Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Uma Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade

Francisco Vanderson de Moura Alves

Número de Ordem PPGCC: M001

Teresina-PI, 28 de Agosto de 2018

Francisco Vanderson de Moura Alves

Uma Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Pedro de Alcântara dos Santos Neto

Teresina-PI

28 de Agosto de 2018

Francisco Vanderson de Moura Alves
Uma Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade/ Francisco Vanderson de Moura Alves. – Teresina-PI, 28 de Agosto de 2018-
132 p. : il. (algumas color.) ; 30 cm.

Orientador: Pedro de Alcântara dos Santos Neto

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, 28 de Agosto de 2018.

1. Desenvolvimento de Software. 2. Engenharia Reversa. 3. Familiaridade de Código. 4. *Feature Location* 5. Identificação de Funcionalidades. 6. Manutenção de Software 7. Mineração de Repositórios de Software. I. Pedro de Alcântara dos Santos Neto. II. Universidade Federal do Piauí. IV. Uma Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade

CDU 02:141:005.7

"Uma Análise de Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade"

FRANCISCO VANDERSON DE MOURA ALVES

Dissertação apresentada ao Programa de Pós–Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovada por:

Prof. Pedro de Alcântara dos Santos Neto
(Presidente da Banca Examinadora)

Prof. Lincoln Souza Rocha
(Examinador Externo)

Prof. Fábio de Jesus Lima Gomes
(Examinador Externo)

Prof. Raimundo Santos Moura
(Examinador Interno)

Teresina, 28 de agosto de 2018

Aos meus pais Vera Lúcia de Moura Silva e Edmilson Alves Silva (in memoriam), por sempre terem batalhado muito pela minha educação.

Agradecimentos

Agradeço primeiramente a Deus por sempre ter guiado meu caminho e ter me proporcionado alcançar mais esse importante objetivo.

Agradeço aos meus pais, Vera Lúcia de Moura Silva e Edmilson Alves Silva (*in memoriam*), por nunca terem medido esforços para educar seus filhos, apesar das muitas dificuldades enfrentadas. Sempre se preocuparam em mostrar o valor do respeito e companherismo. Se hoje estou obtendo mais esta conquista, vocês meus pais, sem sombra de dúvidas, foram uma das principais razões.

Aos meus irmãos, Vailson, Edmilson e Vanessa, por sempre demonstrarem que eu poderia contar com eles para o que fosse necessário. Além de irmãos, sempre foram e serão meus melhores amigos.

Agradeço também aos meus avós, tios, primos, sobrinhos e a toda minha família. Em especial quero agradecer aos meus avós Esmeraldina, José, Antônia e Abel por todo o carinho que tiveram comigo e todos os seus netos, sempre nos tratando com todo amor e zelo. Agradeço especialmente também à minha tia Miramar, que sempre foi minha segunda mãe, contribuindo diretamente com minha formação, sempre demonstrando satisfação em ajudar.

Agradeço ao meu orientador, Prof. Dr. Pedro de Alcântara dos Santos Neto, pela dedicação, conselhos, paciência e ajuda durante todo esse processo. Serei sempre grato por todos os seus ensinamentos e críticas construtivas, que me fizeram evoluir bastante, acadêmica e pessoalmente. É um exemplo de pessoa comprometida, competente, imparcial, justa e honesta, admirado não só por alunos mas por todos os demais professores, uma referência de pessoa, professor e pesquisador.

Aos amigos que obtive durante toda a minha vida, em especial àqueles que tive a imensa oportunidade de conviver durante toda minha trajetória discente.

Aos professores pela paciência, dedicação, partilha de conhecimento e pelos ensinamentos para a vida. Todos deixaram sua marca, estiveram presentes e contribuíram para minha formação como pessoa.

À Universidade Federal do Piauí e toda sua equipe de funcionários competentes que contribuíram muito para essa realização, em especial àqueles ligados ao Núcleo de Computação de Alto Desempenho.

Muito obrigado a todos!

"A ciência nunca resolve um problema sem criar mais dez".
(George Bernard Shaw)

Resumo

Durante as atividades de manutenção de software, especialmente evolutivas, corretivas e perfectivas, gerentes ou líderes de projetos geralmente realizam a distribuição de tarefas com base na relação que cada desenvolvedor possui com o código fonte associado. Ou seja, quanto maior a relação entre um determinado desenvolvedor e o código fonte associado, maior será a probabilidade desse desenvolvedor assumir o desenvolvimento. Contudo, é comum que gerentes de projetos não tenham uma noção real de como está a relação entre desenvolvedores e o código fonte relacionado a cada tarefa a ser desenvolvida. Em um estudo (dissertação de mestrado) anterior, foi proposto um método baseado em *Mining Software Repositories* (MSR) para obter a familiaridade de código dos desenvolvedores, a partir da mineração de dados de repositórios de software. O cálculo da familiaridade é feito considerando a perspectiva de módulo (pacote) ou arquivo. Neste trabalho, é proposta uma abordagem baseada no método citado anteriormente, para inferência da familiaridade de desenvolvedores, considerando também uma nova perspectiva: funcionalidade de software. Essa perspectiva normalmente é mais apropriada considerando um contexto real de desenvolvimento de software, onde as correções de *bugs* e melhorias no sistema são realizadas considerando as funcionalidades do software. Algumas das principais etapas da abordagem proposta visam identificar funcionalidades de software e unidades computacionais (arquivos de código) que compõem cada funcionalidade identificada, com base na utilização de técnicas de *Reverse Engineering* (RE). Um dos objetivos principais deste trabalho é a realização de uma análise da familiaridade de código considerando as perspectivas de módulo e funcionalidade. Neste estudo foi realizado também, um Mapeamento Sistemático da Literatura (MSL) que apresenta um *overview* da área de inferência da familiaridade e outros tipos de relação entre desenvolvedor e código fonte por meio de técnicas baseadas em MSR. Por fim, foi realizada neste trabalho, uma avaliação em duas etapas. Na primeira fase a abordagem proposta foi avaliada no processo de identificação de funcionalidades e arquivos relacionados. Por meio da análise de sistemas reais, pôde-se constatar que a abordagem obteve um desempenho satisfatório durante esse processo. Durante a segunda etapa de avaliação da abordagem foi realizada a inferência e análise da familiaridade de código entre desenvolvedores considerando a perspectiva de módulo e funcionalidade. Os resultados mostram que uma visão por funcionalidade fornece informações importantes a respeito da familiaridade de código, e que tende a ser mais apropriada em um contexto real de desenvolvimento de software.

Palavras-chaves: Desenvolvimento de Software. Engenharia Reversa. Familiaridade de Código. *Feature Location*. Identificação de Funcionalidades. Manutenção de Software. Mineração de Repositórios de Software.

Abstract

During software maintenance activities, especially evolutionary, corrective and perfective, managers or project leaders usually perform the task distribution based on the relationship that each developer has with the associated source code. Thereby, how larger the relationship between a particular developer and the source code associated with the task to be performed, will greater the possibility of that developer assuming the responsibility of develop it. However, it is common for project managers do not have an approximate notion of the developer relationship with the source code related with each task to be developed. In a previous study (master's thesis), a method based on Mining Software Repositories (MSR) was proposed to obtain the code familiarity of the developers from the mining of data of a software repositorie. The calculation of familiarity is done considering the package or file perspective. In this work, it is proposed an approach based on the previously described method, for inference of the developers familiarity considering also a new perspective: software functionality. This perspective is more appropriate considering a real context of software development, where the bug fixes and improvements in the system are realized considering the functionalities of the software. Some of the main steps of the proposed approach are to identify software functionalities and computational units (code files) that integrate each feature identified, based on the use of Reverse Engineering (RE) techniques. One of the main objectives of this work is perform a comparative analysis of the inference of code familiarity considering the package and functionality perspectives. In this study has conducted also a Systematic Literature Mapping (MSL) which presents an overview of the familiarity inference area and other types of relationship between developer and source code through MSR-based techniques. Finally, was permformed an evaluation in two stages. In the first phase the proposed approach was evaluated in the process of identifying functionalities and related files. Through the analysis of real systems, it was possible to verify that the approach obtained a satisfactory performance during this process. During the second stage of evaluation of the approach, was performed the inference and analysis of the code familiarity between developers considering the module and functionality perspective. The results show that a functionality view provides important information about code familiarity, and is more appropriate in a real software development context.

Keywords: Code Familiarity. Feature Location. Functionality Identification. Mining Software Repositories. Reverse Engineering. Software Development. Software Maintenance.

Lista de ilustrações

Figura 1 – Níveis de funcionalidades de software (COCKBURN (2000)). 9
Figura 2 – Histórico de *commits* armazenado por um SCV. 14
Figura 3 – Arquitetura de funcionamento de um SCVC. 15
Figura 4 – Arquitetura de funcionamento de um SCVD. 16
Figura 5 – Etapas típicas do processo de MSR. 18
Figura 6 – Atividades e componentes do processo de engenharia reversa. 19
Figura 7 – Exemplo de *diff*. 30
Figura 8 – Etapas da abordagem proposta. 35
Figura 9 – Etapas do processo para identificação de funcionalidades. 36
Figura 10 – Mapeamento do código fonte Java em uma AST. 40
Figura 11 – Grafo de referência entre classes Java. 41
Figura 12 – Separação de arquivos por pacote. 43
Figura 13 – Seleção de pacotes *controllers* com base nos graus de referência interna e externa. 44
Figura 14 – Inclusão de arquivos não selecionados ao mesmo pacote *controller* ou candidato. 45
Figura 15 – Referência entre pacotes de classes candidatas. 46
Figura 16 – Processo de identificação de classes relacionadas a uma funcionalidade. 47
Figura 17 – Classes de exemplo para identificação de arquivos ligados a uma funcionalidade. 50
Figura 18 – Cálculo da familiaridade por funcionalidade de software. 51
Figura 19 – Página onde é possível realizar a agregação de *features*. 53
Figura 20 – Página para visualização da familiaridade sobre funcionalidades. 54
Figura 21 – Formulário para configuração de parâmetros para métricas na ferrmaneta *CoDiVision*. 55
Figura 22 – Etapas do processo de MSL. 100
Figura 23 – Etapas do processo de busca. 103
Figura 24 – Etapas do processo de seleção. 106
Figura 25 – Etapas do processo de extração. 109
Figura 26 – Distribuição das publicações identificadas entre os anos de 1998 e 2016. 113
Figura 27 – Contribuição *vs.* Tipo de Pesquisa 114
Figura 28 – Quantidade de estudos por relação entre desenvolvedor e código fonte. 115
Figura 29 – Quantidade de estudos por elementos utilizados na definição de métricas. 117
Figura 30 – Quantidade de estudos por granularidade de relação de código. 118
Figura 31 – Quantidade de estudos por restrições de linguagem de programação. . . 119

Figura 32 – Porcentagem de estudos por objetivo em inferir a relação entre desenvolvedor e código. 120
Figura 33 – Quantidade de estudos por repositórios identificados. 123
Figura 34 – Métodos de avaliação utilizados pelos estudos. 125

Lista de tabelas

Tabela 1 – Exemplos de repositórios de software. 17
Tabela 2 – Matriz de confusão. 21
Tabela 3 – Tipos de *nodes* analisados a partir de uma AST. 40
Tabela 4 – Matriz de confusão para a classificação de arquivos controladores. . . . 61
Tabela 5 – Desempenho na identificação de classes controladoras. 62
Tabela 6 – Matriz de confusão para a identificação de funcionalidades. 63
Tabela 7 – Desempenho na identificação de funcionalidades. 64
Tabela 8 – Matriz de confusão para a identificação classes relacionadas às funcionalidades. 65
Tabela 9 – Desempenho na identificação de arquivos relacionados às funcionalidades. 66
Tabela 10 – Parâmetros para as métricas utilizadas na inferência da familiaridade. . 70
Tabela 11 – Resultados da análise para o sistema SIGAA. 71
Tabela 12 – Resultados da análise para o sistema SERCA. 72
Tabela 13 – Familiaridade dos desenvolvedores sobre algumas funcionalidades do sistema SERCA. 74
Tabela 14 – Termos utilizados na *string* de pesquisa. 104
Tabela 15 – Estrutura da *string* final de pesquisa. 104
Tabela 16 – Resultado da busca nas bases de dados. 105
Tabela 17 – Divisão de estudos entre os grupos. 107
Tabela 18 – Critérios para exclusão de trabalhos 109
Tabela 19 – Locais de publicação com trabalhos aceitos no MSL. 113
Tabela 20 – Estudos por relação entre desenvolvedor e código fonte. 116
Tabela 21 – Elementos considerados para cálculo de métricas. 116
Tabela 22 – Estudos por elementos utilizados na definição de métricas. 117
Tabela 23 – Estudos por nível de granularidade de relação de código. 118
Tabela 24 – Estudos por restrições quanto à linguagem de programação. 119
Tabela 25 – Estudos por objetivo em inferir a relação entre desenvolvedor e código. 121
Tabela 26 – Estudos por tipos de repositórios de software. 122
Tabela 27 – Estudos e disponibilidades dos dados extraídos. 123
Tabela 28 – Estudos por tipos de dados extraídos. 124
Tabela 29 – Mecanismos de extração de dados utilizados. 124
Tabela 30 – Estudos por métodos de avaliação utilizados. 126
Tabela 31 – Objetos de avaliação utilizados pelos estudos. 126
Tabela 32 – Estudos que realizam discussão sobre ameaças à validade. 127
Tabela 33 – Referências para estudos aceitos. 129

Lista de abreviaturas e siglas

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
ER	Engenharia Reversa
ES	Engenharia de Software
FL	*Feature Location*
FN	Falso Negativo
FP	Falso Positivo
JDK	*Java Developer Kit*
JDT	*Java Developer Tools*
JEE	*Java Enterprise Edition*
MRS	Mineração de Repositório de Software
MSL	Mapeamento Sistemático da Literatura
MSR	*Mining Software Repositories*
MVC	Padrão Arquitetural ***M**odel-**V**iew-**C**ontroller*
OO	Orientação a Objetos
QP	Questão de Pesquisa
QS	Qualidade de Software
RCS	*Revision Control Systems*
SAGRES	Sistema de Acompanhamento da Gestão dos Recursos da Sociedade
SCCS	*Source Code Control System*
SCV	Sistema de Controle de Versão
SCM	*Source Code Management*
SERCA	Sistema Emissor de Relatórios de Contas Anuais

SIGAA	Sistema Integrado de Gestão de Atividades Acadêmicas
SVN	*Subversion*
TCE/PI	Tribunal de Contas do Estado do Piauí
TF	*Truck Factor*
UFPI	Universidade Federal do Piauí
VCS	*Version Control System*
VN	Verdadeiro Negativo
VP	Verdadeiro Positivo

Sumário

Contexto e Motivação . . . 1
Definição do Problema . . . 2
Objetivos . . . 4
Visão Geral da Proposta . . . 4
Justificativa . . . 5
Estrutura do Trabalho . . . 7

1 FUNDAMENTAÇÃO TEÓRICA . . . 9
1.1 Funcionalidade de Software . . . 9
1.2 *Feature Location* . . . 10
1.3 Sistema de Controle de Versão . . . 12
1.4 Mineração de Repositórios de Software . . . 16
1.5 Engenharia Reversa . . . 19
1.6 *Truck Factor* (TF) . . . 20
1.7 Medidas de Avaliação de Classificadores . . . 20
1.8 Considerações Finais . . . 22

2 TRABALHOS RELACIONADOS . . . 25
2.1 Principais Trabalhos Relacionados . . . 25
2.2 Método para Inferência da Familiaridade de Código na Perspectiva de Módulos . . . 29
2.3 Considerações Finais . . . 34

3 ABORDAGEM PROPOSTA . . . 35
3.1 Etapas da Abordagem . . . 36
3.2 Implementação da Abordagem . . . 38
3.3 Representação Sinóptica da Abordagem . . . 52
3.4 Intregação da Abordagem com a Ferramenta *CoDiVision* . . . 53
3.5 Considerações Finais . . . 55

4 AVALIAÇÃO . . . 57
4.1 Avaliação do Processo de Identificação de Funcionalidades . . . 57
4.2 Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade . . . 68
4.3 Ameaças à Validade . . . 75
4.4 Considerações Finais . . . 77

5 CONCLUSÕES E TRABALHOS FUTUROS 79
5.1 Publicações 81
5.2 Desafios e Limitações 82
5.3 Trabalhos Futuros 83

REFERÊNCIAS 85

APÊNDICES 97

APÊNDICE A – MAPEAMENTO SISTEMÁTICO DA LITERATURA 99
A.1 Metodologia 99
A.2 Relações entre Desenvolvedor e Código Fonte 100
A.3 Questões de Pesquisa 101
A.4 Ameaças à Validade 127
A.5 Considerações Finais 128

APÊNDICE B – REFERÊNCIAS PARA TRABALHOS RELACIONADOS AO MAPEAMENTO SISTEMÁTICO . . 129

Introdução

Contexto e Motivação

Um software representa instruções (programa de computador) que quando executadas fornecem um conjunto de características, funções e performance desejados (PRESSMAN, 2005). Sistemas de software estão presentes em boa parte da vida moderna. Softwares estão contidos em celulares, carros, na compra de mercadorias, em transações bancárias, etc. Cada vez mais, depende-se do correto funcionamento do software e por conseguinte, da equipe de manutenção que trabalha para deixá-lo apto às demandas diárias, mantendo-o atualizado e funcional.

Desenvolver software vai muito além da atividade de programação. Para que um projeto de software seja bem sucedido, gerando um produto de qualidade, é necessário a execução de várias atividades, como por exemplo, análise dos riscos envolvidos, levantamento de recursos necessários, estimativa de esforço e custo do projeto, dentre outros (PRESSMAN, 2005). Com isso, surgiu a Engenharia de Software (ES), que engloba uma série de conceitos, princípios, métodos e ferramentas para o desenvolvimento de software com qualidade (PRESSMAN; MAXIM, 2016).

O processo de desenvolvimento de software com qualidade depende diretamente de um bom gerenciamento de projeto (CLELAND; IRELAND, 2006). A gerência de projetos pode ser definida como o processo de controle da realização dos objetivos do projeto, tendo como funções principais a definição do requisito de trabalho e estabelecimento de sua extensão, a alocação dos recursos necessários, o planejamento da execução do trabalho, o acompanhamento do progresso de atividades e os ajustes durante desvios de planejamento (MUNNS; BJEIRMI, 1996).

O gerenciamento de projetos engloba uma das principais etapas do ciclo de desenvolvimento de software: a **manutenção** (LEHMAN, 1980). Esta pode ser definida como a modificação do sistema após a entrega para corrigir falhas, incluir novas funcionalidades, melhorar o desempenho ou se adaptar a um ambiente alterado (COMMITTEE, 1983). A manutenção é uma etapa muito importante, pois é bastante comum que um sistema de software precise passar por melhorias ou ser incrementado com novas funcionalidades ao longo da sua vida útil. A manutenção de software pode ser classificada em LIENTZ; SWANSON (1978): (I) Manutenção Corretiva: trabalho para corrigir erros de projeto, codificação e implementação; (II) Manutenção Adaptativa: para satisfazer mudanças no ambiente de processamento ou dados; e (III) Manutenção Perfectiva: para aumentar a eficiência, o desempenho ou a manutenção do processamento e para melhor atender as

necessidades dos usuários.

Contudo, as atividades de manutenção de software podem ser classificadas em muitas outras categorias importantes. HARJANI; QUEILLE (1992) apresentam, além das atividades apresentadas em (LIENTZ; SWANSON, 1978), outros importantes tipos de manutenção de software: (I) Manutenção Evolutiva: as atividades destinadas a adicionar novas funcionalidades para o software; (II) Manutenção Preventiva: as atividades destinadas a melhorar a capacidade de manutenção do software, isto é, facilitando o entendimento e modificação do software (o que está diretamente relacionado com a manutenibilidade de sistema, ou seja, a facilidade com que um sistema de software pode ser modificado (MARTIN; MCCLURE, 1983)); (III) Manutenção Antecipativa: as atividades destinadas a antecipar problemas com o uso do software ou com a evolução do seu ambiente de execução.

Uma atividade bastante comum no processo de gerenciamento de projeto durante as atividades de manutenção de software (especialmente evolutivas, corretivas e perfectivas) é o planejamento da execução do trabalho, que envolve a definição de quais desenvolvedores devem executar cada tarefa necessária. É comum que gerentes ou líderes de projetos realizem esse processo de alocação de tarefas com base na relação que cada desenvolvedor possui com o código fonte associado a cada uma das tarefas, isto é, o quão cada desenvolvedor conhece partes do software associada a cada tarefa. Dessa forma, quanto maior a relação entre um determinado desenvolvedor e o código fonte associado à tarefa que deverá ser executada, maior será a probabilidade desse desenvolvedor assumir a responsabilidade de executá-la.

Com isso, é de extrema importância que gerentes de projetos tenham informações das atividades realizadas por cada desenvolvedor durante o processo de desenvolvimento de software, a fim de inferir o grau de relação que cada desenvolvedor possui com partes desse software.

Definição do Problema

É importante ressaltar, que neste trabalho, a relação entre desenvolvedor e código fonte é considerada como sendo a **familiaridade** que um determinado desenvolvedor possui sobre o código. De acordo com *Oxford Dictionaries*[1], o termo familiaridade refere-se a "conhecimento próximo", "conhecimento de algo" ou ainda à "intimidade entre as pessoas". Com base nisso, o termo "familiaridade", utilizado neste estudo, refere-se a quão um determinado desenvolvedor é "íntimo" em relação a uma parte específica do software.

Como discutido na seção anterior, a informação sobre relação entre desenvolvedor e código fonte é de suma importância para que gerentes ou líderes de projetos. A partir

[1] https://en.oxforddictionaries.com/.

dessa informação é possível definir melhores estratégias de distribuição de tarefas, levando em consideração a familiaridade que cada desenvolvedor possui com a parte do software relacionada a cada tarefa a ser realizada, aumentando também a chance de cada tarefa ser bem sucedida. Dessa forma, com a melhoria no processo de manutenção, a tendência que é o software desenvolvido também tenha seu nível de qualidade elevado.

Contudo, estimar a familiaridade de código não é uma tarefa simples (FRITZ et al., 2014). Geralmente, um desenvolvedor de software realiza alterações em centenas de linhas de código diariamente, e o trabalho realizado pelos seus companheiros de equipe muitas vezes resulta em uma alta taxa de variação nesse código, dificultando saber qual membro da equipe está relacionado com qual parte do código (FRITZ et al., 2010).

Ainda assim, apesar da dificuldade em inferir a relação entre desenvolvedor e código fonte, alguns trabalhos já foram desenvolvidos com esse objetivo, como os apresentados por LIRA (2016), FRITZ et al. (2010), FRITZ et al. (2013), HATTORI; LANZA (2009) e FRANTZESKOU et al. (2006b). Contudo, a grande maioria desses estudos visam inferir a relação entre desenvolvedor e código fonte considerando perspectivas menos granulares, isto é, visam em sua maioria inferir a familiaridade que desenvolvedores possuem sobre linhas, arquivos ou módulos (que podem ser considerados também pacotes com um conjunto de arquivos e/ou subpacotes) de software.

Um dos fatores que contribuem para existir poucos estudos que visam inferir a familiaridade de código considerando a perspectiva de funcionalidade, é que o processo de inferência considerando essa perspectiva não é trivial. Isso se deve principalmente a dois fatores. O primeiro deles é a dificuldade em identificar funcionalidades de um software de maneira automática. O segundo fator refere-se ao fato de que geralmente uma funcionalidade é implementada por vários arquivos de código que podem, por sua vez, estarem localizados em diferentes módulos ou pacotes de um projeto. Dada a complexidade do processo de identificação de funcionalidades e seus arquivos relacionados, existem até mesmo estudos voltados especificamente à área de identificação de *features* de software (DIT et al., 2013), como os apresentados por CHEN; RAJLICH (2001), BUCKNER et al. (2005) e ROBILLARD; MURPHY (2007).

É importante ressaltar que neste trabalho o termo ***feature*** é utilizado como sinônimo para funcionalidade. Uma *feature* pode ser entendida como um requisito funcional, ou ainda um comportamento do sistema que pode ser disparado pelo usuário (EISENBARTH; KOSCHKE; SIMON, 2003). Vale ressaltar ainda que o termo *feature* pode se referir também a requistos não funcionais de um sistema.

Objetivos

Este trabalho possui como objetivo principal realizar uma avaliação da familiaridade de código considerando a perspectiva de funcionalidades, que tende a ser mais apropriada no auxílio a gerentes de projetos durante o planejamento de alocação de tarefas na etapa de manutenção de software. Para alcançar o objetivo geral deste trabalho, foram definidos os seguintes objetivos específicos:

1. Realização de um mapeamento sistemático sobre estimação de familiaridade (ou algum aspecto similar) com o objetivo de identificar os principais trabalhos nessa linha de pesquisa e fornecer uma visão geral em relação a métodos, técnicas ou ferramentas que auxiliam nessa atividade. Esse estudo serviu para validar decisões tomadas durante o desenvolvimento do método proposto neste trabalho, pois apresentou lacunas deixadas pelos estudos relacionados.

2. Desenvolvimento de uma abordagem para inferir a familiaridade de código de desenvolvedores em projetos de software, levando em consideração um novo ângulo de visão: funcionalidades de sistema, ou seja, inferir a familiaridade que cada desenvolvedor possui com o código fonte relacionado a cada *feature* do software. A abordagem proposta possui como objetivo base identificar funcionalidades de software e os arquivos que compõem cada uma delas. A partir disso, é possível computar a familiaridade de código com base no subconjunto de arquivos que compõem cada funcionalidade identificada.

3. Desenvolvimento de um novo módulo para integrar à ferramenta *CoDiVision* (LIRA, 2016). Esse módulo é responsável por computar a familiaridade de código em nível de funcionalidades de software e gerar informações para que sejam visualizadas via interface *Web*, facilitando a experiência dos usuários.

4. Avaliação da abordagem proposta em projetos de desenvolvimento de software reais.

Visão Geral da Proposta

Como discutido anteriormente, um dos principais objetivos deste trabalho é o desenvolvimento de uma abordagem para inferência da familiaridade de código considerando a perspectiva de funcionalidades de software. Para inferir a familiaridade de código baseado em requisitos funcionais, é necessário então identificar todas as funcionalidades, assim como os arquivos de código que compõem cada funcionalidade. Com isso, a abordagem proposta neste trabalho oferece também o conjunto de passos que visam alcançar tal objetivo. Essa abordagem é baseada na extração de dados de repositório de software e em técnicas de Engenharia Reversa (ER) (vide Seção 1.5 do Capítulo 1).

A abordagem proposta utiliza também conceitos relacionados à Teoria de Grafos (BONDY; MURTY, 1976; WEST et al., 2001) na descoberta de arquivos principais (arquivos que controlam o início da execução de uma funcionalidade do sistema). A partir disso, são aplicadas técnicas de engenharia reversa para identificar o relacionamento entre arquivos principais do sistema e os demais arquivos do projeto. Uma funcionalidade será composta por um subconjunto de arquivos do sistema. Para atingir o objetivo de identificação de *features* e arquivos relacionados, a abordagem proposta apresenta uma técnica alternativa para ***Feature Location*** (FL) (DIT et al., 2013). Esse conceito é melhor discutido na Seção 1.2 do Capítulo 1. Contudo, a FL pode ser entendida como uma técnica para identificação de funcionalidades e componentes de software (arquivos, métodos/funções, *statements*, etc.) relacionados.

A familiaridade de código de um desenvolvedor sobre uma determinada *feature* será dada pela computação da familiaridade que esse desenvolvedor possui sobre os arquivos que compõem o requisito funcional específico. Para inferir a familiaridade sobre arquivos de uma funcionalidade, a abordagem utiliza um método já existente. Esse método foi proposto em uma dissertação de mestrado desenvolvida por LIRA (2016). O método é baseado em Mineração de Repositórios de Software (MRS) (vide Seção 1.4 do Capítulo 1) para resolução de tal problema. O método busca calcular as contribuições de desenvolvedores para código do projeto, obtidas a partir de commits feitos ao Sistema de Controle de Versão (SCV) (vide Seção 1.3 do Capítulo 1), para inferir a familiaridade que cada membro da equipe tem com o código fonte, considerando arquivos ou módulos (conjunto de um ou mais arquivos) de software.

Neste trabalho é apresentado então uma abordagem que utiliza uma outra forma de cálculo da familiaridade de código, mas que estende o método proposto por LIRA (2016). Essa abordagem tem como objetivo inferir a familiaridade de código de desenvolvedores considerando um novo ângulo de visão: funcionalidades (*features*) de software. A Seção 1.1 apresenta com mais detalhes os conceitos sobre funcionalidades. Contudo, é importante frisar que a abordagem proposta identifica *features* em nível de **subfunção**. Essas funcionalidades representam componentes (métodos) principais em um arquivo controlador (***controller***), dentro de projetos de software baseados no padrão arquitetural MVC (KRASNER; POPE et al., 1988).

Justificativa

A informação sobre a familiaridade de código considerando a perspectiva de funcionalidades tende a ser mais apropriada em um contexto real de desenvolvimento de software. Isso se deve ao fato de que é comum que durante a manutenção de software, seja considerada a perspectiva de funcionalidade como base no processo de distribuição

de tarefas. Na prática, quando um usuário do sistema reporta um problema ou uma recomendação de melhoria, é comum que ele cite a funcionalidade do sistema em que o problema ocorreu ou que necessite de alterações para melhorias. Dessa forma, o gerente de projetos precisa da informação sobre qual desenvolvedor possui maior familiaridade sobre a funcionalidade que precisa ser modificada, seja para a tarefa corretiva ou perfectiva. Com isso, é de extrema importância que exista uma forma de obtenção da familiaridade de código considerando a perspectiva de funcionalidade.

Com a visão da familiaridade sobre funcionalidades é possível até mesmo analisar diferentes fatores. Pode-se verificar se a perspectiva por funcionalidades está relacionada com a visão por módulos, uma vez que em um projeto de software, hipoteticamente constituído por vários integrantes na equipe de desenvolvimento e que possua 2 (dois) desenvolvedores, por exemplo, detendo 50% de familiaridade sobre o software como um todo, pode ter metade das funcionalidades ligadas a um desses desenvolvedores e outra metade ao outro, indicando assim que o projeto se encontra em um situação crítica em termo do conceito de código coletivo. Em contrapartida, pode ocorrer um cenário totalmente diferente, em que existam desenvolvedores com uma maior familiaridade considerando a perspectiva de módulo mas que não possuem um alto grau de familiaridade sobre a maioria das funcionalidades do sistema, isto é, que "dominam" um menor número de funcionalidades em relação aos demais desenvolvedores.

Dessa forma, em uma situação hipotética, um gerente de projeto pode ser levado a tomar decisões errôneas com base em uma informação "parcial" da familiaridade de código. Em uma situação hipotética, um desenvolvedor que possui um alto grau de familiaridade sobre o módulo do projeto como um todo, pode ser designado a realizar uma tarefa de manutenção em um funcionalidade específica, apenas porque o gerente de projeto tenha entendido que esse desenvolvedor poderia estar mais familiarizado com a funcionalidade em questão, por ter uma maior familiaridade sobre os arquivos do projeto como um todo. Entretanto, esse desenvolvedor pode, por sua vez, ter uma baixa (ou nenhuma) familiaridade com a funcionalidade específica.

Apesar da informação sobre a familiaridade de código considerando as perspectivas de menor granularidade (linha, arquivo ou módulo) servirem como base no entendimento da familiaridade de código, uma visão por funcionalidade pode oferecer uma noção ainda maior sobre a familiaridade que desenvolvedores de software possuem sobre o código fonte. Contudo, após um levantamento da área, por meio da realização de um Mapeamento Sistemático da Literatura (MSL) (KITCHENHAM, 2004), e que será melhor discutido no Capitulo 2, pôde-se constatar que os estudos cujo objetivo é inferir a familiaridade de código entre desenvolvedores de software, baseiam-se em sua grande maioria na inferência de familiaridade em nível de linhas, arquivos ou módulos.

Portanto, este trabalho busca oferecer um nova maneira de visualização da fami-

liaridade que desenvolvedores de software possuem sobre o código fonte. Por meio da informação da familiaridade de código sobre funcionalidades ou *features*, pode-se obter uma diferente visão sobre a familiaridade de código, considerando que a perspectiva de funcionalidades oferece uma visão, em um maior nível de granularidade, da familiaridade de código. Além disso, durante o gerenciamento de projetos, gerentes ou líderes podem definir novas estratégias para melhor realizar a alocação de tarefas durante a fase de manutenção, levando em consideração as funcionalidades do software que deverão ser modificadas ou incrementadas. A informação de familiaridade em nível de funcionalidades deve contribuir ainda mais para o processo de desenvolvimento, especialmente para a etapa de manutenção do produto de software gerado.

