	Bom – 1 Regular – 2 Ruim – 3
Acoplamento eferente - Ce	Bom – Ce <=6 Regular – 6 < Ce <= 16 Ruim – Ce > 16	Bom – 1 Regular – 2 Ruim – 3
Instabilidade – I ou RMI	Bom/estável – $I = 0$ Ruim/instável – $I = 1$	Bom/estável – 1 Ruim/instável – 2
Abstração – A ou RMA	Bom/abstrata – A = 1 Ruim/concreta – A = 0	Bom/abstrata – 1 Ruim/concreta – 2

Durante a pesquisa foi possível verificar que alguns desenvolvedores contribuem com mais *commits* do que outros em um mesmo projeto. Desta forma, a possibilidade de influência dos *commits* de um desenvolvedor tende a ser consideravelmente maior do que a de outros, dependendo do número de *commits* que ele realiza no projeto. Para corrigir esta distorção o grau de contribuição do desenvolvedor foi normalizado, dividindo-se o valor calculado no grau de contribuição do desenvolvedor pelo número de *commits* que o desenvolvedor realizou no projeto.

Mantendo-se o resultado do cálculo, uma variação negativa do valor das métricas indicaria um bom grau de contribuição (o desenvolvedor melhorou as métricas de qualidade) e, ao contrário, uma variação positiva indicaria um grau de contribuição ruim. Assim, multiplicando-se o resultado final por -1, uma variação positiva indica um bom grau de contribuição e uma variação negativa um grau de contribuição ruim. Por esta

razão, o valor final do cálculo do grau de contribuição é multiplicado por -1, facilitando a interpretação do resultado.

Desta forma, o cálculo envolve um somatório de termos que são obtidos das métricas agrupadas pela influência no código-fonte, multiplicado pelo peso. Segue a equação proposta para o cálculo do grau de contribuição do desenvolvedor (Equação 5.1):

$$GC[d] = ((\Sigma_{i=1 \text{ até } n} \Delta \text{ da medida da métrica de manutenibilidade} * \text{peso do risco} + \Sigma_{i \text{ de 1 até } n} \Delta \text{ da medida da métrica de testabilidade} * \text{peso do risco} + \Sigma_{i \text{ de 1 até } n} \Delta \text{ da medida da métrica de reusabilidade} * \text{peso do risco} + \Sigma_{i \text{ de 1 até } n} \Delta \text{ da medida da métrica de complexidade} * \text{peso do risco}) / \text{número de } \textit{commits}) * -1 \quad (5.1)$$

Onde:

GC = grau de contribuição

d = desenvolvedor/contribuinte

n = número de métricas que influenciam cada índice de qualidade (manutenibilidade, testabilidade, reusabilidade, complexidade)

peso do risco = valores definidos com base na Tabela 5.1

número de *commits* = número de *commits* realizados pelo desenvolvedor no projeto

Como resultado do cálculo do grau de contribuição do desenvolvedor considera-se um grau de contribuição bom ou razoável um valor de GC[d] igual a 0. Um GC[d] = 0 indica que o desenvolvedor realizou a alteração necessária no código-fonte e manteve o nível de qualidade do produto de software. Um GC[d] > 0, ou seja, um GC[d] positivo é considerado um grau de contribuição ótimo, já que neste caso o desenvolvedor realizou a alteração necessária no código-fonte e melhorou o nível de qualidade do produto de software. Por outro lado, um GC[d] < 0, ou GC[d] negativo, indica que a alteração realizada pelo desenvolvedor, de forma geral, degradou o nível de qualidade do produto de software. Neste caso, o grau de contribuição do desenvolvedor é considerado ruim. O Quadro 5.2 resume os níveis de contribuição a partir do GC[d] calculado.

Quadro 5.2: Grau de contribuição a partir do GC[d] calculado

Resultado do GC[d]	Nível de contribuição	Consideração
GC[d] < 0	Ruim	Diminuiu o nível de qualidade do produto de software
GC[d] =0	Bom	Manteve o nível de qualidade do produto de software
GC[d] > 0	Ótimo	Aumentou o nível de qualidade do produto de software

O valor do grau de contribuição calculado é apresentado ao desenvolvedor, que também conta com informação referente à métrica e ao artefato onde ocorreu a variação, possibilitando que ele reveja a alteração/implementação realizada e melhore o código onde houve decréscimo da qualidade. Com base nos estudos realizados e no retorno das informações referente às métricas de código-fonte, foi possível montar uma tabela de dicas para o desenvolvedor melhorar o código e onde (método/classe/pacote) esta alteração pode ser realizada. Segue na Tabela 5.2 as recomendações enviadas para o desenvolvedor a partir dos retornos das métricas de código-fonte caso o índice da métrica não seja o desejável/bom/ausente/baixo.

Tabela 5.2: Recomendações retornadas para o desenvolvedor baseado no retorno não desejável/bom/ausente/baixo das métricas de código-fonte

Artefato	Métrica de código-fonte	Recomendação
Método	Complexidade Ciclomática de McCabe - CCM ou VG	O <metodo:classe:pacote> tem muitos pontos de decisão/seleção (if-then-else). Resultando em alta complexidade. Tente dividir o problema que o método resolve em problemas menores. Crie outros métodos e chame-os dentro do original.
	Número de parâmetros - PAR	O <metodo:classe:pacote> tem muitos parâmetros. Significa que ele é muito especializado e pode ser difícil de ser reutilizado. Reduza o número de parâmetros, tente passar como parâmetro uma estruturada específica ou um objeto.
	Profundidade de blocos aninhados – NBD	O <metodo:classe:pacote> tem estruturas muito profundas. Aparentemente existem muitas estruturas dentro de outras, como loops, if-then-else, e ainda outras estruturas if-then-else dentro destas, e assim por diante. Isto significa que esta parte do código está muito especializada e pode ser difícil de ser reutilizada. Tente dividir o problema que o método resolve em problemas menores. Crie outros métodos e chame-os dentro do original.
	Número de linhas de código do método - MLOC	O <método:classe:pacote> tem muitas linhas de código. Métodos assim tornam-se complexos e difíceis de serem entendidos. Se possível, reduza o número de linhas do método.

(continua)

Artefato	Métrica de código-fonte	Recomendação
Classe	Métodos Ponderados por Classe - WMC	Os métodos da <classe:pacote:projeto> aparentemente está muito complexo. Isto pode tornar esta classe difícil de se manter e reutilizar. Para cada método desta classe, tente criar outros métodos e chamá-los dentro do original.
	Profundidade da Árvore de Herança - DIT	A <classe:pacote:projeto> está muito profundo na arvore de heranças das classes. Significa que ela está herdando muitos métodos e atributos. Isto pode deixar muito complexo para uma manutenção futura. Tente reduzir a profundidade de herança nas classes.
	Número de filhos - NSC	A <classe:pacote:projeto> tem muitos filhos (subclasses). Quando uma classe tem muitos filhos pode ser bom para o reuso do código, mas muitos filhos podem indicar problemas de abstração nesta classe base. É melhor reduzir o número de filhos nesta classe.
	Número de métodos sobrescritos - NORM	A <classe:pacote:projeto> contém muitos métodos sobrescritos da classe pai. Muitos métodos redefinidos implicam em diferenças muito grandes entre a classe pai e a herança pode não fazer sentido. Tente criar novos métodos dentro da classe, ao invés de sobrescrever muitos métodos da classe pai.
	Falta de coesão em Métodos - LCOM*	A <classe:pacote:projeto> tem problemas de coesão. As classes com problemas de coesão executam mais itens do que seus próprios objetivos, isto é, fazem outras tarefas. Falta de coesão implica em classes que provavelmente deveriam ter sido separadas em duas ou mais subclasses.
	Número de campos - NOF	A <classe:pacote:projeto> possui diversos atributos. Classes com este tipo de problema são difíceis de serem entendidas e mantidas. Tente reduzir o número de atributos desta classe.
	Número de métodos - NOM	A <classe:pacote:projeto> possui muitos métodos. Isto pode indicar que a classe tem mais de um propósito além do seu próprio objetivo. Procure reduzir o número de métodos da classe, tornando-a mais reutilizável.
	Respostas para uma classe - RFC	A <classe:pacote:projeto> possui muitos métodos invocados em resposta a uma mensagem recebida por um objeto desta classe. Isto indica uma classe complexa, que a torna mais difícil de ser entendida, mantida e testada.
	Número de métodos estáticos – NSM	A <classe:pacote:projeto> possui diversos métodos estáticos. Isto pode indicar que o código tende a ser mais do paradigma estruturado e não OO. Tente reduzir o número de métodos estáticos desta classe.
	Acoplamento entre Classes - CBO	A <classe:pacote:projeto> apresenta um forte acoplamento, indicando que ela depende de muitas outras classes. Um forte acoplamento entre classes torna o código difícil de manter, modificar e reutilizar. Verifique a possibilidade de diminuir este acoplamento.

(continua)

Artefato	Métrica de código-fonte	Recomendação
	Índice de especialização - SIX	A <classe:pacote:projeto> tem profundidade de herança e sobrescreve muitos métodos da classe pai. Isto indica que o código é complexo e tende a ser difícil de manter. Pode ser um problema de abstração, as subclasses de uma herança estão sobrescrevendo diversos métodos da superclasse. Crie novos métodos dentro da classe, ao invés de sobrescrever diversos métodos da classe pai.
Pacote	Número de classes - NOC	O <pacote:projeto> possui um alto número de classes o que pode trazer problemas de entendimento, manutenibilidade e reusabilidade deste pacote/projeto.
	Acoplamento aferente - Ca	O <pacote:projeto> parece conter muitas classes que estão sendo utilizadas por diversas classes que pertencem a outros pacotes. Estas classes podem ser muito dependentes deste pacote. Isto aumenta o nível de acoplamento do código, dificultando seu reúso e manutenção.
	Acoplamento eferente - Ce	Muitas classes deste <pacote:projeto> aparentam dependerem de classes de outros pacotes. Isto aumenta o acoplamento do código, dificultando o reuso, manutenção e possíveis modificações futuras. É importante verificar a necessidade desta dependência.
	Instabilidade – I ou RMI	A instabilidade do <pacote:projeto> está alta. Isto significa que esse pacote é muito dependente de outros pacotes. As alterações realizadas nos outros pacotes podem afetar este. Tente reduzir a dependência deste pacote em relação aos outros pacotes.
	Abstração – A ou RMA	O número de classes abstratas (ou interfaces) neste <pacote:projeto> está baixo. Isto significa que é um pacote/projeto concreto e não contém classes abstratas suficientes. Podendo ser difícil de ampliá-lo, reusá-lo ou mantê-lo. Seria melhor aumentar o número de classes abstratas neste pacote/projeto.

O resultado desta tabela é aplicado como recomendação para o desenvolvedor melhorar os índices de qualidade do código-fonte do projeto e podem ser acatados ou não pelo desenvolvedor. Se o desenvolvedor acatar a recomendação e realizar a alteração no código-fonte, um novo cálculo do grau de contribuição é realizado a partir da alteração, possibilitando que ele melhore seu grau e aprenda com a ferramenta, proposta nesta pesquisa.

5.3. Considerações Finais

Neste capítulo foi apresentado o conceito de contribuição que permeia o objetivo do trabalho. Também foi apresentado o resultado da análise realizada a partir da revisão

sistemática de literatura referente às métricas de qualidade de código-fonte e suas influências nos índices de qualidade do produto de software referentes à manutenibilidade, reusabilidade, testabilidade e entendimento do código (complexidade), conforme revisão de literatura apresentada na Seção 2.5 do Capítulo 2. Estas métricas foram agrupadas para melhor qualificar sua influência e foram alocados pesos também baseados na influência do resultado da métrica na qualidade do produto de software. Estas análises e agrupamentos levaram a equação para o cálculo do grau de contribuição do desenvolvedor.

No próximo capítulo são apresentados os resultados dos experimentos realizados conforme o método de pesquisa proposto no Capítulo 4 e baseado nos estudos apresentados aqui. Também são realizadas discussões em relação aos resultados encontrados.

Capítulo 6

Resultados Experimentais e Discussões

Defende-se nesta tese a hipótese de que os valores das métricas obtidos após cada *commit* podem indicar mudanças nos índices de qualidade de software do código, e que esta situação traz subsídios para se calcular o grau de contribuição individual do desenvolvedor no desenvolvimento colaborativo de software. A partir do método proposto e dos dados coletados, conforme citado no Capítulo 4, gráficos foram plotados para estudar casos em que os *commits* aumentaram, diminuíram ou não influenciaram a manutenibilidade, testabilidade, reusabilidade e entendimento do código. Estes gráficos são analisados neste capítulo. Considerando como referência a revisão de literatura citada no Capítulos 2 (Seção 2.5), foi proposta a fórmula para o cálculo do grau de contribuição do desenvolvedor citada no Capítulo 5 e, os resultados destes experimentos são apontados neste capítulo. Também o experimento, que procura buscar analisar se houve comentários do gerente de projetos, é comentado neste capítulo.

6.1. Experimento 1

O primeiro experimento foi realizado com o objetivo de verificar e mapear a variação das métricas de código-fonte entre os *commits* realizados pelos desenvolvedores durante o desenvolvimento colaborativo do projeto de software.

6.1.1 Coleta de Dados

Este experimento foi realizado utilizando a base de dados composta por códigos-fontes disponíveis no GitHub. Foram escolhidos dois projetos, a partir de um conjunto de premissas: projetos finalizados implementados em Java e cujos códigos-fontes foram

possíveis de serem reconstruídos integralmente a partir da base histórica disponibilizada no repositório de projetos. A Tabela 6.1 apresenta as características destes projetos.

Tabela 6.1: Base de dados dos experimentos

Projeto	CSSEmbed	Elastic/Search
# de *commits*	32	18
# de classes	8	5
# de linhas de código	1516	277
# de contribuintes	6	5
Duração do projeto (em meses)	37	25
Link do projeto	https://github.com/nzakas/cssembed/ Acesso em Julho/2015	https://github.com/elastic/elasticsearch-transport-wares/ Acesso em Dezembro/2015

Para criar a base de dados histórica, visando o cálculo do grau de contribuição do desenvolvedor considerou-se que os projetos de software desenvolvidos de forma colaborativa em um ambiente SCV (*P*) contém, cada um, *m commits* (cada *commit* desenvolvido por um contribuinte), logo um projeto (*P*) pode ter *k* contribuintes realizando *commits* no mesmo código (Equação 6.1), seguindo a linha de tempo do desenvolvimento do projeto.

$$\forall m \in \mathrm{P}: \exists k \text{ que desenvolveu } m \tag{6.1}$$

Desta forma, para a coleta de dados, foi necessário armazenar os dados do *commit* que são basicamente: número sequencial do *commit* na linha do tempo do projeto, responsável pelo *commit* e para cada uma das 20 métricas capturadas (conforme Tabela 2.3): valor da métrica, artefato influenciado pela métrica (classe, método ou pacote) e o delta de variação em relação ao *commit* imediatamente anterior. A partir da coleta dos dados referente a cada *commit* realizado no projeto, foi calculado o delta da variação entre o *commit* N e o *commit* N-1 de cada métrica coletada. O valor deste delta é armazenado junto aos dados de cada *commit* realizado no projeto, e para cada métrica calculada.

A partir da seleção dos projetos, foi aplicado o Algoritmo 6.1, considerando os pressupostos também citados no Capítulo 5 em cada um destes projetos. Segue abaixo o Algoritmo 6.1 de coleta de dados da base histórica utilizado neste experimento.

```
procedure applyMetricsToProjects;
artifact = {class | method | package};
for project 1 to n
   commit = 1
   while commit <= m do
       Recover the commit source code
       Rebuilt the commit source code
       Compute the commit source code metrics
       Save the commit source code metrics
   end-while
   Combine all n projects metrics keeping the m commits time line by
   metrics
   for metric 1 to 20
       for each artifact do
            for commit 1 to m
                 diff = abs (commit metric value – (commit -1) metric
                        value)
                 if diff <> 0 then
                      for the whole artifact metric do
                          Save the metric, the artifact, the commit
                          sequence, the commit contributor and the
                          metric value
                      end-for
                 end-if
             end-for
       end-for
   end-for
end-for
end-procedure
```

Algoritmo 6.1: Experimento de coleta de dados da base histórica
Fonte: a própria autora.

Após computar cada métrica de cada código (ou parte dele), os valores destas métricas foram armazenados para serem utilizados no processo de análise. A evolução de uma dada métrica é obtida comparando-se um dado *commit* com seu predecessor. É importante registrar qual contribuinte foi responsável pelo *commit* realizado. Este fato auxiliou na identificação do responsável pela alteração de uma determinada métrica.

6.1.2. Análise das Métricas dos *Commits*

Baseado nos dados plotados é possível observar as variações dos índices das métricas de qualidade de software na medida em que os *commits* foram sendo realizados por cada um dos contribuintes do projeto.

O Gráfico 6.1 apresenta a métrica WMC da classe *NodeServlet* do projeto Elastic/Search. A métrica Métodos Ponderados por Classe (WMC) calcula o número de métodos de uma classe. Diminuindo o número de métodos nas classes, diminui-se a complexidade deste código, aumentando o nível de qualidade do produto de software

considerando a complexidade. No decorrer do projeto esta métrica se mantém no nível de baixo risco (Tabelas 2.3 e 2.4). Nos *commits* 8 e 9, realizados por Kimchy, ela diminui um pouco (incrementando a qualidade), aumentando novamente com o *commit* subsequente de Cwensel.

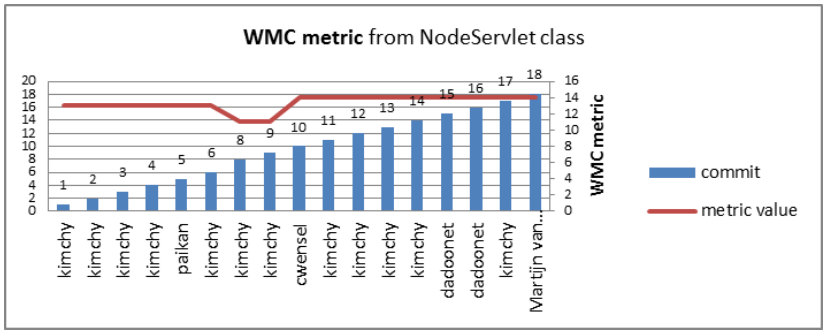


Gráfico 6.1: Aplicação da métrica WMC na classe NodeServlet do projeto Elastic/Search

A melhoria da métrica WMC no *commit* 8 vem da exclusão de algumas chamadas a métodos, conforme pode ser verificado nos códigos-fonte das Figuras 6.1a e 6.1b. A Figura 6.1a apresenta um trecho do código antes da contribuição realizada pelo desenvolvedor Kimchy, mostrando em vermelho os comandos excluídos pelo desenvolvedor.

```
6  src/main/java/org/elasticsearch/wares/NodeServlet.java
@@ -102,12 +102,8 @@ protected void service(HttpServletRequest req, HttpServletRespon
        ServletRestRequest request = new ServletRestRequest(req);
        ServletRestChannel channel = new ServletRestChannel(request, resp);
        try {
-           if (!restController.dispatchRequest(request, channel)) {
-               throw new ServletException("No mapping found for [" + request.uri()
+ + "]");
-           }
            channel.latch.await();
-       } catch (ServletException e) {
-           throw e;
        } catch (Exception e) {
            throw new IOException("failed to dispatch request", e);
        }
```

Figura 6.1a: Split do código-fonte do *commit* 8 do Projeto Elastic/Search antes da contribuição do desenvolvedor

Fonte https://github.com/elastic/elasticsearch-transport-wares/commit/a8538fcf01cb66967e4afcde8dda6f1887d44b2a?diff=split

A Figura 6.1b apresenta um trecho do código depois da contribuição do desenvolvedor, mostrando em verde os comandos incluídos pelo desenvolvedor.

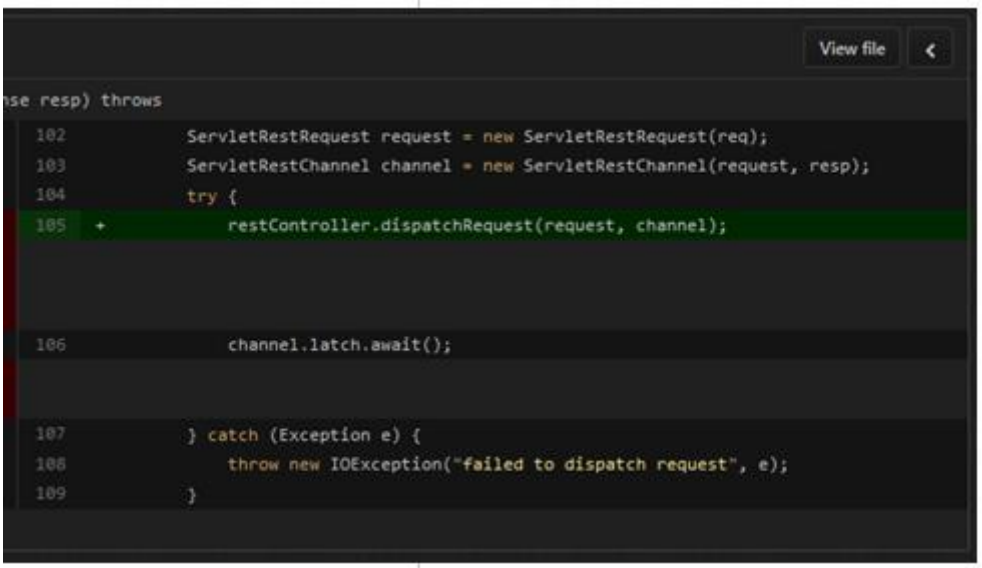


Figura 6.1b: Split do código-fonte do *commit* 8 do Projeto Elastic/Search após a contribuição do desenvolvedor

Fonte https://github.com/elastic/elasticsearch-transport-wares/commit/a8538fcf01cb66967e4afcde8dda6f1887d44b2a?diff=split

Como pode ser percebido neste exemplo, muitas vezes, simples alterações no código podem trazer melhoria na qualidade do produto de software como um todo.

Ainda considerando o Gráfico 6.1, percebe-se o incremento do WMC a partir do *commit* 10. A Figura 6.2a apresenta um trecho do código do *commit* 10 do Projeto Elastic/Search antes da contribuição realizada pelo desenvolvedor Cwenzel.

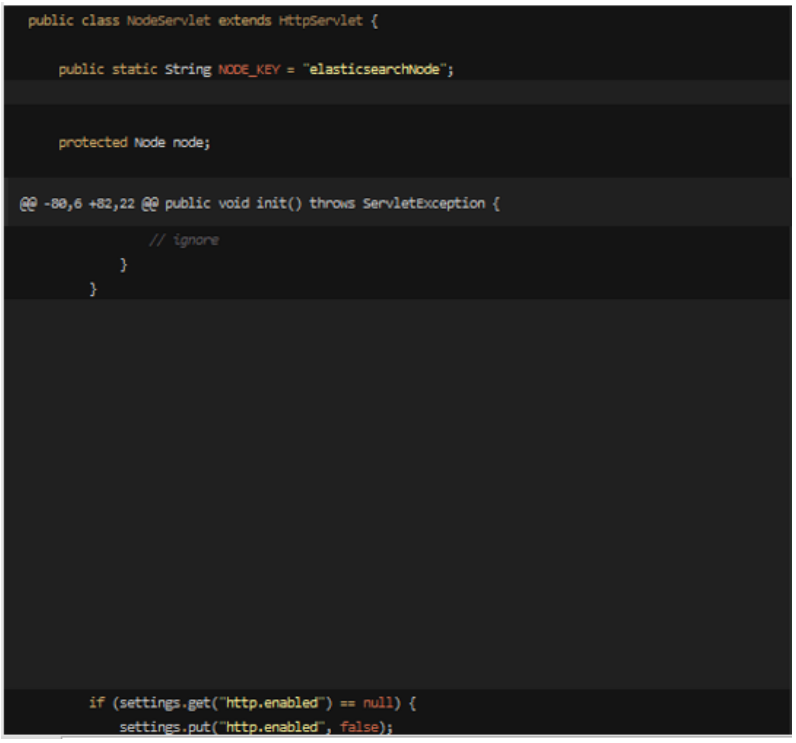
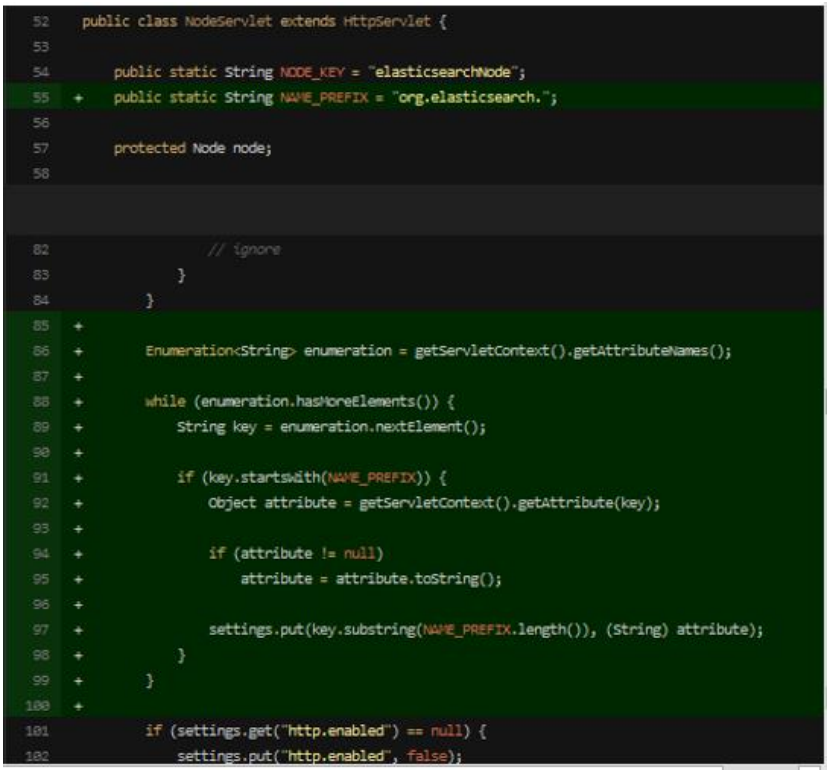


Figura 6.2a: Split de trecho do código-fonte do *commit* 10 do Projeto Elastic/Search antes da contribuição do desenvolvedor

Fonte https://github.com/elastic/elasticsearch-transport-wares/commit/7da6921418b6e709a282bf384d4c4a08eae9eedc?diff=split

Na Figura 6.2b, em verde, é possível perceber a inclusão de chamada a diversos métodos no código realizado pelo desenvolvedor Cwensel. Esta contribuição está aumentando o valor da métrica WMC do código e o desenvolvedor pode não ter percebido este fato. Conhecendo o WMC desta contribuição o desenvolvedor Cwensel poderia ter realizado esta alteração de forma a manter os índices dos itens de qualidade do software.



```
52   public class NodeServlet extends HttpServlet {
53
54       public static String NODE_KEY = "elasticsearchNode";
55   +   public static String NAME_PREFIX = "org.elasticsearch.";
56
57       protected Node node;
58
82                   // ignore
83               }
84           }
85   +
86   +       Enumeration<String> enumeration = getServletContext().getAttributeNames();
87   +
88   +       while (enumeration.hasMoreElements()) {
89   +           String key = enumeration.nextElement();
90   +
91   +           if (key.startsWith(NAME_PREFIX)) {
92   +               Object attribute = getServletContext().getAttribute(key);
93   +
94   +               if (attribute != null)
95   +                   attribute = attribute.toString();
96   +
97   +               settings.put(key.substring(NAME_PREFIX.length()), (String) attribute);
98   +           }
99   +       }
100  +
101          if (settings.get("http.enabled") == null) {
102              settings.put("http.enabled", false);
```

Figura 6.2b: Split de trecho do código-fonte do *commit* 10 do Projeto Elastic/Search após a contribuição realizada pelo desenvolvedor

Fonte https://github.com/elastic/elasticsearch-transport-wares/commit/7da6921418b6e709a282bf384d4c4a08eae9eedc?diff=split

O Gráfico 6.2 apresenta a evolução da métrica CCM aplicada ao método *embedImages* do projeto CSSEmbed. A métrica CCM (Complexidade Ciclomática de McCabe) é uma medida da quantidade de caminhos lógicos linearmente independentes do código (McCabe, 1976). O resultado desta medição pode ser utilizado para indicar o número máximo de testes que precisam ser realizados para garantir que todos os comandos foram executados pelo menos uma vez. Esta medida está relacionada ao grau de dificuldade no entendimento, testabilidade e manutenibilidade de um código, conforme Tabelas 2.3 e 2.4.

O Gráfico 6.2 apresenta os *commits* realizados por todos os contribuintes do projeto e os valores da métrica CCM para cada um dos *commits*. Como pode ser verificado, o valor da métrica varia na medida em que os *commits* são realizados. Do *commit* 1 ao 15 o valor do CCM é considerado moderado; do *commit* 16 até o final os valores aumentam o que pode indicar um risco alto de dificultar de testabilidade do método. O método perde qualidade em relação à testabilidade, mantenibilidade e entendimento na medida em que os *commits* são realizados. Isto significa que a

organização irá gastar mais tempo e dinheiro quando for testar este projeto e/ou realizar ajustes. Ao mesmo tempo, o valor CCM dos *commits* 27 e 28, realizados pelo contribuinte Tivac, diminui um pouco indicando um pequeno aumento na qualidade. É possível que o contribuinte Tivac não tenha notado esta variação assim como os demais contribuintes do projeto.

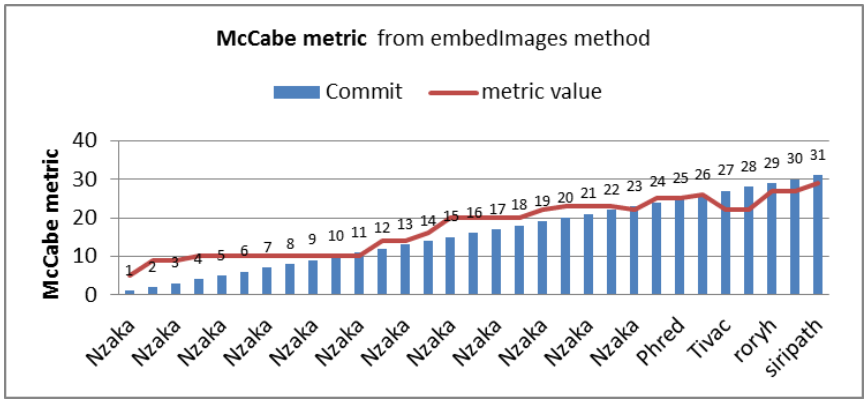


Gráfico 6.2: Aplicação da métrica CCM ao método embedImages do projeto CSSEmbed

A mesma situação pode ser observada quando os valores de CCM são analisados no método *uri* do projeto Elastic/Search (Gráfico 6.3). O CCM aumenta após o *commit* 17 do contribuinte Kimchy, degradando a qualidade do método com relação à manutenibilidade e testabilidade. Considerando esta situação é possível que se o contribuinte Kimchy tivesse sido alertado sobre o aumento do CCM, que indica que existem muitos pontos de decisão/seleção (*if-then-else*) no método, ele poderia ter realizado o *commit* dividindo o problema que o método *embedImages* resolve em problemas menores, criando outros métodos e chamando-os dentro do original, por exemplo. Esta ação poderia ter, no mínimo, mantido a qualidade do software, mesmo que estruturas com CCM em torno de 3 sejam consideradas de baixa complexidade e com baixo risco associado, conforme Tabelas 2.3 e 2.4.

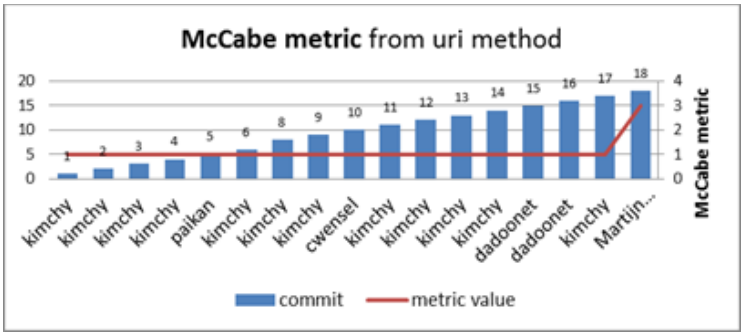


Gráfico 6.3: Aplicação da métrica CCM do método uri do projeto Elastic/Search

O Gráfico 6.4 apresenta a evolução da métrica LCOM* na classe *CSSURLEmbedder* do projeto CSSEmbed. A métrica LCOM* mede a coesão entre os métodos de uma classe (Chidamber e Kemerer, 1991) e sua análise foi baseada nas Tabelas 2.3 e 2.4. É possível perceber no Gráfico 6.4 que a LCOM* inicia perto de zero e, aproximando-se do final do projeto, ela fica próxima de 1, o que indica que a qualidade referente a coesão está decrescendo a medida em que os *commits* estão sendo realizados. Claramente, o *commit* 15 realizado pelo contribuinte Nzaka degradou a qualidade de coesão nesta classe e esta situação perdura até o final do projeto. Logo, em um cenário ideal, esta classe deveria ter sido projetada analisando melhor o código e levando em consideração os valores de coesão, o que poderia ter ocorrido caso Nzaka fosse advertido desta situação em tempo.

As outras duas métricas apresentadas no Gráfico 6.4 são NSF e NSM, que calculam o número de atributos estáticos de uma classe e o número de métodos estáticos de uma classe, respectivamente. As análises destas métricas levaram em conta as Tabelas 2.3 e 2.4.

Como pode ser visto no Gráfico 6.4, NSM inicia em 3, vai para 5, reduz para 0 e termina em 1. Esta variação mostra que a aplicação dos conceitos de programação OO aumenta na medida em que os *commits* ocorrem, significando que os contribuintes utilizam o conhecimento de programação OO de forma correta. O gráfico também apresenta que a NSF inicia em 1 aumenta para 4, depois 8 e termina em 6. Mesmo que seja percebida uma redução no valor da métrica, ela termina em 6 mostrando que o entendimento do código (complexidade) e a manutenibilidade teve sua dificuldade aumentada, diminuindo a qualidade do código.

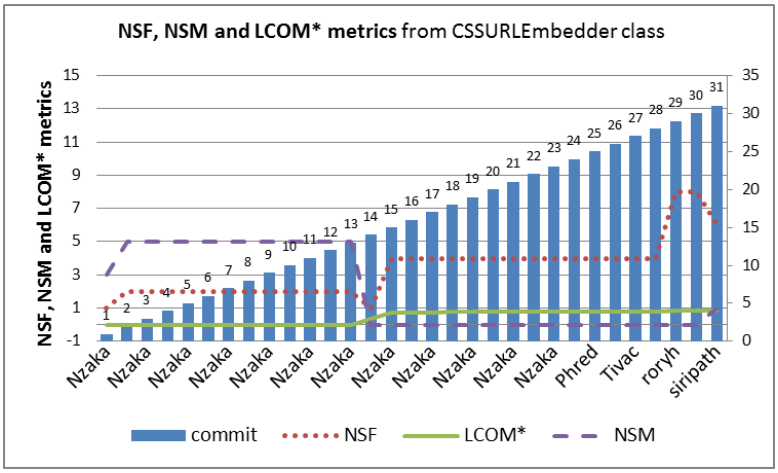


Gráfico 6.4: Aplicação das métricas NSF, NSM e LCOM* da classe CSSURLEmbedder do projeto CSSEmbed

A métrica WMC mede a complexidade de uma classe levando em consideração a complexidade de seus métodos. O Gráfico 6.5 apresenta a variação do WMC na medida em que os *commits* são realizados na classe *CSSURLEmbedder* do projeto CSSEmbed. É possível perceber que no *commit* 4 realizado por Nzaka o WMC aumenta de baixo risco para risco moderado (Tabelas 2.3 e 2.4), indicando uma degradação da qualidade relacionada a manutenibilidade. No *commit* 19 (realizado por Nzaka), 27 e 28 (realizado por Tivac) o WMC diminui, indicando um pequeno aumento na qualidade. Entretanto, o projeto finaliza como um WMC 68 (*commit* 31 realizado por Siripath), que indica um risco moderadamente alto na qualidade relativa a manutenibilidade, conforme as Tabelas 2.3 e 2.4.

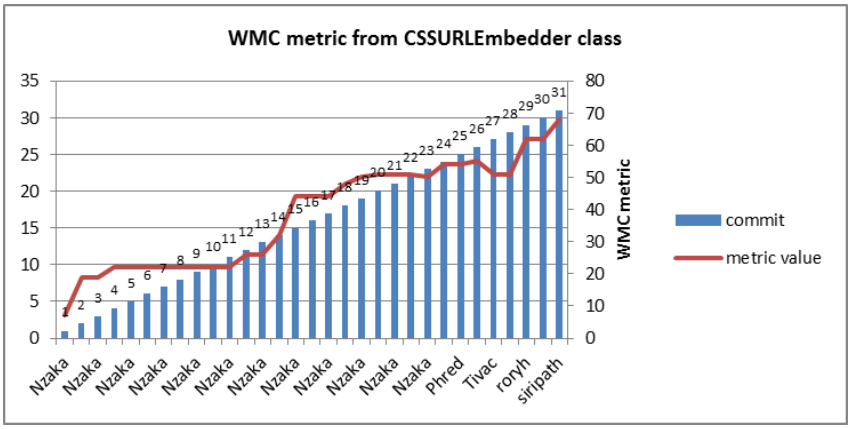


Gráfico 6.5: Aplicação da métrica WMC na classe CSSURLEmbedder do projeto CSSEmbed

O Gráfico 6.6 apresenta a evolução das métricas Ce, Ca e RMI. Ca diz respeito ao número de classes de outros pacotes que dependem da classe considerada no pacote. Ce refere-se ao número de classes em um pacote que depende de classes de outros pacotes e RMI diz respeito à métrica de instabilidade, conforme as Tabelas 5.1 e 5.2, e refere-se à relação entre Ce a o acoplamento total.

No Gráfico 6.6 é possível verificar que a métrica RMI do pacote *net.nczonline.web.cssembed* do projeto CSSEmbed está em sua instabilidade máxima (valor 1). De acordo com (Martin, 2002), um pacote é instável quando ele depende de muitos outros pacotes. A métrica Ce no Gráfico 6.6 apresenta que mesmo com a alteração do valor de Ce não houve alteração do RMI. Neste caso, o valor da métrica Ca, também no Gráfico 6.6, permanece em 0 durante todo o projeto. Este fato explica o valor de RMI com máxima instabilidade, conforme Tabelas 2.3 e 2.4.

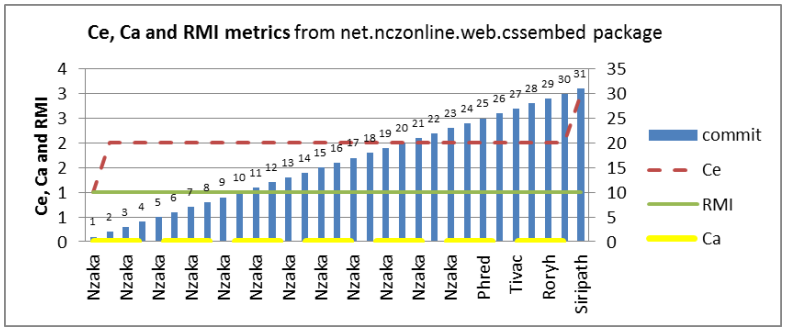


Gráfico 6.6: Aplicação das métricas Ce, Ca e RMI do pacote net.nczonline.web.cssembed do projeto CSSEmbed

6.1.3 Resultados e Discussões do Experimento 1

Conforme citado em (Bassi *et al.*, 2018), a partir da análise dos gráficos apresentados foi possível perceber a variação nas métricas de código-fonte a cada *commit* realizado pelo desenvolvedor no projeto de desenvolvimento colaborativo de software. Esta variação traz uma influência positiva, negativa ou não altera os níveis de qualidade de produto do projeto com relação à complexidade, reusabilidade, testabilidade e manutenibilidade do software. Com base nestes resultados, foi demonstrado que é possível recuperar as métricas de código-fonte a cada *commit* realizado pelo

desenvolvedor e que, a partir destas métricas, é possível verificar se houve uma variação positiva, negativa ou manteve-se os valores das métricas de código-fonte do projeto após a interferência do desenvolvedor no código-fonte do projeto que está sendo desenvolvido de forma colaborativa. Também se demonstrou que é possível identificar o desenvolvedor que realizou a alteração e qual(is) artefato(s) foi(ram) alterado(s) permitindo o uso desta informação na pesquisa. O Apêndice A apresenta outros gráficos analisados do Projeto Elastic.