Estrutura do Trabalho

Além da introdução apresentada anteriormente e que aborda os principais aspectos deste trabalho, que são: contexto, motivação, objetivos gerais e específicos e as principais contribuições deste estudo, este trabalho está dividido em 5 capítulos. O Capítulo 1 apresenta os conceitos fundamentais sobre técnicas e componentes utilizados neste trabalho, abordando, por exemplo, conceitos sobre funcionalidades de software, Mineração de Repositórios de Software (MRS) e Engenharia Reversa (ER).

O Capítulo 2 discute sobre os principais trabalhos relacionados identificados durante a realização de um Mapeamento Sistemático da Literatura (MSL), apresentando uma visão geral da área de inferência da familiaridade que desenvolvedores de software possuem sobre o código fonte. No Capítulo 3 é detalhada a abordagem proposta neste trabalho para inferência da familiaridade de código considerando a perspectiva de funcionalidades de software.

O Capítulo 4 apresenta uma avaliação da abordagem proposta neste trabalho. Essa avaliação foi dividida em duas etapas. A abordagem foi avaliada no processo de identificação de funcionalidades, assim como os componentes computacionais que compõem cada funcionalidade identificada. Na segunda etapa, foi realizada uma análise da familiaridade de código considerando as perspectivas de módulo e funcionalidade. Por fim, o Capítulo 5 apresenta as conclusões deste trabalho, assim como desafios e limitações da abordagem proposta e as perspectivas para trabalhos futuros.

1 Fundamentação Teórica

1.1 Funcionalidade de Software

Segundo EISENBARTH; KOSCHKE; SIMON (2003) uma funcionalidade pode ser entendida como um requisito funcional de um sistema, ou ainda um comportamento observável do sistema que pode ser disparado pelo usuário. Além disso, como apresentado em (COCKBURN, 2000), uma funcionalidade pode ser classificada em três níveis: de I) ***summary*** (agregador), II) ***user*** (usuário) e III) ***subfuntions*** (subfunção). A Figura 1 apresenta os níveis de funcionalidade citados. Para exemplificar, COCKBURN (2000) associa cada um dos níveis a uma determinada cor, respectivamente: branco, azul ou índigo.

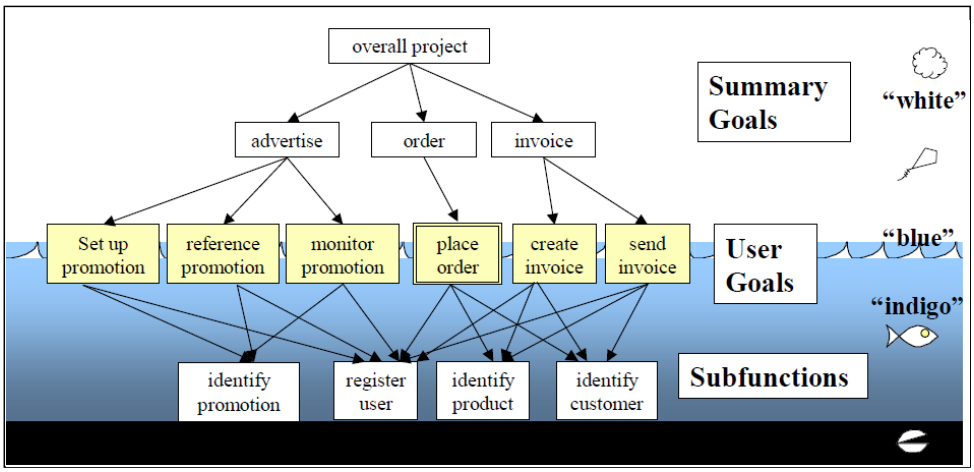


Figura 1 – Níveis de funcionalidades de software (COCKBURN (2000)).

Fonte – COCKBURN (2000).

De maneira resumida, as funcionalidades de nível *agregador* (cor branca) englobam várias tarefas ou *features* de nível *usuário* (cor azul), que por sua vez, representam uma única tarefa a ser executada pelo usuário (comportamento observável), de forma que este obtenha um resultado que o atenda. Por fim, mas não menos importante, as funcionalidades de nível *subfunção* (cor índigo) são aquelas que *features* de menor granularidade, que normalmente representam subtarefas. Pode-se perceber que quanto maior o tom da cor associada a uma determinada funcionalidade, menor será seu nível.

As funcionalidades de nível *subfunção* representam sub tarefas executadas dentro de uma funcionalidade de nível *usuário*. É importante frisar que essas *features* podem representar um comportamento observável (tarefas executadas diretamente pelo usuário) do sistema, como por exemplo: **realizar *login***. Além disso, elas podem representar tarefas

indiretamente executas pelo usuário, isto é, não representam um comportamento observável do sistema, como por exemplo: **validar *login***.

1.2 *Feature Location*

A *Feature Location* (FL) pode ser entendida como a atividade para identificação de partes do código fonte que implementam uma funcionalidade no sistema (BIGGERSTAFF; MITBANDER; WEBSTER, 1993; RAJLICH; WILDE, 2002). A localização de *features* é um passo fundamental em tarefas de evolução de software, como depuração, compreensão, e reutilização (HILL et al., 2013). É uma das atividades de manutenção mais frequentes realizadas por desenvolvedores, porque é uma parte do processo de mudança incremental (RAJLICH; GOSAVI, 2004).

A manutenção e evolução de software envolve o incremento de novas funcionalidades aos programas, melhorando as *features* existentes e removendo bugs, que é análogo à remoção de funcionalidades indesejadas. Nenhuma atividade de manutenção pode ser concluída sem primeiro localizar o código que é relevante para a tarefa em questão, tornando localização característica essencial para manutenção de software, uma vez que é realizada no contexto de mudança incremental (DIT et al., 2013). Dessa forma, é importante que desenvolvedores tenham a informação sobre componentes computacionais que compõem uma determinada funcionalidade. Esses componentes podem ter diferentes granularidades, tais como método/função, *statement* (bloco de instruções) ou arquivo, por exemplo.

Em DIT et al. (2013) é apresentada uma taxanomia sobre a área de FL. São apresentadas quatro principais classificações ligadas a essa área, que são: **Tipos de análise**, **Entrada de usuário**, **Fontes de dados** e **Resultado** (*Output*). A seguir é detalhada cada uma dessas classificações:

- **Tipos de análise**:
  1. **Dinâmica**: Análise em que é necessária a execução do sistema.
  2. **Estática**: Análise em que não é necessária a execução do sistema.
  3. **Textual**: Uma análise textual é utilizada.
  4. **Histórica**: Informações de repositórios de software são utilizadas.
- **Entrada de usuário**:
  1. **Consulta via Linguagem Natural**: Uma descrição textual de uma *feature* (ou uma descrição do relatório de erro) ou uma consulta especificada pelo usuário é usada como entrada.
  2. **Cenário de Execução**: Utiliza-se um cenário para executar uma *feature* (ou reproduzir um bug) de interesse a fim de coletar rastros de execução.

3. **Histórica**: Informações de repositórios de software são utilizadas.
4. **Artefato do Código Fonte**: Um artefato de software é usado como ponto de partida (ou propagação).

- **Fonte de dados**:

1. **Código Fonte**:
   a) **Compilável/executável**: Um programa executável é usado como entrada para extrair informações derivadas.
   b) **Não compilável**: Um código fonte que pode ou não conter erros que impeçam a sua conformidade é usado como uma entrada para extrair informações derivadas.
2. **Dados de análise derivada do código-fonte**:
   a) **Gráfico de Dependência**: O gráfico de dependência pode ser derivado diretamente do código fonte (isto é, estático) ou pode ser derivado de informações de execução (ou seja, dinâmico).
   b) **Rastreamento de execução**: Um conjunto de métodos que foram executados ao se exercitar um cenário.
   c) **Informações Históricas**: Informações históricas de repositórios de código fonte sobre as alterações no código-fonte. Esta informação inclui mensagens de *log* e mudanças reais.

- **Resultado (*Output*)**:

1. **Código fonte:**
   a) **Arquivo/classe**: Os resultados produzidos pela técnica de FL possuem granularidade em nível de arquivo/classe.
   b) **Método/função**: Os resultados produzidos pela técnica de FL possuem granularidade em nível de método/função.
   c) **Instruções (*statments*)**: Os resultados produzidos pela técnica de FL possuem granularidade em nível de instrução.
2. **Artefato que não é código fonte**: A saída produzida não é um artefato de código fonte (relatório de erro, por exemplo).
3. **Apresentação dos resultados**:
   a) **Classificação**: Os resultados produzidos pela técnica de FL têm pontuações que podem ser utilizadas para classificar os resultados com base em sua relevância para a entrada do usuário.
   b) **Visualização**: Os resultados produzidos pela técnica de FL são apresentados usando uma maneira avançada de visualizar informações.

A linguagem de programação na qual um sistema de software é escrito pode influenciar no tipos de técnicas de FL que podem ser aplicadas a esse sistema. Análises dos tipos **estática** e **dinâmica**, por exemplo, podem ser limitadas devido ao suporte do sistema para uma determinada plataforma ou linguagem de programação. Isso porque diferentes linguagens refletem em distintas formas com que os componentes (como arquivos) de software se relacionam.

1.3 Sistema de Controle de Versão

Um Sistema de Controle de Versão (SCV) ou VCS (do inglês *Version Control System*) é basicamente um local para armazenamento e gerência de artefatos gerados durante o desenvolvimento de sistemas de software (MASON, 2005). Esse sistema registra as alterações realizadas sobre artefatos, para que seja possível recuperar, posteriormente, versões específicas desses arquivos (CHACON; STRAUB, 2014). Normalmente, o principal artefato armazenado é o código fonte, mas há a possibilidade de armazenamento até mesmo de documentos, tais como especificação de requisitos (DAVIS et al., 1993).

Durante o estudo sobre SCV, pode-se encontrar alguns termos em comum, tais como: *Revision Control Systems* (RCS) (TICHY, 1982), *Software Configuration Management* (BERLACK, 1992; SCOTT; NISSE, 2001; KEYES, 2004), *Source Code Management* (SCM) (VESPERMAN, 2006) ou *Source Code Control System* (SCCS) (ROCHKIND, 1975). Mesmo que esses termos apresentem diferentes formas, eles têm basicamente o mesmo significado e podem ser usados indistintamente (OTTE, 2009).

O uso de sistemas de versionamento é cada vez mais comum durante o desenvolvimento de software, e diversos fatores contribuem para isso. Esses sistemas habilitam a aceleração e simplificação do processo de desenvolvimento de software, proporcionando novos fluxos de trabalho e mantendo um modelo de acesso concorrente aos artefatos do projeto (OTTE, 2009).

Um SCV é capaz de manter um sistema de software composto por muitas versões e configurações bem organizadas (TICHY, 1985), proporcionando também que o código fonte seja facilmente compartilhado entre toda a equipe de desenvolvimento. Outra característica importante, é que os sistemas de controle de versão mantêm um histórico de todas as operações (adição, remoção ou modificação) realizadas sobre os artefatos do projeto (BALL et al., 1997), bem como o desenvolvedor responsável e a data de cada ação realizada (SPINELLIS, 2005).

Um sistema de versionamento pode ser classificado em centralizado ou distribuído. Cada um desses tipos de sistema possui, obviamente, características peculiares, que serão apresentadas nas Seções 1.3.1 e 1.3.2. Porém, alguns termos e características em comum também são encontradas em ambos os modelos de SCV:

- **Versão**: Indica o estado do artefato ou projeto após receber atualizações. Cada versão possui um identificador único;
- ***Revisão***: Está diretamente relacionada a um *commit*. Quando esta última operação é feita, é gerada então uma nova revisão ou identificador;
- ***Merge***: É a mesclagem entre duas diferentes versões, com o objetivo de gerar uma versão que agregue todas as alterações realizadas;
- **Repositório**: Local de armazenamento dos artefatos do projetos, assim como suas versões geradas no decorrer do desenvolvimento de um produto de software;
- ***Checkout***: É usado para denominar o primeiro *download* dos artefatos do projeto a partir do repositório;
- ***Update***: É a atualização dos artefatos locais por meio do *download* das modificações contidas no repositório;
- ***Commit***: É o envio para o repositório, das alterações realizadas por cada desenvolvedor sobre os artefatos;
- ***Synchronize***: É o processo de verificar quais artefatos foram alterados localmente, que podem ser enviados para o repositório, e quais foram alterados e já enviados por outro desenvolvedor;
- ***Branch***: É um ramo de desenvolvimento paralelo ou alternativo a outro já existente. Esses ramos geralmente são criados para indicar diferentes estados do software.

A propriedade mais importante de um *commit* é a imutabilidade, ou seja, após essa operação ser realizada, seu conteúdo não pode ser alterado, sendo necessário um novo *commit*, mesmo em casos que o objetivo seja reverter algo que foi feito (MAGNUSSON; ASKLUND, 1996). A cada *commit* realizado o SCV vai mantendo o histórico de versões dos artefatos ou arquivos alterados, assim como o tipo de alteração realizada. A Figura 2 apresenta um exemplo de histórico de *commits* armazenado pelo SCV.

Pode-se perceber, que após a realização do primeiro *commit*, (c_1) o software é composto apenas pelos arquivos A e B, que passam a possuir uma determinada versão v_1. O arquivo C, por sua vez, foi adicionado no *commit* posterior (c_2), obtendo uma versão v_1. No mesmo *commit* o arquivo B foi modificado, obtendo assim uma nova versão v_2. No último *commit* realizado (c_3) os arquivos A e C foram modificados, possuindo assim uma nova versão v_2, e o arquivo B foi deletado do projeto. Pode-se perceber também, que o primeiro registro que o SCV possui em relação a um determinado artefato ou arquivo, é referente à sua inclusão (ADD) no sistema, seguido de uma ou mais modificações (MOD) e de uma provável deleção (DEL). Os artefatos podem ser adicionados no decorrer

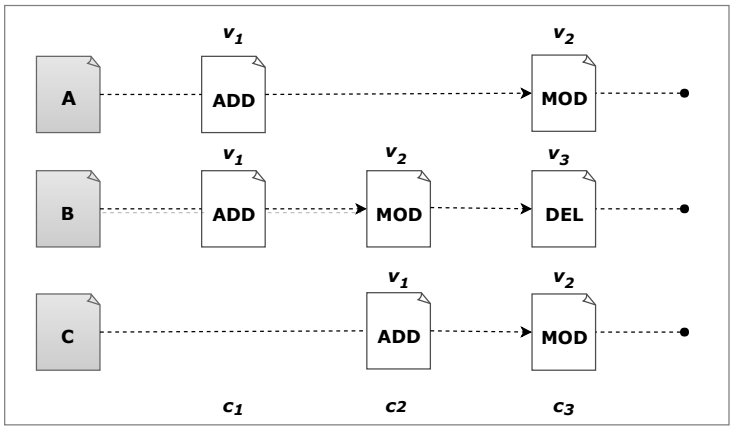


Figura 2 – Histórico de *commits* armazenado por um SCV.

Fonte – Adaptada de LIRA (2016)

do desenvolvimento (como ocorre com o arquivo C), e podem, ou não, serem alterados simultaneamente em um mesmo *commit*.

É importante frisar, que o termo "versão" no contexto de sistemas de versionamento, representa um identificador único que é atualizado a cada nova alteração (*commit*) realizada sobre um artefato (SPINELLIS, 2005). Esse identificador é tipicamente representado por um valor inteiro (OTTE, 2009). No contexto da Engenharia de Software (ES), o termo "versão" está associado a uma *release*, que indica a entrega do sistema de software como produto, após um processo que inclui a execução de testes para certificação de que o produto se encontra em um estado funcional (HUMBLE; FARLEY, 2010).

Quando se trata de diferentes estados de software, *branch* é outro conceito que vale ser ressaltado. Um *branch* ou ramificação é uma divisão da linha principal de desenvolvimento (que normalmente contém o estado do sistema que está sendo utilizado em produção, ou seja, já foi disponibilizado para usuários finais), que existe de forma independente, e ainda, compartilha um histórico em comum (SUSSMAN; BRIAN; PILATO, 2007). A utilização de ramos independentes é importante, por exemplo, quando é necessária a implementação de funcionalidades críticas ou complexas no sistema.

1.3.1 Sistema de Controle de Versão Centralizado

Em um Sistema de Controle de Versão Centralizado (SCVC), os artefatos do projeto ficam armazenados em um único repositório central (ALWIS; SILLITO, 2009; OTTE, 2009) ou servidor, onde todos os desenvolvedores podem obter a última versão de um arquivo do projeto e trabalhar simultaneamente sobre o mesmo arquivo, as operações de *commit* e *update* acontecem entre cliente e servidor. A Figura 3 apresenta a arquitetura de funcionamento de um SCVC.

As operações de *commit* e *update* são ações básicas utilizadas durante o uso de

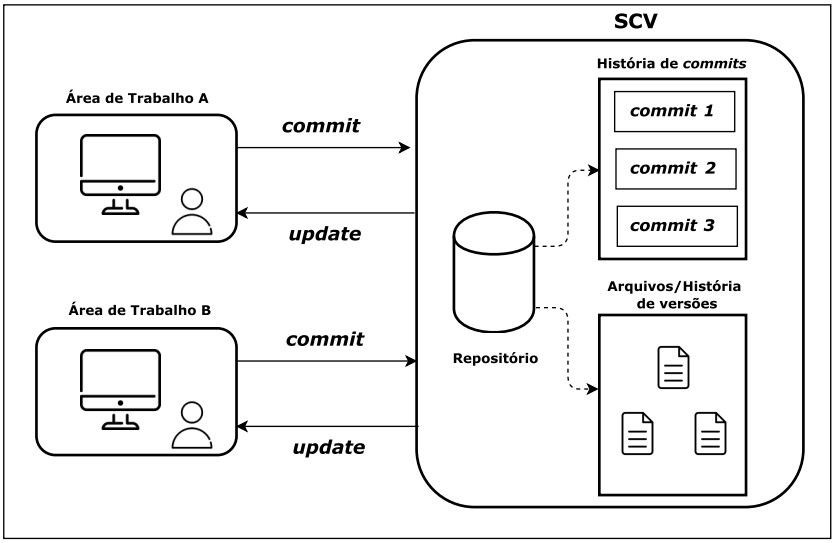


Figura 3 – Arquitetura de funcionamento de um SCVC.

Fonte – Autor.

um SCVC. Ao realizar um *update* o desenvolvedor mantém atualizados (de acordo com a última versão de cada arquivo) os artefatos do projeto em sua área de trabalho, que pode ser entendida como o ambiente (máquina/computador local) de trabalho do desenvolvedor. Para realização de um *update* é necessário que tenham sido feitas alterações (*commits*) no repositório. Essas mudanças podem incluir correções de erros ou inclusão de novas funcionalidades (SPINELLIS, 2005). Após a realização de alterações em artefatos, o desenvolvedor pode então enviar essas informações ao repositório.

A ideia de um servidor centralizado pode ser um gargalo a esses tipos de sistemas, já que o desenvolvimento poderá ser interrompido caso o servidor venha a ficar indisponível. Apesar disso, os riscos podem ser controlados e os SCV's centralizados são comumente utilizados por oferecerem simplicidade de uso e uma rápida curva de aprendizagem (BRINDESCU et al., 2014). Além disso, esses sistemas oferecem a possibilidade de respostas rápidas em relação ao envio e recebimento de arquivos do servidor. Dentre os sistemas de versionamento centralizados mais populares está o Subversion ou SVN (SUSSMAN; BRIAN; PILATO, 2007).

1.3.2 Sistema de Controle de Versão Distribuído

A diferença fundamental de um Sistema de Controle de Versão Distribuído (SCVD) para um SCVC, é que em um SCVD não necessariamente deve haver a existência de um servidor central com apenas um repositório único (OTTE, 2009). Além de permitir a iteração simultânea da equipe de projeto com o servidor, o versionamento distribuído permite também a iteração direta entre membros da equipe de desenvolvimento (RODRIGUEZ-BUSTOS; APONTE, 2012). Cada desenvolvedor possui um repositório local em sua área

de trabalho.

No SCVD o comando *checkout* é também conhecido como *clone*. Tal comando não realiza o *download* apenas de arquivos, mas sim de todo o projeto, de modo que cada máquina ligada ao servidor receberá a última atualização realizada. Em outras palavras, cada máquina de trabalho da equipe irá possuir um repositório contendo todos os artefatos e configurações do projeto. Assim, cada máquina de trabalho da equipe servirá como um servidor, deixando de lado a exigência de um repositório central (ALWIS; SILLITO, 2009).

No modelo distribuído, os comandos *commit* e *update*, ocorrem localmente em cada área de trabalho. A comunicação entre repositórios diferentes é realizada por meio do comando *pull* que traz as atualizações de outro repositório para o repositório local, e o comando *push* que envia as atualizações do repositório local para outro repositório. A Figura ?? apresenta a arquitetura de funcionamento de um SCVD.

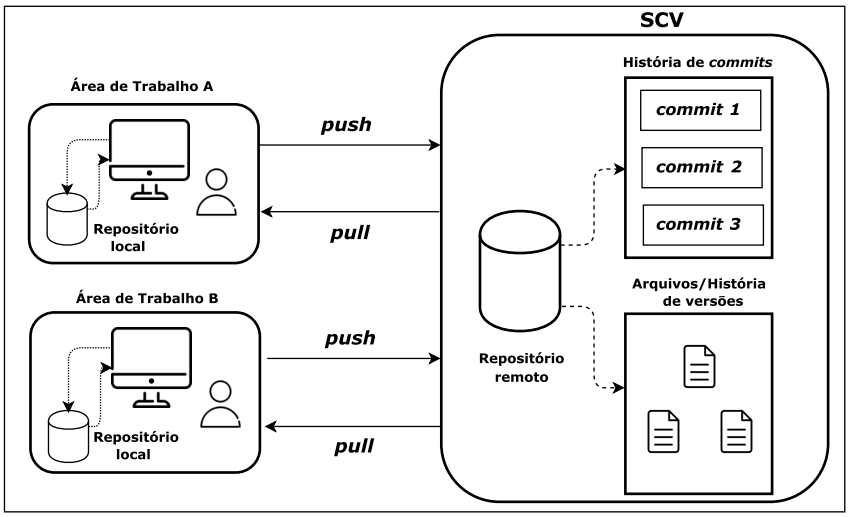


Figura 4 – Arquitetura de funcionamento de um SCVD.

Fonte – Autor.

Os sistemas de versionamento distribuído proporcionam o trabalho simultâneo de várias maneiras organizacionais e mantém o repositório muito bem atualizado. Neste modelo de versionamento, como os artefatos estão duplicados entre os vários repositórios, a tolerância à perca de arquivos é maior, reduzindo o risco de grandes problemas (ALWIS; SILLITO, 2009).

1.4 Mineração de Repositórios de Software

A Mineração de Repositórios de Software (MRS) ou MSR (do inglês *Mining Software Repositories*) é um campo de pesquisa que analisa dados históricos disponíveis em repositórios de software, tais como sistemas de controle de versão (SPINELLIS, 2005;

OTTE, 2009), arquivos de listas de discussão (como *e-mail*, por exemplo), sistemas de rastreamento de *bugs* (ZIMMERMANN et al., 2009), sistemas de rastreamento de questões (JANÁK, 2009), etc., para descobrir informações úteis sobre projetos e sistemas de software (HASSAN, 2008; HERZIG; ZELLER, 2010).

A mineração de repositórios de software emergiu como uma direção de pesquisa sobre a década passada, sendo realizada com êxito substancial tanto na pesquisa quanto na prática para apoiar várias tarefas relacionadas ao processo de desenvolvimento de projetos de software (SUN et al., 2015). O campo de MSR tornou-se uma fundamental área de pesquisa entre a comunidade de Engenharia de Software (ES) (PRESSMAN, 2005), e possui importância vital em casos de estudos empíricos (ROBLES, 2010).

No ano de 2004, surgiu o principal evento da área de MSR, denominado *Working Conference on Mining Software Repositories* (MSRConf)[1]. Atualmente, a MSRConf é a principal conferência na área de disseminação de resultados de trabalhos sobre novos processos, métodos, técnicas, ferramentas e idéias nesse campo (HEMMATI et al., 2013). A área de pesquisa em MSR vem ganhando cada vez mais notoriedade desde que surgiu, e um dos fatores que devem estar contribuindo cada vez mais para isso, além dos benefícios para área da ES, é o surgimento de novas tecnologias ou repositórios de software. A Tabela 1 apresenta exemplos de alguns tipos de repositórios.

Tabela 1 – Exemplos de repositórios de software.

Repositório	**Descrição**
Sistemas de controle de versão	Esses repositórios registram o histórico de desenvolvimento de um projeto. Eles rastreiam todas as alterações no código-fonte junto com meta-dados sobre cada alteração, por exemplo, o nome do desenvolvedor que realizou a alteração, o tempo a mudança foi realizada e uma mensagem curta descrevendo a alteração. Esses repositórios são mais comumente disponíveis e usados em projetos de software. Git e Subversion são exemplos de repositórios de controle de versão que são muito usados na prática.
Sistemas de rastreamento de *bugs*	Esses repositórios rastreiam o histórico de descoberta e resolução de bugs ou solicitações de recursos relatados por usuários e desenvolvedores de grandes projetos de software. Bugzilla e Jira são exemplos de repositórios de bugs.
Listas de discussão	Esses repositórios rastreiam as discussões sobre vários aspectos de um projeto de software ao longo de sua vida útil. As listas de endereços, e-mails, chats IRC e mensagens instantâneas são exemplos de comunicações arquivadas sobre um projeto.
Logs de *Deployment* (Implantação do software)	Esses repositórios registram informações sobre a execução de uma implantação de um aplicativo de software ou implementações diferentes dos mesmos aplicativos. Por exemplo, os *logs* de implantação podem registrar as mensagens de erro relatadas por um aplicativo em vários sites de implantação.
Repositórios de projetos	Esses repositórios arquivam o código-fonte para um grande número de projetos. Sourceforge.net e Google Code são exemplos de grandes repositórios de projeto.

Fonte – Adaptada de HASSAN (2008))

As pesquisas na área de MSR focam principalmente em dois aspectos: I) Criação de técnicas para automatizar e melhorar a extração de informações de repositórios e II) Descoberta e validação de novas técnicas e abordagens para extrair informações importantes desses repositórios. A primeira linha de trabalho é essencial para facilitar a adoção de técnicas de MSR, enquanto que a segunda linha de trabalho demonstra a importância e o

[1] http://msrconf.org/.

valor da informação armazenada em repositórios de software e encoraja outros a adotar técnicas de MSR (HASSAN, 2008).

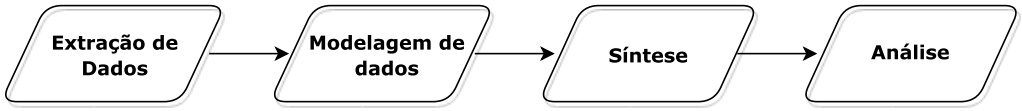


Figura 5 – Etapas típicas do processo de MSR.

Fonte – Autor

HEMMATI et al. (2013) apresentam as etapas típicas de um processo de MSR, que são: I) Extração, II) Modelagem de dados, III) Síntese e IV) Análise. A Figura 5 apresenta as etapas citadas. Primeiramente é realizada a extração de dados "brutos" (ou "crus", como definido por ROBLES (2010)) de vários tipos de repositórios de artefatos de desenvolvimento, tais como: repositórios de código, SCV, sistemas de rastreamento de bugs, listas de discussão, etc. Isso é geralmente seguido por uma fase de modelagem de dados onde é feita alguma preparação dos dados brutos, e a vinculação a dados de diferentes repositórios podem ser necessários. Na etapa de síntese é realizado um processamento dos dados extraídos. Finalmente, é necessária uma análise e interpretação dos resultados para formulação de uma conclusão.

Segundo HASSAN (2008), o processo de MSR, até recentemente, não era o foco de pesquisas na área de ES, por dois motivos principais: o acesso limitado à repositórios e a dificuldade de realizar extração de dados. O primeiro problema é consequência do fato de que empresas não estão dispostas, em geral, a fornecer aos pesquisadores acesso a informações detalhadas sobre seus sistemas de software. Contudo, esse problema vem sendo amenizado com a disponibilização de repositórios com informações de projetos industriais, tais como o repositório PROMISE (BOETTICHER; MENZIES; OSTRAND, 2007). Outro fator que também vem contribuindo é o advento de projetos de software *open source* (JAEGER; METZGER, 2002).

A dificuldade na realização de extração de dados, originou-se do fato de que muitos tipos de repositórios não terem sido projetados para permitir a extração de dados de forma automatizada. Contudo, esse problema vem sendo contornado, pois com as pesquisas crescentes nessa área, muitas ferramentas já foram desenvolvidos para facilitar o processo de MSR, especialmente na etapa de extração de dados, como por exemplo, as ferramentas ProM (VERBEEK et al., 2010) e FRASR (PONCIN; SEREBRENIK; BRAND, 2011).

Diversos problemas podem ser solucionados por meio da utilização de técnicas baseadas em MSR, tais como: detecção de entidades de código semelhantes (KIM et al., 2005; SAGER et al., 2006), predição de *bugs* (GRAVES et al., 2000; HASSAN; HOLT, 2005;

KIM et al., 2007; HASSAN, 2009; HATA; MIZUNO; KIKUNO, 2012), estimativa de esforço em desenvolvimento de software (YU, 2006; MOSER et al., 2008; ROBLES et al., 2014), análise da evolução de sistemas (GALL; LANZA, 2006; KOCH, 2007; YAN; MENARINI; GRISWOLD, 2014; CHAIKALIS et al., 2014) e reutilização de código (TANGSRIPAIROJ; SAMADZADEH, 2005; XIE; PEI, 2006). Com isso, é evidente a gama de benefícios que a aplicação de técnicas baseadas em MSR pode oferecer para a área de Engenharia de Software.

1.5 Engenharia Reversa

A Engenharia Reversa (ER) é um processo que tem como objetivo a análise de sistemas de software para identificar seus componentes atuais e dependências entre esses componentes, além de extrair e prover abstrações desses sistemas e informações de projeto (CHIKOFSKY; CROSS, 1990). Esse processo não envolve mudanças no sistema de software, mas sim atua como um processo de análise, não envolvendo alterações ou replicações (CHIKOFSKY; CROSS, 1990; MÜLLER et al., 1992).

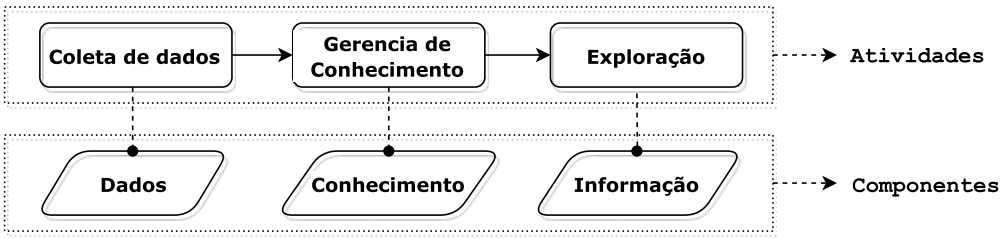


Figura 6 – Atividades e componentes do processo de engenharia reversa.

Fonte – Autor.

De forma resumida, pode-se afirmar que a ER possui ênfase no entendimento de um sistema de software, fornecendo informações adicionais sobre o mesmo (MÜLLER et al., 1992; MÜLLER et al., 2000). O processo de ER envolve três componentes básicos (TILLEY, 1995; TILLEY, 2000): I) Dados: objetos concretos utilizados como base para o estudo, raciocínio ou discussão; II) Conhecimento: a soma do que é conhecido, que inclui dados e informações, tais como relações e regras progressivamente derivadas dos dados; III) Informação: a comunicação ou recepção de conhecimentos obtidos de investigação, estudo ou instrução. Esses componentes são a base para o processo de compreensão de um sistema de software.

Com base na descrição dos componentes manipulados pela ER, surgem três atividades básicas desse processo: coleta de dados, gerenciamento de conhecimento e exploração (navegação, análise e apresentação) (TILLEY, 1995; TILLEY, 2000). Cada uma dessas

atividades está relacionada com seu respectivo componente do processo de ER. A Figura 6 apresenta o que foi citado anteriormente.