Considerando o conceito de contribuição proposto nesta pesquisa, que diz: *"a contribuição é a participação mensurável de um contribuinte no processo de desenvolvimento em um projeto colaborativo de desenvolvimento de software"* e os resultados apresentados nos gráficos comentados nesta seção, foi possível validar a primeira hipótese secundária da pesquisa que procurou validar "e*xiste "contribuição" entre os integrantes de uma equipe de desenvolvimento colaborativo de software.*

6.2. Experimento 2

Este segundo experimento foi realizado com o objetivo de verificar a equação do grau de contribuição do desenvolvedor baseado nos estudos apresentados nos Capítulos 2 e 5.

Para aplicação deste experimento a equipe de desenvolvimento pode ou não trabalhar de forma colaborativa, mas precisa fazer uso de uma ferramenta de controle de versão, uma vez que este experimento utiliza a base histórica dos dados dos projetos disponíveis nos SCV e retorna para o desenvolvedor os valores referentes ao índice de qualidade das alterações realizadas por ele no *commit*.

A visualização do resultado é realizada por um *dashboard* no qual o desenvolvedor verificar a influência do seu *commit* nos índices de qualidade do produto do projeto. Também é apresentado ao desenvolvedor recomendações de como, e em qual artefato (método/classe/pacote), ele pode atuar para incrementar ou manter os índices de qualidade.

6.2.1 Coleta de Dados

Este experimento foi realizado utilizando a base de dados composta por código-fonte disponível no GitHub. Foram escolhidos projetos implementados em Java e cujos código-fonte foram possíveis de serem reconstruídos integralmente a partir da base

histórica disponibilizada no repositório de projetos. Na escolha dos projetos também foi considerada a presença de um mesmo desenvolvedor participando das equipes de desenvolvimento colaborativo de software, com o objetivo de verificar o comportamento do desenvolvedor em diferentes projetos em relação ao grau de contribuição. Da mesma forma que o Experimento 1, foi aplicado o método apresentado no Algoritmo 6.1 e considerando os pressupostos citados no Capítulo 4 em cada um destes projetos. Para cada *commit* do contribuinte foram computadas as 20 métricas de qualidade, conforme as Tabelas 2.3 e 2.4, as quais foram coletadas dos projetos selecionados do GitHub. A Tabela 6.2 apresenta as características destes projetos.

Tabela 6.2: Base de dados dos experimentos

Projeto	CSSEmbed	Elastic/Search	Elastic/Hadoop
# de *commits*	32	18	200
# de classes	8	5	11
# de linhas de código	1516	277	1783
# de contribuintes	6	5	7
Duração do projeto (em meses)	37	25	73
Link do projeto	https://github.com/nzakas/cssembed/ Acesso em Julho/2015	https://github.com/elastic/elasticsearch-transport-wares/ Acesso em Dezembro/2015	https://github.com/costin/elasticsearch-hadoop/ Acesso em Novembro/2017

Como realizado no experimento anterior, após computar cada métrica de cada código (ou parte dele), os valores destas métricas foram armazenados e a evolução de uma dada métrica foi obtida comparando-se um dado *commit* com seu predecessor. O contribuinte responsável pelo *commit* também é identificado e esta informação armazenada.

Segue o Algoritmo 6.2 utilizado para recuperar o valor das métricas do *commit*, cálculo do GC[d] e apresentação do *dashboard* com o valor do GC[d] calculado, percentual de cada item de qualidade do produto de software (complexidade, reusabilidade, manutenibilidade e testabilidade) e, caso os valores das métricas identificadas ficarem na média ou ruins, são apresentadas recomendações de alteração baseadas na métrica e qual o artefato influenciado.

```
procedure applyContributorDegree;
artifact = {class | method | package};
vet = Ø;
commit = 1;
while committing do
    Recover the commit sequence and the commit contributor;
    for metric 1 to 20 do
         Compute the commit source code artifact metric value;
         diff = abs (commit metric value – (commit – 1) metric value);
         if diff <> 0 then
              vet[metric, artifact, commit sequence, commit
                  contributor] = metric value;
         end-if
    end-for
    calculate GC[commit contributor]
    save GC[commit contributor]
    show GC[commit contributor]
    show software quality item´s dashboard
    if any software quality items values classified as "regular" or
       "bad" then
          show artifact metric´s recommendation
    end-if
    commit = commit + 1;
end-while
end-procedure
```

Algoritmo 6.2: Experimento para cálculo do GC[d]

Fonte: a própria autora.

6.2.2. Cálculo do Grau de Contribuição do Desenvolvedor

No momento da coleta dos valores das métricas do código-fonte a partir do *commit* realizado pelo desenvolvedor, é aplicada a equação do cálculo do grau de contribuição do desenvolvedor (Equação 5.1). As variações nos valores de cada métrica de cada artefato do código analisado, comparado com o *commit* imediatamente anterior, são aplicados na Tabela 5.1. Esta tabela relaciona o grau de risco do valor da métrica influenciar em determinados itens de qualidade do produto de software e retorna o peso desta implicação na qualidade do software. O peso associado varia de 1 até 4 e permite potencializar o grau de risco associado ao efeito da contribuição do desenvolvedor na qualidade do software. Este efeito pode ser positivo, negativo ou não alterar o nível de qualidade, conforme descrito na seção 5.3 no Capítulo 5. Este peso é utilizado na equação, assim como o número de *commits* realizado pelo desenvolvedor até o momento do projeto, para normalizar o valor final do grau de contribuição do desenvolvedor.

Nas Tabelas 6.3, 6.4 e 6.5 é possível verificar a evolução do cálculo do grau de contribuição do desenvolvedor e o valor final do grau de cada desenvolvedor que participou do projeto de forma colaborativa. Na segunda coluna da tabela é apresentado

o valor do grau de contribuição sem a normalização em relação ao número de *commits* realizado pelo desenvolvedor. Considerando-se o número de *commits* realizados pelo desenvolvedor (terceira coluna da tabela) o valor indicativo do grau de contribuição do desenvolvedor torna-se mais justo, já que quanto mais *commits* o desenvolvedor realiza no código mais chances ele tem de melhorar/piorar os índices de qualidade do produto final. A coluna quatro da tabela mostra o grau de contribuição do desenvolvedor normalizado em relação ao número de *commits* realizado por ele no código. Na última coluna da tabela o valor do grau de contribuição é multiplicado por -1 para melhorar a usabilidade do resultado. Um valor negativo indica que a contribuição diminuiu os índices de qualidade do código, enquanto um valor positivo indica que o desenvolvedor contribuiu de forma a melhorar os índices de qualidade do produto final.

Tabela 6.3: Cálculo do grau de contribuição de cada desenvolvedor do projeto Elastic/Search

Desenvolvedor	(Σ i = 1 até n (Δ métrica * peso do risco))	# commits	((Σ i = 1 até n (Δ métrica * peso do risco)) / # commits)	GD[d]
kimchy	6,000	12	0,500	-0,500
paikan	0,000	1	0	0
cwensel	3,000	1	3	-3
dadoonet	0,000	2	0	0
Martijn	7,000	1	7	-7

O desenvolvedor que aparece na primeira linha das tabelas é também o coordenador/gerente do projeto colaborativo.

Tabela 6.4: Cálculo do grau de contribuição de cada desenvolvedor do projeto CSSEmbed

Desenvolvedor	(Σ i = 1 até n (Δ métrica * peso do risco))	# commits	((Σ i = 1 até n (Δ métrica * peso do risco)) / # commits)	GD[d]
Nzaka	101,225	22	4,601	-4,601
Fear	5,000	1	5	-5
Phred	32,034	3	10,678	-10,678
Tivac	-29,034	2	-14,517	14,517
Roryh	87,058	2	43,529	-43,529
Siripath	15,000	1	15	-15

O cálculo do grau de contribuição do desenvolvedor *Roryh* na Tabela 6.4 mostra que sua atuação no projeto não foi adequada, pois apresenta seu GC[Roryh] de -43,529,

indicando um grau de contribuição negativo, que é considerado de nível ruim. Este valor poderia ser melhorado caso ele tivesse conhecimento do que sua contribuição causou na qualidade do produto deste projeto e de como fazer para melhorar sua atuação. Já o desenvolvedor *Tivac* (GC[Tivac]) = 14,517) melhorou de forma significativa a qualidade do produto deste projeto. De forma geral, considerando os valores do grau de contribuição dos desenvolvedores deste projeto, pode-se dizer que o projeto CSSEmbed não tem um bom nível de qualidade do produto de software.

Tabela 6.5: Cálculo do grau de contribuição de cada desenvolvedor do projeto Elastic/Hadoop

Desenvolvedor	(Σ i = 1 até n (Δ métrica * peso do risco))	# commits	((Σ i = 1 até n (Δ métrica * peso do risco)) / # commits)	GD[d]
cjcenizal	0,0000	7	0,0000	0,0000
kimchy	6,0000	36	0,5000	-0,5000
dadoonet	3,0000	19	3,0000	-3,0000
jimczi	0,0000	9	0,0000	0,0000
costin	7,0000	28	7,0000	-7,0000
tlrx	96,2250	86	5,6011	-5,6011
astefan	5,0000	15	5,0000	-5,0000

Analisando-se o projeto Elastic/Hadoop de forma geral (Tabela 6.5), apesar de ser um projeto com muitos *commits*, o nível de qualidade do produto final do projeto é bom e foi mantido por seus colaboradores contribuintes durante os *commits* realizados.

A análise de dois projetos pertencentes ao Elastic (Search e Hadoop – Tabelas 6.4 e 6.5, respectivamente) permitiu que fosse comparada a atuação de dois desenvolvedores (Kimchy e Dadoonet) em projetos diferentes. Em relação ao desenvolvedor Kimchy foi possível perceber que, mesmo em projetos com equipes diferentes, o grau de contribuição dele é igual nos dois projetos. A quantidade de *commits* realizados por Kimchy nos projetos analisados é diferente (18% de *commits* no Elastic/Hadoop e 67% de *commits* no Elastic/Search). E mesmo com diferente participação nos projetos seu grau de contribuição manteve-se o mesmo (GC[Kimchy] = -0,5 nos dois projetos). Já o desenvolvedor Dadoonet teve pouca participação nos dois projetos, mas seu grau de contribuição foi diferente nos projetos verificados. No Elastic/Search sua contribuição manteve o nível de qualidade do produto de software (GC[Dadoonet] = 0), já no Elastic/Hadoop sua contribuição diminuiu o nível de qualidade do produto de software

(GC[Dadoonet] = -3).

Os dados das métricas coletadas do *commit* realizado pelo contribuinte são apresentados na tela do usuário em forma de *dashboard* informando o grau de contribuição do desenvolvedor no código e os percentuais de influência da alteração nos itens de qualidade – complexidade, testabilidade, reusabilidade e manutenibilidade. Este retorno é baseado nos conceitos das faixas de risco compilados na Tabela 5.1. A cor verde utilizada no *dashboard* indica a quantidade de métricas que compõem o item de qualidade que estão na faixa *baixo/ausente/fraco/bom*, conforme a Tabela 5.1. A cor amarela por sua vez, sinaliza a quantidade de métricas que compõem o item de qualidade e que estão na faixa *moderado/desejável/aceitável/regular*. Já a cor vermelha indica a quantidade de métricas que compõem o item de qualidade sinalizado e que estão na faixa *alto/muito alto/profundo/forte/ruim*. O desenvolvedor consegue saber a número de métodos/classes/projetos, que estão em cada faixa (boa/regular/ruim) de grau de contribuição, no cabeçalho das roscas representando cada item de qualidade de software afetado. A Figura 6.3 apresenta um exemplo de *dashboard* da aplicação.

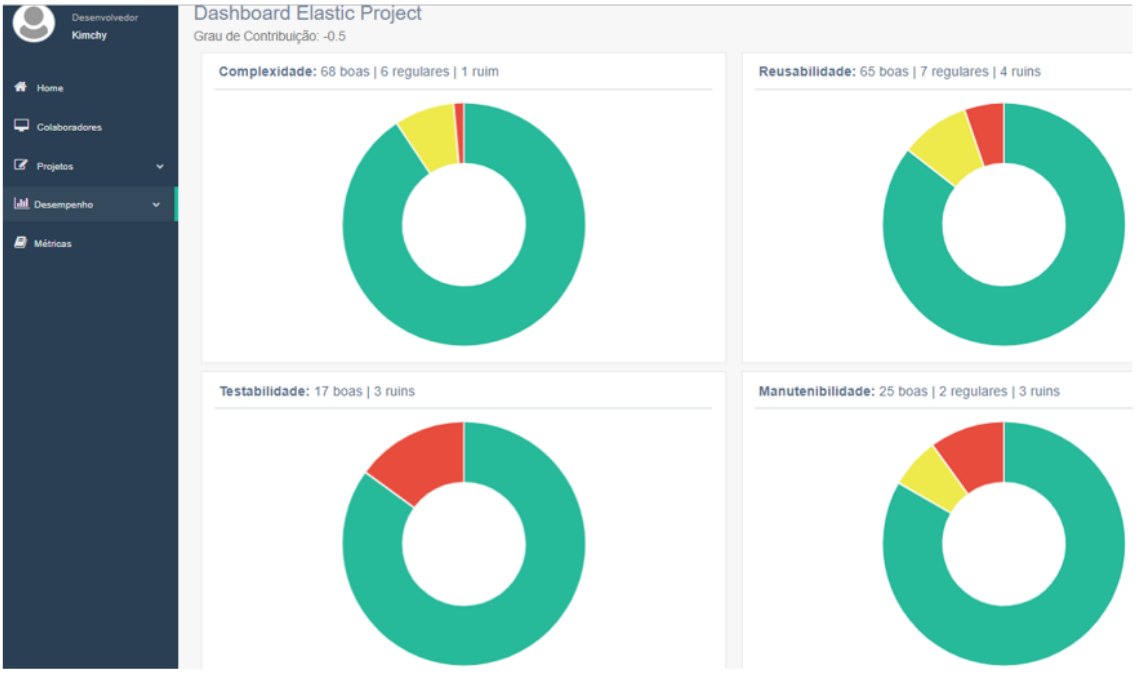


Figura 6.3: Exemplo de *dashboard* da aplicação

Na sequência, caso o desenvolvedor tenha tido um nível de contribuição ruim/alto/forte/profundo/muito alto em relação aos itens de qualidade do produto de

software, o aplicativo retorna “recomendações” para que o desenvolvedor entenda o que ocorreu a partir da inserção do seu *commi*t no projeto de software em relação aos itens de qualidade do produto de software. As “dicas” permitem que o desenvolvedor tenha conhecimento do artefato (método/classe/pacote) no qual a degradação de qualidade ocorreu e qual item de qualidade foi afetado. Em função do valor das métricas, também é retornado para o desenvolvedor qual ação ele pode realizar para melhorar a métrica e consequentemente os índices de qualidade do produto de software que está sendo desenvolvido. As Figuras 6.4a e 6.4b apresentam um exemplo de retorno de “recomendação” para o desenvolvedor. Na Figura 6.4a é possível analisar por meio de um gráfico de barras o valor da métrica analisada nos artefatos do código referente a uma classe específica do projeto nos quais a métrica VG (Complexidade ciclomática de McCabe) foi calculada.

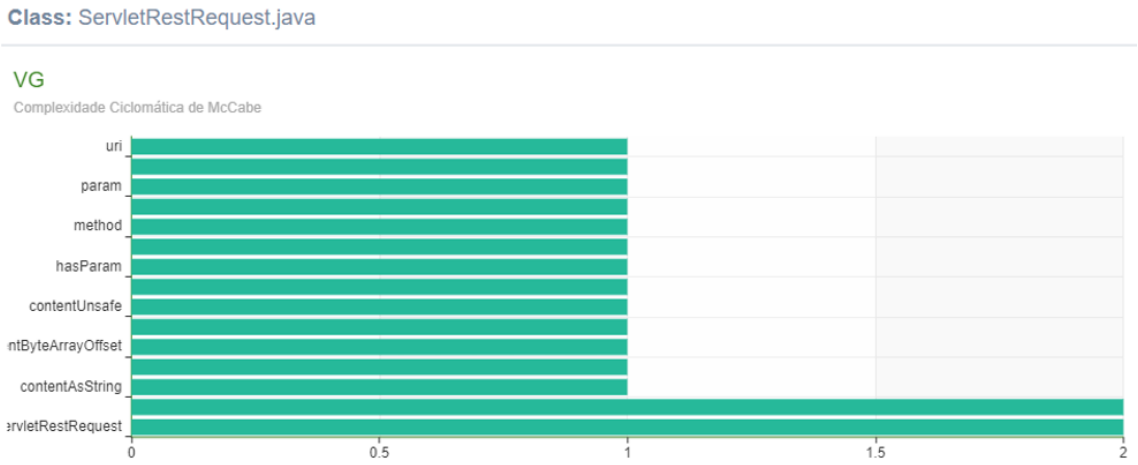


Figura 6.4a: Exemplo de gráfico dos valores das métricas por artefato do projeto

A Figura 6.4b mostra a “recomendação” sugerida pelo aplicativo com base nos valores das métricas calculadas para cada artefato analisado. O desenvolvedor tem acesso ao nome do artefato e qual ação pode ser realizada para melhorar a métrica e consequentemente seu grau de contribuição no projeto.

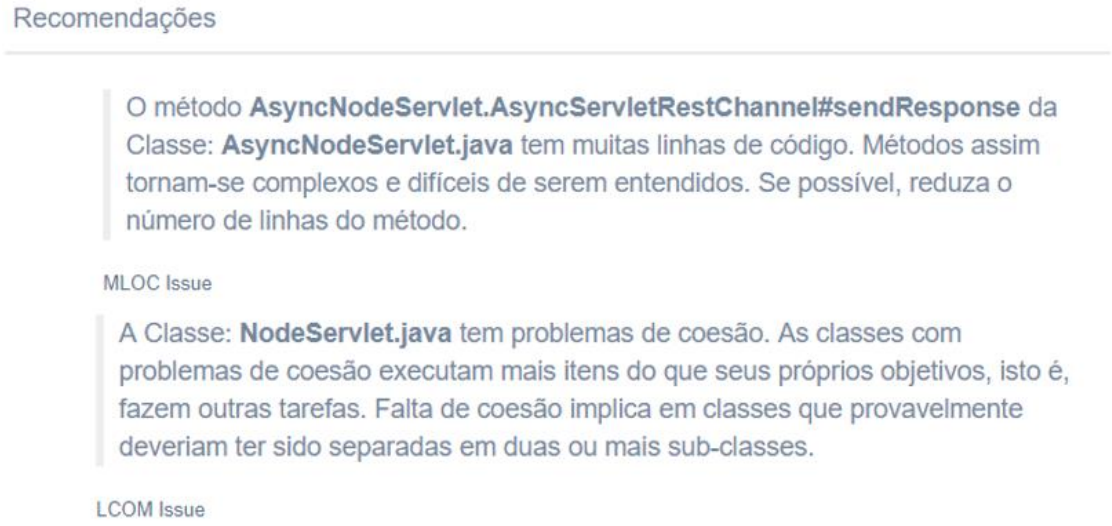


Figura 6.4b: Exemplo de "recomendação" por métrica e artefato para o desenvolvedor ter condições de melhorar o código

Assim, o desenvolvedor pode voltar atrás e melhorar a forma de codificação no artefato identificado, estudando melhor o código e buscando alternativas mais adequadas em relação à coesão, acoplamentos, complexidade, correto uso das classes e métodos já existentes no código. A partir do momento que o desenvolvedor realiza as alterações recomendadas, ele faz um novo *commit* no código do projeto e é realizado um novo cálculo de seu grau de contribuição.

Os retornos enviados para o desenvolvedor em relação aos valores das métricas calculados são baseados nos estudos realizados nesta pesquisa e estão compilados na Tabela 5.2.

6.2.3 Resultados e Discussões do Experimento 2

Este experimento permitiu validar o cálculo do grau de contribuição do desenvolvedor. Desta forma foi importante a escolha de projetos que abordassem no mínimo um dos desenvolvedores em mais do que um projeto analisado permitindo verificar seu grau de contribuição em equipes diferentes de desenvolvimento.

Os dados coletados a partir da base de dados histórica do GitHub permitiu o ajuste da fórmula em relação a normalização, definição dos pesos e forma de apresentação do resultado do grau de contribuição do desenvolvedor. Cada um destes itens da equação está explicado na Seção 5.3 no Capítulo 5 deste trabalho.

A partir da análise dos resultados parciais da aplicação da equação do cálculo do

grau de contribuição foi possível verificar a necessidade da normalização do valor das métricas em função do número de *commits* realizado por cada um dos desenvolvedores. Esta situação pode ser percebida no cálculo dos desenvolvedores *Kimchy* da Tabela 6.3, *Nzaka* da Tabela 6.4 e *Tlrx* da Tabela 6.5.

Também a necessidade de multiplicar o resultado por -1 mostrou-se importante para uma melhor visualização do resultado do grau de contribuição do desenvolvedor. Assim um valor negativo indica uma contribuição ruim e um valor positivo indica uma boa contribuição por parte do desenvolvedor no projeto.

Analisando os resultados do grau de contribuição de cada desenvolvedor no contexto do projeto algumas situações identificadas no ambiente de desenvolvimento colaborativo de software, o SCV, chamaram atenção e foram considerados nesta pesquisa.

Conforme proposto no Capítulo 1, a pesquisa tem como um dos objetivos permitir que o desenvolvedor melhore sua contribuição em relação a qualidade do produto de software, e, além disso, aprenda com esta ferramenta. A ferramenta utilizada para recuperar as métricas de código-fonte (*Plugin* Metrics do Eclipse) permite recuperar o valor das métricas e o artefato (método/classe/pacote) que foi medido. Desta forma a métrica que obteve um valor com grau de risco considerado ruim/alto/forte/profundo/muito alto associado ao item de qualidade que foi infringido pode apresentar para o desenvolvedor uma "dica" de onde e como ele pode melhorar a qualidade do código.

A partir do momento que o desenvolvedor aceita a "dica" e procura aplicar os conceitos de influência no código que a métrica retorna para ele, alterando o código-fonte, um novo *commit* é associado ao projeto e o cálculo do grau de contribuição para este desenvolvedor é refeito. De forma constante, o desenvolvedor pode melhorar seu desempenho de contribuição na qualidade do produto de software e aprender a interpretar o retorno dos valores da métricas informado pela ferramenta. Também o gerente de projetos pode aproveitar esta informação e sugerir treinamentos específicos para seus colaboradores, além de poder definir as equipes dos projetos baseado na performance mensurável dos desenvolvedores, a contribuição de cada um na qualidade do produto de software.

A partir da aplicação e dos resultados apresentados nesta seção em relação ao cálculo do grau de contribuição do desenvolvedor foi possível comprovar a segunda hipótese desta pesquisa: *dada uma coleção de métricas de qualidade de software, estas*

são capazes de expressar o grau de contribuição individual dos participantes do projeto de desenvolvimento colaborativo de software de qualidade no nível de código-fonte. O Apêndice B apresenta detalhes da aplicação desenvolvida.

6.3. Experimento 3

Os dados coletados nas mensagens dos *commits* e *issues* deixadas pelos coordenadores dos projetos colaborativos (gerentes de projeto) e desenvolvedores no ambiente de controle de versão associados às variações das métricas de qualidade calculadas podem ser combinados e processados para contribuir com informações relevantes quanto à motivação da alteração/implementação realizada pelo desenvolvedor.

Para buscar traços da influência do gerente de projetos no processo de alteração do código realizado pelo desenvolvedor no *commit,* as mensagens trocadas entre os membros da equipe de desenvolvimento colaborativo deste *commit* e seus *issues* foram analisadas por meio de técnicas de recuperação da informação, como a análise de sentimento. A proposta deste experimento foi de verificar se houve traços de "ordem" por parte do coordenador/gerente do projeto na forma como o desenvolvedor deveria conduzir a alteração no código-fonte.

Segundo Alali e colegas (Alali *et al.*, 2008), dentro do contexto do desenvolvimento colaborativo de software o objetivo das mensagens trocadas entre os membros das equipes de desenvolvimento colaborativo de software é no sentido de descrever as alterações realizadas e ajudar a identificar a lógica da codificação realizada. Estas mensagens são essenciais para a compreensão e evolução geral do projeto, ajudando os desenvolvedores a entender e validar as alterações, localizar e endereçar *bugs*, rastrear alterações entre os artefatos do software.

Conforme pode ser verificado na documentação do GitHub, dada a importância destas mensagens, existe um código de conduta entre os membros das equipes de desenvolvimento colaborativo de software em relação a estrutura e conteúdo das mensagens destes *commits*. Isto ocorre porque os desenvolvedores que estão colaborando no projeto tem consciência de que escrever uma boa mensagem de *commit* serve, pelo menos, para três importantes motivos: aumentar a velocidade das revisões; ajudar a escrever uma boa nota sobre a nova versão e; ajudar os futuros membros da equipe a identificar os motivos das alterações realizadas no código e o motivo de um requisito

especifico ter sido inserido no código. A Figura 6.5 apresenta um exemplo de mensagem de *commit* de um projeto.

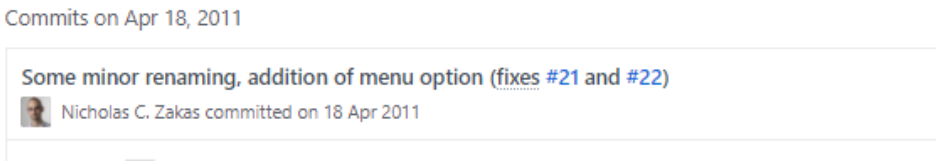


Figura 6.5: Exemplo de uma mensagem de *commit*
Fonte: https://github.com/nzakas/cssembed/commits/master

Também é possível verificar na documentação do GitHub (https://guides.github.com/features/issues/), que os *issues* contém informações importantes para os desenvolvedores em um ambiente SCV. Os *issues* são uma forma colaborativa de os desenvolvedores, que estão trabalhando em um mesmo projeto, compartilharem ideias e discutirem sobre situações do projeto. São também uma forma dos colaboradores do projeto acompanharem as atividades, as evoluções e solucionarem os *bugs* do projeto. Os *issues* podem ser postados por membros do projeto que sejam somente os observadores/cooperadores, os desenvolvedores/contribuintes e coordenadores do projeto. A Figura 6.6 apresenta um exemplo de um *issue* de um projeto.

Figura 6.6: Exemplo de um *issue*
Fonte: https://github.com/nzakas/cssembed/issues/10

Como as mensagens de *commit* e os *issues* dos projetos são escritos em linguagem natural, para encontrar se existem ou não traços de influência do gerente de projeto/coordenador nas instruções que guiaram as alterações no código realizada pelo desenvolvedor, é possível aplicar a análise de sentimento nas mensagens de *commit* e nos

issues dos projetos. A proposta utilizada foi de buscar a presença ou não de "palavras de ordem" dentro dos textos das mensagens de *commit* e *issues* postadas no projeto.

Conforme descrito no Capítulo 2, a análise de sentimento se caracteriza pelo processo de avaliar a quantidade de emoção - positiva, negativa ou neutra – associada a um trecho de texto. Transportando o contexto da análise de sentimento para este experimento, foi considerado que, analisando-se as mensagens dos *commits* e *issues*: "emoção positiva" indica que houve interferência nas alterações de código realizada pelo desenvolvedor; "emoção negativa" indica que não houve interferência na alteração de código realizada pelo desenvolvedor. Na sequência o método aplicado é explicado em detalhes.

6.3.1 Análise das Mensagens dos *Commits* e *Issues*

Neste experimento, primeiramente foi realizada uma análise manual em um pequeno conjunto de amostras de mensagens de *commit* e percebeu-se que estas mensagens eram normalmente curtas, escritas em linguagem natural e informal, além de seguir as regras dos ambientes CSCW de mensagens de *commits*.

A partir de estudos realizados na área de análise de sentimentos (Pang e Lee, 2008) (Sadegh *et al.*, 2012) (Shaikh e Deshpande, 2016) foi possível entender os conceitos e aplicá-los nesta pesquisa (Capítulo 2). Como o objetivo do experimento foi identificar a presença ou não de palavras específicas nas mensagens de *commit* e nos *issues*, e estes textos são geralmente curtos, em linguagem natural informal, a técnica de análise léxica se mostrou interessante para ser aplicada neste experimento.

A ferramenta de análise léxica escolhida para este experimento foi o SentiStrenght Tool (http://sentistrength.wlv.ac.uk), que tem como característica a capacidade de processar textos de diversos formatos como, textos simples, linguagem de comando, textos recebidos da internet entre outros específicos da área de Tecnologia da Informação. Outra característica importante desta ferramenta é a possibilidade de alterar o dicionário léxico utilizado.

Para cumprir o objetivo deste experimento, que é identificar se existe ou não influência do gerente de projetos/coordenador, foram utilizadas "palavras de ordem" como polaridade positiva para identificar se o gerente de projeto influenciou nas alterações do código realizada pelo desenvolvedor. O Dicionário Thesaurus foi utilizado para selecionar as "palavras de ordem" e seus sinônimos para a criação do dicionário léxico a ser utilizado no SentiStrenght. O Dicionário Thesaurus, além do sinônimo,

identifica quanto afastada a palavra analisada está do significado da palavra original, mas ainda assim pode indicar uma "palavra de ordem", porém, com menor intensidade, isto é, quanto positiva a "palavra de ordem" analisada é. A Figura 6.7 apresenta o método aplicado neste experimento.

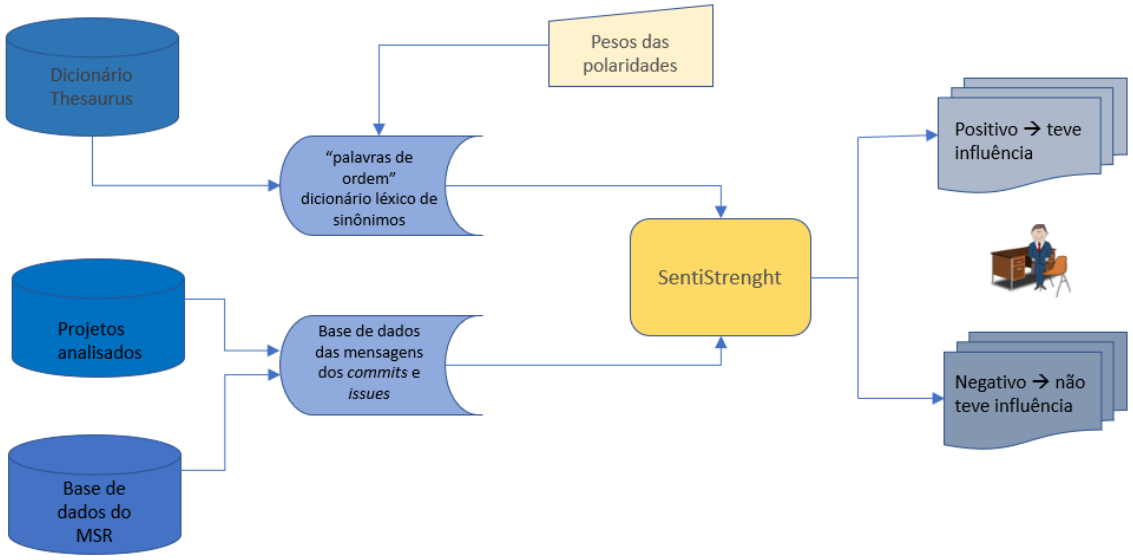
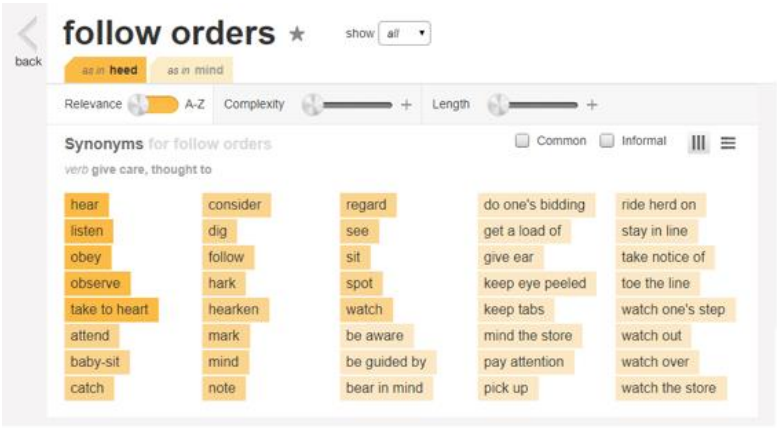


Figura 6.7: Método do experimento de análise léxica das mensagens de *commit* e *issues*
Fonte: a própria autora.

6.3.2 Coleta de dados

A primeira coleta de dados realizada foi para compor o dicionário léxico do SentiStrenght de forma a encontrar as "palavras de ordem" e, através dos pesos, indicar a quanto próxima do sinônimo principal a "palavra de ordem" está. Foram selecionadas as palavras mais relevantes com o mesmo significado no Dicionário Thesaurus (www.thesaurus.com). Com esta base de dados selecionada foram associados os pesos: 2, para palavra com pouca ênfase de "ordem" até 5, para palavras com alto efeito de "ordem". Os pesos +1, 0 e -1 não são utilizados, uma vez que, nesta técnica, estes pesos são aplicados as palavras neutras e os termos neutros são ignorados, pois não identificam "palavras de ordem" ou "de não ordem". A Figura 6.8 apresenta um exemplo do Dicionário Thesaurus e a tabela ao lado os pesos associados as palavras de acordo com sua distância da palavra de ordem principal. No Dicionário Thesaurus, esta distância é apresentada pela cor: cor mais forte, significando mais próximo da palavra original; ou cor mais franca, significando mais afastado da palavra original.



Word	Weight
Hear	5
Listen	5
....	
Should	5
....	
Attend	4
Consider	4
...	

Figura 6.8: Resultado da pesquisa no Dicionário Thesaurus e os pesos da polaridade das palavras de ordem

Como esta pesquisa tem como foco buscar informações de forma não-invasiva, o método foi aplicado nas mensagens deixadas pelos colaboradores do projeto. Assim, a segunda coleta de dados, diz respeito à coleta das mensagens dos *commits* e dos *issues*. Foram utilizadas para compor o *corpus*, mensagens de *commits* e *issues* de projetos disponibilizados no *MSR 2014 Mining Challange Dataset* provido pelo GHTorrent (http://ghtorrent.org). Desta base de dados foram retiradas 600 mensagens de *commit*, e foram incluídas 32 mensagens de *commits* dos projetos que estão sendo analisados nesta pesquisa. De suas 632 mensagens de *commits* e *issues*, 100 foram anotadas manualmente.

A coleta das mensagens dos *commits* e *issues* dos projetos que estão sendo analisados nesta pesquisa, ocorreu juntamente com a coleta das métricas do código-fonte dos experimentos 1 e 2 já relatados. A base de dados de treinamento foi criada utilizando as 100 mensagens de *commits* e *issues* anotadas manualmente.

6.3.3 Resultados e Discussões do Experimento 3

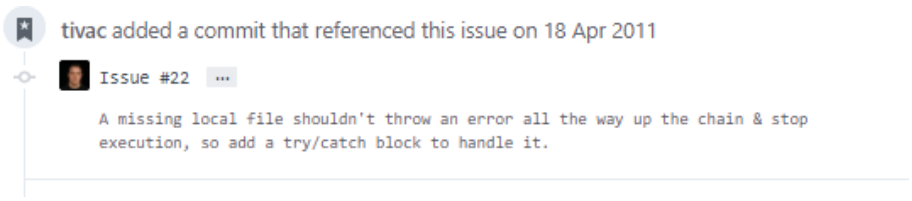
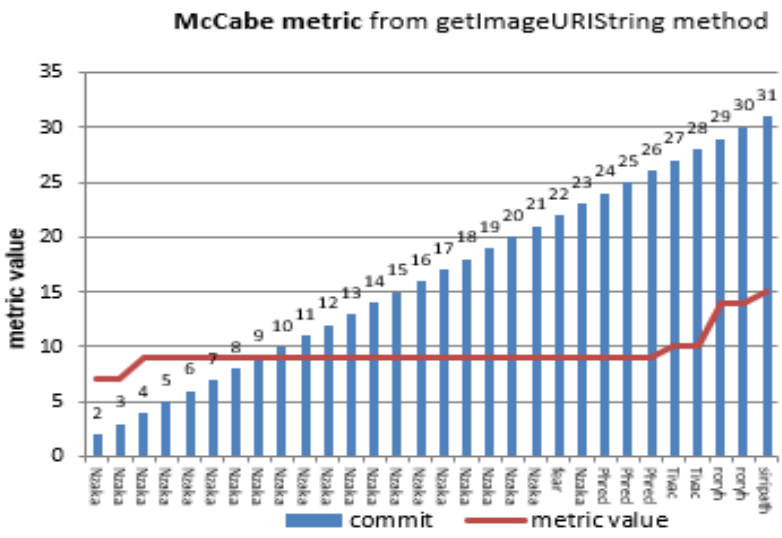
O experimento foi executado e os valores do dicionário léxico utilizado foram ajustados, conforme a ferramenta SentiStrenght prevê em sua documentação.

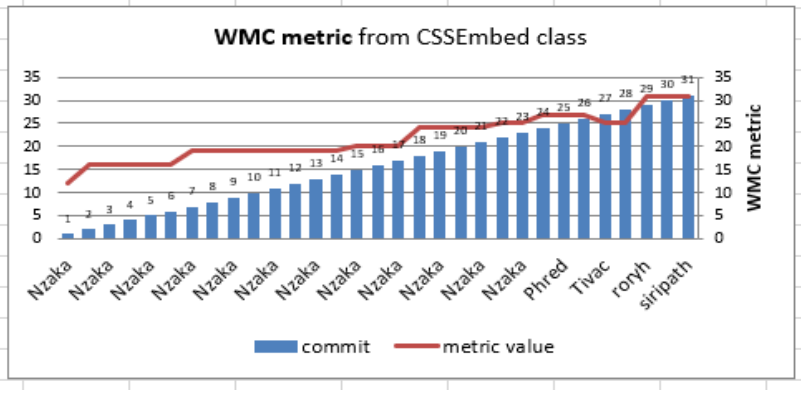
Para validar o experimento foi utilizada a acurácia, que ficou em 0,25. Estudos apresentados no manual de referência do Sentistrength mostram que a acurácia 0,2 é considerada Ok, enquanto acurácia 0,4 ou mais é considerada excelente. Os valores

citados aqui são os aplicados na descrição da ferramenta SentiStrenght, disponível em http://sentistrength.wlv.ac.uk/documentation/language_changes.html.

Como resultado, a Tabela 6.6 apresenta o *issue* do *commit* 28 classificado como positivo em função da "palavra de ordem" *shouldn´t*. Porém, os gráficos da métrica McCabe e WMC, na 2ª e 3ª linhas da Tabela 6.6, mostram que houve um aumento nos índices McCabe e WMC dos códigos, indicando que a qualidade do código melhorou após o *commit* 28. Desta forma, é possível perceber que o grau de contribuição do desenvolvedor pode ser positivo, mesmo a alteração tendo um viés de indicação de influência do gerente de projeto/coordenador na alteração realizada pelo desenvolvedor.

Tabela 6.6: *Issue* do *commit* 28 e suas métricas de código-fonte McCabe e WMC

28º *Commit*	
Métricas de qualidade do	

código-fonte	

O resultado apresentado mostrou que é possível identificar se houve ou não influência do gerente de projeto/coordenador na medida em que os desenvolvedores realizam *commits* no projeto, aplicando métodos e ferramentas de análise de sentimentos. Desta forma, é possível comprovar a terceira hipótese secundária desta pesquisa referente a: *os comentários e mensagens deixados pelos coordenadores e desenvolvedores nas mensagens dos* commits *e* issues *do projeto podem mostrar influência na contribuição individual do desenvolvedor.*

6.4. Considerações Finais

Este capítulo apresentou os experimentos realizados nesta pesquisa, na qual foram analisadas as métricas de qualidade de código-fonte, a influência de cada métrica nos índices de qualidade referentes à manutenibilidade, reusabilidade, testabilidade e entendimento do código. Estas métricas foram agrupadas em razão dos índices de qualidade que influenciam. Foram coletados dados históricos de projetos de software para obter tais métricas que foram analisadas focando na necessidade de se comprovar as influências destas métricas nos índices de qualidade. Com base neste resultado foi conduzido o experimento para validar o cálculo do grau de contribuição do desenvolvedor no desenvolvimento colaborativo de software. Também foi apresentado o experimento que busca verificar a possibilidade de identificar se houve influência do gerente de projetos no modo como o desenvolvedor realizou a alteração no projeto.

Os experimentos realizados comprovaram as hipóteses secundárias desta pesquisa. Assim, na sequência será apresentada a conclusão da pesquisa abordando sua contribuição científica, suas limitações e dificuldades e, por fim, seus trabalhos futuros.

Capítulo 7

Considerações Finais

Apresenta-se neste capítulo as considerações finais referentes aos resultados obtidos nesta tese. Para tanto, a seguir são discutidas a relevância, as contribuições e as limitações da pesquisa, na esfera da utilização de meios para aprimorar a contribuição dos membros das equipes de desenvolvimento colaborativo de software, assim como as conclusões da pesquisa realizada. Também é apresentado, ao final do capítulo, possibilidades de trabalhos futuros.