A atividade de coleta de dados pode ser realizada por técnicas de análise do sistema, que por sua vez são classificadas em estáticas (análise do código fonte) ou dinâmicas (análise da execução do sistema). A atividade de gerência de conhecimento envolve modelagem de domínio, para criar, representar e raciocinar sobre componentes e relações de interesse. Por fim, a tarefa de exploração é considerada a mais importante, pois fornece a maior parte da compreensão do sistema analisado (TILLEY, 2000).

1.6 *Truck Factor* (TF)

O ***Truck Factor*** (TF) pode ser definido como o "o número de pessoas em seu time que devem ser atropeladas por um caminhão antes que o projeto entre em sérios problemas" (WILLIAMS; KESSLER, 2002). Segundo RICCA; MARCHETTO (2010a), essa medida pode ser utilizada para identificar possíveis riscos no projeto, assinalando o quanto o projeto é dependente de determinados desenvolvedores; o custo de substituir um determinado desenvolvedor no projeto; e de que forma o conhecimento está distribuído entre os membros da equipe de desenvolvimento.

Segundo AVELINO et al. (2016a), não existe uma definição concreta de como o TF deve ser calculado. Em RICCA; MARCHETTO; TORCHIANO (2011) são discutidas as dificuldades que levam a esse fato. Apesar disso, alguns trabalhos na área propõem diferentes definições para o cálculo do TF (AVELINO et al., 2016a; ZAZWORKA et al., 2010; RICCA; MARCHETTO, 2010a).

Contudo, de maneira geral, o cálculo da métrica TF leva como base a relação entre desenvolvedor e o código fonte. Quanto menor o valor para a métrica TF, maior são os riscos para o projeto. Por exemplo, em uma equipe que tem TF com valor 1, caso esse membro entre de férias ou adoeça e tenha que se ausentar por alguns dias, provavelmente o projeto sofreria grandes problemas, seguindo em um ritmo bem desacelerado se comparado ao período em que esse desenvolvedor com maior familiaridade no projeto estava presente.

1.7 Medidas de Avaliação de Classificadores

Este trabalho tem com uma das principais contribuições uma abordagem que possui como etapas base a identificação de funcionalidades e arquivos relacionados a cada uma delas. Esse processo se comporta como uma etapa de classificação, e pode ser avaliado por meio de medidas de avaliação de classificadores. Com isso, nesta seção são apresentados os conceitos relacionados às medidas comumente utilizadas por trabalhos encontrados na literatura. Para facilitar o entendimento dos conceitos, são apresentadas

medidas e exemplos para avaliação de classificadores em problemas binários, no entanto, todas as medidas também podem ser aplicadas em problemas multi-classes (SOKOLOVA; LAPALME, 2009).

1.7.1 Matriz de Confusão

A matriz de confusão nada mais é que uma tabela para organização dos resultados obtidos da classificação realizada pelos algoritmos. Seus dados são preenchidos com o objetivo de comparar a classificação do algoritmo com o resultado esperado. A Tabela 2 apresenta um exemplo de uma matriz de confusão para problemas binários.

Tabela 2 – Matriz de confusão.

a = Positivo		**Classificação**	
b = Negativo		**a**	**b**
Resultado	**a**	Verdadeiros Positivos	Falsos Negativos
Esperado	**b**	Falsos Positivos	Verdadeiros Negativos

Fonte – Adaptado de DAVIS; GOADRICH 2006.

Os valores de a e b representam as classes hipotéticas do problema. Os valores para **Verdadeiros Positivos (VP)** indicam a quantidade de elementos da classe a que realmente foram classificados como a. Os **Falsos Positivos (FP)** indicam a quantidade de elementos da classe b que foram classificados como a. Já os **Falsos Negativos (FN)** indicam a quantidade de elementos da classe a, mas que foram classificados como b. Finalmente, os valores para **Verdadeiros Negativos (VN)** representam os elementos da classe b que realmente foram classificados como b.

1.7.2 Acurácia (A)

A Acurácia de um classificador representa a porcentagem de exemplos corretamente classificados pelo algoritmo utilizado. Por meio dessa medida é possível determinar a qualidade do classificador de forma geral, ou seja, calcula a porcentagem de acertos em relação ao total de elementos (POWERS, 2011). A Acurácia de um classificador em problemas binários pode ser calculada por meio da Equação 1.1.

$$A = \frac{VP + VN}{VP + FP + FN + VN} \tag{1.1}$$

1.7.3 Erro (E)

O valor da taxa de erro na avaliação do desempenho, representa a porcentagem de elementos classificados incorretamente. Essa medida pode ser calculada pelo complemento

da Acurácia (POWERS, 2011), conforme apresentada na Equação 1.2

$$E = 1 - A \tag{1.2}$$

1.7.4 Precisão (P)

A precisão é a medida que calcula a porcentagem de elementos verdadeiros positivos em relação ao total de elementos positivos. Essa medida mostra o desempenho do classificador em relação a cada uma das classes dos problema (DAVIS; GOADRICH, 2006). A precisão pode ser calculada por meio da Equação 1.3.

$$P = \frac{VP}{VP + FP} \tag{1.3}$$

1.7.5 *Recall* (R)

O *recall* ou sensitividade é a medida que infere a porcentagem de verdadeiros positivos em relação a soma dos elementos da classe. Assim, é possível identificar a proporção de elementos de uma classe x em relação a todos que deveriam ter sido classificados como x (DAVIS; GOADRICH, 2006). O *recall* pode ser calculado por meio da Equação 1.4.

$$R = \frac{VP}{VP + FN} \tag{1.4}$$

1.7.6 Especificidade (ES)

A especificidade, ou taxa de verdadeiros negativos, é a medida que calcula a porcentagem de amostras negativas identificadas corretamente em relação ao total de amostras negativas (POWERS, 2011). Essa medida é calculada por meio da Equação 1.5.

$$ES = \frac{VN}{VN + FP} \tag{1.5}$$

1.8 Considerações Finais

Neste capítulo foram apresentados os principais conceitos sobre técnicas e componentes utilizados neste estudo. Na Seção 1.1 foram apresentados alguns conceitos básicos sobre funcionalidades de software. Nas Seções 1.2, 1.3, 1.4, 1.5 e 1.6 foram apresentados os principais conceitos relacionados à abordagem proposta neste trabalho. A Seção 1.7 apresentou algumas das principais medidas de avaliação de classificadores, que foram

utilizadas durante a avaliação da abordagem proposta, especificamente quanto a etapa de classificação ou identificação de funcionalidades e arquivos relacionados.

2 Trabalhos Relacionados

Neste capítulo são apresentados alguns dos principais estudos que propõem métodos baseados em Mineração de Repositórios de Software (MRS) para inferir a relação de desenvolvedores com o código ou software desenvolvido. A grande maioria desses trabalhos foram identificados por meio da realização de um Mapeamento Sistemática da Literatura (MSL) (KITCHENHAM, 2004), que pode ser entendida como uma metodologia cujo objetivo é identificar, sintetizar e classificar estudos dentro de uma àrea de interresse. O processo de realização e os resultados do MSL realizado são apresentados no Apêndice A deste trabalho.

Por meio da realização desse MSL foi possível obter várias constatações importantes. A principal delas foi que a grande maioria dos estudos identificados propõem métodos para inferir o grau de relação entre desenvolvedor e código fonte considerando as perspectivas de linhas de código, arquivo ou módulo. Apenas um dos estudos identificados apresentou uma abordagem para inferir essa relação em granularidade de funcionalidade. Os principais estudos identificados são apresentados na seção seguinte.

2.1 Principais Trabalhos Relacionados

Esta seção destaca alguns dos principais trabalhos identificados durante a realização do MSL, descrevendo de forma sucinta o método proposto em cada um deles. Além disso é apresentado também, de forma geral, o trabalho proposto por LIRA (2016), que é utilizado como base para a abordagem proposta neste trabalho.

Em (LIRA, 2016) é apresentado um método para inferência da familiaridade que desenvolvedores de software possuem com o código fonte, considerando a perspectiva de arquivos ou módulos. O método tem como base as alterações realizadas sobre o código fonte de um software, obtidas a partir da extração de dados de SCV's. O cálculo da familiaridade usa métricas para tratar de aspectos relacionados ao desenvolvimento de software e ao próprio ser humano, como a sua capacidade de esquecer algo que implementou ao longo do tempo ou a possibilidade de alterações por outro desenvolvedor naquilo que foi feito. Para facilitar a visualização da familiaridade de código, foi criada uma ferramenta denominada *CoDiVision*. O método foi avaliado em diferentes contextos: em um contexto educacional, auxiliando professores na atribuição de notas à alunos de disciplinas de programação; em um contexto profissional, auxiliando gerentes de projeto no processo de identificação de membros com maior familiaridade com o código. Os resultados obtidos mostraram que as métricas desenvolvidas e a ferramenta criada são importantes aliados no desenvolvimento de software.

Em (MENG et al., 2013) os autores utilizam mineração de dados a partir de repositórios de código de um SCV. São apresentados dois modelos que visam determinar autoria de código sobre linhas de arquivo. Para determinar autoria sobre cada linha de código, esses modelos levam em conta última modificação sobre o arquivo e o histórico de modificações sobre cada linha ao decorrer do desenvolvimento. Antes da utilização destes modelos é criado um esquema de representação em formato de grafo do repositório de código utilizado. O grafo contem informações sobre as revisões do projeto e desenvolvedores responsáveis por gerar uma dada revisão, bem como as versões de código geradas em cada revisão. Os modelos apresentados foram comparados com modelos de autoria em nível de arquivo existentes. Foi verificado que os modelos propostos obtiveram melhores resultados para o conjunto de dados utilizado na análise de autoria de código.

Em (FRITZ et al., 2014) é apresentado um modelo para definição do grau de conhecimento dos desenvolvedores sobre arquivos de código do projeto. Esse modelo utiliza como base uma métrica principal chamada DOK (*Degree-of-Knowledge*) que é definida como base em duas outras métricas: DOI (*Degree-of-Interest*) e DOA (*Degree-of-Authorship*), apresentadas em trabalhos anteriores (FRITZ; MURPHY; HILL, 2007; FRITZ et al., 2010). A métrica DOI representa a quantidade de seleções e edições que um desenvolvedor específico realizou sobre uma entidade de código, e aumenta à medida que desenvolvedor continua interagindo com essa parte de código e é deteriorado gradualmente à medida que essa interação deixa de acontecer, ou seja, o grau de interesse começa a diminuir a partir da interação do desenvolvedor com outras entidades de código. Já a métrica DOA, tem como objetivo determinar a autoria sobre uma entidade de código e é baseada em três métricas auxiliares FA (*first authorship*), DL (*number of deliveries*) e AC (*number of acceptances*).

A métrica FA indica se um determinado desenvolvedor é ou não o primeiro autor de uma entidade de código específica. O valor de DL indica o número de interações feitas por um desenvolvedor sobre um arquivo. Já o valor de AC indica o número de interações realizadas por outros desenvolvedores sobre o mesmo arquivo. Com isso, a métrica principal DOK é definida e visa modelar o conhecimento de um desenvolvedor sobre uma entidade de código. O modelo proposto foi avaliado com base em projetos de software comerciais. O modelo apresentou melhores resultados em comparação às propostas anteriores.

Em (GREEVY; GIRBA; DUCASSE, 2007) os autores propõem uma abordagem para determinar quais desenvolvedores são responsáveis ou autores de arquivos e funcionalidades de um sistema de software. A abordagem para identificar autores de funcionalidades possui duas etapas: I) Determinar propriedade de código (*ownership*) por arquivo e II) Identificar quais arquivos fazem parte de cada funcionalidade, por meio de uma técnica apresentada em um trabalho anterior (GREEVY; DUCASSE, 2005), que se baseia no mapeamento dinâmico de arquivos a partir do fluxo de execução de sistemas *web* escritos

em linguagens de programação orientadas à objetos. Considera-se que uma funcionalidade inicia sua execução a partir de *links* identificados em páginas do sistema.

A propriedade de código sobre arquivos é determinada pelo número de linhas autorais dos desenvolvedores. Essa técnica é apresentada em um trabalho anterior (GIRBA et al., 2005), que também foi incluído no processo de mapeamento sistemático, descrito neste capítulo. O desenvolvedor com maior propriedade sobre linhas de um arquivo é então determinado como seu proprietário, enquanto o grupo de desenvolvedores proprietários de uma funcionalidade é composto por aqueles que são donos dos arquivos que compõem tal funcionalidade. Com isso então, é determinado quais desenvolvedores são responsáveis por cada uma das funcionalidades do sistema. A abordagem foi avaliada com base um projeto *open source* e projetos de software acadêmicos desenvolvidos em Java. Os resultados mostraram que apesar da abordagem proposta não conseguir mapear todo o código fonte relacionado a funcionalidade, prover um bom ponto de partida para determinação da relação entre desenvolvedor e funcionalidades de software.

Dentre os estudos apresentados, o que foi proposto por LIRA (2016) é utilizado como base para a abordagem proposta neste trabalho para inferir a familiaridade de código sobre os arquivos que compõem uma dada funcionalidade de software. Os demais estudos apresentados, apesar de estarem diretamente relacionados a este trabalho, cada um possui suas peculiaridades e oferecem diferentes abordagens para inferir a relação entre desenvolvedor e código fonte.

O método proposto por MENG et al. (2013) utiliza métricas que se baseiam em modificações de código em nível de linhas, que pode representar de forma mais precisa a relação entre desenvolvedor e código. A abordagem proposta neste trabalho além de considerar modificações em nível de linhas, também considera métricas que retratam um contexto real de desenvolvimento de software, como modificações em um mesmo arquivo realizadas por outros desenvolvedores ou mesmo período de tempo decorrido desde a última alteração sobre arquivos de código.

O modelo proposto por FRITZ et al. 2014 assemelha-se bastante à este trabalho, pois visa inferir a relação entre desenvolvedor e código fonte de maneira mais precisa, levando em consideração a perspectiva de arquivos de código. Os autores utilizam uma combinação de diferentes tipos de métricas baseadas nas alterações ou modificações de código realizadas pelos desenvolvedores, mas sem considerar as alterações feitas em nível de linhas de código. O método utilizado neste trabalho baseia-se em alterações considerando linhas de código e além disso utiliza diferentes métricas para inferência da relação entre desenvolvedor e código fonte, considerando a perspectiva de arquivos. Dentre essas métricas está, por exemplo, a ponderação por tipo de alteração realizada sobre linhas de código: adição, modificação ou remoção.

A abordagem apresentada por GREEVY; GIRBA; DUCASSE 2007 é a que mais

assemelha-se à proposta deste trabalho. Os autores apresentam uma abordagem para inferência da relação de código considerando a perspectiva de arquivos e também visam relacionar cada arquivo às funcionalidades do sistema, para então inferir a relação de código considerando também a perspectiva de funcionalidades.

Contudo, o método para inferência da relação de código considerando a perspectiva de arquivo é baseada na última modificação realizada sobre uma linha de código para então definir o proprietário do arquivo, com base o desenvolvedor com maior número de linhas proprietárias. Além disso, o método proposto não identifica funcionalidades de forma automática, que são consideradas *links* presentes em páginas *web* e a identificação dos arquivos de cada funcionalidade depende da execução do sistema, o que pode se tornar um gargalo. O método utilizado no trabalho aqui apresentado, por sua vez, considera todo o histórico de alterações em linhas de código, e além disso utiliza as outras diferentes métricas que visam inferir de maneira mais precisa a relação de código considerando a perspetiva de arquivo. Além disso, as funcionalidades do sistema e os arquivos que compõem cada uma são identificados de maneira automática e sem a necessidade de execução do sistema.

A abordagem proposta no trabalho aqui apresentado visa fornecer uma nova alternativa para inferência da relação entre desenvolvedor e código fonte, considerando a perspectiva de funcionalidades e utiliza com base para o cálculo da familiaridade o LIRA (2016), que considera a combinação de diferentes métricas que visam inferir de maneira mais precisa a relação entre desenvolvedor e código fonte, que é tratada neste trabalho como a familiaridade de código. Apesar do método proposto em (LIRA, 2016) ter sido brevemente descrito, é de suma importância que o mesmo seja explanado mais detalhadamente neste trabalho. Com isso, a apresentação do método é realizada na Seção 2.2.

É importante ressaltar também que este trabalho está diretamente relacionado com a área de ***Feature Location*** (FL) (DIT et al., 2013). Essa área apresenta estudos que utilizam técnicas para identificar locais/componentes de software (linhas, métodos/funcões, arquivos, etc.) que fazem parte de uma funcionalidade. Em (DIT et al., 2013) são apresentados de maneira geral estudos que trazem técnicas para *featute location*.

Dentre os trabalhos que propõe técnicas de FL, pode-se destacar os realizados por CHEN; RAJLICH (2000), CHEN; RAJLICH (2000), ROBILLARD; MURPHY (2002), BUCKNER et al. (2005) e ROBILLARD; MURPHY (2007). Esses estudos citados, relacionados a *feature location*, são os que mais se assemelham a este trabalho, por utilizarem técnicas de análise estática de código. Contudo, grande parte desses estudos identificam componentes (locais) relacionados a *features* em nível de granularidade de método, ao contrário deste trabalho que se baseia em nível de granularidade de arquivos.

É importante ressaltar que os estudos citados, relacionados a FT, não foram identificados no mapeamento realizado porque esses focam exclusivamente na identificação

de funcionalidades e componentes relacionados, e não visam identificar algum tipo de relação entre desenvolvedor e código fonte.

2.2 Método para Inferência da Familiaridade de Código na Perspectiva de Módulos

Esta seção apresenta em detalhes o método que visa inferir a familiaridade de código na perspectiva de arquivos ou módulos de software, que foi proposto por LIRA (2016), e compõe a abordagem proposta neste trabalho.

O método descrito nesta seção possui como etapas a **extração de dados** de Sistemas de Controle de Versão (SCV), e a definição de métricas para **inferência da familiaridade** de código sobre módulos de software, que são formados por um ou mais arquivos de código fonte. Essas etapas são descritas a seguir.

2.2.1 Extração dos Dados

Essa etapa consiste em extrair os dados de repositórios de software para identificar os responsáveis por cada *commit* realizado para o repositório. A partir desses *commits* são obtidos os arquivos que foram alterados para então determinar a quantidade de linhas de código modificadas em cada um. Com isso, é mensurada a quantidade de alterações feitas pelos desenvolvedores em cada *commit*. As alterações são analisadas de duas maneiras. A primeira é a análise do arquivo alterado como um todo, isto é, ao modificar qualquer parte de um arquivo, é considerado como uma alteração, independente da quantidade de linhas de código alteradas. A segunda maneira consiste em identificar cada linha alterada em um arquivo. Essas alterações podem ser: uma adição, deleção ou modificação.

A adição é considerada a inclusão de uma nova linha no arquivo, assim como a deleção é a remoção de uma linha existente nesse arquivo. Já a modificação é identificada por meio da utilização do algoritmo de Levenshtein (SANKOFF; KRUSKAL, 1983), que calcula a diferença entre duas cadeias de caracteres, que é dada pelo número de operações necessárias (mudanças de caracteres) para transformar uma cadeia na outra. Desse modo, é verificado se em um conjunto de operações adição/deleção se o valor dado pelo algoritmo de Levenshtein representa menos de 25% do tamanho da primeira cadeia, ou seja, é verificado se quantidade de caracteres que tem de ser alterada na primeira cadeia para transformá-la na segunda representam menos de 25% do total caracteres, em caso afirmativo, não considera-se uma adição seguido de deleção, mas uma modificação.

As linhas alteradas em cada arquivo de *commits* podem ser obtidas por meio de um arquivo de *diff*. O *diff* de um arquivo exibe exatamente o que mudou nesse arquivo entre uma versão e outra. A Figura 7 mostra um exemplo de *diff* no formato unificado. As

duas primeiras linhas (“—” e “+++”) indicam o arquivo no qual está sendo feito o *diff* e a versão desse arquivo. Em seguida, podem ocorrer um ou mais trechos que iniciam com “@@” que apresentam a linha inicial do trecho que segue e a quantidade de linhas desse trecho na versão anterior (“-”) e na versão atual (“+”) do arquivo respectivamente. Em seguida são exibidas as linhas adicionadas na versão atual (“+”), as linhas que existiam na versão anterior, mas foram removidas na versão atual (“-”) e as linhas inalteradas.

```
Index: ArquivoA
===================================
--- ArquivoA    (revision X)
+++ ArquivoA    (revision Y)

@@ -10,1 +10,2 @@

-     linha 1
+     linha 1.1
+     linha 2
```

Figura 7 – Exemplo de *diff*.

Fonte – LIRA (2016)

A quantidade de linhas alteradas que envolvem comandos condicionais (*Se... Então*) também são computadas. Isso porque LIRA (2016) acredita que linhas de código que envolvem esses tipos de comandos possuem uma maior complexidade, e por esse motivo são levadas em consideração. Durante a extração de dados não é armazenado o código fonte. São armazenados apenas metadados gerados pelo SCV. Os *diffs* obtidos são utilizados apenas para extrair a quantidade de alterações (adições, modificações, deleções) realizadas em cada *commit*. A quantidade de alterações de cada desenvolvedor sobre módulos do software, é a informação efetivamente armazenada.

2.2.2 Inferência da Familiaridade de Código

A segunda etapa do método proposto por LIRA (2016) consiste no processo de inferência da familiaridade de código sobre módulos de software. Para isso, são utilizadas algumas métricas para auxiliar nesse processo. As métricas propostas têm o objetivo de transformar o valor absoluto das alterações dos desenvolvedores em um valor relativo por meio da adição de aspectos da natureza humana e do próprio processo de desenvolvimento. Essas características envolvem a importância de determinados tipos de alteração, o esquecimento por parte dos desenvolvedores em relação ao que foi feito e a sobreposição disso por outros desenvolvedores durante o desenvolvimento. Essas métricas são apresentadas nas subseções seguintes.

2.2.2.1 Ponderação das Alterações

Como apresentado anteriormente, são considerados 4 tipos de alteração (adição, modificação, deleção e condição) nas linhas dos arquivos contidos nos *commits*, além das alterações nos próprios arquivos. LIRA (2016) considera que cada tipo de alteração possui uma esforço diferente para ser realizada. Por exemplo, alguns desenvolvedores consideram que é mais fácil remover, ao invés de adicionar, uma linha de código. Desse modo, é atribuído peso a cada tipo de alteração, com a finalidade de balancear o esforço necessário associado a cada tipo de alteração. O cálculo das alterações realizadas de acordo com seus respectivos pesos é dado pela Equação 2.1.

$$\begin{aligned} W(d, a(v)) = (ADD^{d,a(v)} * W_{ADD}) + (MOD^{d,a(v)} * W_{MOD}) + \\ (DEL^{d,a(v)} * W_{DEL}) + (COND^{d,a(v)} * W_{COND}) \end{aligned} \tag{2.1}$$

Onde:

- $W(d, a(v))$: é o valor ponderado da quantidade de alterações realizadas pelo desenvolvedor d na versão v do arquivo a após a aplicação dos respectivos pesos;
- ADD: é a quantidade de linhas adicionadas pelo desenvolvedor d na versão v do arquivo a;
- MOD: é a quantidade de linhas modificadas pelo desenvolvedor d na versão v do arquivo a;
- DEL: é a quantidade de linhas apagadas pelo desenvolvedor d na versão v do arquivo a;
- $COND$: é a quantidade de linhas contendo instruções condicionais alteradas pelo desenvolvedor d na versão v do arquivo a;
- W_{ADD}: é o peso associado às adições;
- W_{MOD}: é o peso associado às modificações;
- W_{DEL}: é o peso associado às deleções;
- W_{COND}: é o peso associado às condições.

2.2.2.2 Degradação por Tempo

A segunda métrica é baseada em algo bastante comum do ser humano, que é o esquecimento de algo feito antes, principalmente se isso foi realizado há algum tempo. Quando se trata de código não é diferente. À medida que desenvolvedores de software passam algum tempo sem interagir com um código, mesmo sendo o autor, é inevitável

não esquecer de como foi implementado, o que requer um esforço maior para dar manutenção nessa parte do software. A métrica de degradação por tempo pretende simular esse esquecimento, como algo natural do ser humano. Essa degradação é calculada da seguinte maneira: um pequeno valor, que aumenta de acordo com a quantidade de dias passados desde a data do *commit*, é subtraído da quantidade total de alterações feitas pelo desenvolvedor. A Equação 2.2. apresenta o que foi discutido.

$$T(d, a(v)) = \{1 - [(D_{atual} - D_v) * P_t]\} \tag{2.2}$$

Onde:

- $T(d, a(v))$: é a porcentagem de familiaridade que restará após a aplicação da degradação por tempo;
- D_{atual}: é a data da realização do cálculo;
- D_v: é a data em que foi gerada a versão v;
- P_t: Porcentagem aplicada sobre o tempo decorrido (fator de esquecimento).

2.2.2.3 Degradação por Nova Alteração

A familiaridade que um membro da equipe possui sobre um arquivo pode ser determinado pela quantidade de alterações feitas por ele nesse arquivo. Porém é importante observar que como um projeto de software é realizado em equipe, vários usuários podem alterar os mesmos arquivos constantemente. Como consequência disso as alterações feitas por um membro podem ser sobrescritas por outro, ou um desenvolvedor diferente pode adicionar um novo conteúdo e com isso a familiaridade inicial acerca do arquivo pode não ser a mesma depois dessas várias alterações pelas quais o arquivo passou.

Semelhante ao que ocorre com a degradação por tempo, a métrica de degradação por nova alteração pretende simular o impacto dessas novas alterações feitas por outros membros da equipe na familiaridade acerca do arquivo para o membro que fez as alterações anteriores. Para isso, um valor é subtraído da quantidade total de alterações, baseado na quantidade de alterações realizadas por outros desenvolvedores após a versão do arquivo que está sendo considerado no cálculo. Esse cálculo pode ser observado na Equação 2.3.

$$N(d, a(v)) = [1 - (F * P_n)] \tag{2.3}$$

Onde:

- $N(d, a(v))$: é o valor da quantidade de alterações realizadas pelo desenvolvedor d na versão v do arquivo a após a aplicação da degradação por nova alteração;

- F: é a quantidade de vezes que o arquivo a foi alterado por um desenvolvedor $x \neq d$ em versões $y > v$;

- P_n: é a porcentagem aplicada sobre a quantidade de novas alterações.

2.2.2.4 Cálculo da Familiaridade de Código

Para inferir a familiaridade de código de cada desenvolvedor é utilizada a Equação 2.4, que leva em consideração cada uma das métricas descritas anteriormente.

$$M(d,p) = \sum_{a=1}^{A}[\sum_{v=1}^{V}(W(d,a(v)) * T(d,a(v)) * N(d,a(v)))] \tag{2.4}$$

Onde:

- p: é o conjunto de arquivos nos quais serão aplicadas as métricas;

- d: é o desenvolvedor para o qual serão aplicadas as métricas;

- a: é o índice do arquivo atual;

- A: é o n-ésimo arquivo;

- v: é o índice da versão atual do arquivo;

- V: é a n-ésima versão;

- $W(d,a(v))$: é valor calculado pela Equação 2.1;

- $T(d,a(v))$: é valor calculado pela Equação 2.2;

- $N(d,a(v))$: é valor calculado pela Equação 2.3.

Portando, o resultado de $M(d,p)$ indica a familiaridade de código que cada desenvolvedor possui sobre um módulo de software específico. Esse método foi implementado em uma ferramenta denominada *CoDiVision*.

Com base nisso, a abordagem proposta neste trabalho utiliza o método apresentado nesta seção para inferir a familiaridade de código em um conjunto de arquivos de uma dada funcionalidade. Assim, como proposta final, tem-se uma abordagem para inferência da familiaridade de código considerando um novo ângulo de visão, isto é, funcionalidades de software. A abordagem proposta neste trabalho é apresenta detalhadamente no Capítulo 3.

2.3 Considerações Finais

Este capítulo apresentou os principais estudos indentificados durante a realização de um Mapeamento Sistemático realizado como o objetivo de identificar, analisar e sumarizar o estado da arte sobre estudos que apresentam técnicas baseadas em mineração de repositórios de software para inferir a relação entre desenvolvedor e código fonte considerando diferentes perspectivas. Esse estudo é apresentado com detalhes no Apêndice A, e fornece uma visão geral dessa linha de pesquisa, identificando os principais tópicos abordados por trabalhos anteriores e destacando lacunas passíveis de investigação. Isso atende um dos objetos desta dissertação.

Pôde-se perceber que grande parte dos estudos visam inferir a relação de código fonte considerando a perspectiva de linhas de código, arquivos ou módulo de software. Dentre eles, foi apresentado em mais detalhes o método (LIRA, 2016) englobado pela abordagem proposta neste trabalho, para inferir a familiaridade de código sobre a perspectiva de funcionalidades. Este trabalho, portanto, apresenta uma nova abordagem que visa explorar essa direção de pesquisa.

É importante ressaltar ainda, que o trabalho proposto por LIRA (2016) não apareceu no Mapeamento Sistemático realizado porque apesar de ter sido publicado em eventos científicos, ainda não foi traduzido para o inglês, e um dos critérios de exclusão utilizados para seleção de estudos foi a restrição de que os trabalhos fossem escritos nesse idioma.

3 Abordagem Proposta

A abordagem proposta neste trabalho tem como principal objetivo inferir a familiaridade de código dos desenvolvedores em relação às funcionalidades de um software. A abordagem proposta tem como algumas das etapas principais, a identificação de funcionalidades e os respectivos arquivos de código fonte que as compõem. Assim, é possível então realizar a inferência da familiaridade de código sobre funcionalidades de software. A abordagem proposta neste trabalho é composta basicamente por 4 (quatro) etapas: 1) Extração de Dados; 2) Identificação de Funcionalidades; 3) Identificação de Arquivos Relacionados às Funcionalidades e 4) Inferência da Familiaridade por Funcionalidades. A Figura 8 apresenta as etapas da abordagem proposta.

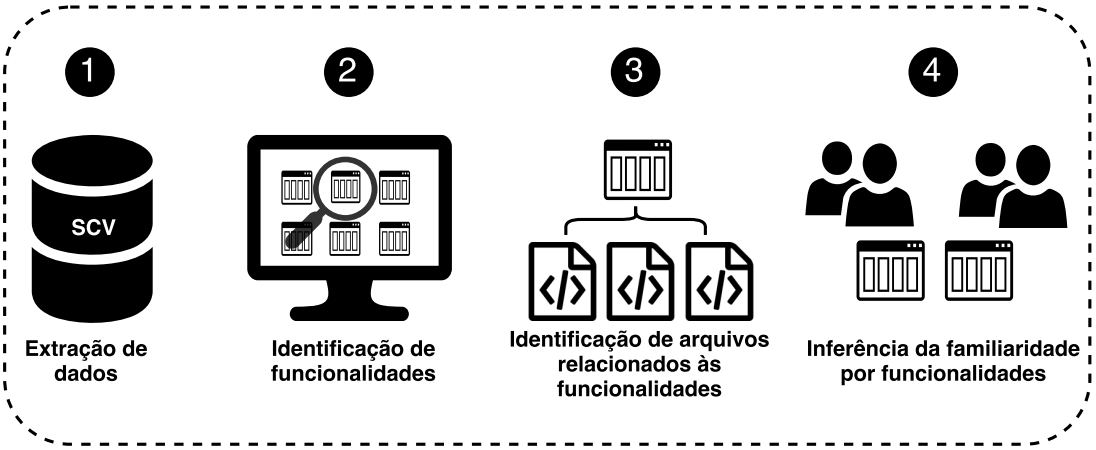


Figura 8 – Etapas da abordagem proposta.

Fonte – Autor.

Na etapa de **Extração de Dados**, devem ser obtidos, a partir de um Sistema de Controle de Versão (SCV), todos os arquivos de código fonte do software, assim como o histórico de alterações de código sobre cada arquivo. Durante a etapa de **Identificação de Funcionalidades**, devem ser identificados os requisitos funcionais do sistema. Na etapa de **Identificação de arquivos Relacionados a Funcionalidades**, devem ser mapeados os arquivos de código fonte do sistema, que estão ligados a cada uma das funcionalidades identificadas na Etapa 2. Por fim, durante a etapa de **Inferência da Familiaridade por Funcionalidades** deve-se realizar o cálculo da familiaridade que cada desenvolvedor possui sobre o conjunto de arquivos que compõe cada funcionalidade identificada. É importante ressaltar que as Etapas 2 e 3 compõe uma técnica alternativa para **Feature Location** (FL) (DIT et al., 2013), uma área que tem por objetivo especificamente a identificação de componentes ligados às funcionalidades de software.

3.1 Etapas da Abordagem

Esta seção apresenta detalhadamente as etapas que compõem a abordagem proposta neste trabalho.