7.1. Relevância da Pesquisa

O desenvolvimento colaborativo de software é uma prática que vem crescendo nas organizações e principalmente nos ambientes que aplicam a metodologia ágil e recursos *open source*. O perfil dos desenvolvedores que estão chegando ao mercado (gerações X e Y); as ferramentas disponíveis para este modelo de desenvolvimento de software (cada vez mais difundidas e de utilização simples); e a globalização das organizações (permitindo que equipes de desenvolvimento de software sejam compostas por membros de diferentes localizações geográficas e de diferentes culturas) facilitam ainda mais a utilização deste recurso para o desenvolvimento de software.

Com esta diversidade de desenvolvedores trabalhando de forma colaborativa em um mesmo sistema de software vem à tona a necessidade de manter um nível mínimo de qualidade do produto deste software, dao que esta forma de desenvolvimento de software diminui os custos do produto final, porém, deixa lacunas no nível de qualidade final do produto de software. Embora existam diversas técnicas para avaliar o nível de qualidade do produto de software elas não consideram os ambientes colaborativos de

desenvolvimento. E, em sua maioria, as técnicas aplicadas em ambientes colaborativos de desenvolvimento de software são invasivas e não tratam do produto de software – o código-fonte em si.

A motivação principal desta pesquisa foi prover uma forma não-invasiva de calcular o grau de contribuição do desenvolvedor no desenvolvimento colaborativo de software, focando nos itens de qualidade do produto de software. Para isso, foi feito uso das métricas de código-fonte que influenciam nos itens de qualidade do produto de software – a complexidade, manutenibilidade, reusabilidade e testabilidade. A partir do cálculo do grau de contribuição do desenvolvedor foi possível validar o conceito de Contribuição no ambiente colaborativo de desenvolvimento de software. A Contribuição vem a somar e diferenciar o item Cooperação, no Modelo 3Cs do desenvolvimento colaborativo de software – coordenação, comunicação e cooperação.

Conhecendo-se o grau de colaboração do desenvolvedor em um ambiente colaborativo de desenvolvimento de software as organizações têm condições de melhor organizar seus recursos humanos nas equipes de desenvolvimento, além de qualificá-los de maneira pontual.

7.2 Contribuições da pesquisa

Considerando os resultados obtidos por meio do método de pesquisa aplicado, pode-se ressaltar as seguintes contribuições:

- Definição do conceito de Contribuição do desenvolvedor dentro do ambiente colaborativo de desenvolvimento de software, somando e diferenciando a Cooperação que compõem o Modelo 3Cs do desenvolvimento colaborativo de software – coordenação, comunicação e cooperação;
- Definição do conceito de grau de contribuição do desenvolvedor a partir da equação de cálculo do grau de contribuição do desenvolvedor no desenvolvimento colaborativo de software no nível de código-fonte;
- Identificação das métricas de qualidade de software capazes de sinalizar as variações (aumento, diminuição ou não influência) que as alterações realizadas pelo desenvolvedor (contribuinte) no código-fonte exercem sobre a qualidade tendo em vista a manutenibilidade, testabilidade,

reusabilidade e complexidade do código.

7.3 Limitações da pesquisa

O objetivo principal desta pesquisa foi propor um método não-invasivo para medir o grau de contribuição individual entre os desenvolvedores participantes de um projeto colaborativo de desenvolvimento de software. Para atingir este objetivo foram utilizados os valores das métricas de código-fonte, suas variações entre os *commits* realizados pelos desenvolvedores/contribuintes e o agrupamento das métricas em relação a sua influência nos itens de qualidade do produto de software – complexidade, manutenibilidade, testabilidade e reusabilidade.

Foi necessário realizar o cálculo das métricas de código-fonte considerando a linha do tempo dos *commits* realizados pelos desenvolvedores contribuintes do respectivo projeto. Para tanto, foi utilizada uma base de dados histórica, de projetos de software *open source* desenvolvidos de forma colaborativa, de onde foram retirados e analisados tais códigos-fonte. No entanto, algumas limitações desta pesquisa podem ser discutidas:

- Para ter a visão global do projeto, foram selecionados projetos finalizados ou há algum tempo sem atualização. Desta forma, os projetos utilizados nos experimentos não eram recentes. Para realizar o cálculo dos valores das métricas de código-fonte a cada *commit* realizado pelos desenvolvedores/contribuintes os códigos-fonte precisaram ser *buildados* e compilados um a um e este fato gerou a necessidade de se criar todo o ambiente de produção do projeto novamente. Na grande maioria dos projetos, este ambiente de produção não foi possível de ser organizado, dificultando a coleta dos dados para geração da base de dados. Este fato, somado a não serem projetos recentes, limitou o número de projetos analisados.
- As métricas de código-fonte utilizadas no cálculo do grau de contribuição do desenvolvedor foram baseadas na influência que tais métricas têm nos itens de qualidade do produto de software. Desta forma, foi necessário utilizar o *Plugin* Metrics do Eclipse. O Metrics retorna os valores das métricas puras sem interpretações, diferente do Sonar, por exemplo.

Então, o cálculo proposto aplica-se somente a projetos desenvolvidos no ambiente Eclipse.

- Estudos mostram que, apesar das métricas de código-fonte serem focadas no paradigma orientado a objetos, as linguagens de programação orientadas a objeto como C++ e Java, por exemplo, podem ter interpretações diferentes em relação aos níveis de risco da influência do valor da métrica nos itens de qualidade do produto de software e isto ocorre devido à forma de construção do compilador de cada linguagem. Logo o cálculo proposto aqui aplica-se a projetos desenvolvidos na linguagem de programação Java.
- O protótipo desenvolvido mostra um retorno em forma de *dashboard* para o desenvolvedor/contribuinte informando como e em qual artefato ele poderia melhorar a qualidade do produto de software. No entanto, a efetividade deste retorno necessita de interações com o desenvolvedor/contribuinte em tempo real de produção do código.

7.4 Conclusões

A relevância do tema é justificada pela inserção desta pesquisa em uma área que está em constante desenvolvimento e necessita construir software de forma ágil, descentralizada e com qualidade. Além disto, as pesquisas realizadas e os retornos das revisões dos pesquisadores, quando da submissão de artigos sobre esta pesquisa ((Bassi *et al.*, 2018), (Silva *et al.*, 2019)) e relacionados a ela (Beal *et al.*, 2017) para eventos científicos, mostraram que inexiste um conceito ou um método semelhante ao que foi estudado aqui, o que também é um item motivador desta tese.

Após análise dos resultados disponibilizados nos experimentos realizados pôde-se comprovar que é possível relacionar as métricas de qualidade de software de cada *commit* realizado por cada membro da equipe de desenvolvimento do software com manutenibilidade, testabilidade, reusabilidade e complexidade do código-fonte. Também, conforme os resultados dos experimentos, foi possível calcular o grau de contribuição do desenvolvedor, identificando se a sua contribuição no código-fonte ocorreu de forma positiva, negativa ou manteve o nível de qualidade do produto de software considerando a manutenibilidade, testabilidade, reusabilidade e complexidade do código. Estes

resultados também permitiram validar o conceito proposto nesta pesquisa em relação à contribuição do desenvolvedor no desenvolvimento colaborativo de software, diferenciando a contribuição da cooperação, no momento em que a contribuição se define como sendo *a participação mensurável de um contribuinte no processo de desenvolvimento em um projeto colaborativo de desenvolvimento de software*. E também o papel do contribuinte neste contexto, como sendo *o participante cuja contribuição pode ser mensurada.*

Por meio dos experimentos realizados e descritos, a hipótese proposta nesta pesquisa de que é *possível medir o grau de contribuição individual entre os participantes de um projeto colaborativo de desenvolvimento de software no nível do código-fonte*, pôde ser comprovada.

Além disto, é possível extrair contribuições deste trabalho, uma vez que, a partir deste conhecimento os gerentes de projetos de desenvolvimento de software têm auxílio na condução e melhor organização das equipes de desenvolvimento de software, levando em consideração o grau de contribuição do desenvolvedor em relação aos níveis de qualidade do produto de software. Também, os desenvolvedores podem identificar qual está sendo sua interferência na qualidade do código do projeto e corrigir esta situação, caso necessário, melhorando, assegurando a qualidade do produto final e mostrando comprometimento com o produto final de software.

Em um maior grau de maturidade das empresas desenvolvedoras de software, o grau de contribuição do desenvolvedor pode vir a compor os níveis de progressão de carreira dos desenvolvedores/programadores. O setor de Recursos Humanos pode aplicar planos de cargos e salários focando, não somente no tempo de experiência do colaborador, mas também no seu nível de contribuição para o desenvolvimento de software de qualidade e grau de interesse em melhorar o produto de software desenvolvido pela organização.

Como consequência, aumentando a cooperação e contribuição entre os membros da equipe de desenvolvimento de software é possível melhorar a produtividade e qualidade do código, uma vez que serão produzidos códigos com maiores chances de facilidade de reuso, com baixo custo de manutenção e capazes de serem efetivamente testados, itens diretamente ligados a dívida técnica do produto de software. Também como consequência das análises realizadas nesta tese, acredita-se que, em alguns casos, o desenvolvedor/contribuinte não apresenta um bom ou mau estilo de programação, mas

provavelmente, ele não tenha conhecimento da influência que a alteração realizada por ele no código esteja refletindo no nível de qualidade do produto do projeto de software. A partir das informações desta tese o desenvolvedor/contribuinte pode ter acesso a esta informação e, além disto, agir sobre ela.

Uma indústria de desenvolvimento colaborativo de software que se mostra preocupada com o grau de contribuição dos desenvolvedores, com foco na melhoria e/ou manutenção dos níveis de qualidade do produto de software, além de ter uma boa visibilidade no mercado, também tem a tendência de diminuir a dívida técnica referente aos itens de complexidade, reusabilidade, testabilidade e manutenibilidade do aplicativo de software, itens estes que são considerados custosos durante o ciclo de vida do software. Desenvolvedores preocupados com sua carreira também terão interesse em colaborar nas organizações que tenham esta visão de trabalho colaborativo, pois percebem que podem crescer individualmente, mesmo trabalhando de forma colaborativa.

7.5 Trabalhos futuros

Considerando os resultados desta pesquisa, foram identificadas algumas possibilidades de desdobramentos em trabalhos futuros, tais como:

- Testar o protótipo em tempo real em um ambiente de produção industrial de desenvolvimento colaborativo de software.
- Analisar o comprometimento do desenvolvedor/contribuinte com a qualidade do produto de software. Verificando-se, a partir do *dashboard* e retorno para o desenvolvedor/contribuinte de como e em qual artefato ele pode melhorar o código-fonte para aumentar o nível de qualidade do produto de software, ele volta atrás e melhora o código.
- Relacionar os resultados da recuperação de informação das mensagens dos *commits* e *issues*, em relação a influência do gerente de projeto na alteração realizada pelo desenvolvedor, com o resultado do grau de colaboração do desenvolvedor.
- Incluir o grau de contribuição do desenvolvedor como característica do perfil do desenvolvedor conforme proposto por Beal e colegas em (Beal *et al.*, 2017).
- Analisar, testar e validar os níveis de influência nos itens de qualidade do

produto de software das demais métricas de código-fonte, que não foram utilizadas na equação do cálculo do grau de contribuição, para poder incluí-las.

Referências

ALALI, Abdulkareem; KAGDI, Huzefa; MALETIC, Jonathan I. **What's a typical commit? a characterization of open source software repositories.** In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, 2008. p. 182-191.

ALLEN, James. **Natural language understanding.** Pearson, 1995.

ALMUGRIN, Saleh; ALBATTAH, Waleed; MELTON, Austin. **Using indirect coupling metrics to predict package maintainability and testability**. Journal of systems and software, v. 121, p. 298-310, 2016.

ALTMANN, Josef; POMBERGER, Gustav**. Cooperative software development: concepts, model and tools.** In: Proceedings of Technology of Object-Oriented Languages and Systems-TOOLS 30 (Cat. No. PR00278). IEEE, 1999. p. 194-207.

AMBU, Walter et al. **Studying the evolution of quality metrics in an agile/distributed project.** In: International Conference on Extreme Programming and Agile Processes in Software Engineering. Springer, Berlin, Heidelberg, 2006. p. 85-93.

ANDERSON, J. L. **Using software tools and metrics to produce better quality test software.** In: Proceedings AUTOTESTCON 2004. IEEE, 2004. p. 293-297.

ANTTILA, Juhani. **Advanced Web 2.0 based interactive technology to support informal learning for enhancing quality of business management**. In: proceedings of the ICELW Conference, New York, NY. 2008. Disponível em www.qualityintegration.biz/Mumbai2006.html Acesso em Agosto de 2013.

DE ARAUJO, Renata Mendes; BORGES, Marcos R S. **The role of collaborative support to promote participation and commitment in software development teams.** Software Process: Improvement and Practice, v. 12, n. 3, p. 229-246, 2007.

BAKER, Kevin; GREENBERG, Saul; GUTWIN, Carl. **Empirical development of a heuristic evaluation methodology for shared workspace groupware**. In: Proceedings of the 2002 ACM conference on Computer supported cooperative work. ACM, 2002. p. 96-105.

BANNON, L. Schmidt. K. **CSCW: Four characters in search of a context**. Studies in Computer Supported Cooperative Work: Theory, Practice and Design. Amsterdam: North Holland, p. 3-16, 1991.

BASILI, Victor R. **Software modeling and measurement: The Goal/Question/Metric paradigm.** 1992.

BASSI, Patricia Rücker et al. **Measuring Developers' Contribution in Source Code using Quality Metrics**. In: 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD)). IEEE, 2018. p. 39-44.

BEAL, Franciele. **Um método para a construção do perfil dinâmico do desenvolvedor no desenvolvimento colaborativo de software**. 2013. Qualificação de Mestrado. Pontifícia Universidade Católica do Paraná.

BEAL, Franciele; BASSI, Patricia Rücker; PARAISO, Emerson Cabrera. **Developer Modelling using Software Quality Metrics and Machine Learning**. In: ICEIS (1). 2017. p. 424-432.

BEHNAMGHADER, Pooyan et al. **Towards better understanding of software quality evolution through commit-impact analysis**. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2017. p. 251-262.

BHATT, Pankaj et al. **Influencing factors in outsourced software maintenance**. ACM SIGSOFT Software Engineering Notes, v. 31, n. 3, p. 1-6, 2006.

BRERETON, Pearl et al. **Lessons from applying the systematic literature review process within the software engineering domain**. Journal of systems and software, v. 80, n. 4, p. 571-583, 2007.

BRUNTINK, Magiel; VAN DEURSEN, Arie. **Predicting class testability using object-oriented metrics.** In: Source Code Analysis and Manipulation, Fourth IEEE International Workshop on. IEEE, 2004. p. 136-145.

BitBucket Disponível em www.bitbucket.org Acesso em Janeiro de 2017.

CAMPAGNOLO, Bruno et al. **An architecture for supporting small collocated teams in cooperative software development**. In: 2009 13th International Conference on Computer Supported Cooperative Work in Design. IEEE, 2009. p. 264-269.

The Cambridge English Dictionary. Disponível em: www.dictionary.cambridge.org. Acesso em Janeiro de 2016.

CAPITÁN, Lorena; VOGEL-HEUSER, Birgit. **Metrics for software quality in automated production systems as an indicator for technical debt**. In: 2017 13th IEEE Conference on Automation Science and Engineering (CASE). IEEE, 2017. p. 709-716.

CARD, D.N. SCALZO, B. Measurement for object-oriented software projects. 6th International Symposium on Software Metrics. 1999.

CHAUHAN, Ritu et al. **Estimation of Software Quality using Object Oriented Design Metrics.** International Journal of Innovative Research in Computer and Communication Engineering, v. 2, n. 1, p. 2581-2586, 2014.

CHESBROUGH, Henry W. **The era of open innovation**. Managing innovation and change, v. 127, n. 3, p. 34-41, 2006.

CHIDAMBER, Shyam R.; KEMERER, Chris F. **Towards a metrics suite for object-oriented design**. 1991.

CHOWDHURY, Gobinda G**. Introduction to modern information retrieval**. Facet publishing, 2010.

CHU-CARROLL, Mark C.; SPRENKLE, Sara. **Coven: Brewing better collaboration through software configuration management.** IBM Thomas J. Watson Research Division, 2000.

CLARET, M. D. **Métricas para colaboração em processos de negócios**. Dissertação de Mestrado, Informática, UniRio, 2012

CMMI – Capability Maturity Model Integration. Disponível em www.cmmiinstitute.com. Acesso em Janeiro de 2014.

CONTANDRIOPOULOS, André-Pierre *et al*. **Saber preparar uma pesquisa**. 3. ed. São Paulo/Rio de Janeiro: HUCITEC/Abrasco, 1999.

COOK, C. CHURCHER, N. **Modeling and Measuring Collaborative Software Engineering.** Proceedings of ACSC2005. 28th Computer Science Conference, volume 38 of Conferences in Research and Practice in Information Technology, p. 267-276, 2005.

CUGINI, John et al. **Methodology for evaluation of collaboration systems**. The evaluation working group of the DARPA intelligent collaboration and visualization program, Rev, v. 3, 1997.

DAGPINAR, Melis; JAHNKE, Jens H. **Predicting maintainability with object-oriented metrics-an empirical comparison**. In: 10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings. IEEE, 2003. p. 155-164.

DAMIAN, Daniela; KWAN, Irwin; MARCZAK, Sabrina. **Requirements-driven collaboration: Leveraging the invisible relationships between requirements and people**. In: Collaborative software engineering. Springer, Berlin, Heidelberg, 2010. p. 57-76.

DASH, Yajnaseni; DUBEY, Sanjay Kumar; RANA, Ajay. **Maintainability prediction of object-oriented software system by using artificial neural network approach.** International Journal of Soft Computing and Engineering (IJSCE), v. 2, n. 2, p. 420-423, 2012.

DE SOUZA, C. R. B. MARCZAK, S. PRIKLADNICKI, R. **Desenvolvimento colaborativo de software.** In: Sistemas Colaborativos, Elsevier Editora Ltda, 2012.

DIAS, A. F. **Conceitos Básicos de Controle de Versão de Software - Centralizado e Distribuído**. 2009. Disponível em http://www.xmarks.com/s/site/www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/conceitos_basicos_controle_versao_centralizado_e_distribuido.php. Acesso em Abril de 2016.

DODDS, Peter Sheridan et al. **Temporal patterns of happiness and information in a global social network: Hedonometrics and Twitter.** PloS one, v. 6, n. 12, p. e26752, 2011. Disponível em www.plossone.org Acesso em Janeiro de 2018.

DROUIN, Nicholas; BADRI, Mourad; TOURÉ, Fadel. **Analyzing software quality evolution using metrics: An empirical study on open source software**. Journal of software, v. 8, n. 10, p. 2462-2473, 2013.

ELLIS, Clarence A.; GIBBS, Simon J.; REIN, Gail**. Groupware: some issues and experiences**. Communications of the ACM, v. 34, n. 1, p. 39-58, 1991.

ESTUBLIER, Jacky et al. **Impact of software engineering research on the practice of software configuration management**. ACM Transactions on Software Engineering and Methodology (TOSEM), v. 14, n. 4, p. 383-430, 2005.

FAGUNDES, I.C. SATO, L.S. MAÇADA, D.L. **Aprendizes do Futuro – as inovações já começaram**. Brasilia. MEC. 1999.

FENTON, Norman E.; NEIL, Martin. **Software metrics: roadmap**. In: Proceedings of the Conference on the Future of Software Engineering. ACM, 2000. p. 357-370.

FERREIRA, Kecia AM et al. **Identifying thresholds for object-oriented software metrics.** Journal of Systems and Software, v. 85, n. 2, p. 244-257, 2012.

FILÓ, Tarcısio GS; BIGONHA, Mariza; FERREIRA, Kecia. **A catalogue of thresholds for object-oriented software metrics.** Proceedings of the 1st SOFTENG, p. 48-55, 2015.

FINLAY, Jacqui; PEARS, Russel; CONNOR, Andy M. **Data stream mining for predicting software build outcomes using source code metrics**. Information and Software Technology, v. 56, n. 2, p. 183-198, 2014.

FONTANA, Francesca Arcelli; FERME, Vincenzo; SPINELLI, Stefano. **Investigating the impact of code smells debt on quality code evaluation.** In: Proceedings of the Third International Workshop on Managing Technical Debt. IEEE Press, 2012. p. 15-22.

FOUCAULT, Matthieu; FALLERI, Jean-Rémy; BLANC, Xavier. **Code ownership in open-source software.** In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM, 2014. p. 39.

FRANCONI, E. **Description Logics for Natural Language Processing**. Description Logics Handbook. Cambridge University Press. 2001.

FRANZAGO, Mirco et al. **Collaborative model-driven software engineering: a classification framework and a research map**. IEEE Transactions on Software Engineering, v. 44, n. 12, p. 1146-1175, 2018.

GALLIANO, G. **O método científico: teoria e prática**. São Paulo: Mosaico, 1979.

GHTorrent. Disponível em http://ghtorrent.org/ Acesso em Março/2018.

GIL, A. C. **Como elaborar projetos de pesquisa.** 4ª edição. São Paulo: Atlas, 2002.

GitHub. Disponível em www.github.com Acesso em Novembro de 2017.

GitHub Guides. Disponível em https://guides.github.com/features/issues/ Acesso em Novembro de 2017.

GitLab. Disponível em www.gitlab.com Acesso em Novembro de 2017.

GÓMEZ, Verónica Uquillas; DUCASSE, Stéphane; D'HONDT, Theo. **Visually characterizing source code changes.** Science of Computer Programming, v. 98, p. 376-393, 2015.

GOYAL, Rinkaj; CHANDRA, Pravin; SINGH, Yogesh. **Why interaction between metrics should be considered in the development of software quality models: a preliminary study.** ACM SIGSOFT Software Engineering Notes, v. 39, n. 4, p. 1-4, 2014.

GRAVES, Liana N. **Creating a community context for cooperative learning**. Handbook of cooperative learning methods, p. 283-299, 1994.

GRUDIN, Jonathan. **Computer-supported cooperative work: History and focus.** Computer, v. 27, n. 5, p. 19-26, 1994.

SADEGH, Mohammad; IBRAHIM, Roliana; OTHMAN, Zulaiha Ali. **Opinion mining and sentiment analysis: A survey.** International Journal of Computers & Technology, v. 2, n. 3, p. 171-178, 2012.

HARRISON, R. COUNSELL, S. NITHI, R. **An overview of oriented-object design metrics**. International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering, p. 230-235, 1997.

SELLERS, Brian H. **Object-oriented metrics: Measures of complexity.** PH PTR, New Jersey, 1996.

HONGLEI, Tu; WEI, Sun; YANAN, Zhang. **The research on software metrics and software complexity metrics**. In: 2009 International Forum on Computer Science-Technology and Applications. IEEE, 2009. p. 131-136.

HORSTMANN, C. **Big Java.** Porto Alegre: Bookman, 2004.

IEEE Std. 610.12-1990. ***Standard Glossary of Software Engineering Terminology***, IEEE Computer Society Press, Los Alamitos, CA, 1990.

ISO – International Organization for Standardization. Disponível em www.iso.org/ Acesso em Março de 2017.

JERMAKOVICS, Andrejs; SCOTTO, Marco; SUCCI, Giancarlo. **Visual identification of software evolution patterns**. In: Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting. ACM, 2007. p. 27-30.

JOHNSON, P. **Collaborative software development laboratory – LEAP**. Department of Information and Computer Sciences at the University of Hawaii. 1997.

JONGELING, Robbert et al. **On negative results when using sentiment analysis tools for software engineering research.** Empirical Software Engineering, v. 22, n. 5, p. 2543-2584, 2017.

KALLIAMVAKOU, Eirini et al. **Measuring Developer Contribution from Software Repository Data.** MCIS, v. 2009, p. 4th, 2009.

KARUS, Siim; DUMAS, Marlon. **Code churn estimation using organizational and code metrics: An experimental comparison**. Information and software technology, v. 54, n. 2, p. 203-211, 2012.

KITCHENHAM, B. CHARTERS, S. **Guidelines for performing systematic literature reviews in software engineering.** Software Engineering. Version 2.3 - EBSE-2007-01, Durham University, 2007. Disponível em: http//www.dur.ac.uk/ebse/resource/guidelines/Systematic-reviews-5-8.pdf. Acesso em Maio de 2014.

KITCHENHAM, Barbara. **What's up with software metrics?–A preliminary mapping study.** Journal of systems and software, v. 83, n. 1, p. 37-51, 2010.

KLING, Rob. **Cooperation, coordination and control in computer-supported work.** Communications of the ACM, v. 34, n. 12, p. 83-88, 1991.

KONONENKO, Oleksii; BAYSAL, Olga; GODFREY, Michael W. **Code review quality: how developers see it.** In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016. p. 1028-1038.

LAVALLÉE, Mathieu; ROBILLARD, Pierre N. **The impacts of software process improvement on developers: A systematic review.** In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012. p. 113-122.

LEE, Young; YANG, Jeong; CHANG, Kai H. **Metrics and evolution in open source software.** In: Seventh International Conference on Quality Software (QSIC 2007). IEEE, 2007. p. 191-197.

LEHMAN, Meir M. et al. **Metrics and laws of software evolution-the nineties view**. In: Proceedings Fourth International Software Metrics Symposium. IEEE, 1997. p. 20-32.

LI, Huan. **A novel coupling metric for object-oriented software systems**. In: 2008 IEEE International Symposium on Knowledge Acquisition and Modeling Workshop. IEEE, 2008. p. 609-612.

LIN, Bin et al. **Sentiment analysis for software engineering: How far can we go?** In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018. p. 94-104.

MAGDALENO, Andréa Magalhães et al. **Collaboration optimization in software process composition**. Journal of Systems and Software, v. 103, p. 452-466, 2015.

MALONE, Thomas W.; MALONE, Thomas W.; CROWSTON, Kevin. **The interdisciplinary study of coordination**. ACM Computing Surveys (CSUR), v. 26, n. 1, p. 87-119, 1994.

MARCONI, M. A. LAKATOS, E. M. **Fundamentos de metodologia científica**. 7.ed. São Paulo: Atlas, 2007.

MARLOWE, T. NOUSALA, S. JASTROCH, N. KIROVA, V. **The Collaborative Future**. Journal of Systemics, Cybernetics and Informatics, v. 9, n. 5, p. 1-5, 2011.

MARTIN, Robert. **OO design quality metrics-an analysis of dependencies**. In: Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94. 1994. Disponível em http://www.objectmentor.com/resources/articles/oodmetrc.pdf. Acesso em Agosto de 2013.

MARTIN, Robert C. **Agile software development: principles, patterns, and practices.** Prentice Hall, 2002.

MATSUMOTO, Shinsuke et al. **An analysis of developer metrics for fault prediction**. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, 2010. p. 18.

MATTSSON, S. **Measures of Collaboration in CSCW: Usability and collective measures in remote and co-located problem-solving** Dissertação de mestrado em Intelligence System Design, University of Gothenburg, Sweden, 2011

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

MEHDI, Ahmed-Nacer; URSO, Pascal; CHAROY, François. **Evaluating software merge quality**. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM, 2014. p. 9.

MENS, Tom; DEMEYER, Serge. **Future trends in software evolution metrics**. In: Proceedings of the 4th international workshop on Principles of software evolution. ACM, 2001. p. 83-86.

MIDDLETON, Justin et al. **Which contributions predict whether developers are accepted into github teams.** In: Proceedings of the 15th International Conference on Mining Software Repositories. ACM, 2018. p. 403-413.

MIJAÉ, Marko; STAPIÉ, Zlatko. **Reusability metrics of software components: survey**. In: 26th Central European Conference on Information and Intelligent Systems (CECIIS 2015). 2015.

MISRA, Sanjay; AKMAN, Ibrahim. **Weighted class complexity: a measure of complexity for object-oriented system**. Journal of Information Science and Engineering, v. 24, p. 1689-1708, 2008.

MISTRÍK, Ivan et al. **Collaborative software engineering: challenges and prospects**. In: Collaborative Software Engineering. Springer, Berlin, Heidelberg, 2010. p. 389-403.

MITCHELL, Lewis et al. **The geography of happiness: Connecting twitter sentiment and expression, demographics, and objective characteristics of place**. PloS one, v. 8, n. 5, p. e64417, 2013. Disponível em www.plossone.org Acesso em Janeiro de 2018.

Moderno Dicionário Português Michaelis. Disponível em: www.michaelis.uol.com.br. Acesso em Janeiro de 2016.

MOECKEL, Alexandre; FORCELLINI, Fernando Antonio. **Framework to support collaboration and knowledge management in the product pre-development**. In: 19th International Congress of Mechanical Engineering. 2007.

MORCH, A I. **Computer-supported cooperative work: an introduction**. InterMedia, University of Oslo, Norway. 2012.

MOHTASHAMI, Mojgan; MARLOWE, Thomas J.; KU, Cyril S. **Metrics are needed for collaborative software development**. Journal of Systemics, Cybernetics, and Informatics, v. 9, n. 5, p. 41-47, 2011.

MPS.BR, Melhoria do Processo de Software Brasileiro. Disponível em https://softex.br/mpsbr/. Acesso em Maio de 2019.

MUNSON, John C.; ELBAUM, Sebastian G. **Code churn: A measure for estimating the impact of code change**. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, 1998. p. 24-31.

MURTA, Leonardo Gresta P.; WERNER, Claudia Maria L.; ESTUBLIER, Jacky. **The configuration management role in collaborative software engineering**.

In: Collaborative Software Engineering. Springer, Berlin, Heidelberg, 2010. p. 179-194.

NAKAKOJI, Kumiyo; YE, Yunwen; YAMAMOTO, Yasuhiro. **Supporting expertise communication in developer-centered collaborative software development environments**. In: Collaborative Software Engineering. Springer, Berlin, Heidelberg, 2010. p. 219-236.

NITZKE, Julio; CARNEIRO, Mara; GELLER, Marlise. **Aprendizagem cooperativa/colaborativa apoiada por computador** (**ACAC**). Trabalho apresentado no SBIE, 1999.

OLAGUE, Hector M.; ETZKORN, Letha H.; COX, Glenn W. **An Entropy-Based Approach to Assessing Object-Oriented Software Maintainability and Degradation-A Method and Case Study**. In: Software Engineering Research and Practice. 2006. p. 442-452.

OLIVEIRA, Marcio FS et al. **Software quality metrics and their impact on embedded software**. In: 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software. IEEE, 2008. p. 68-77.

OLIVEIRA, S. L. DE. **Tratado de metodologia científica**. Projetos de pesquisas, TGI, TCC, monografias, dissertações e teses. São Paulo: Pioneira, 2001.

PANG, Bo et al. **Opinion mining and sentiment analysis**. Foundations and Trends® in Information Retrieval, v. 2, n. 1–2, p. 1-135, 2008.

PERKUSICH, Mirko et al. **A Bayesian network approach to assist on the interpretation of software metrics.** In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, 2015. p. 1498-1503.

PIMENTEL, Mariano et al. **Modelo 3C de Colaboração para o desenvolvimento de Sistemas Colaborativos**. Anais do III Simpósio Brasileiro de Sistemas Colaborativos, p. 58-67, 2006.

PINTO, Gustavo; STEINMACHER, Igor; GEROSA, Marco Aurélio. **More common than you think: An in-depth study of casual contributors**. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2016. p. 112-123.

PLOSCH, Reinhold et al. **A method for continuous code quality management using static analysis.** In: 2010 Seventh International Conference on the Quality of Information and Communications Technology. IEEE, 2010. p. 370-375.

RAHMAN, Foyzur; DEVANBU, Premkumar. **How, and why, process metrics are better.** In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013. p. 432-441.

RAWAT, Mrinal Singh; MITTAL, Arpita; DUBEY, Sanjay Kumar. **Survey on impact of software metrics on software quality**. IJACSA International Journal of Advanced Computer Science and Applications, v. 3, n. 1, 2012.

RAYMOND, Eric. **The cathedral and the bazaar**. Knowledge, Technology & Policy, v. 12, n. 3, p. 23-49, 1999.

RIAZ, Mehwish; MENDES, Emilia; TEMPERO, Ewan. **A systematic review of software maintainability prediction and metrics**. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, 2009. p. 367-377.

RICHARDSON, Ita et al. Global software engineering: A software process approach. In: Collaborative software engineering. Springer, Berlin, Heidelberg, 2010. p. 35-56.

ROBBINS, Harvey; FINLEY, Michael. **The new why teams don't work: What goes wrong and how to make it right**. Berrett-Koehler Publishers, 2000.

ROBILLARD, Pierre N.; ROBILLARD, Martin P. **Types of collaborative work in software engineering**. Journal of Systems and Software, v. 53, n. 3, p. 219-224, 2000.

ROBINSON, Hugh; SHARP, Helen. **Collaboration, communication and co-ordination in agile software development practice**. In: Collaborative software engineering. Springer, Berlin, Heidelberg, 2010. p. 93-108.

ROSCHELLE, Jeremy; TEASLEY, Stephanie D. **The construction of shared knowledge in collaborative problem solving**. In: Computer supported collaborative learning. Springer, Berlin, Heidelberg, 1995. p. 69-97.

ROSENBERG, Linda H.; STAPKO, Ruth; GALLO, Albert. **Risk-based object oriented testing**. 24th SWE, 1999. Disponível em http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.7509&rep=rep1&type=pdf. Acesso em Agosto de 2013.

RUSSELL, S. J. NORVIG, P. **Artificial Intelligence – a modern approach**. Prentice Hall, 1995.

SACKMAN, Harold; ERIKSON, Warren J.; GRANT, E. Eugene. **Exploratory experimental studies comparing online and offline programing performance**. SYSTEM DEVELOPMENT CORP SANTA MONICA CA, 1968.

SAE-LIM, Natthawute; HAYASHI, Shinpei; SAEKI, Motoshi. **An Investigative Study on How Developers Filter and Prioritize Code Smells**. IEICE TRANSACTIONS on Information and Systems, v. 101, n. 7, p. 1733-1742, 2018.

SARMA, Anita et al. **Continuous coordination tools and their evaluation**. In: Collaborative Software Engineering. Springer, Berlin, Heidelberg, 2010. p. 153-178.

SATO, Danilo; GOLDMAN, Alfredo; KON, Fabio. **Tracking the evolution of object-oriented quality metrics on agile projects**. In: International Conference on Extreme Programming and Agile Processes in Software Engineering. Springer, Berlin, Heidelberg, 2007. p. 84-92.

SAVCHENKO, D. HYNNINEN, T. TAIPALE, O. **Code quality measurement: case study**. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), p.1455-1459, 2018.

SenticNet disponível em http://sentic.net/projects/ Acesso em Novembro de 2017.

SentiStrenght disponível em http://sentistrength.wlv.ac.uk/ Acesso em Novembro de 2017.

SentiWordNet disponível em http://sentiwordnet.isti.cnr.it/ Acesso em Novembro de 2017.

SOLLA, Marwa; PATEL, Ahmed; WILLS, Christopher. **New metric for measuring programmer productivity.** In: 2011 IEEE Symposium on Computers & Informatics. IEEE, 2011. p. 177-182.

STAHL, G. KOSHMANN, T. SUTHERS, D. **CSCL: A history perspective.** 2006. Disponível em www.cis.drexel.edu/faculty/gerry/cscl. Acesso em Março de 2015.

SCHROEDER, Mark. **A practical guide to object-oriented metrics**. IT professional, v. 1, n. 6, p. 30-36, 1999.

SCHWIND, Michael; WEGMANN, Christian. **SVNNAT: Measuring collaboration in software development networks**. In: 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services. IEEE, 2008. p. 97-104.

SHAIKH, Tahura; DESHPANDE, Deepa. **A review on opinion mining and sentiment analysis**. International Journal of Computer Applications, v. 975, p. 8887, 2016.

DE SILVA, D. I.; KODAGODA, N.; PERERA, H. **Applicability of three complexity metrics**. In: International Conference on Advances in ICT for Emerging Regions (ICTer2012). IEEE, 2012. p. 82-88.

SILVA, M. C. BASSI, P. R. WANDERLEY, G. M. P. TACLA, C. A. PARAISO, E. C. **A Visual Tool for Supporting Collaborative Code Quality**, 23rd International Conference on Computer Supported Cooperative Work in Design – CSCWD, p. 368-373, 2019.

SINGH, Gagandeep. **Metrics for measuring the quality of object-oriented software**. ACM SIGSOFT Software Engineering Notes, v. 38, n. 5, p. 1-5, 2013.

SHAH, Chirag. **Working in collaboration-what, why, and how**. In: Proceedings of collaborative information retrieval workshop at CSCW 2010. 2010.

SMITH, Karl A. **Cooperative learning: Making "groupwork" work**. New directions for teaching and learning, v. 1996, n. 67, p. 71-82, 1996.

SOLIMAN, Taysir Hassan A.; EL-SWESY, Adel; AHMED, Saddam Hussein. **Utilizing ck metrics suite to uml models: A case study of microarray midas software**. In: 2010 The 7th International Conference on Informatics and Systems (INFOS). IEEE, 2010. p. 1-6.

TAHIR, Amjed et al. **Can you tell me if it smells? A study on how developers discuss code smells and anti-patterns in Stack Overflow**. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. ACM, 2018. p. 68-78.

TAIBI, Fathi. **Reusability of open-source program code: a conceptual model and empirical investigation**. ACM SIGSOFT Software Engineering Notes, v. 38, n. 4, p. 1-5, 2013.

TAPSCOTT, Don; WILLIAMS, Anthony D. **Wikinomics: How mass collaboration changes everything**. Penguin, 2008.

The Thesaurus Dictionary. Disponível em http://www.thesaurus.com/ Acesso em Março de 2018.

THOMSON, Ann Marie; PERRY, James L.; MILLER, Theodore K. **Conceptualizing and measuring collaboration.** Journal of Public Administration Research and Theory, v. 19, n. 1, p. 23-56, 2009.

UMARJI, Medha; SEAMAN, Carolyn. **Gauging acceptance of software metrics: Comparing perspectives of managers and developers**. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, 2009. p. 236-247.

VAN DER HOEK, André; SARMA, Anita. **Palantír: raising awareness among configuration management workspaces.** In: 25th International Conference on Software Engineering, 2003. Proceedings. IEEE, 2003. p. 444-454.

VARELA, Alberto S. N. et al. **Source code metrics: A systematic mapping study**. Journal of Systems and Software, v. 128, p. 164-197, 2017.

VYTOVTOV, Petr; MARKOV, Evgeny. **Source code quality classification based on software metrics**. In: 2017 20th Conference of Open Innovations Association (FRUCT). IEEE, 2017. p. 505-511.

WALLACE, Linda G.; SHEETZ, Steven D. **The adoption of software measures: A technology acceptance model (TAM) perspective**. Information & Management, v. 51, n. 2, p. 249-259, 2014.

WANDERLEY, Gregory Moro Puppi et al**. An advanced collaborative environment for software development**. In: 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 2016. p. 002917-002922.

WARNARS, Harco Leslie Hendric Spits et al. **Object-oriented metrics to measure the quality of software upon PHP source code with PHP_depend study case request online system application**. In: 2017 International Conference on Applied Computer and Communication Technologies (ComCom). IEEE, 2017. p. 1-5.

WASHIZAKI, Hironori; YAMAMOTO, Hirokazu; FUKAZAWA, Yoshiaki. **A metrics suite for measuring reusability of software components**. In: Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717). IEEE, 2004. p. 211-223.

WHITEHEAD, Jim. **Collaboration in software engineering: A roadmap**. In: Future of Software Engineering (FOSE'07). IEEE, 2007. p. 214-225.

WHITEHEAD, Jim et al. **Collaborative software engineering: concepts and techniques**. In: Collaborative Software Engineering. Springer, Berlin, Heidelberg, 2010. p. 1-30.

WINER, Michael; RAY, Karen. **Collaboration Handbook: Creating, Sustaining, and Enjoying the Journey**. Amherst H. Wilder Foundation, 919 Lafond, St. Paul, MN 55104., 1994.

XIE, Guowu; CHEN, Jianbo; NEAMTIU, Iulian. **Towards a better understanding of software evolution: An empirical study on open source software**. In: 2009 IEEE International Conference on Software Maintenance. IEEE, 2009. p. 51-60.

YU, Liguo; RAMASWAMY, Srini; NAIR, Anil. **Using bug reports as a software quality measure**. 16th International Conference on Information Quality ICIQ, p. 277-286, 2011.

YU, Sheng; ZHOU, Shijie. **A survey on metric of software complexity**. In: 2010 2nd IEEE International Conference on Information Management and Engineering. IEEE, 2010. p. 352-356.

Apêndice A – Gráficos das métricas do experimento 1

Na sequência são apresentados diversos gráficos referentes a métricas de código-fonte dos *commits* realizados pelos desenvolvedores que contribuíram no Projeto Elastic/Search.