3.1.1 Extração de Dados

O objetivo principal desta etapa é realizar a extração de dados a partir de um Sistema de Controle de Versão (SCV) (SPINELLIS, 2005). Como visto na Seção 1.3 do Capítulo 1, tais sistemas armazenam, além do código fonte de um software, todas as informações a respeito de alterações realizadas sobre tal código.

Dessa forma, é possível obter todos os arquivos de código de um sistema. Além disso, é possível identificar o trabalho realizado por todos os desenvolvedores de software, em termos de adição, remoção ou modificações de arquivos ou linhas de código, durante toda a fase de desenvolvimento do sistema.

3.1.2 Identificação de Funcionalidades

O objetivo principal desta etapa é identificar as funcionalidades pertencentes a um sistema de software. É importante ressaltar que as *features* identificadas são em nível de **subfunção**, representadas neste trabalho por métodos principais em um arquivo principal ou controlador no sistema. Vale ressaltar também que esses arquivos controladores são mais comuns em sistemas que utilizam o padrão arquitetural MVC (*model-view-controller*) ((KRASNER; POPE et al., 1988)), tipicamente utilizados em sistemas **Web**, por exemplo. Para alcançar o objetivo desta etapa, são seguidas 4 (quatro) subetapas: I) Extração de informações dos arquivos de código; II) Determinação do relacionamento entre arquivos III) Identificação de arquivos controladores; IV) Identificação de métodos principais em arquivos controladores. A Figura 9 ilustra as etapas apresentadas para identificação de funcionalidades.

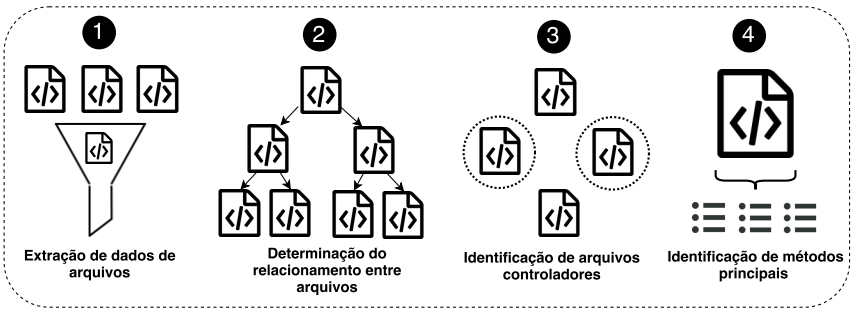


Figura 9 – Etapas do processo para identificação de funcionalidades.

Fonte – Autor.

Durante a subetapa I devem ser extraídos dados referentes ao código contido em cada arquivo do sistema. Isso é feito com base na obtenção da *Abstract Syntax Tree* (AST) (PFENNING; ELLIOTT, 1988) de cada arquivo de código. As informações são referentes a todos os dados contidos em um arquivo de código, como por exemplo, variáveis e métodos (incluindo seus parâmetros e tipo de retorno). Com tais informações é possível inferir uma ligação entre um arquivo de código e diversos outros, por exemplo, por existirem variáveis que são de um tipo declarado em outro arquivo. Durante a subetapa II deve-se identificar o relacionamento entre arquivos, isto é, determinar para cada arquivo a, o conjunto de outros arquivos do sistema referenciados por a. Isso é feito com base nos dados de cada arquivo de código extraídos na subetapa anterior.

Durante a subetapa III deve-se identificar arquivos principais, denominados aqui por **controladores**, isto é, aqueles responsáveis por iniciar a execução de funcionalidades do sistema. Para isso, é criado um grafo (WEST et al., 2001) de relacionamento entre arquivos de código para então identificar, com base em referências realizadas, os arquivos principais. Por fim, na subetapa IV, devem ser identificados os métodos ou funções principais de cada arquivo controlador. Os métodos ou funções principais de um arquivo controlador representam os pontos de início de execução das funcionalidades do sistema. Um método principal é identificado principalmente pelo número de chamadas a outros métodos internos ou não ao mesmo arquivo.

Assim, como resultado desta etapa de identificação de funcionalidades de software, deve ser gerado um conjunto definido por $F = \{f_1, f_2, ..., f_n\}$, onde n é o número de funcionalidades identificadas e f_i representa a i-ésima funcionalidade.

É importante ressaltar que durante esta etapa sejam identificadas falsas funcionalidades. Dessa forma, a abordagem possibilita o refinamento ou exclusão manual de falsas *features* identificadas. Além disso, a abordagem permite que seja feita de forma manual a agregação de *features*, para que seja possível formar funcionalidades de maiores níveis (**usuário e/ou agregador**).

3.1.3 Identificação de Arquivos Relacionados às Funcionalidades

O objetivo principal desta etapa é identificar os arquivos que pertencem a cada funcionalidade identificada.

Para alcançar esse objetivo, deve-se identificar, a partir de cada método ou função principal de um arquivo controlador, quais os arquivos referenciados durante o fluxo de chamadas a outros métodos, que podem pertencer ao mesmo arquivo ou a outros arquivos do sistema. Em outras palavras, deve mapear arquivos de código que são referenciados no fluxo que uma funcionalidade pode fazer durante sua execução. É importante frisar que essa etapa é realizada por meio de técnicas de Engenharia Reversa (CHIKOFSKY;

CROSS, 1990), mas de maneira estática. Isto quer dizer que não é necessário a execução do sistema, mas especificamente da funcionalidade, para mapear as classes referenciadas durante o fluxo de chamadas.

Como resultado desta terceira etapa deve ser gerado para cada funcionalidade f do sistema, um conjunto A_f de arquivos associados, sendo que A_f é composto por um subconjunto de arquivos do sistema.

3.1.4 Inferência da Familiaridade por Funcionalidades

Nesta etapa deve-se inferir a familiaridade de código de desenvolvedores para com as funcionalidades do sistema. Para alcançar esse objetivo, deve-se inferir a familiaridade de código que cada desenvolvedor possui sobre os arquivos relacionados a uma dada funcionalidade.

A familiaridade dos desenvolvedores sobre arquivos do sistema é realizada com base nas alterações realizadas sobre cada arquivo. Essas alterações incluem principalmente adições, remoções ou modificações sobre linhas de código. Com isso então, a familiaridade de um desenvolvedor sobre uma determinada funcionalidade é dada pela familiaridade detida sobre os arquivos de código que pertencem à funcionalidade específica.

3.2 Implementação da Abordagem

Esta seção detalha a implementação da abordagem proposta neste trabalho, apresentando como cada uma das etapas foi implementada e as tecnologias utilizadas, para que seja possível alcançar o objetivo principal de inferência da familiaridade sobre funcionalidades de software.

A abordagem proposta engloba um método para identificação de funcionalidades, bem como arquivos de código fonte que compõem cada funcionalidade identificada. Para alcançar esse objetivo, utiliza-se técnicas de Engenharia Reversa (ER).

Essa análise é dependente do tipo de componente ou arquivo de código fonte a ser analisado. Isso significa, que são utilizadas diferentes técnicas de análise para diferentes tipos de arquivos de código fonte, isto é, para diferentes linguagens de programação. Isso se deve ao fato de que, dependendo da linguagem de programação, existem diferentes maneiras para arquivos de código fazerem referências a outros arquivos. A realização de uma análise para um tipo de linguagem de programação requer então um bom nível de conhecimento na linguagem em questão.

Assim, inicialmente, o método proposto para identificação de funcionalidade e arquivos relacionados, foi implementado para identificar funcionalidades em sistemas desenvolvidos em linguagem de programação Java (GOSLING; MCGILTON, 1995). A

linguagem Java possui uma série de especificações contidos na plataforma *Java Enteeprise Edition* (JEE) (PRAJAPATI; DABHI, 2009), que permite o desenvolvimento de sistemas comerciais de grande porte, como sistemas *Web*, por exemplo. Além disso, essa linguagem é baseada no paradigma de Orientação a Objetos (OO) (KIM; LOCHOVSKY, 1989). Esse paradigma fornece um modelo natural para construção de sistemas heterogêneos, autônomos e distribuídos (PITOURA; BUKHRES; ELMAGARMID, 1995). Por essas razões, muitas empresas de desenvolvimento de software costumam desenvolver sistemas escritos nessa linguagem. A linguagem Java é também comumente utilizada em meios acadêmicos. Essas circunstâncias motivaram a uma implementação do método para identificação de funcionalidades e arquivos relacionados, sobre sistemas desenvolvidos em Java.

É importante ressaltar que existem sistemas, especialmente *Web*, que são desenvolvidos por meio da utilização de uma ou mais linguagens de programação. Nesses casos, durante a análise dos arquivos de código, são considerados apenas aqueles que foram escritos em Java.

3.2.1 Extração de Dados

Nesta etapa são extraídos dados a partir de um Sistema de Controle de Versão (SCV). Os arquivos podem ser obtidos dos principais tipos de sistemas de versionamento: Git (CHACON; STRAUB, 2014) ou SVN (SUSSMAN; BRIAN; PILATO, 2007). Os dados extraídos são: a) Histórico de *commits* realizados; b) *Diffs* (apresenta a diferença entre duas versões de um mesmo arquivo) de arquivos alterados e c) Arquivos de código fonte do sistema. Para a realização desta etapa de extração de dados, são utilizados os módulos já existentes na ferramenta *CoDiVision* (LIRA, 2016). A ferramenta utiliza-se das API's JGit[1] e SVNKit[2] para extração de repositórios Git e SVN, respectivamente.

3.2.2 Identificação de Funcionalidades

Esta etapa é dividida em 4 (quatro) fases ou subetapas. Durante a primeira fase, são extraídas informações de cada arquivo de código do sistema, que neste caso, são referentes a classes Java. Durante a segunda fase é construída uma representação, baseada em um grafo direcional (BONDY; MURTY, 1976), de referências entre arquivos, para identificar o referenciamento entre classes do sistema.

Durante a terceira fase é definido o conjunto de classes controladoras (*controllers*) do sistema, isto é, que são responsáveis pelo início da execução de funcionalidades. Por fim, durante a quarta fase são identificados os métodos principais de cada uma das classes controladoras. Esses métodos representarão as funcionalidades identificadas no sistema.

[1] http://www.eclipse.org/jgit/.
[2] https://svnkit.com/.

3.2.2.1 Extração de Dados de Arquivos

Esta seção descreve como foi implementada a subetapa para extração de dados a partir de arquivos de código fonte Java. Primeiramente, o conteúdo de cada classe é processado por um *parser* desenvolvido por meio da API Eclipse JDT (PLATFORM, 2012). Esse *parser* mapeia o código fonte Java em uma *Abstract Syntax Tree* (AST) (PFENNING; ELLIOTT, 1988), como apresentado na Figura 10.

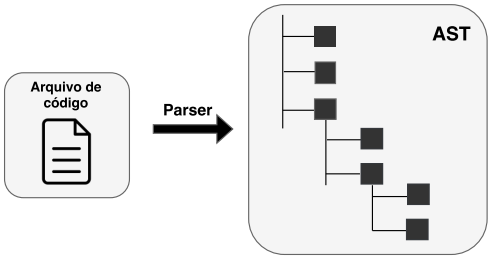


Figura 10 – Mapeamento do código fonte Java em uma AST.

Fonte – Autor.

Uma AST é mais conveniente e confiável para analisar o código fonte. A AST permite uma rápida identificação de dados a respeito de uma classe. Todas essas informações estão contidas em *nodes* ou nós da AST gerada para uma determinada classe Java, e podem ser identificadas de forma mais confiável por meio da utilização da API. A Tabela 3 apresenta os tipos de *nodes* analisados a partir de uma AST, para que seja possível obter dados a partir de uma classe Java.

Tabela 3 – Tipos de *nodes* analisados a partir de uma AST.

Node AST	**Descrição**
Import Declaration	Referente a *imports* da classe.
Compilation Unit	Referente à unidade de código fonte ou classe, com isso é possível obter a superclasse na relação de herança.
Field Declaration	Referente aos atributos da classe.
Single Variable Declaration	Referente às varáveis que são parâmetros em métodos da classe, incluindo o construtor.
Variable Declaration Statement	Referente às variáveis declaradas em métodos da classe, incluindo o construtor.
Simple Name	Referente a *tokens* (não inclui palavras reservadas) encontrados no conteúdo do código fonte da classe.
Anonymous Class Declaration	Referente às classes anônimas declaradas.
Method Declaration	Referentes a métodos da classe.
Method Invocation	Referentes à chamada de métodos.

Fonte – Autor.

Assim, durante esta subetapa é gerado um conjunto de dados para cada classe (arquivo) do sistema. Para cada uma delas, são mapeados dados, como: identificador (nome) da super classe na relação de herança (caso estenda outra classe); nomes de interfaces implementadas (caso implemente alguma interface do sistema); nomes de atributos da

classe (incluindo tipos) e informações sobre métodos (inclusive nome, parâmetros, retorno e modificadores), classes anônimas declaradas internamente; invocações ou chamadas a métodos (internos ou não a mesma classe), *imports* declarados, nome do pacote em que a classe está inserida e *tokens* (palavras como nome de variáveis, classes, métodos, nomes que compõem *imports*, etc.) presentes na classe, exceto palavras reservadas à linguagem Java.

Dessa forma, é criado um conjunto $C = \{c_1, c_2, ..., c_n\}$, onde c_i representa a i-ésima classe ou arquivo Java e n o número total de classes do sistema. Cada classe c contém todas as informações extraídas a partir da análise realizada por meio do *parser* utilizado.

3.2.2.2 Determinação do Relacionamento entre Arquivos

Esta subetapa visa identificar quais classes são referenciadas por cada classe individual do sistema. Dessa forma, para cada classe c, é criado também um conjunto $C_r = \{c_1, c_2, ..., c_n\}$, onde c_i representa a i-ésima classe referenciada por c. A construção desse conjunto é feita a partir dos dados extraídos durante a subetapa anterior.

Com base nisso, é criada uma representação de referência entre classes do sistema. Essa representação é modelada como um grafo direcional (BONDY; MURTY, 1976), onde cada vértice do grafo representa uma classe do sistema e a aresta $e = (c_i, c_j)$ indica que a classe c_i referencia c_j. A Figura 11 apresenta um exemplo de grafo de referência entre classes de um sistema.

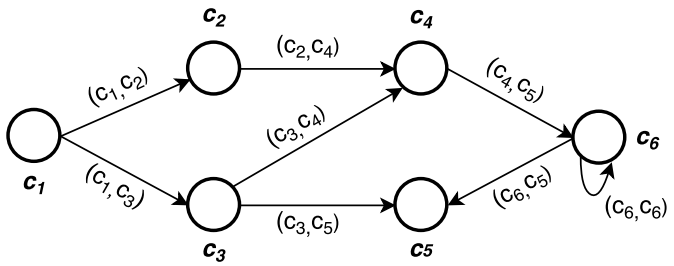


Figura 11 – Grafo de referência entre classes Java.

Fonte – Autor.

Pode-se perceber que a aresta (c_1, c_2), por exemplo, indica que a classe c_1 referencia c_2, e assim por diante. Pode-se observar também, que o nó c_6 possui um ciclo, significando que a classe c_6 faz referência à si própria. Assim, nesta subetapa é gerado um grafo de relacionamento entre os arquivos ou classes do sistema, com base nos conjuntos C, criado durante a subetapa anterior, e C_r. O grafo de referência entre classes é representado por G.

3.2.2.3 Identificação de Arquivos Controladores

Nesta subetapa são identificados os arquivos ou classes principais do sistema, isto é, aquelas que desempenham o papel de *controllers* ou controladoras. Uma classe controladora é responsável pelo início da execução de uma funcionalidade, ou seja, a partir dessa classe são referenciadas as demais classes responsáveis por concluir a execução de uma funcionalidade específica.

Com isso, durante esta subetapa, é gerado um conjunto de classes fortemente candidatas a serem controladoras, com base no grafo G de referência entre classes, que foi definido durante a subetapa anterior (Seção 3.2.2.2). Para identificação de classes controladoras, são utilizadas as seguintes métricas: existência de **ciclo em um vértice**, **grau de entrada** e **saída** de um vértice e **média dos graus de entrada** e **saída** de todos os vértices do grafo de referência entre classes. Os graus de entrada e saída de um vértice v representam, respectivamente, o número de classes que referenciam e são referenciadas por v. A partir disso, são definidas também algumas heurísticas para identificar quais arquivos podem ser, ou não, classes candidatas a serem controladoras. Essas heurísticas foram definidas empiricamente e são apresentadas a seguir:

- h_1: Se um vértice v do grafo possuir grau de saída igual a 0 (zero) ou conter ciclo, a classe referente a v não é considerada *controller*. Isso é justificado pelo fato de que é comum classes controladoras referenciem pelo menos uma outra classe do sistema. Além disso, a chance de um *controller* referenciar a si próprio é pouco provável;

- h_2: Se um vértice v possuir grau de entrada igual a 0 (zero), a classe referente a v é definida como candidata a ser *controller*. Isso é justificado pelo fato de que classe controladoras não são referenciadas (exceto por outros controladores) por outras classes do sistema, pois são responsáveis pelo início da execução de uma funcionalidade;

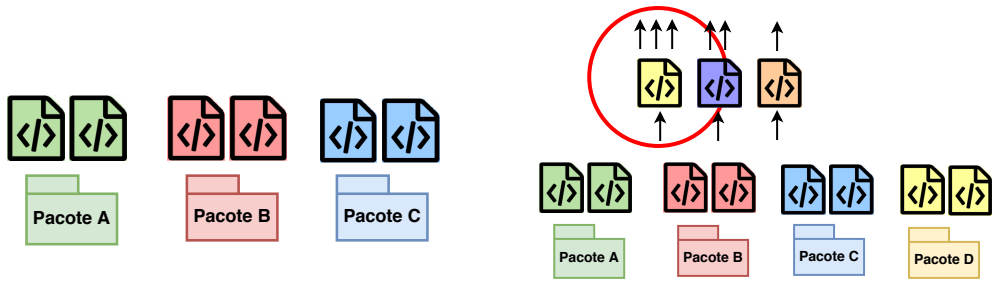
- h_3: Se um vértice v possuir grau de saída maior que a média dos graus de saída de todos os nós do grafo e o grau de entrada menor que a média de todos os graus de entrada do grafo, v é então dita como candidata a ser *controller*. Isso é justificado pelo fato de que uma classe controladora tende a referenciar um maior número de arquivos, além de ser menos referenciada em relação às demais classes.

Com base nas heurísticas apresentadas, são pré-selecionadas as classes que são fortes candidatas a serem controladoras do sistema. Contudo, os conjunto de classes pré-selecionadas ainda não representam os arquivos controladores do sistema, apesar de serem fortemente candidatos. Com isso, é realizado um **processo de filtragem** de classes pré-selecionadas por meio da aplicação das heurísticas definidas. Esse processo de filtragem consiste em 4(quatro) fases, que são apresentadas nas subseções seguintes.

3.2.2.3.1 Processo de Filtragem (1ª Fase – Separação de Arquivos por Pacote)

A primeira fase consiste em realizar a separação de classes candidatas a serem controladores entre seus respectivos pacotes, que serão chamados aqui por pacotes *controllers*. Para isso, as classes que não são referenciadas por nenhum outro arquivo (isto é, cujos vértices associados no grafo G possuem grau de entrada igual a 0) são previamente definidas como controladoras, e então é inserida em seu devido pacote.

Aquelas classes que são referenciadas (ou seja, que o vértice correspondente no grafo G de referência possua grau de entrada maior ou igual a 1), apenas são incluídas caso pertençam a algum pacote de uma das classes que não são referenciadas (cujo vértice correspondente possua grau de entrada igual a 0). A separação de pacotes com base na seleção de um arquivo que não é referenciado por nenhum outro, é ilustrada pela Figura 12a.



(a) Baseando-se na seleção de arquivos cujos vértices associados possuem grau de entrada igual a 0.

(b) Baseando-se em arquivo referenciado, mas cujo vértice associado possui maior grau de saída.

Figura 12 – Separação de arquivos por pacote.

Fonte – Autor

Esta fase gera como resultado um ou mais pacotes candidatos a serem *controllers* e que cada pacote contém pelo menos uma classe que não é referenciada por nenhuma outra classe do sistema. Assim, com o objetivo de permitir que seja incluído ao menos um pacote, em que todas as suas classes sejam referenciadas (isto é, possuam grau de entrada maior ou igual a 1), é definida como candidata a controladora aquela classe que realiza o maior número de referência e que não foi selecionada na primeira fase de filtragem, e assim criado um novo pacote candidato com essa classe. Por fim, são incluídas ao novo pacote candidato, as classes que pertençam a esse mesmo pacote e que não foram selecionadas após a primeira fase do processo de filtragem. A Figura 12b ilustra esse mecanismo.

3.2.2.3.2 Processo de Filtragem (2ª Fase – Definição de *Controllers* com base no Grau de Referência por Pacote)

A segunda fase do processo de filtragem, consiste em **determinar pacotes *controllers*** por meio da verificação do grau de referência interna e externa para cada pacote, que são dados por $\sum_{i=1}^{|C_p|} c_{i_{r_in}}$ e $\sum_{i=1}^{|C_p|} c_{i_{r_out}}$, respectivamente, onde C_p é o conjunto de classes pertencentes ao pacote p, e os valores de r_in e r_out representam o número de referências em que a classe c_i recebe e realiza, respectivamente. A Figura 13 ilustra um conjunto de arquivos com a indicação de referências internas (setas "chegando") e externas (setas "saindo").

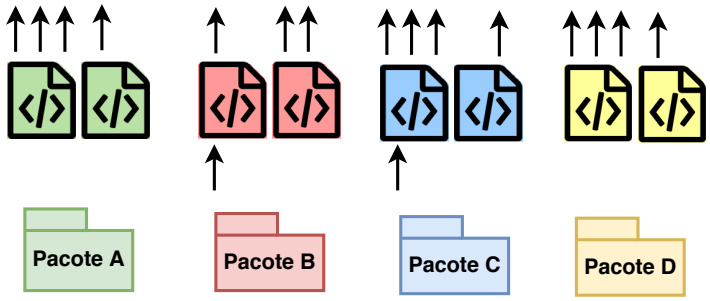


Figura 13 – Seleção de pacotes *controllers* com base nos graus de referência interna e externa.

Fonte – Autor.

Pode-se perceber, que o *Pacote A*, por exemplo, possui **graus de referência externa e interna igual a 4 e 0**, respectivamente. O valor 0 para o grau de referência é dado pelo fato de que os arquivos desse pacote não serem referenciados por outras classes. Além disso, o valor 4 (grau de referência externa) se deve ao fato de que esse pacote é contido por dois arquivos, onde um deles possui grau de referência externa igual a 3 e outro com grau 1. Caso um pacote p possua grau de referência interna menor que a média entre os graus de referências internas de todos os pacotes candidatos a serem controladores, ou grau de referência externa maior que a média entre os graus de referências externas de todos os pacotes candidatos, p é então definido agora como um pacote *controller*.

3.2.2.3.3 Processo de filtragem (3ª Fase – Inclusão de Arquivos não Selecionados ao mesmo Pacote *Controller* ou Candidato)

A terceira fase do processo de filtragem, consiste em incluir todas aquelas classes do sistema que não foram incluídas inicialmente do conjunto de classes candidatas, mas que pertencem a algum pacote candidato (ou que já foi definido na fase anterior de filtragem) a ser *controller*.

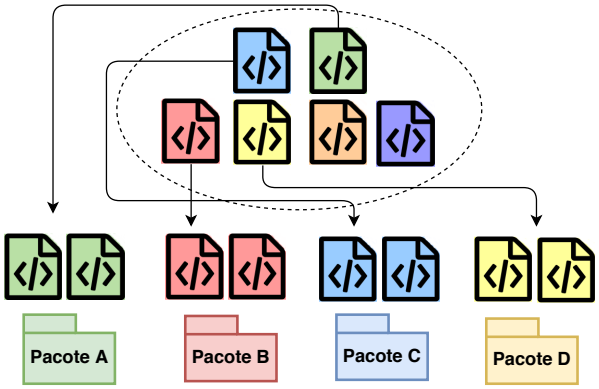


Figura 14 – Inclusão de arquivos não selecionados ao mesmo pacote *controller* ou candidato.

Fonte – Autor.

Esse processo da terceira fase é realizado também para que seja possível executar corretamente a quarta fase do processo de filtragem. Além disso, considera-se que uma classe que pertence a um pacote *controller* possui grande chance de ser *controller*.

3.2.2.3.4 Processo de filtragem (4ª Fase – Definição de Pacotes *Controllers* por meio de Referências para Pacotes Candidatos)

Por fim, a quarta fase do processo de filtragem, consiste em identificar referências feitas entre pacotes. Caso alguma classe c_i de um pacote X referencie indiretamente (isto é, o vértice v_j que representa a classe referenciada não possui uma aresta oriunda diretamente do vértice v_i que representa c_i, mas que pode ser alcançada a partir desse vértice) uma outra classe c_j pertencente a um pacote Y, então as classes pertencentes a Y são desconsideradas como *controllers* (exceto se o pacote correspondente não foi definido anteriormente como controlador). Isso porque as classes que pertencem ao pacote Y estão em um nível de hierarquia menor em comparação com as classes do pacote X, e por isso, é bem improvável que os arquivos no pacote Y sejam controladores. Além disso, os vértices diretamente ligados a v_i são desconsiderados porque classes controladoras podem ser referenciadas de forma direta por outro *controller*, e nesse caso, não são excluídas do grupo de candidatas a serem controladoras.

A Figura 15 exemplifica o que foi exposto anteriormente durante a segunda fase do processo de filtragem. Deve-se considerar que o grafo apresentado contem apenas classes candidatas a serem controladoras e que as cores dos vértices indicam o pacote em que cada classe representante pertence. Pode-se perceber que a classe c_1 referencia diretamente as classes c_2 e c_3 e indiretamente as classes c_4, c_5 e c_6. A classe c_1 não é referenciada por nenhuma outra classe, e com isso possui a maior probabilidade de ser realmente uma controladora.

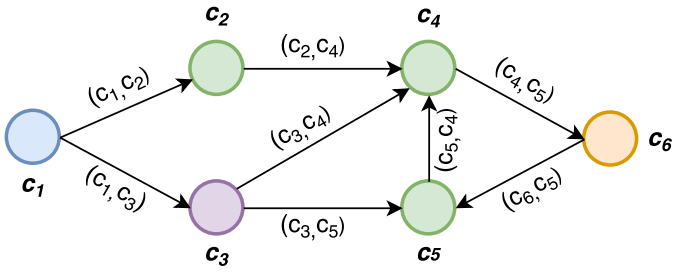


Figura 15 – Referência entre pacotes de classes candidatas.

Fonte – Autor.

Ao executar a estratégia de análise de referência entre pacotes, para exclusão de classes que possuem grande chance de não fazerem parte do grupo de candidatas a serem controladoras, a classe c_6 é excluída por ser referenciada indiretamente pelas classes c_1, c_2 e também c_3, que estão em pacotes diferentes. Seguindo o mesmo raciocínio, as classes c_4, c_5 são excluídas porque são referenciadas indiretamente por c_1. E por fim c_2 é excluída por fazer parte do mesmo pacote de classes que foram excluídas, no caso, c_4 e c_5. Dessa forma, apenas as classes c_1, c_3 permaneceriam no grupo de candidatas a serem controladoras, ambas em pacotes distintos.

É importante ressaltar que a segunda fase do processo de filtragem de classes candidatas a serem controladoras, foi definida com o objetivo de evitar que, durante a quarta fase de filtragem, sejam descartadas classes controladoras contidas em pacotes diferentes dentro do sistema, devido a estratégia de exclusão de pacotes adotada durante a quarta fase. Com isso, após a última fase do processo de filtragem tem-se um conjunto de classes definidas como controladoras do sistema.

3.2.2.4 Identificação de Métodos Principais

Esta subetapa consiste em identificar os métodos principais contidos em cada uma das classes controladoras. Para uma classe ser controladora, ela deve conter métodos principais que são os responsáveis pelo início da execução de uma funcionalidade. Um método principal é representado como uma funcionalidade do sistema. Contudo, é comum que classes Java possuam métodos auxiliares, responsáveis apenas por fornecer ou atribuir valores aos atributos internos da classe, conhecidos por *getters* e *setters*, respectivamente.

Um método é dito principal se as seguintes restrições forem atendidas: **a)** não é um construtor (método chamado implicitamente após a instanciação de um objeto da classe) e **b)** faz pelo menos uma chamada a outros métodos (da própria classe ou não). O primeiro critério é justificado pelo fato de que métodos construtores não serem chamados diretamente por meio de uma ação realizada pelo usuário, e sim apenas no momento da instanciação de um objeto. O segundo critério é justificado pelo fato de que métodos

principais normalmente realizam chamadas a outros métodos, internos ou não a mesma classe.

Portanto, caso um método pertencente a uma classe *controller* satisfaça ambas as restrições elencadas, o método é então considerado principal, indicando assim, uma funcionalidade do sistema. Cada funcionalidade identificada obtém um nome, que é composto pelo nome da classe *controller* mais o nome do método principal que representa tal funcionalidade. O conjunto de funcionalidades do sistema é denominado $F = \{f_1, f_2, ..., f_n\}$, onde n é o número de funcionalidades identificadas e f_i representa a i-ésima funcionalidade.

É importante ressaltar que a abordagem permite que seja realizada de forma manual a exclusão de falsas funcionalidades identificadas pela abordagem. Mesmo que o método satisfaça as restrições citadas anteriormente, pode não representar um funcionalidade do sistema, ou até mesmo pertence a uma classe identificada como controladora, mas que na verdade não representa um *controller*. Além disso, a abordagem permite que seja realizada também de forma manual a agregação de *features*, para que seja possível formar funcionalidades de maiores níveis, de **usuário** ou **agregador**.

3.2.3 Identificação de Arquivos Relacionados às Funcionalidades

Esta seção apresenta a terceira etapa da abordagem proposta neste trabalho. Essa etapa tem por objetivo a identificação de arquivos ou classes do sistema relacionadas a cada funcionalidade identificada.

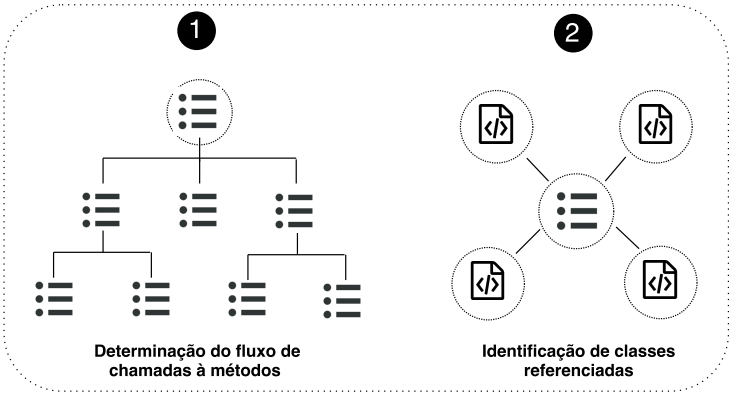


Figura 16 – Processo de identificação de classes relacionadas a uma funcionalidade.

Fonte – Autor.

Para realizar a identificação de arquivos relacionados a uma funcionalidade f qualquer, é necessário identificar quais classes são referenciadas a partir da invocação do método principal correspondente a f. Como apresentado na Seção 3.2.2.4, cada método principal de uma classe controladora, é considerada uma funcionalidade. Assim, são analisados todos os métodos invocados a partir de um método principal, com o objetivo de identificar as classes referenciadas durante esse processo. Cada funcionalidade f possui um

conjunto C_f de classes que a compõe. Esse conjunto está contido em C, que é o conjunto de classes do sistema.

Assim, o processo de identificação de arquivos ou classes relacionadas a uma funcionalidade f qualquer, é definida com base em duas subetapas principais: **I)** Determinação do fluxo de chamadas a métodos a partir de um método principal e **II)** Identificação de classes referenciadas durante o fluxo de chamadas a métodos identificado na subetapa anterior. Essas subetapas são apresentadas na Figura 16.

Durante a primeira subetapa são identificadas os métodos invocados a partir de um método principal (que representa uma funcionalidade) de uma classe controladora. Durante a segunda subetapa são identificadas aquelas classes que são referenciadas apenas pelos métodos contidos no fluxo de chamadas de uma funcionalidade. Essas subetapas são mais detalhadas nas Seções 3.2.3.1 e 3.2.3.2 seguintes.

3.2.3.1 Identificação do Fluxo de Chamadas a Métodos

Esta subetapa tem como objetivo identificar todos os métodos chamados durante o fluxo de execução de uma funcionalidade, que é representada por um método principal em uma classe controladora.