Os Gráficos A.1, A.2 e A.3 apresentam as variações da métrica WMC em diferentes classes do Projeto Elastic/Search. Os resultados destes gráficos permitiram as análises apresentadas no Experimento 1 desta pesquisa.

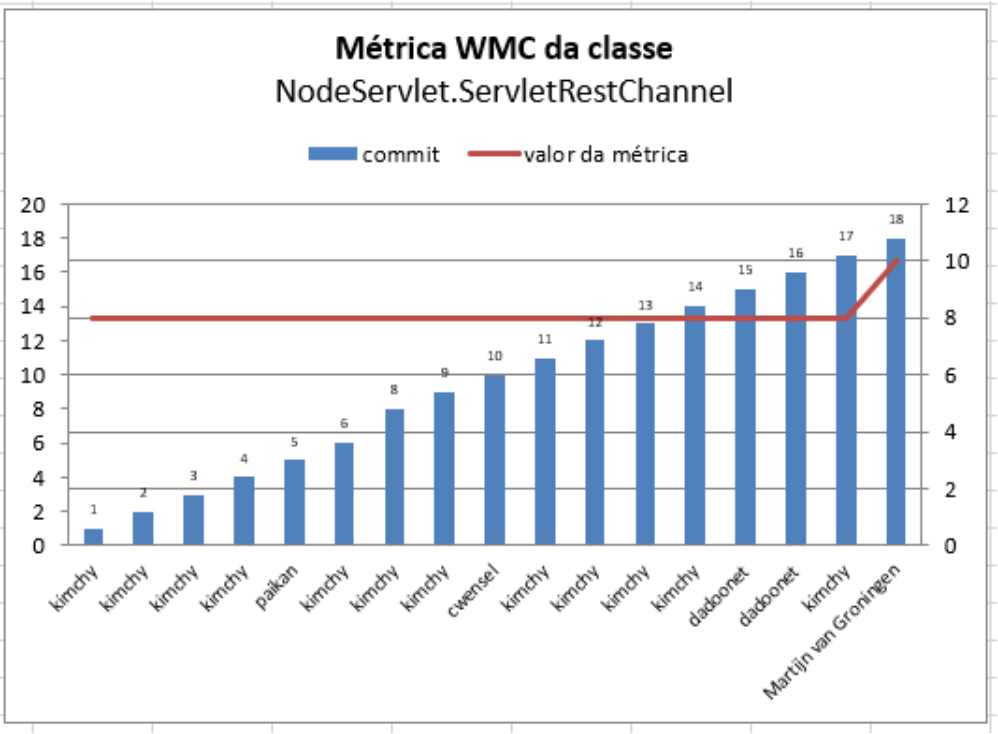


Gráfico A.1: Métrica WMC da classe NodeServlet.ServletRestChannel do Projeto Elastic/Search

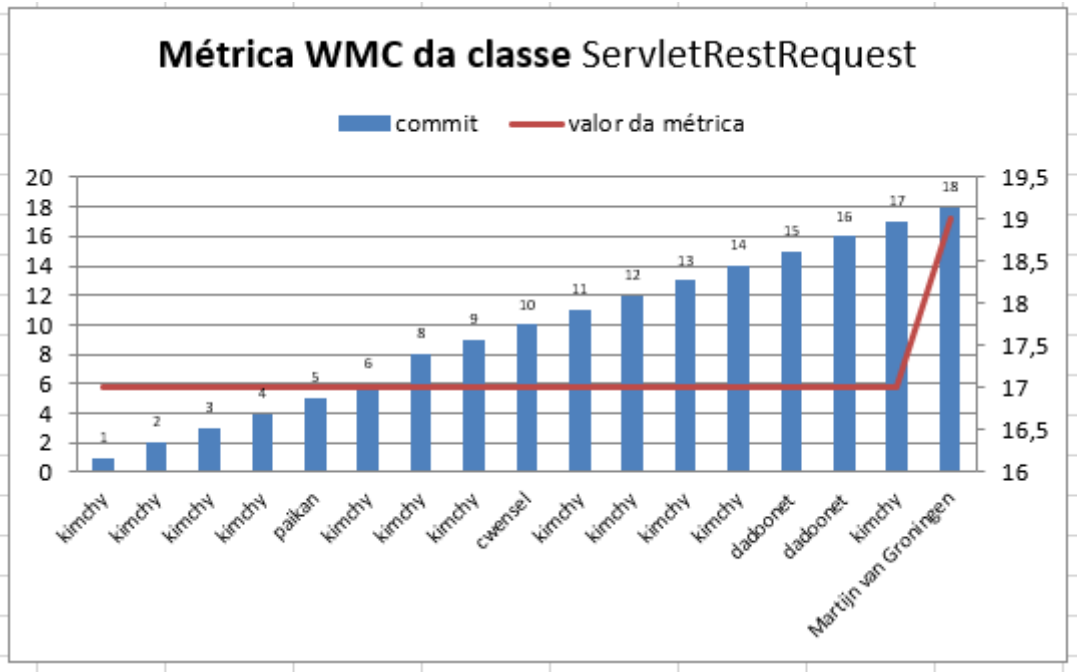


Gráfico A.2: Métrica WMC da classe ServletRestChannel do Projeto Elastic/Search

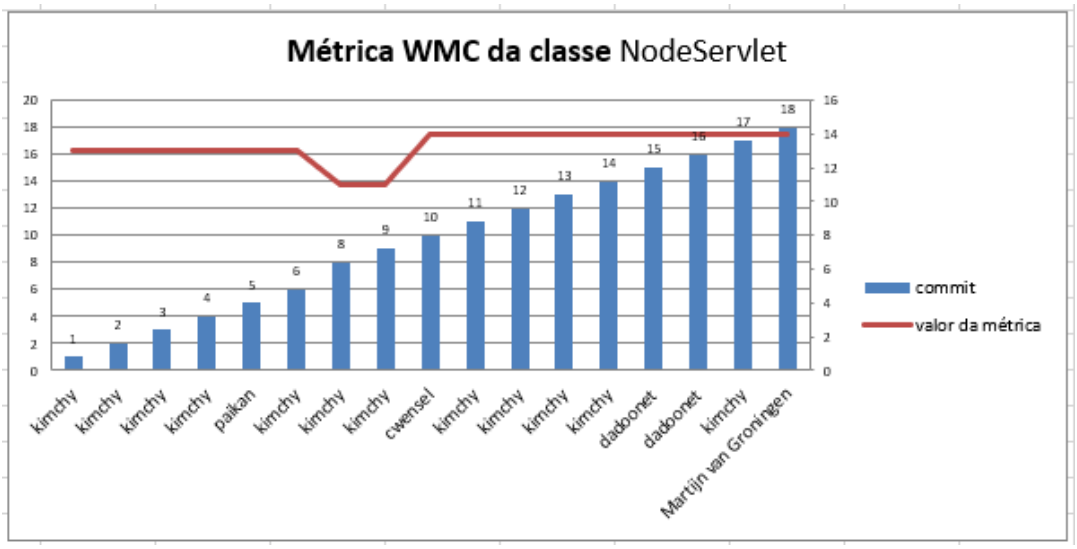


Gráfico A.3: Métrica WMC da classe NodeServlet do Projeto Elastic/Search

Os Gráficos A.4 e A.5 apresentam a métrica CCM de dois métodos do Projeto Elastic/Search, que também foram analisados no experimento 1 deste trabalho.

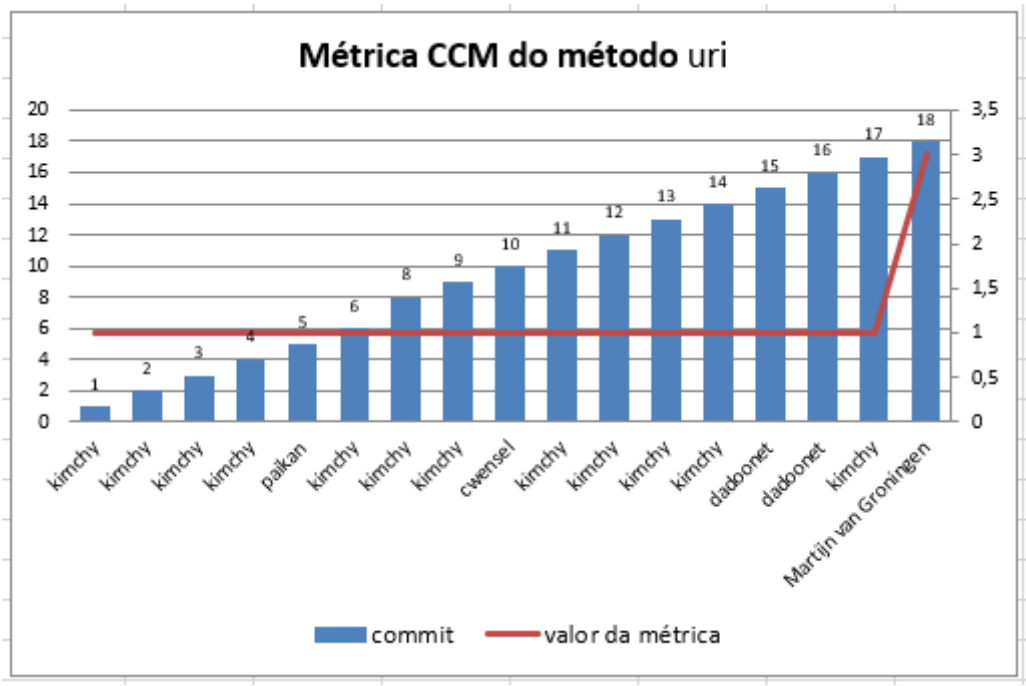


Gráfico A.4: Métrica CCM ou VG do método Uri do Projeto Elastic/Search

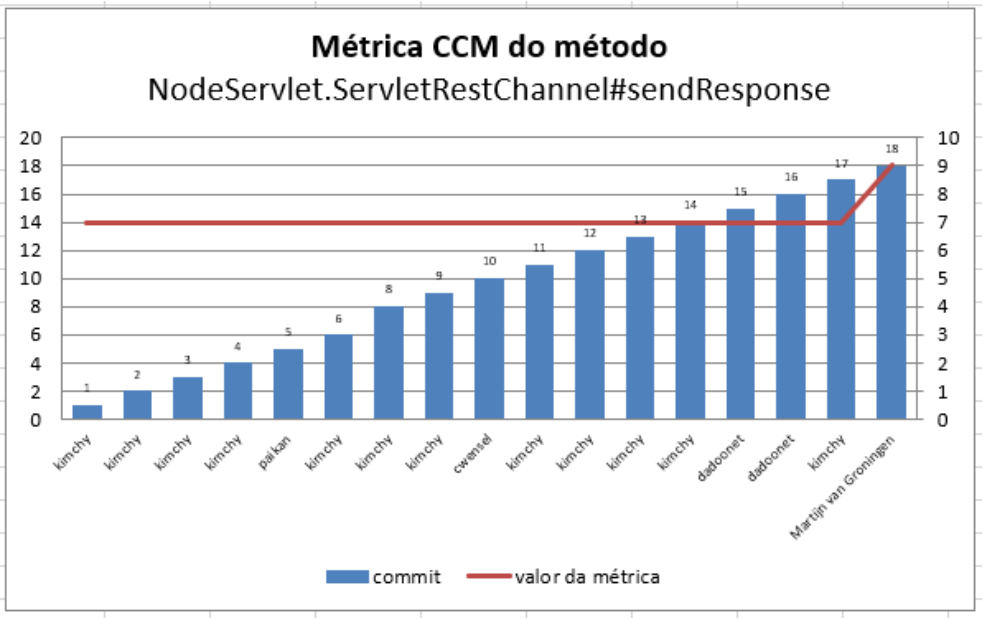


Gráfico A.5: Métrica CCM ou VG do método NodeServlet.ServletRestChannel#sendResponse do Projeto Elastic/Search

As métricas Ce, Ca e RMI medem o acoplamento entre as classes do pacote. A Ce mede o acoplamento eferente, que representa a contagem de quantas classes diferentes a classe atual faz referência por meio de campos ou parâmetros. A Ca trata do acoplamento aferente, representando a contagem de quantas classes diferentes referem-se à classe atual

por meio de campos ou parâmetros. Já a RMI (ou I) é o indicador de instabilidade. Este indicador é obtido a partir dos valores de Ce e Ca, conforme descrito no Capítulo 2 deste trabalho, RMI = Ce / (Ca + Ce). O intervalo para o RMI é de 0 a 1, com RMI = 0 indicando um pacote estável e RMI = 1 indicando um pacote instável.

Os Gráficos A.6, A.7 e A.8 apresentam as variações de Ce, Ca e RMI, respectivamente, no pacote do Projeto Elastic. Estas variações foram analisadas no experimento 1 desta pesquisa.

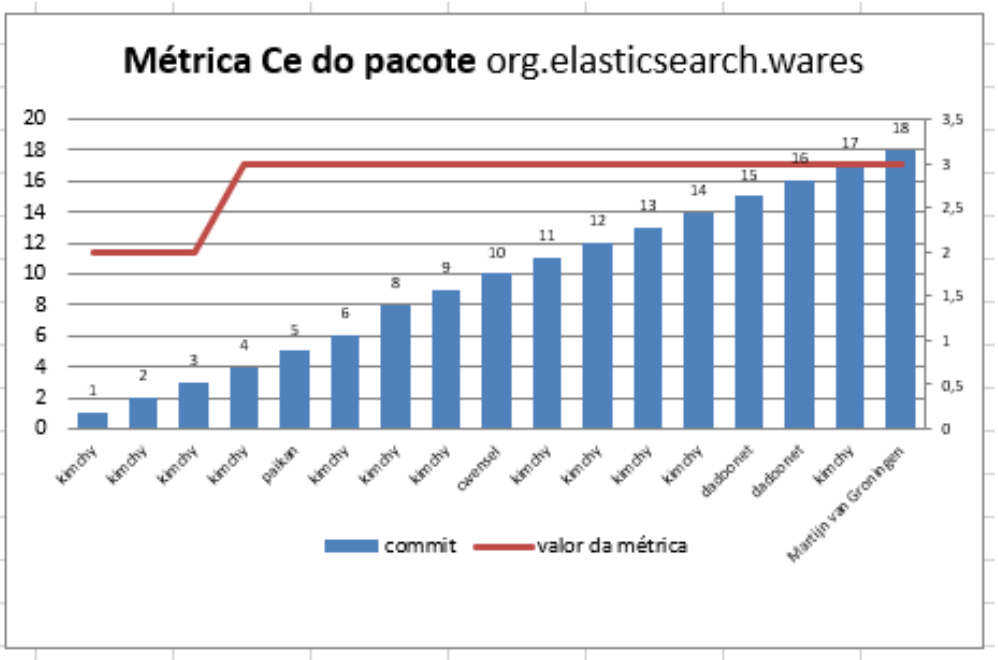


Gráfico A.6: Métrica Ce do pacote org.elasticsearch.wares do Projeto Elastic/Search

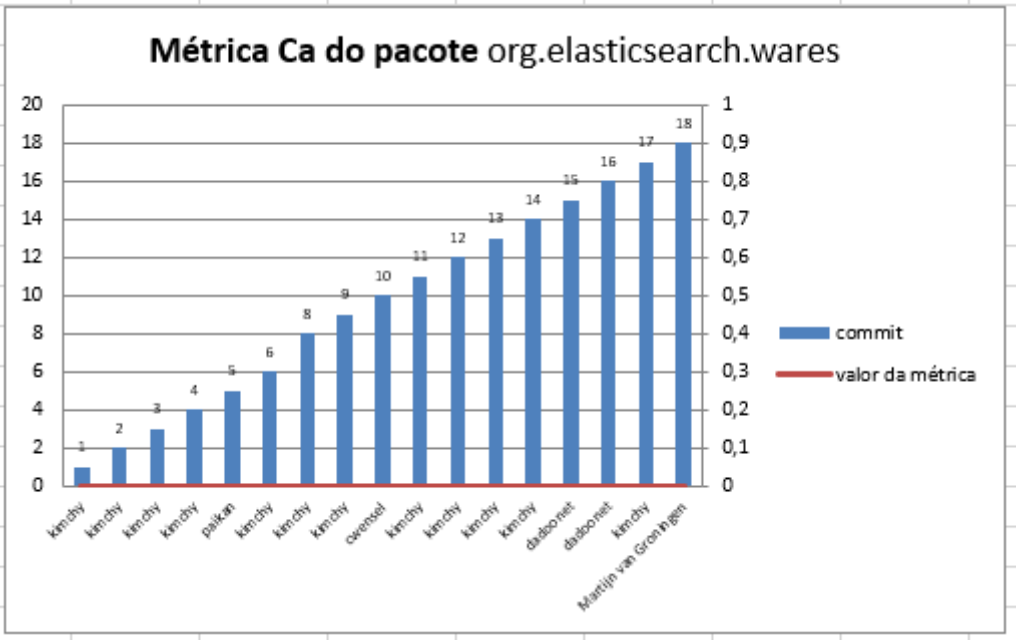


Gráfico A.7: Métrica Ca do pacote org.elasticsearch.wares do Projeto Elastic/Search

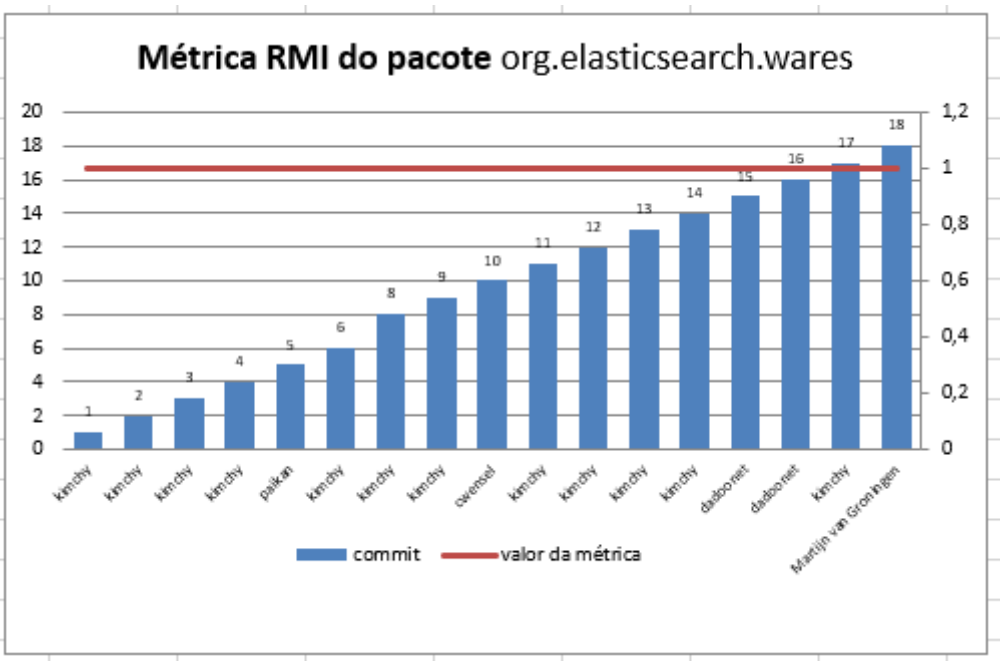


Gráfico A.8: Métrica RMI ou I do pacote org.elasticsearch.wares do Projeto Elastic/Search

As análises realizadas nas métricas do Projeto Elastic/Search permitiram perceber que os *commits* 8 e 18 foram os que mais impactaram nas métricas do projeto de forma geral. Os Gráficos A.9 e A.10 apresentam a variação de diversas métricas nos *commits* 8 e 18, respectivamente, que afetaram diversos artefatos do projeto.

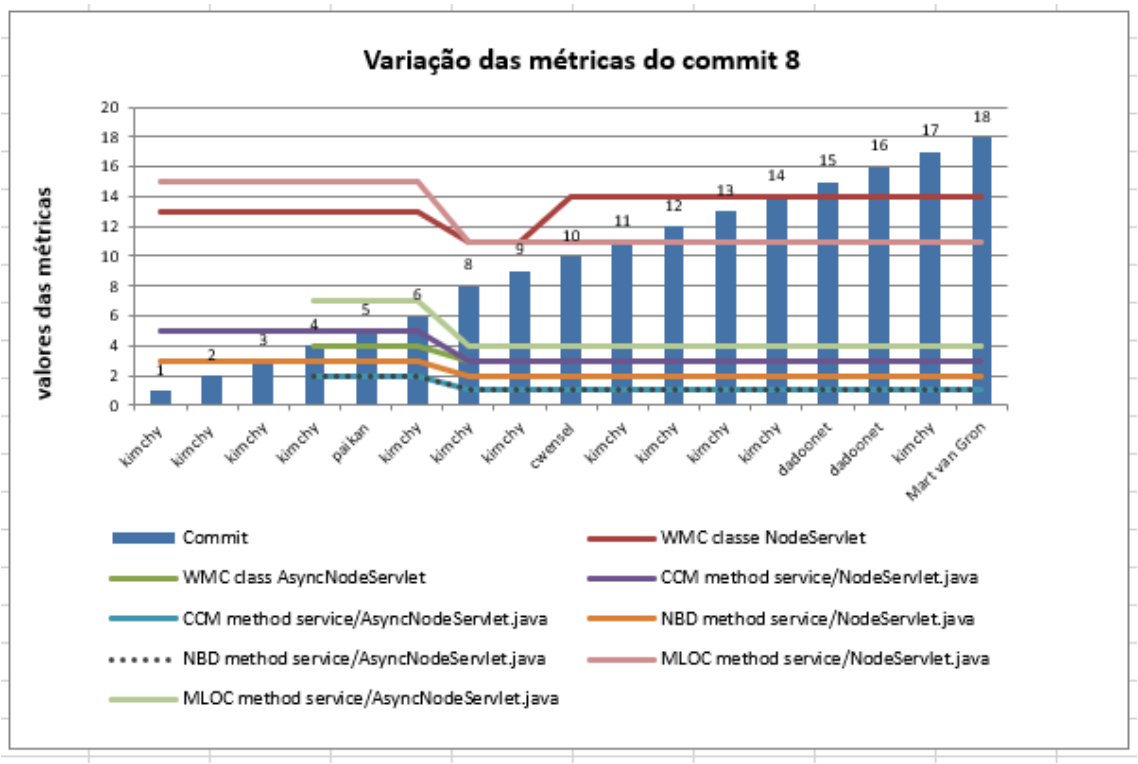


Gráfico A.9: Métricas coletadas do *commit* 8 do Projeto Elastic/Search

No Gráfico A.9 observando-se a métrica NBD (em laranja) é possível perceber que houve uma diminuição no seu valor. A NBD calcula a profundidade de blocos aninhados, que são os blocos de instruções aninhados. Está métrica é utilizada para avaliar a complexidade do código. A Figura A.1 apresenta um trecho do código do *commit* 8, na qual é possível perceber a exclusão de um grupo de decisão *if* (lado esquerdo da figura em vermelho). Esta contribuição do desenvolvedor Kimchy melhorou a métrica NBD do código.

Figura A.1: Split de trecho do código do *commit* 8 do Projeto Elastic/Search

Fonte: https://github.com/elastic/elasticsearch-transport-wares/commit/a8538fcf01cb66967e4afcde8dda6f1887d44b2a?diff=split

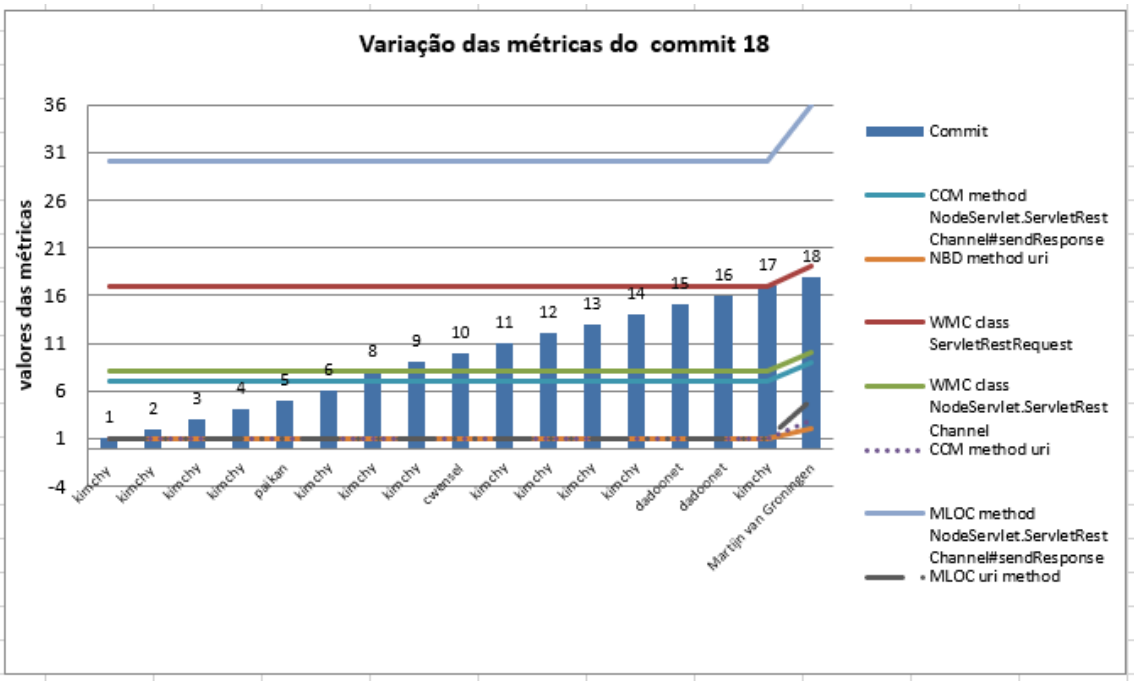


Gráfico A.10: Métricas coletadas do *commit* 18 do Projeto Elastic/Search

Apêndice B – Protótipo da aplicação para cálculo do GC[d]

Para o cálculo do GC[d], conforme proposto no Capítulo 5 e realizado no experimento 2, foram recuperados os códigos-fontes dos projetos a partir do GitHub. Estes códigos foram *buildados* um-a-um obedecendo a ordem dos *commits* realizados pelos desenvolvedores.

O aplicativo utilizado para a recuperação dos códigos-fontes está disponível em: https://github.com/Patriciadebassi/pegadas-do-programador.

E para ser executado é preciso ter instalado no computador os aplicativos Java JDK, Eclipse neon e o git.exe (git-scm.com). Para processar o aplicativo QualityMetrics.jar é preciso inserir o link do projeto no GitHub que se deseja verificar.

Para que seja possível *buildar* cada um dos *commits* é necessário criar o ambiente de desenvolvimento do projeto. Esta atividade requer uma dedicação especial pois, cada projeto disponível no GitHub tem seu ambiente próprio de desenvolvimento e muitas vezes não é possível recriar estes ambientes. Os motivos podem ser desde as versões dos aplicativos utilizados, que não estão mais disponíveis ou não são compatíveis, até a necessidade de aplicativos das instalações de software que são de uso específico destas instalações. No caso de se utilizar uma base de dados histórica, alguns projetos podem não ser possível de *buildar* fora do ambiente de produção da instalação.

Tendo o ambiente do projeto adequadamente instalado no computador a cada *commit buildado* é possível gerar os valores das métricas de código-fonte executando o P*lugin* Metrics do Eclipse. Os dados das métricas de código-fonte foram armazenados no formato XML acrescido do número sequencial do *commit*, mantendo sua linha do tempo, e o nome do desenvolvedor responsável pelo *commit*.

A Figura B.1 apresenta os campos referentes aos dados das métrica de *commits* recuperados do Projeto Elastic/Search.

commit	developer	metric	comment	per	name	source	package	value
1	kimchy	VG	use Extract-method t	method	NodeServlet.ServletRestCh	NodeServlet.java	org.elasticsearch.wares	7
1	kimchy	VG	use Extract-method t	method	init	NodeServlet.java	org.elasticsearch.wares	6
1	kimchy	PAR	Move invoked metho	method	service	NodeServlet.java	org.elasticsearch.wares	2
1	kimchy	PAR	Move invoked metho	method	uri	ServletRestRequest.ja	org.elasticsearch.wares	0
1	kimchy	NBD	use Extract-method t	method	NodeServlet.ServletRestCh	NodeServlet.java	org.elasticsearch.wares	3
1	kimchy	CA		packageFr	org.elasticsearch.wares		org.elasticsearch.wares	0
1	kimchy	CE		packageFr	org.elasticsearch.wares		org.elasticsearch.wares	2
1	kimchy	RMI		packageFr	org.elasticsearch.wares		org.elasticsearch.wares	1
1	kimchy	RMA		packageFr	org.elasticsearch.wares		org.elasticsearch.wares	0
1	kimchy	RMD		packageFr	org.elasticsearch.wares		org.elasticsearch.wares	0
1	kimchy	DIT		type	NodeServlet	NodeServlet.java	org.elasticsearch.wares	3
1	kimchy	DIT		type	ServletRestRequest	ServletRestRequest.ja	org.elasticsearch.wares	2
2	kimchy	VG	use Extract-method t	method	NodeServlet.ServletRestCh	NodeServlet.java	org.elasticsearch.wares	7
2	kimchy	VG	use Extract-method t	method	init	NodeServlet.java	org.elasticsearch.wares	6
2	kimchy	VG	use Extract-method t	method	service	NodeServlet.java	org.elasticsearch.wares	5
5	paikan	DIT		type	NodeServlet.ServletRestCh	NodeServlet.java	org.elasticsearch.wares	1
5	paikan	WMC		type	ServletRestRequest	ServletRestRequest.ja	org.elasticsearch.wares	17
5	paikan	WMC		type	NodeServlet	NodeServlet.java	org.elasticsearch.wares	13
5	paikan	NSC		type	NodeServlet	NodeServlet.java	org.elasticsearch.wares	1
5	paikan	NORM		type	NodeServlet	NodeServlet.java	org.elasticsearch.wares	3
5	paikan	NORM		type	AsyncNodeServlet	AsyncNodeServlet.jav	org.elasticsearch.wares	1

Figura B.1: Exemplo dos dados das métricas dos *commits* realizados referente ao Projeto Elastic/Search

A partir dos dados das métricas de cada *commit* realizado por cada desenvolvedor que contribuiu na construção do projeto de software é possível apresentar os valores do GC[d] do desenvolvedor e o *dashboard* referente a contribuição do desenvolvedor. A Figura B.2 apresenta um exemplo dos valores das métricas no formato XML que é o formato necessário para a aplicação que calcula o GC[d] do desenvolvedor.

```
<?xml version="1.0" encoding="UTF-8"?>
<Metrics scope="elasticsearch" type="Project" date="2015-07-01" xmlns="http://metrics.sourceforge.net/2003/Metrics-First
   <Metric id = "VG" description ="McCabe Cyclomatic Complexity" max ="10" hint ="use Extract-method to split the metho
      <Values per = "method" avg = "1,88" stddev = "1,84" max = "7">
         <Value name="AsyncNodeServlet.AsyncServletRestChannel#sendResponse" source ="AsyncNodeServlet.java" package ="
         <Value name="NodeServlet.ServletRestChannel#sendResponse" source ="NodeServlet.java" package ="org.elasticsear
         <Value name="init" source ="NodeServlet.java" package ="org.elasticsearch.wares" value ="6"/>
        <Value name="service" source ="NodeServlet.java" package ="org.elasticsearch.wares" value ="3"/>
         <Value name="destroy" source ="NodeServlet.java" package ="org.elasticsearch.wares" value ="2"/>
         <Value name="ServletRestRequest" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="
        <Value name="param" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="2"/>
         <Value name="AsyncNodeServlet.AsyncServletRestChannel#AsyncServletRestChannel" source ="AsyncNodeServlet.java"
         <Value name="destroy" source ="AsyncNodeServlet.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="init" source ="AsyncNodeServlet.java" package ="org.elasticsearch.wares" value ="1"/>
         <Value name="services" source ="AsyncNodeServlet.java" package ="org.elasticsearch.wares" value ="1"/>
         <Value name="NodeServlet.ServletRestChannel#ServletRestChannel" source ="NodeServlet.java" package ="org.elasti
        <Value name="contentAsString" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="contentByteArray" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/
        <Value name="contentByteArrayOffset" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value
        <Value name="contentLength" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="contentUnsafe" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
      <Value name="hasContent" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
         <Value name="hasParam" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
         <Value name="header" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="method" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="params" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="param" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="rawPath" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
        <Value name="uri" source ="ServletRestRequest.java" package ="org.elasticsearch.wares" value ="1"/>
      </Values>
   </Metric>
   <Metric id = "PAR" description ="Number of Parameters" max ="5" hint ="Move invoked method or pass an object">
      <Values per = "method" avg = "0,64" stddev = "0,794" max = "2">
```

Figura B.2: Exemplo dos dados das métricas dos *commits* no formato XML referente ao Projeto Elastic/Search

A aplicação que realiza o cálculo do GC[d] e apresenta o resultado da influência da alteração que o desenvolvedor realizou no código referente ao projeto de software está disponível em: https://github.com/Patriciadebassi/pegadas-do-programador

Esta aplicação recebe como entrada um arquivo XML com os dados referentes ao nome do desenvolvedor e as métricas de código-fonte do *commit* realizado pelo desenvolvedor. A partir destes dados é apresentado o grau de contribuição (GC[d]) do desenvolvedor e um *dashboard* mostrando o nível de influência da alteração realizada pelo desenvolvedor no código agrupada em complexidade, reusabilidade, testabilidade e manutenibilidade do código gerado. A Figura B.3 apresenta um exemplo de *commit* realizado no Projeto Elastic/Search.

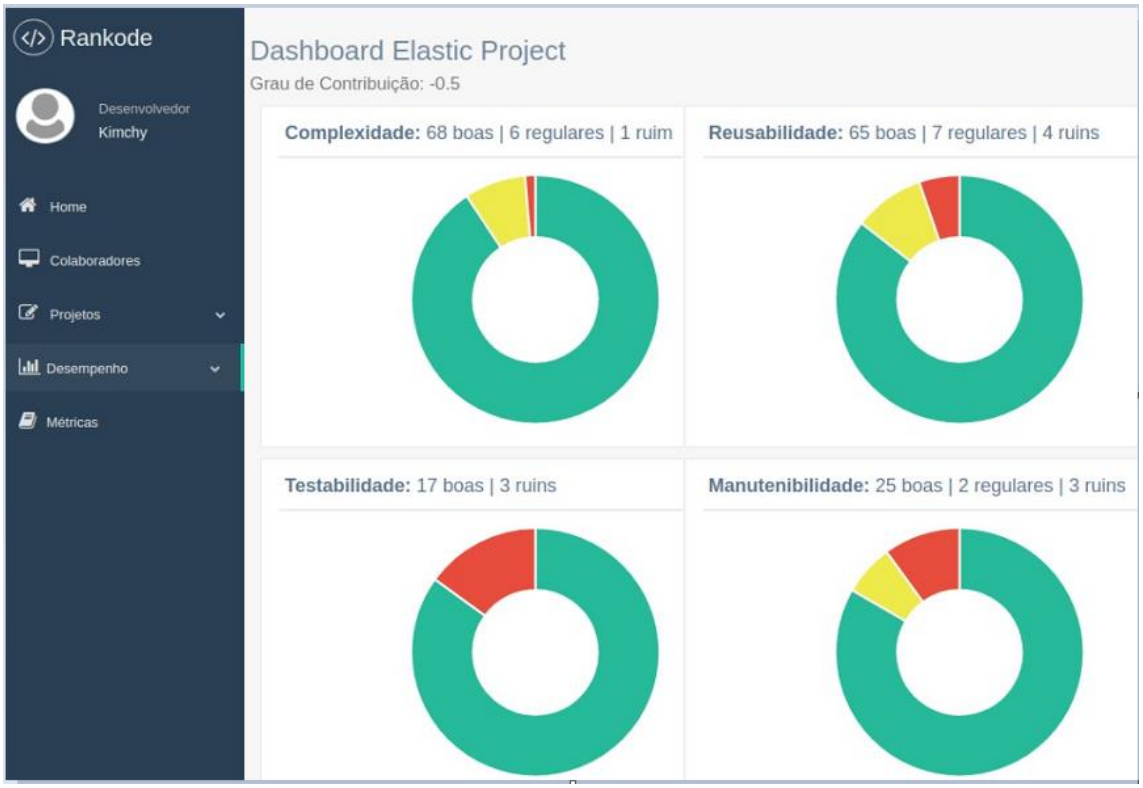


Figura B.3: Exemplo de valor do GC[d] e *dashboard* do nível de qualidade do código no Projeto Elastic/Search

O aplicativo retorna também os valores de cada métrica calculada em forma de gráfico de barras (Figura B.4) e na sequência as “recomendações” de retorno (Figura B.5) com base no valor das métricas do código-fonte avaliado.

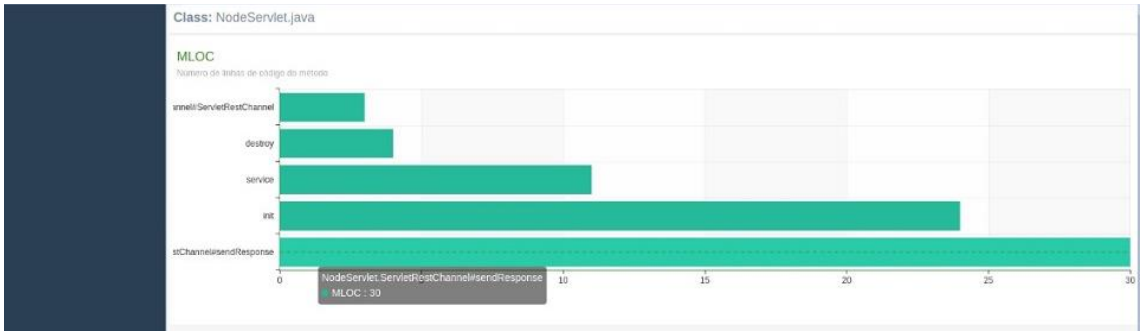


Figura B.4: Exemplo de retorno do valor da métrica em cada artefato do código em forma de gráfico de barras do Projeto Elastic/Search

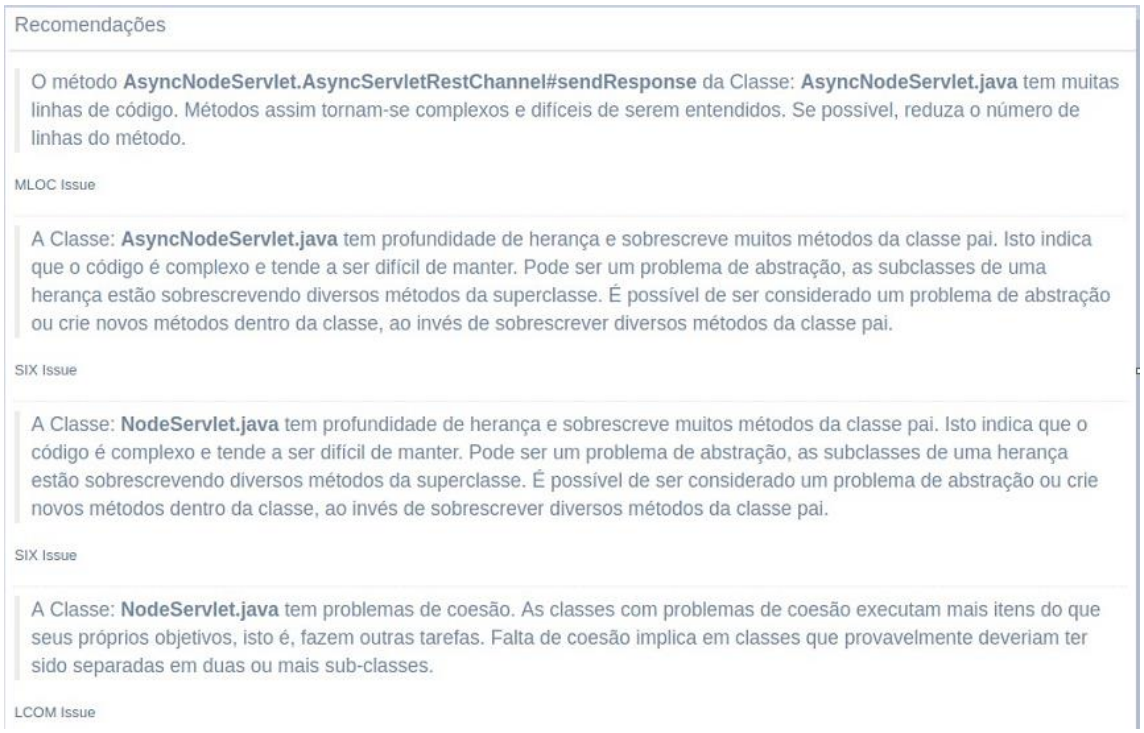


Figura B.5: Exemplo de retorno das “recomendações” com base nos valores das métricas de cada artefato do código do Projeto Elastic/Search