Para alcançar esse objetivo, são identificadas todas as invocações aos métodos a partir de um método principal. É possível identificar informações sobre invocações aos métodos por meio dos dados extraídos de cada classe Java. Em cada chamada de um método é possível obter a ***expression*** (expressão), o **nome do método** chamado e os **parâmetros recebidos**. Uma *expression* pode representar, por exemplo, uma variável de método ou um atributo da classe.

Com base na *expression* de uma invocação de um método, é **determinado a classe** que o contém. Após a identificação da classe que contém o método invocado, é necessário identificar qual método está realmente sendo invocado. Para tal, considerar apenas o nome do método chamado e quantidade de parâmetros recebidos não é suficiente para o processo de identificação, já que em uma classe Java pode existir métodos com nomes iguais e/ou quantidade de parâmetros iguais. Com isso é necessário também identificar os **tipos de parâmetros** passados durante a invocação ao método.

Além disso, outros fatores são considerados durante a identificação de qual método está sendo invocado, por exemplo: a linguagem Java permite a utilização de tipos de dados **abstratos** e **primitivos**, como *Integer* e *int*, respectivamente. Ambos os tipos são utilizados para manipulação de números inteiros. A linguagem permite que métodos que foram definidos com parâmetros do tipo abstrato, sejam invocados com passagem de parâmetros do tipo primitivo, e *vice-versa*. Com isso, no processo de identificação de métodos invocados, caso não seja encontrado um método que contenha tipos de parâmetros

definidos iguais aos tipos recebidos durante a chamada, é investigado então a existência de um método considerando tipos de dados abstratos e primitivos sem distinção, isto é, se o método recebe por padrão uma variável do tipo primitivo e foi invocado recebendo como parâmetro um tipo abstrato, esse método será então definido como aquele que foi invocado.

Assim, para cada método identificado é repetido o mesmo processo descrito anteriormente, até que se tenha todos os métodos referenciados durante o fluxo de execução de uma funcionalidade, isto é, a partir da invocação de um método principal.

3.2.3.2 Identificação de Classes Referenciadas

Esta subetapa consiste em identificar as classes referenciadas durante o fluxo de execução de uma funcionalidade. Para alcançar esse objetivo, é analisado cada método contido no fluxo de chamadas de cada funcionalidade.

A partir disso, para cada método contido no fluxo de chamadas de uma funcionalidade f, o conjunto de classes pertencentes a f será composto por: **a)** classes que contém cada método pertencente ao fluxo de chamadas; **b)** classes que fazem parte do conjunto de **parâmetros**, **variáveis** e **modificadores de acesso**; e **c)**; classes que fazem parte do **tipo de retorno** dos métodos contidos no fluxo de chamadas da funcionalidade.

Dessa forma são identificadas as classes referenciadas a partir da chamada a um método principal em uma classe controladora, que representa uma funcionalidade. Assim, tem-se como resultado, a formação do conjunto C_f de classes ligadas à funcionalidade f específica.

3.2.3.3 Exemplificação do Processo de Identificação de Arquivos Relacionados às Funcionalidades

Para exemplificar o processo de classificação de arquivos referenciados durante um fluxo de invocações de métodos e que fazem parte de uma determinada funcionalidade, é apresentado na Figura 17 um conjunto de classes utilizadas como exemplo. Nesse exemplo, o arquivo $ClasseA$ é considerado uma classe controladora e o $metodoM1A$ representa uma funcionalidade f_i. Com base nisso, inicialmente, é realizado o processo de identificação do fluxo de chamadas aos métodos a partir do método principal.

O conjunto M_i inicial de métodos do fluxo de chamadas da funcionalidade f_i é composto por $M_i = \{metodoM1A\}$. Ao analisar as invocações de métodos realizadas pelo $metodoM1A$ é identificado então que este faz referência ao $metodoM1B$, que passa a integrar o conjunto $M_i = \{metodoM1A, metodoM1B\}$. A partir disso, o $metodoM1B$ passa a ser analisado. Nesse método é realizada duas invocações a outros métodos, uma delas feita ao $metodoM1C$ e outra ao método $metodoM2C$. O con-

junto de métodos do fluxo de chamadas à funcionalidade f_i passa a ser composto por $M_i = \{metodoM1A, metodoM1B, metodoM1C, metodoM2C\}$. Como os dois últimos métodos identificados não realizam chamadas a outros métodos, então é encerrado o processo.

A partir desse momento, passa a ser executada a segunda subetapa do processo de identificação de arquivos relacionados à funcionalidade. Todos os métodos do conjunto M_i são analisados para identificar as classes referenciadas por cada um deles. Primeiramente é analisado o método principal que representa a funcionalidade f_i, que no caso é o $metodoM1A$. Esse método está contido na $ClasseA$ e referencia a $ClasseB$ por meio de sua variável. Além disso, o método $metodoM1B$ recebe como parâmetro uma variável do tipo da $ClasseC$, para realizar as chamadas aos métodos $metodoM1C$ e $metodoM2C$. Portanto, o conjunto de classes relacionadas a funcionalidade f_i passa a ser composto por $C_{f_i} = \{ClasseA, ClasseB, ClasseC\}$.

```
public class ClasseA {
  public void metodoM1A(){
     ClasseB varM1AB = new ClasseB();
     varAM1B.metodoM1B(new ClasseC());
  }
}
```

(a) Classe A.

```
public class ClasseB {
  public void metodoM1B(ClasseC paramP1M1B){
     paramP1M1B.metodoM1C();
     ClasseE<ClasseF> varM1BE;
     varM1BE = paramP1M1B.metodoM2C();
  }
}
```

(b) Classe B.

```
public class ClasseC {
  public ClasseD metodoM1C(){
    return new ClasseD();
  }
  ClasseE<ClasseF> metodoM2C (){
    return new ClasseE<ClasseF>();
  }
}
```

(c) Classe C.

```
public class ClasseD {
  public void metodoM1D(int inteiro){}
  public void metodoM1D(Integer inteiro){}
}
```

(d) Classe D.

```
public class ClasseE<T> {
  T atributoAbstrato;

  public void metodoM1E(){}
  public void metodoM2E(){}
}
```

(e) Classe E.

```
public class ClasseF {
  public void metodoM1F(int paramP1M1F){}
  public void metodoM2F(){}
  public void metodoM2F(){}
}
```

(f) Classe F.

Figura 17 – Classes de exemplo para identificação de arquivos ligados a uma funcionalidade.

Fonte – Autor.

Em seguida é analisado o método $metodoM1B$, que está contido na $ClasseB$ e faz referência às seguintes classes: $ClasseC$, $ClasseE$ e $ClasseF$. A primeira classe é identificada por meio do parâmetro recebido pelo método. As duas últimas são identificadas por meio do tipo de variável contido no método. Assim, o conjunto C_{f_i} passa a ser formado por $C_{f_i} = \{ClasseA, ClasseB, ClasseC, ClasseE, ClasseF\}$.

Logo após, passa a ser analisado o $metodoM1C$, que está contido na $ClasseC$ e que

faz referência a $ClasseD$ por meio de seu tipo de retorno. Dessa forma, o conjunto C_{f_i} passa a ser formado por $C_{f_i} = \{ClasseA, ClasseB, ClasseC, ClasseE, ClasseF, ClasseD\}$. Em seguida é analisado o método $metodoM2C$ que faz referência à $ClasseE$ e também à $ClasseF$ por meio do tipo de retorno desse método. Como essas classes já foram incluídas anteriormente no conjunto C_{f_i}, esse conjunto então é definido finalmente por $C_{f_i} = \{ClasseA, ClasseB, ClasseC, ClasseE, ClasseF, ClasseD\}$, considerando a ordem em que as classes foram identificadas. Por fim, é importante ressaltar que um mesmo arquivo ou classe pode pertencer a diferentes funcionalidades.

3.2.4 Inferência da Familiaridade por Funcionalidades

Esta etapa consiste em inferir a familiaridade de código que cada desenvolvedor possui com as funcionalidades de um software específico. Esse processo é ilustrado na Figura 18.

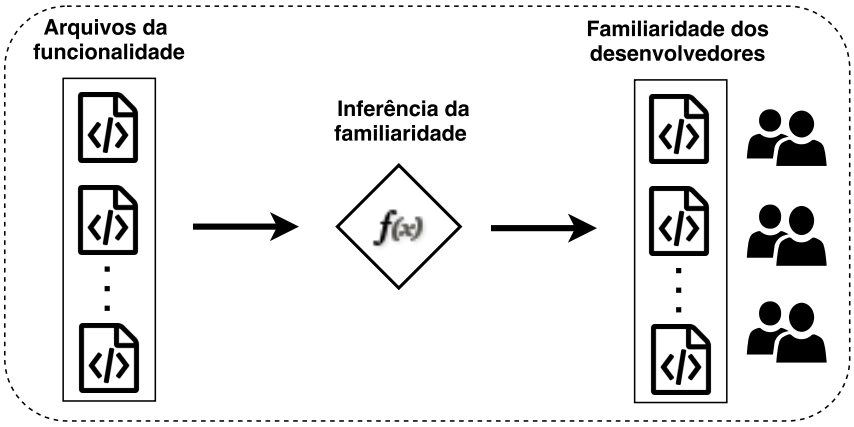


Figura 18 – Cálculo da familiaridade por funcionalidade de software.

Inicialmente, é inferida ou calculada a familiaridade de código que cada desenvolvedor possui sobre cada um dos arquivos ou classes pertencentes a uma funcionalidade. Para isso, é utilizado o método proposto por LIRA (2016), que como apresentado na Seção no Capítulo 2 deste trabalho, utiliza métricas baseadas em alterações de código realizadas por desenvolvedores durante o desenvolvimento de um sistema de software, como por exemplo: remoção, adição ou modificação de linhas de código. Considerando também ponderações para cada tipo de alteração realizada.

Por fim, a familiaridade de código dos desenvolvedores sobre uma funcionalidade identificada pela abordagem, é dada pela familiaridade que esses desenvolvedores possuem sobre cada um dos arquivos relacionados a essa funcionalidade.

3.3 Representação Sinóptica da Abordagem

Com o objetivo de facilitar o entendimento, é apresentada a seguir uma representação em alto nível da abordagem proposta neste trabalho.

1. **Extração de dados**:
   a) Extração de arquivos de código;
   b) Identificação de modificações realizadas em arquivos de código.
2. **Identificação de funcionalidades**:
   a) Extração de informações dos arquivos de código;
      i. Mapeamento do código para uma AST;
      ii. Formação de um conjunto de informações de cada arquivo.
   b) Identificação do relacionamento entre arquivos;
      i. Criação de um grafo de referência entre arquivos.
   c) Identificação de arquivos controladores;
      i. Definição de heurísticas baseadas no grafo de referência entre arquivos;
      ii. Definição de arquivos principais do sistema com base nas heurísticas definidas.
   d) Identificação de métodos principais:
      i. Definição de métodos principais (considerados funcionalidades) com base nas seguinte restrições:
         - Não é um método construtor;
         - Faz pelo menos uma chamada a outros métodos.
      ii. Agregação de funcionalidades e/ou exclusão manual de falsas *features*.
3. **Identificação de arquivos relacionados às funcionalidades**:
   a) Determinação do fluxo de chamadas a métodos a partir de um método principal (funcionalidade);
   b) Identificação de arquivos referenciados em cada fluxo de chamadas a métodos.
4. **Inferência da familiaridade por funcionalidades**:
   a) Determinação da familiaridade em uma funcionalidade com base na familiaridade em cada arquivo de código que a compõe.

Essa representação sintetiza as etapas gerais da abordagem proposta neste trabalho. As principais delas estão destacadas em negrito.

3.4 Intregação da Abordagem com a Ferramenta *CoDiVision*

A abordagem proposta foi implementada como um novo módulo na ferramenta *CoDiVision*. A ferramenta permite visualizar por meio da apresentação de gráficos, a familiaridade de código considerando também a perspectiva de funcionalidade. Como mencionado anteriormente, a abordagem proposta identifica funcionalidades de nível de **subfunção**, representas por métodos em arquivos controladores. Com isso, para que seja possível também visualizar a familiaridade sobre funcionalidades de níveis de **usuário** ou **agregador**.

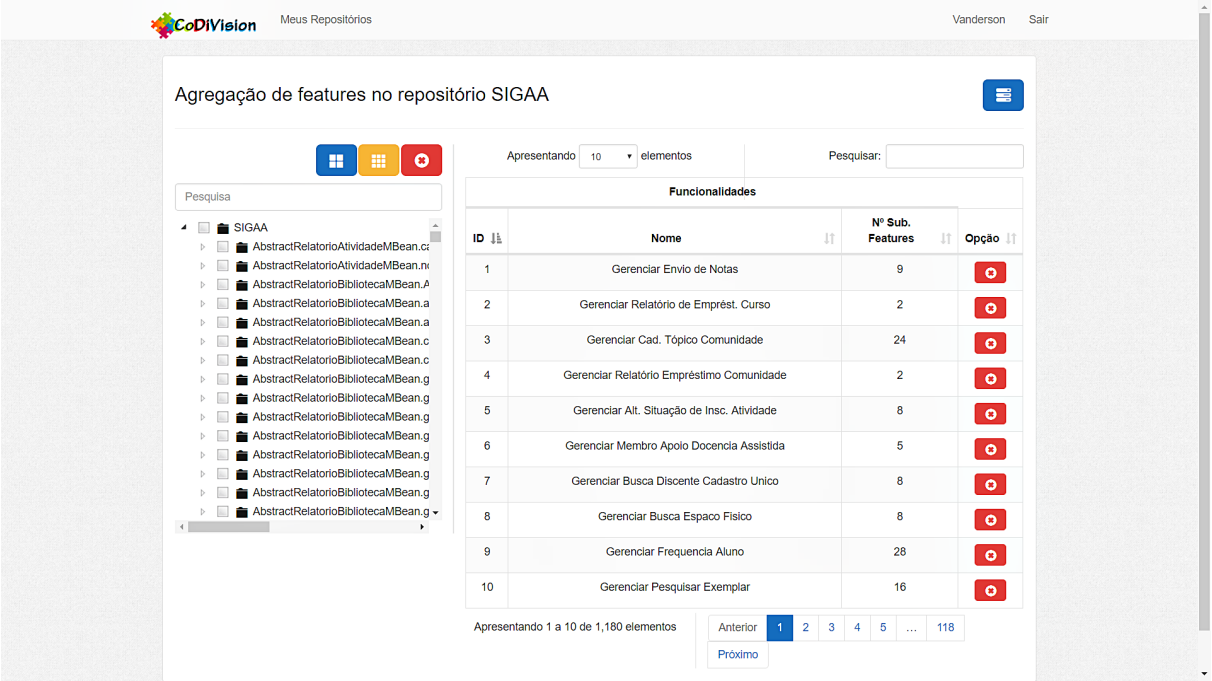


Figura 19 – Página onde é possível realizar a agregação de *features*.

Fonte – Autor.

A Figura 19 apresenta a página na ferramenta *CoDiVision* onde é possível realizar a agregação de funcionalidades de nível de subfunção, formando assim funcionalidades de maiores níveis (de usuário ou agregador). Para isso, é necessário selecionar as respectivas *features* que deverão ser agrupadas e logo após clicar no botão azul no canto superior esquerdo. É possível também definir um nome para cada funcionalidade gerada a partir da agregação. Com o clique no botão de cor laranja é possível também realizar a agregação de *features* que estão no mesmo arquivo *controller*, de forma automática. Os nomes das funcionalidades agrupadas automaticamente são definidos de acordo com os nomes dos respectivos *controllers*. O botão em cor vermelho (também no canto superior esquerdo) permite excluir falsas *features* indicadas pela ferramenta.

As funcionalidades agrupadas podem ser visualizadas na tabela apresentada ao centro da Figura 19. Essa tabela apresenta a quantidade de funcionalidades de nível de

subfunção que compõem cada funcionalidade gerada a partir da agregação. Cada uma dessas funcionalidades geradas podem ser excluídas por meio do clique no botão em cor vermelho na coluna de opção da tabela citada. O botão em cor azul no canto superior direito permite o redirecionamento do usuário à página de visualização da familiaridade sobre as funcionalidades que foram agregadas. É importante ressaltar que para visualizar a familiaridade sobre *features* indicadas pela ferramenta é necessário realizar a "agregação" de cada *feature* de forma individual.

A Figura 20 apresenta a página onde é possível visualizar a familiaridade dos desenvolvedores sobre as funcionalidades do sistema. Para a visualização da familiaridade sobre uma funcionalidade específica deve-se clicar sobre o nó correspondente na àrvore de navegação apresentada à esquerda da imagem. A visualização da familiaridade sobre todas as funcionalidades pode ser obtida por meio do clique na raiz da árvove de navegação.

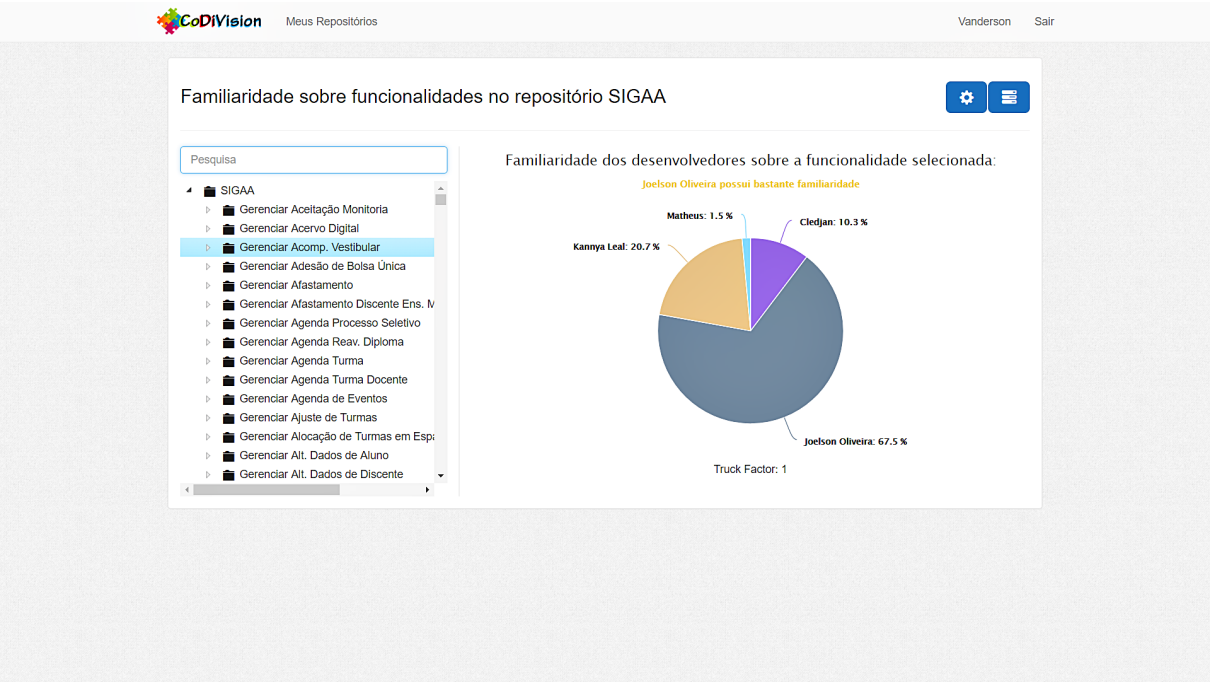


Figura 20 – Página para visualização da familiaridade sobre funcionalidades.

Fonte – Autor.

Ainda pela análise da Figura 20, pode-se perceber que é possível obter o valor da métrica *Truck Factor* (TF) (discutida na Seção 1.6 do Capítulo 1), de acordo com cada funcionalidade selecionada. É importante ressaltar que a métrica TF apresenta, neste caso, o número de desenvolvedores com maior familiaridade sobre um determinada *feature* ou mesmo um conjunto de funcionalidades.

O valor da métrica TF é calculada de acordo com o limiar definido no formulário de configurações de parâmetros. Essas configurações podem ser realizadas após o clique no botão azul (com ícone de "engrenagem") no canto superior direito da Figura 20. O

formulário de configurações das métricas utilizadas na ferramenta *CoDiVision* é apresentado na Figura 21. Pode-se perceber que é possível definir parâmetros para as métricas de degradação da familiaridade por tempo (mês) e por novas alterações, além de valores para ponderação dos tipos de alterações de código. Esses parâmetros para ponderação são dados em valores reais. Cada uma das métricas foi detalhada no Capítulo 2. Dessa forma, é possível visualizar a familiaridade sobre funcionalidades de acordo com diferentes contextos.

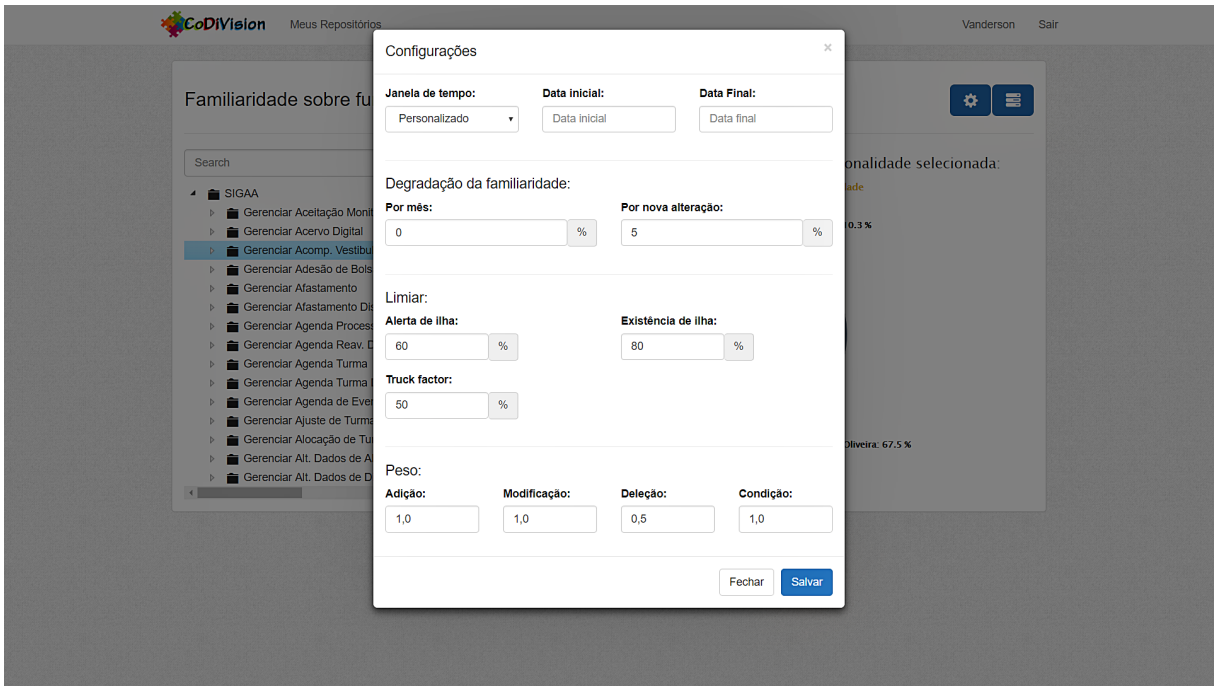


Figura 21 – Formulário para configuração de parâmetros para métricas na ferrmaneta *CoDiVision*.

Fonte – Autor.

É importante citar que a métrica **alerta de ilha** é utilizada para a gereção de alertas de risco quando apenas um desenvolvedor possuir uma porcentagem (60%, neste caso) considerável de familiaridade em relação aos demais membros da equipe. De maneira análoga, a métrica **existência de ilha** é utilizada para indicar uma situação de criticidade, quando um único desenvolvedor possuir sozinho uma grande porcentagem (80%, neste casso) da familiaridade de código.

3.5 Considerações Finais

Este capítulo apresentou a abordagem proposta neste trabalho, que tem como principal objetivo a inferência da familiaridade de código sobre funcionalidades de software. Esta abordagem estende o método proposto em (LIRA, 2016), e fornece uma nova perspectiva da relação entre desenvolvedor e código fonte. Esta abordagem visa apoiar, de

forma mais apropriada, gerentes de projetos durante a etapa de manutenção de software. A informação da familiaridade sobre funcionalidades tende a ser mais apropriada em um contexto de desenvolvimento de software real, pois gerentes de projetos precisam comumente distribuir tarefas que normalmente são tratadas como funcionalidades, e portando, é mais adequada uma noção da familiaridade de desenvolvedores considerando a perspectiva de funcionalidade.

As principais etapas da abordagem proposta consistem em identificar funcionalidades de software e arquivos de código que fazem parte de cada funcionalidade. Assim é possível inferir a familiaridade de desenvolvedores sobre funcionalidades de um sistema. A abordagem proposta foi integrada como um novo módulo de visualização na ferramenta *CoDiVision*, que implementa o método proposto por LIRA (2016). Por meio da ferramenta é possível realizar a agregação de *features* de nível de *subfunção*, formando assim funcionalidades de nível de **usuário** e/ou **agregador**. Além disso, é possível excluir falsas funcionalidades identificadas. A ferramenta permite a visualização da familiaridade sobre uma funcionalidade específica ou mesmo sobre todas as funcionalidades definidas.

No próximo capítulo é apresentada uma avaliação da abordagem proposta, com a utilização da ferramenta *CoDiVision*. A abordagem foi avaliada no processo de identificação de funcionalidades e os arquivos que as compõem. Foram utilizados como base sistemas de software reais, desenvolvidos em linguagem Java. Foi realizada também uma análise da familiaridade de código considerando as perspectivas de módulo e funcionalidade de software. Essa análise constitui um dos principais objetivos deste trabalho.

4 Avaliação

Este capítulo apresenta a avaliação da abordagem proposta neste trabalho. Essa avaliação foi realizada em duas etapas. Na primeira etapa, foi avaliada o processo de identificação de funcionalidades e os arquivos de código que fazem parte de cada *feature* identificada. Na segunda etapa da avaliação, foi realizada uma análise da familiaridade de código considerando as perspectivas de módulo e funcionalidade. Esta avaliação seguiu recomendações prescritas por WOHLIN et al. (2012) para avaliações experimentais no âmbito da Engenharia de Software.

4.1 Avaliação do Processo de Identificação de Funcionalidades

Esta seção apresenta avaliação da segunda e terceira etapas da abordagem proposta, isto é, a avaliação do método no processo de identificação de funcionalidades e os arquivos de código que fazem parte de cada *feature* em um software.

4.1.1 Objetivo

Esta primeira etapa da avaliação tem por objetivo verificar o desempenho do método no processo de identificação dos seguintes elementos: I) arquivos ou **classes controladoras** (*controllers*) do sistema; II) **funcionalidades** e III) arquivos de código ou **classes Java** que compõem as funcionalidades de um software. Esse processo é importante para que seja possível inferir de forma mais precisa a familiaridade de código de desenvolvedores considerando a perspectiva de funcionalidades.

Assim, para cada quesito avaliado (classes principais, funcionalidades e arquivos relacionados às *features*) foi verificado se o método era capaz de classificar corretamente os seguintes casos: (I) arquivos que pertencem, ou não, ao conjunto de classes controladoras do sistema; (II) métodos principais em classes controladoras que pertencem, ou não, ao conjunto de funcionalidades do sistema; e (III) arquivos que pertencem, ou não, ao conjunto de classes que compõe uma funcionalidade específica do software. Com isso, para cada caso (I, II e III) houve duas classificações possíveis: **Pertence**, se o elemento analisado pertence ao conjunto e **Não Pertence**, caso contrário.

4.1.2 Questões de Pesquisa

Esta primeira etapa de avaliação foi proposta para responder as seguintes questões de pesquisa:

- **QP1**: A abordagem proposta é eficaz no processo de *feature location*?

  a) A abordagem proposta é eficaz no processo de identificação de arquivos controladores?

  b) A abordagem proposta é eficaz no processo de identificação de métodos principais (*features*)?

  c) A abordagem proposta é eficaz no processo de identificação de arquivos relacionados às funcionalidades identificadas?

4.1.3 Objetos

Esta seção descreve de maneira geral os softwares utilizados durante esta primeira etapa de avaliação da abordagem proposta neste trabalho. Todos os sistemas analisados são sistemas **Web** e utilizam o padrão arquitetural MVC(*model-view-controller*). Foram utilizados três softwares reais que já se encontram em produção, isto é, em processo de utilização por usuários finais. Esses softwares foram escolhidos principalmente por ter sido possível o acesso ao código fonte de cada um, bem como ao contato próximo com os desenvolvedores responsáveis pela implementação de cada um dos sistemas.

4.1.3.1 SAGRES Web

O **SAGRES**[1] (Sistema de Acompanhamento da Gestão dos Recursos da Sociedade) é um software utilizado por centenas de usuários para o envio de documentos com as informações das prestações de contas dos municípios e/ou órgãos subordinados no Estado do Piauí. O software foi desenvolvido em um convênio da UFPI com o Tribunal de Contas do Estado do Piauí (TCE/PI)[2].

A seguir são descritas algumas características principais desse software, das quais derivam as funcionalidades mais importantes providas pela aplicação:

- **Gerência pedido de chaves**: O sistema permite ao usuário realizar pedidos de chave, que são arquivos criptografados que permitem o envio de alguns tipos especiais de Prestação de Contas (PC).
- **Entrega de prestação de contas**: O sistema permite que o usuário faça o *upload* (envio) do arquivo de prestação de contas e também o *download* dos arquivos de prestações de contas já enviados.
- **Geração de recibo de entrega**: O sistema permite que os usuários obtenham o recibos de entregas de prestações de contas.

[1] https://sistemas.tce.pi.gov.br/sagres-web/.
[2] https://www.tce.pi.gov.br/.

O sistema possui várias outras características que derivam ainda mais funcionalidades do software, contudo, as *features* mais importantes baseiam-se principalmente nas características descritas anteriormente.

4.1.3.2 TheEnd

A ferramenta **TheEnd**[3] é um sistema *Web* para auxiliar pesquisadores na realização de Mapeamento Sistemático da Literatura (MSL) (KITCHENHAM, 2004; KEELE, 2007; HIGGINS; GREEN et al., 2008). Esse sistema tem suporte às etapas de planejamento, execução e sumarização de resultados obtidos durante o processo de seleção e extração de dados de um MSL. Algumas das características principais da ferramenta são descritas a seguir:

- **Gerência de planejamento**: O sistema permite ao usuário definir itens que fazem parte do protocolo de MSL, que são: definição dos objetivos, questões de pesquisa, *string* de busca, critérios de inclusão e exclusão, e o formulário para extração de dados.
- **Gerência de estudos**: O sistema permite que o usuário possa incluir os artigos obtidos após aplicação da *string* de busca nas bibliotecas digitais, utilizando o formato *Bibtex*.
- **Seleção de estudos:** O sistema permite que o usuário analise as informações de artigos para que então possa optar por incluir ou excluir o trabalho.
- **Extração de dados:** o sistema permite que o usuário faça a extração de dados dos estudos de acordo com o formulário que foi definido na etapa de planejamento.
- **Sumarização de resultados:** o sistema permite visualização dos resultados do MSL, disponibilizando gráficos e relatórios referentes ao processo de seleção dos trabalhos e extração de informações.

As *features* mais importantes deste sistema derivam das características descritas anteriormente. Contudo, este software apresenta também outras características associadas.

4.1.3.3 CoDiVision

A ferramenta **CoDiVision**[4] é um sistema *Web* desenvolvido por (LIRA, 2016), que implementa o método contemplado pela abordagem proposta neste trabalho, para inferência da relação entre desenvolvedor e código fonte, considerando a perspectiva de arquivos ou módulos de software. A ferramenta utiliza como base Mineração de Repositórios

[3] http://easii.ufpi.br/theend/.
[4] http://easii.ufpi.br/codivision/.

de Software (MRS) (HASSAN, 2008) para extrair dados a partir de sistemas de controle de versão (SCV) (OTTE, 2009). A seguir são descritas algumas características principais do desse sistema, das quais também derivam as funcionalidades mais importantes providas pela aplicação.

- **Gerência de repositórios**: O sistema permite ao usuário criar, editar, ou excluir informações a respeitos de repositórios em SCV, tais como *url* de acesso, *branch* do projeto de onde deverão ser extraídos os dados a respeito de alterações de código.

- **Extração de dados**: O sistema permite que o usuário inicie a extração de dados a partir de repositórios informados, para que possam ser gerados resultados sobre a relação entre desenvolvedores e código fonte.

- **Exibição gráfica de dados**: O sistema permite que os usuários visualizem gráficos que apresentam a familiaridade de código que os desenvolvedores possuem sobre arquivos e módulos do sistema.

É importante ressaltar que a avaliação utilizando esse software foi realizada sobre a última versão antes da integração com a abordagem proposta neste trabalho, isto é, a versão analisada oferecia suporte apenas à inferência da familiaridade considerando a perspectiva de módulo ou arquivos de software.

4.1.4 Participantes

Esta avaliação contou com a participação direta dos principais desenvolvedores responsáveis pela implementação dos softwares avaliados. Um desenvolvedor de cada sistema auxiliou na definição de classes controladoras, funcionalidades (métodos principais em classes controladoras) e também classes relacionadas à cada funcionalidade do sistema, para que assim fosse possível a comparação dos resultados obtidos pela aplicação da abordagem proposta.

4.1.5 Contexto

É importante frisar que o conjunto de dados utilizados durante esta primeira etapa de avaliação é constituído por arquivos de código fonte de sistemas desenvolvidos em linguagem de programação Java. As versões analisadas para os sistemas **SAGRES**, **TheEnd** e **CoDiVision** possuem um total de **99**, **95** e **65 classes** ou arquivos de código fonte Java. Todos os dados obtidos a partir desses sistemas, foram extraídos de SCV's do tipo Git.

4.1.6 Operação

Para avaliar o desempenho do método durante a etapa de identificação de funcionalidades e arquivos relacionados, foram utilizadas medidas comumente aplicadas durante a avaliação de classificadores e que foram apresentadas na Seção 1.7 do Capítulo 1, que são: **acurácia**, **erro**, **precisão**, ***recall*** e **especificidade**.

As métricas utilizadas são baseadas no número de elementos **Verdadeiros Positivos**, **Falsos Positivos**, **Verdadeiros Negativos** e **Falsos Negativos**. A **acurácia** apresenta o desempenho do classificador de forma geral, ou seja, calcula a porcentagem de acertos em relação ao total de elementos. O **erro** (dado por $1 - A$, onde A é o valor da acurácia) representa a porcentagem de elementos classificados incorretamente.

A métrica **precisão** apresenta a porcentagem de elementos classificados como positivos e que realmente são positivos. O ***recall*** apresenta a porcentagem de elementos classificados como positivos em relação a todos os elementos da mesma classe. Por fim, a **especificidade** calcula a porcentagem de amostras negativas identificadas corretamente em relação ao total de elementos negativos.

4.1.7 Resultados

Esta seção apresenta os resultados obtidos a partir da análise dos sistemas descritos na Seção 4.1.3, com o objetivo de avaliar a abordagem proposta no processo de identificação de funcionalidades e arquivos relacionados a cada um delas. Além disso, são apresentadas algumas discussões sobre os resultados obtidos durante esta avaliação.

4.1.7.1 Identificação de Classes Controladoras

A Tabela 4 apresenta a matriz de confusão para resultados obtidos no processo de identificação ou classificação dos arquivos de código ou classes controladoras de cada um dos sistemas analisados.

Tabela 4 – Matriz de confusão para a classificação de arquivos controladores.

a = Pertence **b = Não Pertence**	**Classificação**								
	Sagres			TheEnd			CoDiVision		
Resultado Esperado		**a**	**b**		**a**	**b**		**a**	**b**
	a	8	0	**a**	8	0	**a**	5	0
	b	10	81	**b**	22	65	**b**	2	58

Fonte – Autor.

Pode-se perceber, dentre todos os sistemas, que não houve classes controladoras que deixaram de ser identificadas. Os sistemas **SAGRES** e **TheEnd** tiveram, cada um, as **8 (oito) classes controladoras** identificadas realmente como *controllers*. Além disso,

todas as **5 (cinco) classes controladoras** do sistema **CoDiVision** também foram identificadas.

Pode-se perceber também, que os sistemas **SAGRES**, **TheEnd** e **CoDiVision** tiveram **10**, **22** e **2 arquivos**, respectivamente, classificados como classes controladoras, mas que na verdade não pertenciam ao conjunto de *controllers*, isto é, fazem parte do grupo **Falso Positivo**. Por outro lado, pode-se perceber que os mesmos sistemas tiveram **81**, **65** e **58 arquivos**, respectivamente, classificados corretamente para o grupo **Verdadeiro Negativo**, indicando que grande parte dos arquivos desses sistemas, que não eram realmente controladores, não foram incluídos no grupo de *controllers*.

A Tabela 5 apresenta os valores para as métricas utilizadas na avaliação de desempenho da abordagem proposta, no que diz respeito à identificação de classes controladoras do sistema.

Tabela 5 – Desempenho na identificação de classes controladoras.

Sistema	Acurácia	Erro	Precisão	Recall	Especificidade
SAGRES	0,8990	0,101	0,4167	1,0	0,8901
TheEnd	0,7684	0,2316	0,2667	1,0	0,7471
CoDiVision	0,9692	0,0308	0,7143	1,0	0,9667

Fonte – Autor.

Pode-se perceber, que o valor para **precisão** não foi alto, considerando os sistemas **SAGRES** e **TheEnd**. Contudo, essa métrica leva em consideração apenas o número de elementos para as classes **Verdadeiro Positivo** e **Falso Positivo**. Isso, quer dizer que essa métrica calcula a eficiência baseada apenas na classificação dos elementos contidos no grupo **a** (**Pertence**), que representa os arquivos ou classes controladoras do sistema. Em outras palavras, um valor não muito alto para a métrica **precisão**, indica que o método sempre aponta algumas classes como sendo controladores, mas que na verdade não são *controllers*. Apesar disso, esse fato não é uma questão crítica e mostra que a abordagem é inclusiva, sendo menos criteriosa ao apontar que a classe não é um *controller*.

Além disso, pode-se perceber por meio dos valores para a **acurácia**, que os resultados obtidos foram mais significativos para os sistemas **SAGRES** e **CoDiVision**, atingindo um total de **0,8990** e **0.9692**, respectivamente. O valor da **acurácia** não foi tão elevado para o sistema **TheEnd** em relação aos demais sistemas. Isso está diretamente relacionado ao valor da métrica **precisão**, que não foi tão elevado porque a abordagem indicou um número maior de classes como sendo *controllers*, mas que na verdade não eram. Contudo, esse número de arquivos classificados como *controllers*, mas que não eram, representa menos de **25%** do total de classes do sistema **TheEnd**. Indicando que a abordagem apesar de ser inclusiva não indicou um número excessivo de classes não *controllers* como sendo controladores dos sistemas.

Pode-se perceber também que o valor para ***recall*** foi de **1,0** para todos os sistemas. O valor dessa métrica indica a proporção do número de arquivos classificados como controladores e que realmente são, pelo número de todos os arquivos que deveriam ser classificados como controladores. O resultado de **1,0** para essa métrica reforça o que foi apresentado na Tabela 4, na qual indica que todas as classes que realmente eram controladoras foram relamente classificadas como sendo **controllers**, considerando todos os sistemas.

No contexto deste trabalho, a métrica **recall** é a mais importante, uma vez que valores significativos para essa métrica indicam que a abordagem consegue encontrar grande parte dos elementos da classe **Pertence**, que é mais importante nesse contexto, pois a implementação da abordagem permite a exclusão de falsas *features* (oriundas da identificação de falsos *controllers*), para que assim não sejam utilizadas na avaliação da familiaridade na visão de funcionalidades. Assim, garantindo-se que todos os controladores sejam identificados permite que um refinamento nessa classificação seja feito, com um esforço bem menor que fazer essa análise utilizando-se de todos os arquivos.

4.1.7.2 Identificação de Funcionalidades

Nesta seção são apresentados os resultados obtidos no processo de classificação ou identificação de funcionalidades (representadas por métodos principais em classes controladoras) para os sistemas analisados. A Tabela 6 apresenta a matriz de confusão para resultados obtidos no processo de identificação de funcionalidades.

Tabela 6 – Matriz de confusão para a identificação de funcionalidades.

a = Pertence **b = Não Pertence**	**Classificação**								
	SAGRES			TheEnd			CoDiVision		
Resultado Esperado		**a**	**b**		**a**	**b**		**a**	**b**
	a	53	0	**a**	78	0	**a**	23	0
	b	40	122	**b**	5	29	**b**	2	5

Fonte – Autor.

É importante ressaltar, que uma funcionalidade é representada por um método principal em uma classe controladora do sistema. Assim, pode-se perceber, por meio da análise da Tabela 6, que houve um total de **40**, **5** e **2 métodos** dos sistemas **SAGRES**, **TheEnd** e **CoDiVision**, respectivamente, que foram classificadas como representantes de funcionalidades do sistema, e que na verdade não eram (**Falso Positivo**). Contudo, pode-se observar que todos os métodos principais ou funcionalidades pertencentes às classes *controllers* foram identificados.

Pode-se perceber também, que a maioria dos métodos foram classificados corretamente como sendo funcionalidades ou não do sistema. Isso pode ser percebido por meio da

análise da diagonal principal da matriz de confusão para cada sistema analisado, como apresenta a Tabela 6. É possível observar também, que a grande maioria dos métodos contidos em classes controladores não representam funcionalidades dos sistemas, e grande parte desses métodos foram realmente definidos como não representantes de funcionalidades (**Verdadeiro Negativo**).

A Tabela 7 apresenta os valores para as métricas utilizadas na avaliação de desempenho da abordagem proposta, no que diz respeito à identificação de funcionalidades dos sistemas analisados.

Tabela 7 – Desempenho na identificação de funcionalidades.

Sistema	Acurácia	Erro	Precisão	Recall	Especificidade
SAGRES	0,8140	0,186	0,5699	1,0	0,7531
TheEnd	0,9554	0,0446	0,9398	1,0	0,8529
CoDiVision	0,9333	0,0667	0,9200	1,0	0,7143

Fonte – Autor.

Pode-se perceber que a **acurácia** obtida para o sistema **SAGRES** foi menor em relação aos demais sistemas, **TheEnd** e **CoDiVision**. Isso é justificado pelo fato de que para o sistema **SAGRES** houve um maior número de falsas funcionalidades identificadas, isso porque muitos métodos nos arquivos controladores desse sistema, apesar de terem satisfeito as restrições da abordagem para definição de métodos principais, eram responsáveis apenas pela manipulação de elementos (botões, caixas de texto, menus, etc.) da interface de gráfica, não representando realmente uma *feature* útil ao usuário. Essa característica (gerência de elementos da *view* em arquivos controllers) do sistema **SAGRES** é determinada pelo uso de um *framework* denominado **Primefaces**[5].

Contudo, de maneira geral os valores para a métrica de **acurácia** foram significativos, atingindo um total de **0,8140**, **0,9554** e **0,9333** para os sistemas **SAGRES**, **TheEnd** e **CoDiVision**, respectivamente, apesar de que para o sistema **SAGRES** o valor foi um pouco menor em relação aos demais, pelo fato de que para esse sistema houve um maior número de falsas funcionalidades identificadas pela abordagem. Contudo, de maneira geral, a abordagem apresentou um bom desempenho durante a identificação de funcionalidades dos sistemas. Isso pode ser percebido também por meio do valor para a **taxa de erro**, que atingiu, respectivamente, os valores **0,186**, **0,0446** e **0,0667**, para os mesmos sistemas.

Além disso, os valores para ***recall*** (que foi de **1,0** para todos os sistemas) e **especificidade** (que foi de **0,7531**, **0,8529** e **0,7143**, respectivamente, para os sistemas **SAGRES**, **TheEnd** e **CoDiVision**) reforçam que a abordagem proposta obteve uma

[5] https://www.primefaces.org/.

maior taxa de acerto, em vez de erro, na identificação de métodos que representam, ou não, funcionalidades dos sistemas analisados.

Os valores mais baixos obtidos para a métrica **especificidade** é justificado pelo fato de que alguns métodos apontados como *features* na verdade não representam de fato funcionalidades. Contudo, todos os métodos que realmente representavam *features* foram identificados, por isso os valores de ***recall*** igual a **1,0**. Como mencionado anteriomente, para o contexto deste trabalho, o valor maior para a métrica **recall** é mais importante para este trabalho.

4.1.7.3 Identificação de Classes Relacionadas às Funcionalidades

Nesta seção são apresentados os resultados obtidos no processo de classificação ou identificação de arquivos de código ou classes Java pertencentes às funcionalidades dos sistemas analisados. A Tabela 8 apresenta a matriz de confusão para resultados obtidos no processo de identificação de classes relacionadas às funcionalidades dos sistemas **SAGRES**, **TheEnd** e **CoDiVision**. É importante ressaltar que os dados apresentados são em relação à classificação geral, isto é, considerando todas as funcionalidades para cada um dos sistemas analisados.

Pode-se perceber, por meio da análise da diagonal principal da matriz de confusão, que a abordagem obteve um bom desempenho na classificação de arquivos que pertencem, ou não, às funcionalidades dos sistemas. Além disso, não houve casos em que o método classificou uma determinada classe como pertencente a uma funcionalidade, mas que na verdade não era, ou seja, não houve elementos para o conjunto **Falso Positivo**.

Tabela 8 – Matriz de confusão para a identificação classes relacionadas às funcionalidades.

a = Pertence **b = Não Pertence**		**Classificação**							
		Sagres			TheEnd			CoDiVision	
Resultado Esperado		**a**	**b**		**a**	**b**		**a**	**b**
	a	309	16	**a**	941 **(834)**	0 **(107)**	**a**	289 **(266)**	0 **(23)**
	b	0	4914	**b**	0	6276	**b**	0	1336

Fonte – Autor.

Pode-se perceber também, que para o sistema **SAGRES** houve poucos casos em que uma determinada classe (arquivo de código) deixou de ser classificada como pertencente a uma determinada funcionalidade, isto é, o número de elementos do conjunto **Falso Negativo** foi bastante pequeno.

Durante a validação dos dados junto ao desenvolvedor responsável pelo sistema **SAGRES**, foi constatado que esse sistema utiliza-se de bibliotecas externas que controlavam chamadas a métodos internos do sistema, e com isso, essas classes que pertenciam a

métodos internos chamados por meio de bibliotecas externas, não puderam ser mapeadas durante a chamada a métodos contidos no fluxo de chamadas de algumas funcionalidades. Esse fato contribuiu para a não identificação de classes relacionadas a algumas funcionalidades. Contudo, a abordagem conseguiu classificar corretamente a grande maioria dos arquivos relacionados às funcionalidades desse sistema.

Quanto aos sistemas **TheEnd** e **CoDiVision**, alguns arquivos relacionados às suas funcionalidades não puderam ser identificadas em um primeiro momento. Os valores destacados entre parênteses na Tabela 8 indicam que em uma primeira análise desses sistemas houve arquivos relacionados que não puderam ser identificados para algumas funcionalidades. Isso motivou-se pelo fato de que esses sistemas foram desenvolvidos com a utilização de um *framewok Web* conhecido por **Vraptor** (FREIRE; SILVEIRA, 2004). Esse *framework* (e outros similares) fornece uma maneira específica de chamadas entre métodos de classes controladoras. Por esse motivo, alguns fluxos de chamadas não puderam ser totalmente mapeados.

Contudo, foi verificado que o **Vraptor**, e demais *frameworks* de mesma natureza, sempre permitem a identificação para a classe controladora e ao método invocado. Com isso, foram realizadas melhorias na implementação da abordagem proposta para que fosse possível mapear chamadas a métodos realizadas por meio da utilização do *framework* **Vraptor** e outros similares. Com isso, foi possível identificar então todos os arquivos relacionados às funcionalidades dos sistemas **TheEnd** e **CoDiVision**. A Tabela 9 apresenta o desempenho obtido pela abordagem proposta durante o processo de identificação de arquivos relacionados às funcionalidades dos sistemas analisados.

Tabela 9 – Desempenho na identificação de arquivos relacionados às funcionalidades.

Sistema	**Acurácia**	**Erro**	**Precisão**	**Recall**	**Espec.**
SAGRES	0,9969	0,0031	1,0	0,9508	1,0
TheEnd	1,0 **(0,9852)**	0,0 **(0,0148)**	1,0 **(1,0)**	1,0 **(0,8863)**	1,0 **(1,0)**
CoDiVision	1,0 **(0,9858)**	0,0 **(0,0142)**	1,0 **(1,0)**	1,0 **(0,9204)**	1,0 **(1,0)**

Fonte – Autor.

Pode-se perceber pela análise da Tabela 9 que existem alguns valores destacados entre parênteses. Esses valores indicam o desempenho obtido pela abordagem considerando os sistemas **TheEnd** e **CoDiVision**, durante a primeira análise desses sistemas, antes das melhorias realizadas na implementação da abordagem para que fosse possível mapear chamadas a métodos realizadas por meio de *frameworks Web*, como o **Vraptor**, utilizado pelos sistemas citados.

Por meio dos valores apresentados na Tabela 9, pode-se constatar que o método proposto obteve uma **acurácia** de **0,9969** e uma **taxa de erro** de apenas **0.0031** para o sistema **SAGRES** e uma **acurácia** de **1,0** e **taxa de erro** de **0,0** para os sistemas

TheEnd e **CoDiVision**. Além disso, os valores de ***recall*** também foram significativos. Esses resultados indicam que o desempenho da abordagem no processo de identificação de arquivos relacionados às funcionalidades do sistema, foi bastante satisfatório.

4.1.8 Discussões

Por meio dos resultados obtidos durante esta primeira etapa de avaliação pode-se concluir que as respostas para as questões de pesquisa (**QP1a**, **QP1b** e **QP1c**) elencadas na Seção 4.1.2 são afirmativas, isto é, a abordagem proposta obteve desempenho satisfatório durante a: **QP1a)** identificação de arquivos controladores do sistema, **QP1b)** identificação de métodos principais (*features*) em classes *controllers* e **QP1c)** identificação de arquivos relacionados às funcionalidades identificadas. Isso porque o valores para a métrica **recall** sempre foram igual ou bem próximo de **1,0** para todos os processos analisados.

No contexto deste trabalho a métrica ***recall*** é a mais importante, pois essa medida indicou que abordagem proposta identificou corretamente todos os elementos da classe de interesse. Em outras palavras, a abordagem conseguiu identificar todas as classes *controllers* dos sistemas analisados, assim como todos os métodos principais que representavam uma funcionalidade. Quanto ao processo de identficação de arquivos relacionados às funcionalidades, apenas para o sistema **SAGRES** não foi possível identificar todas as classes pertencentes às funcionalodades identificadas. Apesar disso, o valor obtido para a métrica ***recall*** foi bem próximo de **1,0**, considerando o mesmo sistema.

Além disso, os valores obtidos para a métrica de **acurácia** foram de maneira geral significativos. É importante ressaltar que essa medida é considerada pela literatura como sendo uma métrica tendenciosa (POWERS, 2011), pois a abordagem poderia estar classificando incorretamente muitos elementos da classe de interesse e ainda assim serem obtidos valores significativos. Contudo, com os resultados obtidos para as métricas ***recall*** e **especificidade** indicam que, a abordagem proposta, de maneira geral, classificou corretamente os elementos de ambas as classes tratadas na avaliação.

Os valores obtidos para a métrica **precisão** não foram muito altos no processo de identificação de arquivos *controllers*. Contudo, vale ressaltar que isso ocorreu por conta da abordagem proposta ser inclusiva. Dessa forma, é priorizado a indicação de um maior número de possíveis classes como sendo controladoras. Pois é mais importante identificar todos os *controllers* em vez de não encontrar alguns deles. Além disso, para os processos de identificação de *features* e arquivos relacionados às funcionalidades identificadas, os valores obtidos para a **precisão** foram de maneira geral próximos ou iguais a **1,0**. Foi possível constatar que dentre os métodos indicados como funcionalidades, a maioria realmente representava *features* do sistema. Além disso, todos os arquivos em que abordagem classificou como pertencentes a uma determinada funcionalidade, realmente pertenciam.

Portanto, por meio dos resultados obtidos, principalmente pelos valores da métrica ***recall***, pode-se concluir que a resposta para a questão **QP1** é afirmativa, ou seja, a abordagem proposta obteve de maneira geral um bom desempenho no processo de *feature location* ou identificação de funcionalidades.

É importante ressaltar que a implementação da abordagem proposta permite que sejam manualmente excluídas falsas funcionalidades identificadas pelo mecanismo automático. Além disso, é possível ainda realizar a agregação de funcionalidades de nível **usuário** em *features* de nível **usuário** e/ou **agregador**. Dessa forma, é possível obter a familiaridade de código considerando diferentes níveis de funcionalidade.

4.2 Análise da Familiaridade de Código entre as Perspectivas de Módulo e Funcionalidade

Esta seção apresenta a aplicação da abordagem descrita neste trabalho por meio de uma análise da familiaridade de código entre as perspectivas de módulo e funcionalidade. Vale ressaltar que os resultados foram obtidos utilizando a ferramenta *CoDiVision* (LIRA, 2016), que implementa a abordagem proposta.

4.2.1 Objetivo

O objetivo principal desta etapa de avaliação é identificar o ganho de informação que pode ser obtido por meio da visão da familiaridade de código de desenvolvedores na perspectiva de funcionalidade, identificando a utilidade dessa informação em um contexto real de desenvolvimento de software.

4.2.2 Questão de Pesquisa

Esta segunda etapa de avaliação foi proposta para responder a seguinte questão de pesquisa:

- **QP1**: A abordagem proposta fornece informações úteis para um contexto real de desenvolvimento de software?

4.2.3 Objetos

Nesta etapa de avaliação foram avaliados dois sistemas de software reais: **SIGAA** (Sistema Integrado de Gestão de Atividades Acadêmicas)[6] e **SERCA** (Sistema Emissor de Relatórios de Contas Anuais)[7]. O sistema **SIGAA** é utilizado por diversas universidades do

[6] Sistema SIGAA: <http://www.sigaa.ufpi.br>.
[7] Sistema de acesso restrito, pertencente ao **Tribunal de Contas do Estado do Piauí (TCE-PI)**.

Brasil, e a versão analisada pertence à Universidade Federal do Piauí (UFPI).Esse sistema conta com diversos módulos, abrangendo desde o ensino infantil até a pós graduação, e possui milhares de usuários e milhões de linhas de código, permitindo o controle de diversas atividades como projetos de pesquisa, ações de extensão, projetos de ensino, registro e relatórios da produção acadêmica, ensino a distância, etc., (FILHO; AQUINO; ROSA, 2013).

O sistema **SERCA** é utilizado pelo Tribunal de Contas do Estado do Piauí (TCE/PI), para registro de dados sobres auditorias de contas e obras/serviços de jurisdicionados ou representantes/gestores do poder político no estado do Piauí. O sistema permite também a emissão automática de relatórios com dados de auditorias realizadas. Esse sistema foi desenvolvido em linguagem de programação Java, assim como o sistema **SIGAA**.

4.2.4 Participantes

Esta segunda etapa de avaliação também contou com a participação direta dos principais desenvolvedores responsáveis pela implementação dos softwares avaliados. Um desenvolvedor de cada sistema auxiliou na discussão dos resultados obtidos por meio de informações úteis de cada projeto avaliado. Além disso, cada desenvolvedor apoiou no processo de agregação de funcionalidades realizado durante esta etapa de avaliação.

4.2.5 Contexto

Para o sistema **SIGAA** foram analisados *commits* feitos desde **05/09/2017**, quando deixaram de ser realizados por um usuário único, por meio da utilização de um repositório de código *SVN*. A partir desse período passou-se a utilizar o *Git*, com o registro de alterações feitas por cada desenvolvedor. Para o sistema **SIGAA** foram extraídos um total de **250** *commits* realizados durante o período de análise (de 05/09/2017 a 06/03/2018). Para o sistema **SERCA** foram analisados os *commits* feitos desde o inicio do desenvolvimento do software. O primeiro *commit* registrado no repositório Git foi em **13/04/2017**. No total foram analisados **3864** *commits* realizados no sistema **SERCA**, no período de 13/04/2017 a 09/05/2018.

Foram identificadas, especialmente no sistema **SIGAA**, um grande número de funcionalidades em nível de *subfunção*. Por essa razão, essas funcionalidades foram agrupadas em *features* de nível *agregador*. Dessa forma, foram obtidas um total de **1179** e **29** funcionalidades, para os sistemas **SIGAA** e **SERCA**, respectivamente. Todos os dados desta avaliação, podem ser acessados *online*[8].

[8] Dados da avaliação: <https://goo.gl/KwTbDC>

É importante ressaltar, que os sistemas analisados na etapa de avaliação anterior não foram utilizados nesta etapa porque as ferramentas **CoDiVision** e **TheEnd** tiveram, respectivamente, apenas 1 e 2 desenvolvedores, números considerados baixos para uma avaliação. O sistema **SAGRES**, por sua vez, teve grande parte do histórico de *commits* perdido por conta de mudança (de *SVN* para *Git*) no repositório de código utilizado.

4.2.6 Operação

É importante ressaltar que a ferramenta *CodiVision*, utilizada durante esta etapa de avaliação, permite a configuração de métricas utilizadas durante o cálculo da familiaridade. Os parâmetros para as métricas utilizadas no processo de inferência da familiaridade foram definidos com base nos valores utilizados em (LIRA, 2016). Esses parâmetros foram estipulados pelo autor após uma avaliação baseada na aplicação de um questionário, realizado com a participação de pesquisadores e desenvolvedores de software.

Tabela 10 – Parâmetros para as métricas utilizadas na inferência da familiaridade.

Tipos de alteração				**Degradação da familiaridade**		**Truck Factor (TF)**
Adição	**Modificação**	**Deleção**	**Condição**	**Tempo (mês)**	**Nova alteração**	
1,0	1,0	0,5	1,0	0%	5%	50% - 60% - 70%

Fonte – Autor.

A Tabela 10 apresenta os parâmetros utilizados nesta avaliação, com base no que foi utilizado por (LIRA, 2016). Para esta avaliação foi estipulado o valor de **1,0** para ponderação da operações de **adição**, **modificação** e alterações em **comandos de condição**. Para a operação de **remoção** foi definido o valor de **0,5**. Esses valores para as métricas de **tipos de alteração** indicam que as operações de remoção de código influenciam menos no cálculo da familiaridade de desenvolvedores. Para a métrica de **degradação** da familiaridade por **novas alterações** foi utilizado a porcentagem de 5%. Isso indica que a familiaridade de um desenvolvedor específico é degradada em **0,5%** (neste caso) a cada nova alteração realizada por outros desenvolvedores no mesmo arquivo de código fonte, e consequentemente na mesma funcionalidade.

Para esta avaliação foi desconsiderada a degradação da familiaridade de código por tempo(mês) recorrido. Isso indica que a familiaridade dos desenvolvedores não sofreu influência por conta do período em que as alterações de código foram realizadas. Dessa forma, foi definida a porcentagem de 0% como parâmetro para a métrica de degradação por tempo. O fatores que levaram à definição desse valor foram os períodos das alterações realizadas nos sistemas analisados durante esta avaliação. Esses períodos de análise são relativamente recentes, e são apresentados na seção seguinte. Por fim, cada sistema foi

avaliado em três situações: com o valor do limiar para a métrica *Truck Factor* definido em 50%, 60% e 70%, respectivamente.

4.2.7 Resultados

Esta seção apresenta os resultados obtidos durante a análise da familiaridade de código entre as perspectivas de módulo e funcionalidade, considerando os sistemas **SIGAA** e **SERCA**. A Tabela 11 apresenta os resultados da análise para o sistema **SIGAA**. As colunas **TF-50**, **TF-60**, **TF-70** indicam a porcentagem de funcionalidades "dominadas" por cada desenvolvedor, quando o limiar da métrica ***Truck Factor* (TF)** (RICCA; MARCHETTO; TORCHIANO, 2011) foi de **50%**, **60%** e **70%**, respectivamente. Um desenvolver d domina uma funcionalidade f quando, para essa funcionalidade f, o valor da métrica TF é igual a 3 (indicando que existem três desenvolvedores que possuem a maior porcentagem de familiaridade sobre a funcionalidade f), por exemplo, e o desenvolver d está entre os três desenvolvedores que mais possuem familiaridade sobre tal funcionalidade.

Tabela 11 – Resultados da análise para o sistema SIGAA.

	Features Dominadas (%)					
Desenv.	**TF-50**	**TF-60**	**TF-70**	**Média**	**FF**	**FM**
$D_{a,SIG}$	2,37%	2,46%	2,46%	2,43%	1,90%	1,70%
$D_{b,SIG}$	0,42%	0,42%	0,42%	0,42%	7,00%	6,30%
$D_{c,SIG}$	4,66%	6,19%	6,19%	5,68%	10,00%	10,10%
$D_{d,SIG}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,50%**	**0,40%**
$D_{e,SIG}$	**36,81%**	**39,44%**	**39,44%**	**38,56%**	**32,60%**	**31,80%**
$D_{f,SIG}$	**22,05%**	**25,53%**	**25,53%**	**24,37%**	**36,40%**	**35,80%**
$D_{g,SIG}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**
$D_{h,SIG}$	**16,12%**	**16,20%**	**16,20%**	**16,17%**	**0,10%**	**12,70%**
$D_{i,SIG}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,50%**	**0,50%**
$D_{j,SIG}$	2,88%	2,97%	2,97%	2,94%	0,80%	0,70%
$D_{k,SIG}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**

FF: Familiaridade sobre todas as funcionalidades.
FM: Familiaridade sobre o módulo de arquivos Java.
Fonte – Autor.

Pode-se perceber que os desenvolvedores $D_{e,SIG}$ e $D_{f,SIG}$ "dominam" o maior número de funcionalidades no sistema **SIGAA**. Além disso, esses desenvolvedores possuem maior grau de familiaridade sobre todas as *features* e também sobre o módulo de arquivos de código fonte Java. É importante notar também que apesar do desenvolvedor $D_{f,SIG}$ possuir mais familiaridade sobre as funcionalidades em geral e também sobre o módulo de arquivos Java, é $D_{e,SIG}$ quem domina o maior número de *features* do sistema.

Isso indica que não se pode afirmar que o desenvolvedor com um maior grau de familiaridade sobre módulo, também será sempre aquele que dominará o maior número de funcionalidades do sistema. Um dos fatores que contribuem para isso é que algumas funcionalidades específicas possuem um maior número de arquivos em relação a outras, e portanto, aquele desenvolvedor que domina menos *features*, mas que essas possuem um maior número de arquivos, tende a possuir mais familiaridade quando se tem uma visão por módulo.

Tabela 12 – Resultados da análise para o sistema SERCA.

	Features Dominadas (%)					
Desenv.	**TF-50**	**TF-60**	**TF-70**	**Média**	**FF**	**FM**
$D_{a,SER}$	**57,14%**	**67,85%**	**71,42%**	**65,47%**	**18,30%**	**17,80%**
$D_{b,SER}$	**25,00%**	**25,00%**	**32,14%**	**27,38%**	**24,00%**	**23,00%**
$D_{c,SER}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**
$D_{d,SER}$	21,42%	28,57%	39,28%	29,76%	13,10%	12,60%
$D_{e,SER}$	10,71%	10,71%	17,85%	13,09%	6,60%	6,00%
$D_{f,SER}$	3,57%	14,28%	25,00%	14,28%	9,60%	10,40%
$D_{g,SER}$	17,85%	32,14%	39,28%	29,76%	6,60%	7,00%
$D_{h,SER}$	**64,28%**	**75,00%**	**75,00%**	**71,43%**	**21,90%**	**23,40%**
$D_{i,SER}$	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**	**0,00%**

FF: Familiaridade sobre todas as funcionalidades.
FM: Familiaridade sobre o módulo de arquivos Java.
Fonte – Autor.

Pode-se observar também, que os desenvolvedores $D_{d,SIG}$ e $D_{i,SIG}$ não dominam nenhuma das funcionalidades do sistema **SIGAA**, mas possuem (apesar de baixa) alguma familiaridade considerando todas as funcionalidades e também sobre o módulo de arquivos Java. Isso pode indicar que esses desenvolvedores realizam alterações em algumas funcionalidades do sistema, mas bem menos em relação a outros desenvolvedores. É possível observar também que os desenvolvedores $D_{g,SIG}$ e $D_{k,SIG}$ não possuem familiaridade sobre o módulo de arquivos Java e também sobre funcionalidades. Esse fato pode ter ocorrido por dois motivos: 1) O grau de familiaridade foi reduzido em razão de alterações feitas por outros desenvolvedores, já que para esta avaliação foi utilizada como configuração na ferramenta **CoDiVision** uma redução de **0,5%** sobre a familiaridade em decorrência de alterações realizadas sobre o mesmo arquivo por diferentes indivíduos; 2) Esses desenvolvedores realizaram alterações apenas em arquivos não Java, como os de interface gráfica e configuração do sistema, por exemplo.

Os resultados obtidos durante a análise do sistema **SERCA** são apresentados na Tabela 12. Pode-se notar que, assim como para o sistema **SIGAA**, houve desenvolvedores

que não apresentaram nenhuma familiaridade com as funcionalidades sistema ou com o módulo de arquivos Java. Em contrapartida, para o sistema **SERCA** não existem casos em que um desenvolvedor específico possui alguma familiaridade sobre as funcionalidades em geral ou sobre o módulo Java, mas que não dominam alguma funcionalidade do sistema. Outro fato importante pode ser notado em relação aos desenvolvedores $D_{e,SER}$, $D_{f,SER}$ e $D_{g,SER}$. O desenvolvedor $D_{g,SER}$ possui familiaridade (considerando as funcionalidades gerais e o módulo de arquivos JAVA) quase equivalente ao desenvolvedor $D_{e,SER}$ e menor em relação à $D_{f,SER}$. Contudo, é $D_{g,SER}$ que domina o maior número de funcionalidades em relação aos mesmos desenvolvedores.

É importante notar também, que o desenvolvedor $D_{b,SER}$ é um dos que possuem maior familiaridade considerando a perspectiva de módulo. Contudo, o desenvolvedor $D_{a,SER}$ que possui um grau de familiaridade um pouco menor, e o desenvolvedor $D_{h,SER}$, que possui um grau de familiaridade (considerando a mesma perspectiva) bem próximo ao que $D_{b,SER}$ possui, dominam um maior número de funcionalidades em relação ao desenvolvedor $D_{b,SER}$. Essa questão é justificada pelo fato de que o desenvolvedor $D_{b,SER}$ foi quem realizou o maior número de alterações em arquivos de uma das principais funcionalidades do sistema. Essa funcionalidade, por sua vez, envolvia o maior número de arquivos relacionados. Esse fato reforça o que foi mencionado anteriormente (durante a análise do sistema **SIGAA**), que não é possível afirmar com certeza que aquele desenvolvedor com maior grau de familiaridade considerando a perspectiva de módulo, será também aquele que domina o maior número de funcionalidades do sistema.

Para exemplificar que uma visão da familiaridade de código, considerando apenas a perspectiva de módulo, pode levar gerentes de projetos a tomar decisões precipitadas no processo de alocação de tarefas, foram selecionadas (com o apoio do desenvolvedor participante) quatro funcionalidades principais pertencentes ao sistema **SERCA**, denominadas por **Gerenciar Relatório (Cadastro, Listagem e Edição)** (F_1), **Gerenciar Achado** (F_2), **Gerenciar Registro de Achado** (F_3) e **Gerenciar Geração de Relatório** (F_4), com o objetivo de mostrar que o processo de alocação de tarefas com base apenas na informação da familiaridade de código sobre a perspectiva de módulo pode não ser a ideal.

A Tabela 13 apresenta o grau de familiaridade dos desenvolvedores do sistema **SERCA** sobre as funcionalidades citadas, bem como o grau de familiaridade dos mesmos desenvolvedores sobre o módulo principal do projeto, considerando todas as classes do sistema. Pode-se perceber que caso o gerente de projeto leve como base apenas a familiaridade de código sobre módulo, os desenvolvedores $D_{a,SER}$, $D_{b,SER}$ e $D_{h,SER}$ provavelmente seriam os mais indicados a realizar alterações de melhorias ou correções de *bugs* sobre alguma das funcionalidades.

Supondo que a funcionalidade F_4 necessite de alterações para correção de problemas, o gerente de projetos tenderia a alocar o desenvolvedor $D_{h,SER}$ para essa tarefa, pois $D_{h,SER}$

possui uma maior familiaridade considerando a perspectiva de módulo. Contudo, para esse caso, o desenvolvedor mais apropriado para corrigir os problemas relacionados a funcionalidade F_4, seria $D_{b,SER}$, que sozinho, possui **41,06%** de familiaridade sobre tal funcionalidade. Isso reforça que uma visão da familiaridade de código considerando apenas a perspectiva de módulo pode influenciar em decisões pouco apropriadas durante o processo de alocação de tarefas.

Tabela 13 – Familiaridade dos desenvolvedores sobre algumas funcionalidades do sistema SERCA.

Desenv.	**Familiaridade sobre funcionalidades**				**Familiaridade sobre módulo**
	F_1	F_2	F_3	F_4	
$D_{a,SER}$	29,55%	24,13%	20,75%	15,12%	17,80%
$D_{b,SER}$	6,62%	4,16%	7,81%	41,06%	23,00%
$D_{c,SER}$	0,00%	0,00%	0,00%	0,00%	0,00%
$D_{d,SER}$	10,51%	12,94%	7,99%	7,57%	12,60%
$D_{e,SER}$	0,89%	4,33%	8,53%	4,78%	6,00%
$D_{f,SER}$	11,15%	6,38%	9,06%	9,06%	10,40%
$D_{g,SER}$	11,98%	7,13%	8,00%	4,10%	7,00%
$D_{h,SER}$	29,30%	40,94%	35,26%	18,32%	23,40%
$D_{i,SER}$	0,00%	0,00%	0,00%	0,00%	0,00%

Fonte – Autor.

Os dados de familiaridade por funcionalidade foram apresentados ao gerente de projeto do software SERCA. Ele gostou da informação obtida e destacou que ela seria muito importante para a decisão de alocação de desenvolvedores para atividades de manutenção. Ele ressaltou que embora tenha-se ideia dos desenvolvedores mais ligados a determinadas partes do software, essa informação perde-se ao longo do tempo e torna-se difícil de ser inferida em projetos com uma grande equipe. Assim, a visualização oferecida é de grande valia para os responsáveis por um projeto de software.

4.2.8 Discussões

Durante a segunda etapa de avaliação da abordagem proposta, pode-se realizar constatações importantes a respeito da familiaridade de código considerando as perspectivas de módulo e funcionalidade. Uma noção da familiaridade de código considerando a perspectiva de funcionalidades pode evitar conclusões distorcidas quando se tem um noção da familiaridade considerando apenas a perspectiva de módulo. Pelos resultados obtidos, foi possível verificar que não se pode afirmar que um desenvolvedor que possui um alto grau de familiaridade sobre o módulo do projeto como um todo, também será aquele que possui uma maior familiaridade sobre a maioria das funcionalidades de um sistema.

Com isso, apesar do indicador sobre a familiaridade de código considerando a perspectiva de módulo oferecer informações importantes, uma visão por funcionalidade pode evitar conclusões equivocadas por parte de gerentes de projetos. Além disso, uma visão por funcionalidade tende a ser mais adequada em um contexto de realização de manutenções em softwares, uma vez que boa parte das solicitações de manutenção tende a levar em conta as funcionalidades do software.

Os resultados obtidos foram apresentadas para cada desenvolvedor participante da segunda etapa de avaliação. Esses desenvolvedores foram indagados se as informações sobre a familiaridade de código considerando a perspectiva de funcionalidade seriam úteis em um contexto real de desenvolvimento de software. A resposta dos desenvolvedores foi positiva, indicando que abordagem proposta fornece informações úteis sobre a familiaridade de código considerando a perspectiva de funcionalidade. A partir disso, e dos resultados obtidos, pode-se indicar que a resposta para a questão de pesquisa **QP2** é afirmativa, isto é, as informações obtidas por meio da noção da familiaridade de código, considerando a perspectiva de funcionalidades, é útil e mais apropriada para um contexto real.

4.3 Ameaças à Validade

Uma das questões fundamentais a respeito de uma avaliação refere-se ao grau de validade dos resultados obtidos. Assim, é importante analisar os riscos relacionados à avaliação e propor alternativas para tentar contornar os problemas que possam ameaçar à validade dos resultados (WOHLIN et al., 2012). Esta seção apresenta as ameaças ligadas à avaliação da abordagem proposta neste trabalho.

4.3.1 Validade Interna

Este aspecto está relacionado à validade dos resultados identificando a relação causa-efeito (WOHLIN et al., 2012), ou seja, uma ameaça relacionada ao controle do processo de avaliação para coletar os dados analisados.

Uma ameaça interna relacionada a este trabalho refere-se aos dados utilizados durante a segunda etapa de avaliação. É comum que durante o processo de desenvolvimento de software as alterações sejam transferidas para o repositório de código por meio de um usuário único cadastrado no sistema de controle de versão. Dessa forma, mesmo que diversos desenvolvedores participem do desenvolvimento de um software específico, as alterações sempre irão estar ligadas a um mesmo usuário ou "desenvolvedor".

Esse contexto aconteceu com um dos sistemas analisados na segunda etapa de avaliação. Dessa forma, foram descartados todos os *commits* realizados por esse usuário único, e considerou-se apenas *commits* feitos por desenvolvedores individualmente. Como consequência o número de *commits* analisados foi reduzido, influenciando assim no total

de alterações de código consideradas no processo de inferência da familiaridade dos desenvolvedores. Contudo, foi obtido um número (250, no total) ainda significativo de *commits* realizados, permitindo assim que os resultados obtidos ainda fossem expressivos.

4.3.2 Validade Externa

A validade externa está relacionada ao risco de generalização dos resultados, pois alguns fatores podem ameaçar a validade e não permitir a generalização dos resultados obtidos durante as avaliações.

Essa ameaça está relacionada a quantidade e tipos de sistemas analisados, especialmente durante a primeira etapa de avaliação. Foram utilizados 3 (três) sistemas reais. Contudo, os resultados obtidos poderiam ser ainda mais expressivos por meio da análise de um maior número de softwares, e em larga escala.

Para atenuar essa ameaça e obter resultados mais significativos, durante a primeira etapa de avaliação foram utilizados sistemas reais e distintos, para que a abordagem pudesse ser avaliada em diferentes situações, considerando o processo de *feature location*. Com isso, foi possível ainda realizar melhorias na implementação da abordagem, no que diz respeito a identificação de arquivos relacionados às funcionalidades. Por meio dos resultados obtidos foi possível perceber que a abordagem obteve desempenho satisfatório durante o processo de identificação de funcionalidades e arquivos relacionados.

A quantidade de sistemas analisados durante a segunda etapa da avaliação também representa uma ameaça à validade. Vale lembrar que os sistemas utilizados na primeira etapa não puderam ser analisados na segunda etapa pela falta do histórico completo de alterações ou porque poucos desenvolvedores participaram do processo de desenvolvimento. Contudo, foram utilizados dois sistemas, de médio e grande porte. Além disso, a análise foi realizada com o apoio dos desenvolvedores responsáveis por cada um dos sistemas. Esses desenvolvedores participantes informaram também que os resultados, de maneira geral, foram de acordo com a percepção que tinham do projeto, como por exemplo, que desenvolvedor tinha maior familiaridade sobre algumas funcionalidades do sistema.

4.3.3 Validade de Construção

A validade de construção se refere à influência que pode ter sido causada pela forma de construção da avaliação. Essa ameaça está relacionada à configuração de parâmetros para as métricas utilizadas no cálculo da familiaridade de código, durante a segunda etapa de avaliação. Com a variação dos valores desses parâmetros, os resultados obtidos também devem variar.

Contudo, durante a análise realizada, os parâmetros das métricas de familiaridade foram definidos em conformidade com o que foi recomendado em (LIRA, 2016), onde

os parâmetros foram definidos com base na aplicação de um questionário de pesquisa, respondido por pesquisadores e desenvolvedores profissionais. De acordo com os resultados foram estipulados os valores que tendem ser mais adequados para os parâmetros das métricas de familiaridade.

4.3.4 Validade de Conclusão

A validade da conclusão está relacionada à capacidade de chegar a conclusões de acordo com os resultados obtidos a partir da aplicação do método. Essa ameaça refere-se à validade dos resultados obtidos a respeito da análise da familiaridade de código considerando a perspectiva de funcionalidade.

A informação sobre a familiaridade por funcionalidades depende diretamente de um bom processo de *feature location* (FL), isto é, as funcionalidades e seus arquivos relacionados necessitam ser corretamente identificados. Quanto mais preciso for esse processo mais significativos serão os resultados obtidos durante a análise da familiaridade de código considerando a perspectiva de funcionalidade. Para atenuar essa ameaça, a primeira etapa da avaliação teve como objetivo averiguar se a abordagem possui um bom desempenho durante o processo de FL. Por meio dos resultados obtidos foi possível observar que a abordagem obteve desempenho satisfatório.

4.4 Considerações Finais

Este capítulo apresentou uma avaliação da abordagem proposta neste trabalho. Por meio dos resultados obtidos pôde-se perceber que a abordagem proposta obteve um bom desempenho no processo de identificação de funcionalidades e arquivos relacionados. A partir disso, foi realizada uma análise da familiaridade de código considerando as perspectiva de módulo e funcionalidade. Por meio dos resultados obtidos durante essa fase da avaliação foi possível perceber o ganho de informações que pode ser obtido quando se tem diferentes visões da familiaridade de código, e que uma visão da familiariridade considerando a perspectiva de funcionalidade é útil e mais apropriada em um contexto de desenvolvimento de software real.

5 Conclusões e Trabalhos Futuros

A manutenção de software que pode ser definida como a modificação do sistema após a entrega para corrigir falhas, incluir novas funcionalidades, melhorar o desempenho ou se adaptar a um novo ambiente (COMMITTEE, 1983). A etapa de manutenção é normalmente a atividade mais onerosa no ciclo de vida de um software, pois é executada durante toda sua vida útil. Uma atividade bastante comum durante a manutenção de software é o planejamento da execução do trabalho, que envolve a definição de quais desenvolvedores devem executar cada tarefa necessária. É comum que a alocação de tarefas a desenvolvedores seja feita levando em consideração a relação que um desenvolvedor possui com o código fonte associado, isto é, o quão cada desenvolvedor conhece partes (arquivos, módulos ou funcionalidades) do software ligados a uma tarefa. Dessa forma, quanto maior a relação entre um determinado desenvolvedor e o código fonte associado a uma tarefa a ser executada, maior será a probabilidade desse desenvolvedor assumir a responsabilidade de executá-la. Assim, é de suma importância que gerentes de projetos tenham uma noção mais próxima da familiaridade de código que os desenvolvedores possuem.

Por meio da realização de um Mapeamento Sistemático da Literatura (MSL) (KITCHENHAM, 2004), realizado também neste trabalho, foi constatado que existem vários estudos que visam inferir a relação entre desenvolvedor e código fonte. Contudo, a grande maioria desses trabalhos focam em inferir a relação de código considerando a perspectiva de arquivo ou módulo de software. Apesar da informação sobre a familiaridade de código considerando essas perspectivas fornecerem um bom suporte durante a manutenção, uma visão por funcionalidade pode oferecer uma noção ainda maior sobre a familiaridade que desenvolvedores de software possuem sobre o código fonte, além de ser mais adequada considerando um contexto real de desenvolvimento de software, pois as tarefas de manutenção geralmente são tratadas em nível de funcionalidades e consequentemente é mais adequada a noção da familiaridade de código considerando a perspectiva de *features*.

Dessa forma, este trabalho apresentou uma abordagem baseada em técnicas de Engenharia Reversa (ER) (CHIKOFSKY; CROSS, 1990; MÜLLER et al., 1992) e Mineração de Repositórios de Software (MRS) (HASSAN, 2008), para inferência da familiaridade de código que desenvolvedores possuem sobre funcionalidades de software. O objetivo principal da abordagem proposta é apoiar gerentes ou líderes de projetos durante o processo de desenvolvimento de software, principalmente durante as atividades de manutenção, sejam elas corretivas, adaptativas ou perfectivas. A abordagem é baseada no método proposto por LIRA 2016, que visa inferir a familiaridade de código considerando a perspectiva de arquivos ou módulo de software. Algumas das principais etapas da abordagem proposta neste trabalho visam a identificação de funcionalidades, assim como os arquivos de có-

digo relacionados à cada funcionalidade identificada, dessa forma, a abordagem proposta engloba uma técnica alternativa para ***Feature Location*** (FL) (DIT et al., 2013). Por fim, é possível então inferir a familiaridade sobre funcionalidades de software, com base na familiaridade de código que desenvolvedores possuem sobre os arquivos relacionados a cada funcionalidade.

A abordagem proposta foi implementada com um novo módulo na ferramenta *CoDiVision* (LIRA, 2016). Por meio do uso dessa ferramenta, gerentes de projetos de software podem por meio de visualização gráfica obter a informação sobre a familiaridade de código que desenvolvedores possuem sobre as funcionalidades do sistema. Com essa informação é possível definir melhores estratégias durante o processo de manutenção de software, mas especificamente quanto a distribuição de tarefas com base na relação que cada desenvolvedor possui com o código fonte ou funcionalidades associados a cada uma delas. Dessa forma, com a melhoria no processo de manutenção, o nível de qualidade do software gerado também tende a ser mais elevado.

Neste trabalho foi realizada também uma avaliação da abordagem proposta. Essa avaliação foi dividida em duas etapas. Na primeira fase foi avaliado o processo de identificação de funcionalidades e arquivos relacionados. Por meio da análise de sistemas reais, pôde-se constatar que a abordagem obteve um desempenho satisfatório durante esse processo. Durante a segunda etapa de avaliação foi realizada a inferência e análise da familiaridade de código entre desenvolvedores considerando a perspectiva de módulo e funcionalidade.

Os resultados da avaliação mostraram que uma visão por funcionalidade fornece informações importantes a respeito da familiaridade de código. Umas das principais constatações é que não se pode afirmar que desenvolvedores com um maior grau de familiaridade de código sobre módulo de arquivos principais, também terão mais familiaridade sobre um maior número de funcionalidades de um sistema. Pôde-se constatar também que a informação da familiaridade de código considerando a perspectiva de funcionalidade é mais apropriada em um contexto real de desenvolvimento de software, evitando decisões precipitadas por parte de gerentes de projetos, no processo de alocação de tarefas.

Por fim, para efeito de sumarização, a seguir são listadas as principais contribuições oriundas da realização deste trabalho:

- **Realização de um Mapeamento Sistemático da Literatura (MSL)** (KITCHENHAM, 2004): desenvolveu-se um mapeamento sistemático (Capítulo 2) com o objetivo de realizar um levantamento da área de pesquisa deste trabalho, construindo assim conhecimento a partir das publicações relacionadas à esta pesquisa. Por meio desse estudo secundário foi possível obter informações importantes da área de inferência da familiaridade com base em técnicas de Mineração de Repositórios

de Software (MSR). Uma das principais descobertas é que grande parte dos estudos desta área propõe métodos que inferem a familiaridade de código considerando as perspectivas de linha, arquivo ou módulo de software.

- **Desenvolvimento de uma abordagem para inferência da familiaridade sobre funcionalidades de software**: foi desenvolvida uma abordagem baseada em técnicas de Engenharia Reversa (ER) e Mineração de Repositórios de Software (MRS) para inferência da familiaridade de código considerando a perspectiva de funcionalidades de software (Capítulo 3). Uma das etapas principais da abordagem proposta, consiste em identificar funcionalidades e arquivos relacionados.

- **Implementação da abordagem proposta**: A abordagem foi implementada como um novo módulo de na ferramenta *CoDiVision* (LIRA, 2016). A ferramenta passou a permitir a visualização da familiaridade de código considerando a perspectiva de funcionalidades. A ferramenta permite também o gerenciamento das *features* identificadas, permitindo a exclusão de falsas funcionalidades e também a realização de agregação de *features* de nível de **subfunção**, formando assim funcionalidades de maior nível (**usuário** e/ou **agregador**).

- **Análise da familiaridade de código entre as perspectivas de módulo e funcionalidade**: Durante a etapa de avaliação foi realizada a inferência e análise da familiaridade de código entre desenvolvedores considerando a perspectiva de módulo e funcionalidade. Os resultados apresentaram o ganho de informações obtido para melhor entender a distribuição da familiaridade entre membros de uma equipe de desenvolvimento de software.

5.1 Publicações

Durante esta pesquisa foram publicados três artigos científicos. O primeiro trabalho apresenta o método para inferência da familiaridade considerando a perspectiva de módulo ou arquivo, e que compõe a abordagem proposta neste trabalho. O segundo estudo apresenta a ferramenta *CoDiVision* e sua utilização em um contexto educacional, apresentando a importância da ferramenta durante a avaliação de estudantes em disciplinas de programação.

I) Irvayne M. S. Ibiapina, F. V. M. Alves, Werney A. L. Lira, Gleison A. Silva, and Pedro A. S. Neto. 2017. **Inferência da Familiaridade de Código por Meio da Mineração de Repositórios de Software**. Simpósio Brasileiro de Qualidade de Software (SBQS).

II) F. V. M. Alves, Irvayne M. S. Ibiapina, Werney A. L. Lira, and Pedro A. S. Neto. 2017. **CoDiVision: Uma Ferramenta para Apoio na Avaliação de Estudantes no Ensino de Programação**. Simpósio Brasileiro de Informática na Educação (SBIE).

III) F. V. M. Alves, Pedro A. S. Neto, Werney A. L. Lira , and Irvayne M. S. Ibiapina. 2018. ***Analysis of Code Familiarity in Module and Functionality Perspectives***. Simpósio Brasileiro de Qualidade de Software (SBQS).

O terceiro artigo científico descreve toda a abordagem proposta neste trabalho, apresentando também os resultados obtidos durante as avaliaçãoes realizadas.

5.2 Desafios e Limitações

Como apresentado durante a etapa de avaliação (Capítulo 4) a abordagem proposta neste trabalho obteve resultados satisfatórios no processo de identificação de funcionalidades e e arquivos de código relacionados, contribuindo assim para uma obtenção mais precisa sobre a familiaridade de código a perspectiva de funcionalidades. Contudo, a abordagem também apresenta algumas limitações. A primeira está relacionada à linguagem de programação utilizada no desenvolvimento do sistema que será analisado. Como mencionado anteriormente, o método proposto é baseado em técnicas de Engenharia Reversa (ER) e realiza a análise estática de código fonte para identificar o relacionamento entre arquivos de sistemas escritos em linguagem de programação Java (GOSLING; MCGILTON, 1995).

Contudo, a linguagem de programação Java permite a criação de sistemas robustos e por conta disso é umas das linguagens de programação mais utilizadas atualmente, tanto em ambientes acadêmicos, quanto profissionais. Essas circunstâncias elevam muito a chance de utilização da abordagem proposta para inferência da familiaridade sobre funcionalidades de software, já que é bastante comum a utilização da linguagem Java durante o desenvolvimento de projetos de software.

Por basear-se na análise estática de código fonte, a abordagem proposta também possui uma limitação quanto à identificação de classes que compõem uma funcionalidade específica. Isso ocorre quando o sistema analisado utiliza-se de classes pertencentes a bibliotecas externas, e assim alguns tipos de invocações ou chamadas a métodos podem não ser mapeados, pois o acesso ao código fonte de classes externas ao sistema é dificultado. Esse fato contribuiu para a não identificação de algumas classes relacionadas às funcionalidades de um dos sistemas analisados durante a primeira etapa da avaliação, apresentada no Capítulo 4. Além disso, algumas classes podem não ser identificadas por conta do uso de *interfaces* (um dos recursos da orientação a objetos presentes na linguagem Java) pelos sistemas analisados. Isso porque essas classes muitas vezes só podem ser identificadas em tempo de execução.

Um desafio encontrado durante a avaliação da abordagem proposta foi a determinação de alguns arquivos de código fonte ou classes como sendo controladores do sistema, mas que na realidade não eram. Dessa forma, foram identificadas falsas funcionalidades do sistema. Contudo, nenhuma das verdadeiras funcionalidades dos softwares analisados deixou de ser inclusa. Além disso, a implementação da abordagem na ferramenta *CoDiVision* permite que falsas funcionalidades sejam facilmente descartadas pelo usuário, por meio de uma interface gráfica provida pela ferramenta.

Em relação à avaliação apresentada no Capítulo 4, existe também outro desafio a ser superado. Como foi apresentado, a avaliação da abordagem no processo de identificação de funcionalidades e arquivos relacionados, bem como na análise da familiaridade de código entre as perspectivas de módulo e funcionalidade, foi realizada com base em diferentes sistemas reais. Apesar disso, a análise de outros softwares durante a primeira etapa de avaliação poderia tornar os resultados ainda mais expressivos. Além disso, durante a segunda etapa, a utilização de mais sistemas poderia auxiliar ainda mais na identificação de padrões associados às perspectivas de módulo e funcionalidade, a respeito da familiaridade de código. Contudo, os resultados obtidos são promissores e sugerem que a abordagem proposta pode ser utilizada como um suporte a gerentes ou líderes de projetos, especialmente durante o processo de manutenção de software.

Outra limitação da abordagem proposta neste trabalho está ligado ao padrão arquitetural dos sistemas analisados. A abordagem tem como uma das principais subetapas identificar arquivos controladores, que armazenam métodos responsáveis pelo início da execução de funcionalidades do sistema. Com isso, a abordagem é aplicável em sistemas que seguem o padrão MVC (*model-view-controller*). Essa limitação fez com que não fossem inclusos, durante a segunda etapa de avaliação (Capítulo 4), outros sistemas de softwares reais que mantinham seus arquivos controladores em uma biblioteca de código externa.

Por fim, é notável que apesar do trabalho já realizado, existem algumas possibilidades de melhorias e trabalhos futuros. Assim, tais limitações aqui apresentadas oferecem novas oportunidades de pesquisa futuramente.

5.3 Trabalhos Futuros

Com base nos desafios e limitações apresentadas na Seção 5.2, foram definidas algumas direções para trabalhos futuros:

- **Extensão e melhorias na implementação da abordagem:** A abordagem proposta se baseia em técnicas de Engenharia Reversa (ER) e realiza a análise estática de código. Dessa forma o processo de identificação de funcionalidades, bem como os arquivos relacionados depende da linguagem de programação utilizada na codifica-

ção do sistema analisado. Dessa forma a abordagem foi inicialmente implementada para análise de sistemas desenvolvidos em linguagem Java. Contudo, é importante também a implementação de uma extensão da abordagem para análise de sistemas escritos em outras linguagens de programação. Além disso é importante também buscar alternativas para o problema de não identificação de arquivos relacionados a funcionalidades identificadas, quando o sistema utiliza de bibliotecas externas de código, e que controlam chamadas a métodos internos do sistema. Além de uma alternativa para análise de sistemas também utilizam bibliotecas externas para manterem arquivos de diferentes camadas MVC.

- **Novas avaliações da abordagem:** Este trabalho apresenta uma avaliação dividida em duas etapas. É importante realizar uma análise entre as perspectivas de módulo e funcionalidade com base em um maior número de sistemas para tornar ainda mais expressivos os resultados obtidos. Dessa forma, ressalta-se a importância da realização de novas avaliações com diferentes aplicações em cenários reais e incluir uma avaliação de gerentes de projeto com relação à informação disponibilizada.

- **Revisão sistemática:** Neste trabalho foi realizado um Mapeamento Sistemático da Literatura (MSL) referente aos trabalhos que objetivam inferir a familiaridade de código, com base em técnicas de Mineração de Repositórios de Software (MRS). No entanto, um mapeamento sistemático é um estudo quantitativo e não qualitativo. Portanto, é interessante a realização de uma Revisão Sistemática da Literatura (KITCHENHAM, 2004) como extensão do mapeamento realizado neste trabalho, e com o objetivo de avaliar a qualidade dos trabalhos identificados.

Referências

ALRABAEE, S. et al. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation*, Elsevier, v. 11, p. S94–S103, 2014. Citado na página 130.

ALWIS, B. D.; SILLITO, J. Why are software projects moving from centralized to decentralized version control systems? In: IEEE. *Cooperative and Human Aspects on Software Engineering, 2009. CHASE'09. ICSE Workshop on*. [S.l.], 2009. p. 36–39. Citado 2 vezes nas páginas 14 e 16.

AVELINO, G. et al. A novel approach for estimating truck factors. In: IEEE. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. [S.l.], 2016. p. 1–10. Citado na página 20.

AVELINO, G. et al. A novel approach for estimating truck factors. In: IEEE. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. [S.l.], 2016. p. 1–10. Citado na página 132.

BALL, T. et al. If your version control system could talk. In: CITESEER. *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*. [S.l.], 1997. v. 11. Citado na página 12.

BANDARA, U.; WIJAYARATHNA, G. Deep neural networks for source code author identification. In: SPRINGER. *International Conference on Neural Information Processing*. [S.l.], 2013. p. 368–375. Citado na página 130.

BANDARA, U.; WIJAYARATHNA, G. Source code author identification with unsupervised feature learning. *Pattern Recognition Letters*, Elsevier, v. 34, n. 3, p. 330–334, 2013. Citado na página 130.

BERLACK, H. R. *Software configuration management*. [S.l.]: Wiley Online Library, 1992. Citado na página 12.

BIGGERSTAFF, T. J.; MITBANDER, B. G.; WEBSTER, D. The concept assignment problem in program understanding. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 15th international conference on Software Engineering*. [S.l.], 1993. p. 482–498. Citado na página 10.

BOETTICHER, G.; MENZIES, T.; OSTRAND, T. Promise repository of empirical software engineering data. *West Virginia University, Department of Computer Science*, 2007. Citado na página 18.

BONDY, J. A.; MURTY, U. S. R. *Graph theory with applications*. [S.l.]: Citeseer, 1976. v. 290. Citado 3 vezes nas páginas 5, 39 e 41.

BRINDESCU, C. et al. How do centralized and distributed version control systems impact software changes? In: ACM. *Proceedings of the 36th International Conference on Software Engineering*. [S.l.], 2014. p. 322–333. Citado na página 15.

BUCKNER, J. et al. Jripples: A tool for program comprehension during incremental change. In: IEEE. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. [S.l.], 2005. p. 149–152. Citado 2 vezes nas páginas 3 e 28.

BURROWS, S.; TAHAGHOGHI, S. M. Source code authorship attribution using n-grams. In: *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*. [S.l.: s.n.], 2007. p. 32–39. Citado na página 130.

BURROWS, S.; UITDENBOGERD, A. L.; TURPIN, A. Application of information retrieval techniques for source code authorship attribution. In: SPRINGER. *International Conference on Database Systems for Advanced Applications*. [S.l.], 2009. p. 699–713. Citado na página 130.

CHACON, S.; STRAUB, B. *Pro git*. [S.l.]: Apress, 2014. Citado 2 vezes nas páginas 12 e 39.

CHAIKALIS, T. et al. Seagle: Effortless software evolution analysis. In: IEEE. *IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014*. [S.l.], 2014. p. 581–584. Citado na página 19.

CHEN, K.; RAJLICH, V. Case study of feature location using dependence graph. In: IEEE. *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*. [S.l.], 2000. p. 241–247. Citado na página 28.

CHEN, K.; RAJLICH, V. Ripples: Tool for change in legacy software. In: IEEE COMPUTER SOCIETY. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. [S.l.], 2001. p. 230. Citado na página 3.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE software*, IEEE, v. 7, n. 1, p. 13–17, 1990. Citado 3 vezes nas páginas 19, 38 e 79.

CLELAND, D. L.; IRELAND, L. R. *Project management*. [S.l.]: McGraw-Hill Professional, 2006. Citado na página 1.

COCKBURN, A. Writing effective use cases, the crystal collection for software professionals. *Addison-Wesley Professional Reading*, 2000. Citado 2 vezes nas páginas 13 e 9.

COHEN, J. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, Sage Publications, v. 20, n. 1, p. 37–46, 1960. Citado na página 108.

COMMITTEE, I. C. S. S. E. T. Ieee standard glossary of software engineering terminology. In: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. [S.l.], 1983. Citado 2 vezes nas páginas 1 e 79.

COSTA, D. A. da et al. Unveiling developers contributions behind code commits: an exploratory study. In: ACM. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. [S.l.], 2014. p. 1152–1157. Citado na página 131.

D'AMBROS, M.; LANZA, M.; GALL, H. Fractal figures: Visualizing development effort for cvs entities. In: IEEE. *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*. [S.l.], 2005. p. 1–6. Citado na página 129.

DAVIS, A. et al. Identifying and measuring quality in a software requirements specification. In: IEEE. *Software Metrics Symposium, 1993. Proceedings., First International*. [S.l.], 1993. p. 141–152. Citado na página 12.

DAVIS, J.; GOADRICH, M. The relationship between precision-recall and roc curves. In: ACM. *Proceedings of the 23rd international conference on Machine learning*. [S.l.], 2006. p. 233–240. Citado 2 vezes nas páginas 21 e 22.

DEMEYER, S. et al. Happy birthday! a trend analysis on past msr papers. In: IEEE PRESS. *Proceedings of the 10th Working Conference on Mining Software Repositories*. [S.l.], 2013. p. 353–362. Citado na página 99.

DIT, B. et al. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, Wiley Online Library, v. 25, n. 1, p. 53–95, 2013. Citado 6 vezes nas páginas 3, 5, 10, 28, 35 e 80.

EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating features in source code. *IEEE Transactions on software engineering*, IEEE, v. 29, n. 3, p. 210–224, 2003. Citado 2 vezes nas páginas 3 e 9.

FARAGÓ, C. et al. Impact of version history metrics on maintainability. In: IEEE. *Advanced Software Engineering & Its Applications (ASEA), 2015 8th International Conference on*. [S.l.], 2015. p. 30–35. Citado na página 129.

FARIAS, M. A. de F. et al. A systematic mapping study on mining software repositories. In: ACM. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. [S.l.], 2016. p. 1472–1479. Citado 2 vezes nas páginas 99 e 102.

FILHO, I. B.; AQUINO, G.; ROSA, J. G. S. Sigaa mobile–o caso de sucesso da ferramenta de gestão acadêmica na era da computação móvel. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2013. v. 24, n. 1, p. 92. Citado na página 69.

FOUCAULT, M.; FALLERI, J.-R.; BLANC, X. Code ownership in open-source software. In: ACM. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.], 2014. p. 39. Citado na página 131.

FOUCAULT, M. et al. On the usefulness of ownership metrics in open-source software projects. *Information and Software Technology*, Elsevier, v. 64, p. 102–112, 2015. Citado na página 131.

FRANTZESKOU, G. et al. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software*, Elsevier, v. 81, n. 3, p. 447–460, 2008. Citado na página 130.

FRANTZESKOU, G. et al. Effective identification of source code authors using byte-level information. In: ACM. *Proceedings of the 28th international conference on Software engineering*. [S.l.], 2006. p. 893–896. Citado na página 130.

FRANTZESKOU, G. et al. Source code author identification based on n-gram author profiles. *Artificial Intelligence Applications and Innovations*, Springer, p. 508–515, 2006. Citado 2 vezes nas páginas 3 e 131.

FREIRE, A.; SILVEIRA, P. Vraptor - simple and quick web framework. In: *Anais do 5º Workshop sobre Software Livre, Porto Alegre*. [S.l.: s.n.], 2004. p. 39–42. Citado na página 66.

FRITZ, T.; MURPHY, G. C.; HILL, E. Does a programmer's activity indicate knowledge of code? In: ACM. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.], 2007. p. 341–350. Citado 2 vezes nas páginas 26 e 131.

FRITZ, T. et al. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 23, n. 2, p. 14, 2014. Citado 6 vezes nas páginas 3, 26, 27, 100, 103 e 129.

FRITZ, T. et al. A degree-of-knowledge model to capture source code familiarity. In: ACM. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. [S.l.], 2010. p. 385–394. Citado 3 vezes nas páginas 3, 26 e 129.

FRONZA, I.; TREBESCHI, S. Overview: Ownership visualization word cloud. *ICEIS 2016*, p. 405, 2016. Citado 2 vezes nas páginas 100 e 129.

GALL, H. C.; LANZA, M. Software evolution: analysis and visualization. In: ACM. *Proceedings of the 28th international conference on Software engineering*. [S.l.], 2006. p. 1055–1056. Citado na página 19.

GIGER, E.; PINZGER, M.; GALL, H. Using the gini coefficient for bug prediction in eclipse. In: ACM. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. [S.l.], 2011. p. 51–55. Citado na página 131.

GIRBA, T. et al. How developers drive software evolution. In: IEEE. *Principles of Software Evolution, Eighth International Workshop on*. [S.l.], 2005. p. 113–122. Citado 2 vezes nas páginas 27 e 129.

GOSLING, J.; MCGILTON, H. The java language environment. *Sun Microsystems Computer Company*, v. 2550, 1995. Citado 3 vezes nas páginas 38, 82 e 119.

GRAVES, T. L. et al. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, IEEE, v. 26, n. 7, p. 653–661, 2000. Citado 2 vezes nas páginas 18 e 19.

GRAY, A.; SALLIS, P.; MACDONELL, S. Identified (integrated dictionary-based extraction of non-language-dependent token information for forensic identification, examination, and discrimination): A dictionary-based system for extracting source code metrics for software forensics. In: IEEE. *Software Engineering: Education & Practice, 1998. Proceedings. 1998 International Conference*. [S.l.], 1998. p. 252–259. Citado na página 130.

GREEVY, O.; DUCASSE, S. Correlating features and code using a compact two-sided trace analysis approach. In: IEEE. *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. [S.l.], 2005. p. 314–323. Citado na página 26.

GREEVY, O.; GIRBA, T.; DUCASSE, S. How developers develop features. In: IEEE. *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*. [S.l.], 2007. p. 265–274. Citado 4 vezes nas páginas 26, 27, 118 e 129.

GREILER, M.; HERZIG, K.; CZERWONKA, J. Code ownership and software quality: a replication study. In: IEEE PRESS. *Proceedings of the 12th Working Conference on Mining Software Repositories*. [S.l.], 2015. p. 2–12. Citado na página 130.

HARJANI, D.-R.; QUEILLE, J.-P. A process model for the maintenance of large space systems software. In: IEEE. *Software Maintenance, 1992. Proceerdings., Conference on*. [S.l.], 1992. p. 127–136. Citado na página 2.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *Frontiers of Software Maintenance, 2008. FoSM 2008.* [S.l.], 2008. p. 48–57. Citado 5 vezes nas páginas 17, 18, 60, 79 e 121.

HASSAN, A. E. Predicting faults using the complexity of code changes. In: IEEE COMPUTER SOCIETY. *Proceedings of the 31st International Conference on Software Engineering*. [S.l.], 2009. p. 78–88. Citado 2 vezes nas páginas 18 e 19.

HASSAN, A. E.; HOLT, R. C. The top ten list: Dynamic fault prediction. In: IEEE. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.* [S.l.], 2005. p. 263–272. Citado 2 vezes nas páginas 18 e 19.

HATA, H.; MIZUNO, O.; KIKUNO, T. Bug prediction based on fine-grained module histories. In: IEEE PRESS. *Proceedings of the 34th International Conference on Software Engineering*. [S.l.], 2012. p. 200–210. Citado 2 vezes nas páginas 18 e 19.

HATTORI, L.; LANZA, M. Mining the history of synchronous changes to refine code ownership. In: IEEE. *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. [S.l.], 2009. p. 141–150. Citado 2 vezes nas páginas 3 e 130.

HATTORI, L. P.; LANZA, M.; ROBBES, R. Refining code ownership with synchronous changes. *Empirical Software Engineering*, Springer, v. 17, n. 4-5, p. 467–499, 2012. Citado na página 131.

HEMMATI, H. et al. The msr cookbook: Mining a decade of research. In: IEEE. *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. [S.l.], 2013. p. 343–352. Citado 3 vezes nas páginas 17, 18 e 99.

HERZIG, K.; ZELLER, A. Mining your own evidence. *Making Software: What Really Works and Why We Believe It. O'Reilly Media*, 2010. Citado na página 17.

HIGGINS, J. P.; GREEN, S. et al. *Cochrane handbook for systematic reviews of interventions*. [S.l.]: Wiley Online Library, 2008. v. 5. Citado 3 vezes nas páginas 59, 99 e 100.

HILL, E. et al. Which feature location technique is better? In: IEEE. *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.], 2013. p. 408–411. Citado na página 10.

HOSSEN, M. K.; KAGDI, H.; POSHYVANYK, D. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In: ACM. *Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.], 2014. p. 130–141. Citado na página 129.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. [S.l.]: Pearson Education, 2010. Citado na página 14.

JAEGER, T.; METZGER, A. *Open-source-software*. [S.l.]: Beck, 2002. Citado na página 18.

JANÁK, J. Issue tracking systems. *Brno, spring*, 2009. Citado na página 17.

JONES, C. C. *Software quality: Analysis and guidelines for success*. [S.l.]: Thomson Learning, 1997. Citado na página 120.

KAVALER, D.; FILKOV, V. Stochastic actor-oriented modeling for studying homophily and social influence in oss projects. *Empirical Software Engineering*, Springer, v. 22, n. 1, p. 407–435, 2017. Citado na página 131.

KEELE, S. Guidelines for performing systematic literature reviews in software engineering. In: *Technical report (EBSE)*. [S.l.: s.n.], 2007. Citado 3 vezes nas páginas 59, 99 e 100.

KERNIGHAN, B. W. et al. *The C programming language*. [S.l.]: prentice-Hall Englewood Cliffs, 1988. v. 2. Citado na página 119.

KEYES, J. *Software configuration management*. [S.l.]: CRC Press, 2004. Citado na página 12.

KILGOUR, R. et al. A fuzzy logic approach to computer software source code authorship analysis. Springer-Verlag, 1998. Citado na página 130.

KIM, M. et al. An empirical study of code clone genealogies. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 2005. v. 30, n. 5, p. 187–196. Citado na página 18.

KIM, S. et al. Predicting faults from cached history. In: IEEE COMPUTER SOCIETY. *Proceedings of the 29th international conference on Software Engineering*. [S.l.], 2007. p. 489–498. Citado 2 vezes nas páginas 18 e 19.

KIM, W.; LOCHOVSKY, F. H. *Object-oriented concepts, databases, and applications*. [S.l.]: ACM Press/Addison-Wesley Publishing Co., 1989. Citado na página 39.

KITCHENHAM, B. Procedures for performing systematic reviews. *Keele, UK, Keele University*, v. 33, n. 2004, p. 1–26, 2004. Citado 8 vezes nas páginas 6, 25, 59, 79, 80, 84, 99 e 100.

KOCH, S. Software evolution in open source projects?a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 19, n. 6, p. 361–382, 2007. Citado na página 19.

KOTHARI, J. et al. A probabilistic approach to source code authorship identification. In: IEEE. *Information Technology, 2007. ITNG'07. Fourth International Conference on*. [S.l.], 2007. p. 243–248. Citado na página 130.

KRASNER, G. E.; POPE, S. T. et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, v. 1, n. 3, p. 26–49, 1988. Citado 2 vezes nas páginas 5 e 36.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, IEEE, v. 68, n. 9, p. 1060–1076, 1980. Citado na página 1.

LIENTZ, B. P.; SWANSON, E. B. Discovering issues in software maintenance. *Data management*, DATA PROCESS MANAGEMENT ASSN 505 BUSSE HIGHWAY, PK RIDGE, IL 60068-3191, v. 16, n. 10, p. 15, 1978. Citado 2 vezes nas páginas 1 e 2.

LINARES-VÁSQUEZ, M. et al. Triaging incoming change requests: Bug or commit history, or code authorship? In: IEEE. *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. [S.l.], 2012. p. 451–460. Citado 2 vezes nas páginas 99 e 130.

LIRA, W. A. L. *Um método para inferência da familiaridade de código em projetos de software*. Dissertação (Mestrado) — Universidade Federal do Piauí, Teresina, 9 2016. Citado 23 vezes nas páginas 3, 4, 5, 14, 25, 27, 28, 29, 30, 31, 34, 39, 51, 55, 56, 59, 68, 70, 76, 79, 80, 81 e 107.

LOELIGER, J.; MCCULLOUGH, M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado na página 102.

MAGNUSSON, B.; ASKLUND, U. Fine grained version control of configurations in coop/orm. *Software Configuration Management*, Springer, p. 31–48, 1996. Citado na página 13.

MARTIN, J.; MCCLURE, C. *Software maintenance: the problem and its solutions*. [S.l.]: Prentice-Hall, 1983. Citado na página 2.

MASON, M. *Pragmatic Version Control Using Subversion*. [S.l.]: Pragmatic Bookshelf, 2005. Citado na página 12.

MENG, X. Fine-grained binary code authorship identification. In: ACM. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2016. p. 1097–1099. Citado na página 131.

MENG, X. et al. Mining software repositories for accurate authorship. In: IEEE. *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.], 2013. p. 250–259. Citado 5 vezes nas páginas 3, 26, 27, 100 e 129.

MOCKUS, A. Succession: Measuring transfer of code and developer productivity. In: IEEE COMPUTER SOCIETY. *Proceedings of the 31st International Conference on Software Engineering*. [S.l.], 2009. p. 67–77. Citado na página 132.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. A case study of open source software development: the apache server. In: IEEE. *Software Engineering, 2000. Proceedings of the 2000 International Conference on*. [S.l.], 2000. p. 263–272. Citado na página 131.

MOSER, R. et al. A model to identify refactoring effort during maintenance by mining source code repositories. In: SPRINGER. *International Conference on Product Focused Software Process Improvement*. [S.l.], 2008. p. 360–370. Citado na página 19.

MÜLLER, C.; REINA, G.; ERTL, T. In-situ visualisation of fractional code ownership over time. In: ACM. *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*. [S.l.], 2015. p. 13–20. Citado na página 132.

MÜLLER, H. A. et al. Reverse engineering: A roadmap. In: ACM. *Proceedings of the Conference on the Future of Software Engineering*. [S.l.], 2000. p. 47–60. Citado na página 19.

MÜLLER, H. A. et al. A reverse engineering environment based on spatial and visual software interconnection models. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1992. v. 17, n. 5, p. 88–98. Citado 2 vezes nas páginas 19 e 79.

MUNNS, A.; BJEIRMI, B. F. The role of project management in achieving project success. *International journal of project management*, Elsevier, v. 14, n. 2, p. 81–87, 1996. Citado na página 1.

ORRÚ, M.; MARCHESI, M. A case study on the relationship between code ownership and refactoring activities in a java software system. In: ACM. *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*. [S.l.], 2016. p. 43–49. Citado na página 131.

OTTE, S. Version control systems. *Computer Systems and Telematics, Institute of Computer Science, Freie Universität, Berlin, Germany*, 2009. Citado 7 vezes nas páginas 12, 14, 15, 16, 17, 60 e 102.

PENTA, M. D.; GERMAN, D. M. Who are source code contributors and how do they change? In: IEEE. *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. [S.l.], 2009. p. 11–20. Citado na página 132.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: SN. *12th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.], 2008. v. 17. Citado na página 100.

PFENNING, F.; ELLIOTT, C. Higher-order abstract syntax. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1988. v. 23, n. 7, p. 199–208. Citado 2 vezes nas páginas 37 e 40.

PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. *Version control with subversion*. [S.l.]: "O'Reilly Media, Inc.", 2008. Citado na página 102.

PITOURA, E.; BUKHRES, O.; ELMAGARMID, A. Object orientation in multidatabase systems. *ACM Computing Surveys (CSUR)*, ACM, v. 27, n. 2, p. 141–195, 1995. Citado na página 39.

PLATFORM, E. *Technical Overview, 2003*. 2012. Citado na página 40.

PONCIN, W.; SEREBRENIK, A.; BRAND, M. V. D. Process mining software repositories. In: IEEE. *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. [S.l.], 2011. p. 5–14. Citado na página 18.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. Bioinfo Publications, 2011. Citado 3 vezes nas páginas 21, 22 e 67.

PRAJAPATI, H. B.; DABHI, V. K. High quality web-application development on java ee platform. In: IEEE. *Advance Computing Conference, 2009. IACC 2009. IEEE International*. [S.l.], 2009. p. 1664–1669. Citado na página 39.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016. Citado na página 1.

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave Macmillan, 2005. Citado 2 vezes nas páginas 1 e 17.

RAHMAN, F.; DEVANBU, P. Ownership, experience and defects: a fine-grained study of authorship. In: ACM. *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.], 2011. p. 491–500. Citado na página 130.

RAJLICH, V.; GOSAVI, P. Incremental change in object-oriented programming. *IEEE software*, IEEE, v. 21, n. 4, p. 62–69, 2004. Citado na página 10.

RAJLICH, V.; WILDE, N. The role of concepts in program comprehension. In: IEEE. *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. [S.l.], 2002. p. 271–278. Citado na página 10.

RICCA, F.; MARCHETTO, A. Are heroes common in floss projects? In: ACM. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2010. p. 55. Citado 2 vezes nas páginas 20 e 129.

RICCA, F.; MARCHETTO, A. Heroes in floss projects: an explorative study. In: IEEE. *Reverse Engineering (WCRE), 2010 17th Working Conference on*. [S.l.], 2010. p. 155–159. Citado na página 129.

RICCA, F.; MARCHETTO, A.; TORCHIANO, M. On the difficulty of computing the truck factor. In: SPRINGER. *PROFES*. [S.l.], 2011. p. 337–351. Citado 5 vezes nas páginas 20, 71, 104, 120 e 131.

RIGBY, P. C. et al. Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. In: ACM. *Proceedings of the 38th International Conference on Software Engineering*. [S.l.], 2016. p. 1006–1016. Citado na página 131.

ROBILLARD, M. P.; MURPHY, G. C. Concern graphs: finding and describing concerns using structural program dependencies. In: ACM. *Proceedings of the 24th international conference on Software engineering*. [S.l.], 2002. p. 406–416. Citado na página 28.

ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 16, n. 1, p. 3, 2007. Citado 2 vezes nas páginas 3 e 28.

ROBLES, G. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In: IEEE. *7th IEEE Working Conference on Mining Software Repositories (MSR), 2010*. [S.l.], 2010. p. 171–180. Citado 2 vezes nas páginas 17 e 18.

ROBLES, G. et al. Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack. In: ACM. *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.], 2014. p. 222–231. Citado na página 19.

ROCHKIND, M. J. The source code control system. *IEEE Transactions on Software Engineering*, IEEE, n. 4, p. 364–370, 1975. Citado na página 12.

RODRIGUEZ-BUSTOS, C.; APONTE, J. How distributed version control systems impact open source software projects. In: IEEE PRESS. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. [S.l.], 2012. p. 36–39. Citado na página 15.

SAGER, T. et al. Detecting similar java classes using tree algorithms. In: ACM. *Proceedings of the 2006 international workshop on Mining software repositories*. [S.l.], 2006. p. 65–71. Citado na página 18.

SANKOFF, D.; KRUSKAL, J. B. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B.*, v. 1, 1983. Citado na página 29.

SCOTT, J. A.; NISSE, D. Software configuration management. *SWEBOK*, p. 103, 2001. Citado na página 12.

SHAW, M. Writing good software engineering research papers. In: IEEE. *Proceedings of the 25th International Conference on Software Engineering, 2003.* [S.l.], 2003. p. 726–736. Citado 2 vezes nas páginas 110 e 114.

SHEVERTALOV, M. et al. On the use of discretized source code metrics for author identification. In: IEEE. *Search Based Software Engineering, 2009 1st International Symposium on*. [S.l.], 2009. p. 69–78. Citado na página 130.

SOKOLOVA, M.; LAPALME, G. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, Elsevier, v. 45, n. 4, p. 427–437, 2009. Citado na página 21.

SPINELLIS, D. Version control systems. *Software, IEEE*, IEEE, v. 22, n. 5, p. 108–109, 2005. Citado 6 vezes nas páginas 12, 14, 15, 16, 17 e 36.

STROUSTRUP, B. *The C++ programming language*. [S.l.]: Pearson Education India, 1995. Citado na página 119.

SUN, X. et al. Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, Elsevier, v. 66, p. 1–12, 2015. Citado na página 17.

SUSSMAN, B. C.; BRIAN, W. F.; PILATO, C. M. Version control with subversion: for subversion 1.5. *TBA, California*, 2007. Citado 3 vezes nas páginas 14, 15 e 39.

TANGSRIPAIROJ, S.; SAMADZADEH, M. H. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In: ACM. *Proceedings of the 2005 ACM symposium on Applied computing*. [S.l.], 2005. p. 1539–1545. Citado na página 19.

TAYLOR, Q. C. et al. An analysis of author contribution patterns in eclipse foundation project source code. In: SPRINGER. *IFIP International Conference on Open Source Systems*. [S.l.], 2011. p. 269–281. Citado na página 129.

TENNYSON, M. F.; MITROPOULOS, F. J. A bayesian ensemble classifier for source code authorship attribution. In: SPRINGER. *International Conference on Similarity Search and Applications*. [S.l.], 2014. p. 265–276. Citado na página 129.

TENNYSON, M. F.; MITROPOULOS, F. J. Choosing a profile length in the scap method of source code authorship attribution. In: IEEE. *SOUTHEASTCON 2014, IEEE*. [S.l.], 2014. p. 1–6. Citado na página 130.

THONGTANUNAM, P. et al. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: IEEE. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. [S.l.], 2016. p. 1039–1050. Citado na página 131.

TICHY, W. F. Design, implementation, and evaluation of a revision control system. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 6th international conference on Software engineering*. [S.l.], 1982. p. 58–67. Citado na página 12.

TICHY, W. F. Rcs - a system for version control. *Software: Practice and Experience*, Wiley Online Library, v. 15, n. 7, p. 637–654, 1985. Citado na página 12.

TILLEY, S. R. Domain-retargetable reverse engineering. iii. layered modeling. In: IEEE. *Proceedings of the International Conference on Software Maintenance, 1995.* [S.l.], 1995. p. 52–61. Citado na página 19.

TILLEY, S. R. The canonical activities of reverse engineering. *Annals of Software Engineering*, Springer, v. 9, n. 1-2, p. 249–271, 2000. Citado 2 vezes nas páginas 19 e 20.

VERBEEK, H. et al. Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, v. 615, p. 34–39, 2010. Citado na página 18.

VESPERMAN, J. *Essential CVS: Version Control and Source Code Management*. [S.l.]: "O'Reilly Media, Inc.", 2006. Citado na página 12.

VIERA, A. J.; GARRETT, J. M. et al. Understanding interobserver agreement: the kappa statistic. *Fam Med*, v. 37, n. 5, p. 360–363, 2005. Citado na página 108.

WEST, D. B. et al. *Introduction to graph theory*. [S.l.]: Prentice hall Upper Saddle River, 2001. v. 2. Citado 2 vezes nas páginas 5 e 37.

WIERINGA, R. et al. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, Springer, v. 11, n. 1, p. 102–107, 2006. Citado 2 vezes nas páginas 110 e 114.

WILLIAMS, L.; KESSLER, R. *Pair programming illuminated*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002. Citado 2 vezes nas páginas 20 e 103.

WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012. Citado 3 vezes nas páginas 57, 75 e 125.

XIE, T.; PEI, J. Mapo: Mining api usages from open source repositories. In: ACM. *Proceedings of the 2006 International Workshop on Mining Software Repositories*. [S.l.], 2006. p. 54–57. Citado na página 19.

YAMASHITA, K. et al. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In: ACM. *Proceedings of the 14th International Workshop on Principles of Software Evolution*. [S.l.], 2015. p. 46–55. Citado na página 129.

YAN, Y.; MENARINI, M.; GRISWOLD, W. Mining software contracts for software evolution. In: IEEE. *IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014*. [S.l.], 2014. p. 471–475. Citado na página 19.

YING, H. et al. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In: ACM. *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*. [S.l.], 2016. p. 29–35. Citado 3 vezes nas páginas 100, 116 e 131.

YU, L. Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 18, n. 5, p. 311–332, 2006. Citado na página 19.

ZANJANI, M. B.; KAGDI, H.; BIRD, C. Using developer-interaction trails to triage change requests. In: IEEE PRESS. *Proceedings of the 12th Working Conference on Mining Software Repositories*. [S.l.], 2015. p. 88–98. Citado na página 132.

ZANJANI, M. B.; KAGDI, H.; BIRD, C. Using developer-interaction trails to triage change requests. In: IEEE PRESS. *Proceedings of the 12th Working Conference on Mining Software Repositories*. [S.l.], 2015. p. 88–98. Citado na página 132.

ZAZWORKA, N. et al. Are developers complying with the process: an xp study. In: ACM. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2010. p. 14. Citado na página 20.

ZIMMERMANN, T. et al. Improving bug tracking systems. In: IEEE. *31st International Conference on Software Engineering-Companion Volume, 2009. ICSE-Companion 2009.* [S.l.], 2009. p. 247–250. Citado na página 17.

Apêndices

APÊNDICE A – Mapeamento Sistemático da Literatura

Muitos estudos já foram realizados com o objetivo de entender ou inferir a relação entre desenvolvedores de software e código fonte. Essas informações podem ser utilizadas para apoiar na resolução de diversos problemas ligados à Engenharia de Software, especialmente, em uma de suas principais etapas: a manutenção. LINARES-VÁSQUEZ et al. (2012), por exemplo, utilizam técnicas de mineração de repositórios de software para detectar autoria de código e utilizar essa informação para recomendação de desenvolvedores para solução de problemas (*bugs*).

Contudo, apesar da grande importância de estudos que utilizam ou aplicam técnicas de mineração de repositórios para inferir a relação entre desenvolvedor e código fonte, não há um trabalho sistemático (Revisão Sistemática ou Mapeamento Sistemático da Literatura) que reúna evidências sobre tais estudos, apresentando suas principais características e oferecendo um mapa completo dessa linha de pesquisa. Alguns estudos secundários, como os apresentados por FARIAS et al. (2016), HEMMATI et al. (2013) e DEMEYER et al. (2013), reúnem estudos primários ligados à área de MSR. Esses estudos, no entanto, reúnem somente trabalhos publicados no principal evento anual da área, conhecida por *Working Conference on Mining Software Repositories* (MSRConf)[1]. Apenas FARIAS et al. (2016) apresentam um estudo realizado de forma sistemática. Todavia, nenhum desses estudos secundários citados focam em mapear estudos primários que utilizam técnicas de MSR para inferir a relação entre desenvolvedor e código fonte.

Com isso, neste apêndice é apresentado detalhadamente um Mapeamento Sistemático da Literatura (MSL) (KITCHENHAM, 2004; KEELE, 2007; HIGGINS; GREEN et al., 2008) realizado neste trabalho e que fornece uma visão geral da área em que este estudo está inserido, apresentado os principais trabalhos que utilizam métodos baseados em Mineração de Repositórios de Software (MRS) para inferir a relação de desenvolvedores com o código ou software desenvolvido.

A.1 Metodologia

Um Mapeamento Sistemático da Literatura (MSL) é um método científico capaz de identificar, interpretar e sumarizar os trabalhos relevantes para determinada linha de pesquisa, área ou fenômeno de interesse de forma não tendenciosa e replicável (KITCHE-

[1] http://msrconf.org/.

NHAM, 2004). O método surgiu na área médica, para que pesquisadores fossem capazes de se manterem atualizados, acompanhando a evolução de estudos realizados na área ao longo do tempo (HIGGINS; GREEN et al., 2008). Contudo, por ser um método que provê grandes benefícios para a ciência, ele foi adaptado para outras áreas de pesquisa, incluindo a Engenharia de Software (PETERSEN et al., 2008).

A realização de mapeamentos sistemáticos engloba diversos objetivos, dentre eles pode-se destacar: a necessidade de sumarizar a evidência empírica sobre determinado tratamento ou tecnologia; identificar "lacunas" em uma determinada área de pesquisa ou sugerir pontos de futura investigação (KITCHENHAM, 2004). Por conta disso, é possível utilizar essa metodologia de investigação para analisar os trabalhos disponíveis sobre um tema de interesse e, assim, melhorar a fundamentação de novas soluções.

Figura 22 – Etapas do processo de MSL.

Fonte – Autor.

Neste trabalho foi conduzido um mapeamento sistemático para analisar o estado da arte na área de Mineração de Repositórios de Software (MSR) com foco na inferência da relação entre desenvolvedor e código fonte. A pesquisa foi realizada seguindo diretrizes propostas por KEELE (2007), KITCHENHAM (2004), e parcialmente o processo de mapeamento fornecido por PETERSEN et al. (2008). O processo de mapeamento empregado está resumido na Figura 22 e descrito mais adiante nas Subseções A.3 à A.3.4.

A.2 Relações entre Desenvolvedor e Código Fonte

Este trabalho tem como principal objetivo inferir a **familiaridade** de código que desenvolvedores de software possuem sobre um determinado sistema. O termo "familiaridade" tratado neste trabalho, refere-se a quão um determinado desenvolvedor é "íntimo" em relação ao código fonte de um software. Ou seja, a familiaridade trata-se de uma relação entre o desenvolvedor e código fonte.

Contudo, outros tipos dessa relação podem ser encontrados na literatura. FRITZ et al. (2014) apresenta um estudo sobre a relação de **knowledge** (conhecimento), que se refere à quão desenvolvedores conhecem partes específicas de código. YING et al. (2016) realizou um estudo no qual apresenta um outro tipo de relação: **expertise** (especialidade). MENG et al. (2013) trata da relação de **authorship** (autoria). FRONZA; TREBESCHI (2016) apresenta um estudo em que aborda a relação de **ownership** (propriedade). Apesar

de serem termos distintos, esses tipos de relação entre desenvolvedor e código fonte se referem, de maneira geral, à associação entre cada desenvolvedor com o código de um determinado sistema de software. Além disso, as relações de familiaridade e conhecimento geralmente são determinadas com base nos demais tipos de relações citadas.

Nesse contexto, durante a realização desse MSL, foram considerados estudos que tratassem de um ou mais tipos de relação entre desenvolvedor e código fonte discutidos anteriormente, incluindo o tipo abordado neste trabalho.

A.3 Questões de Pesquisa

As questões de pesquisa foram definidas em função dos objetivos deste trabalho. Essas questões são expostas a seguir:

- Quais relações entre desenvolvedor e código fonte são inferidas? Familiaridade, conhecimento, autoria, propriedade, especialidade, contribuição de código?

  a) Em que se baseiam as métricas utilizadas no processo de inferência?

  b) Qual a granularidade (linhas de código, arquivos, módulos ou funcionalidades) da relação inferida entre desenvolvedor e código fonte?

  c) Existem restrições em relação a linguagem de programação?

- **QP2**: O estudo busca solucionar algum problema ligado ao processo de desenvolvimento de software, a partir da inferência da relação entre desenvolvedor e código fonte? Qual?

- **QP3**: Quais tipos de repositórios de software foram utilizados no processo de mineração?

  a) Qual o repositório de software?

  b) Os dados obtidos a partir do repositório de software são públicos ou privados?

  c) Qual o objeto de análise obtido (extraído) a partir do repositório?

- **QP4**: Qual mecanismo utilizado para o processo de extração de dados de repositórios?

- **QP5**: Qual o método de avaliação (survey, estudo de caso, estudo exploratório, comparação com outros estudos, etc.) utilizado por cada estudo?

  a) Quais foram os objetos de avaliação?

  b) Foram discutidas ameaças à avaliação?

A questão de pesquisa **QP1** foi definida com o intuito de identificar qual a relação entre desenvolvedor e código fonte o estudo visa inferir: familiaridade, conhecimento, propriedade, autoria ou contribuição/esforço de código. A subquestão **QP1(a)** visa obter informações das métricas utilizadas, mais especificamente do quê é considerado no cálculo dessas métricas, por exemplo: inserção ou modificações de código realizadas por desenvolvedores no decorrer do desenvolvimento de software. A subquestão **QP1(b)** tem como objetivo identificar qual a granularidade da relação inferida entre desenvolvedor e código fonte, ou seja, se essa relação é dada considerando linhas de código, arquivos, módulos ou requisitos funcionais. Já a subquestão **QP1(c)** tem por objetivo identificar se o método utilizado para inferência da relação entre desenvolvedor e código fonte, possui restrições quanto a linguagens de programação.

A segunda questão de pesquisa (**QP2**) foi definida com o intuito de identificar qual o objetivo em inferir a relação entre desenvolvedor e código fonte, no ponto de vista de pesquisadores ou profissionais para o processo de desenvolvimento de software. Essa informação a respeito da relação entre desenvolvedor e código fonte, pode ser utilizada por exemplo, para auxiliar durante a etapa de manutenção de software, mais especificamente na etapa de recomendação de desenvolvedores para solucionar problemas ou adicionar novas funcionalidades ao sistema de software.

A questão de pesquisa **QP3** tem como objetivo identificar quais tipos de repositórios software foram utilizados durante o processo de mineração de dados. A subquestão **QP3(a)** visa identificar qual tipo de repositório utilizado. Exemplos de repositórios de software são *Subversion* (SVN) (PILATO; COLLINS-SUSSMAN; FITZPATRICK, 2008) e *Git* (LOELIGER; MCCULLOUGH, 2012), que são do tipo Sistema de Controle de Versão (OTTE, 2009) (vide Seção 1.3 do Capítulo 1). A subquestão **QP3(b)** tem por objetivo identificar se os dados extraídos são públicos ou privados. Isso porque alguns repositórios de software, possuem tanto versões gratuitas como privadas, limitando o livre acesso a dados armazenados. A subquestão **QP3(c)** foi adaptada de FARIAS et al. (2016) e tem por objetivo identificar quais dados foram extraídos a partir do repositório de código. Arquivos de código fonte são exemplos desses tipos de dados.

A questão de pesquisa **QP4** tem como objetivo identificar também que tipo de artifício foi utilizado para extrair dados de repositórios, ou seja, se foi utilizada alguma ferramenta, *framework*, biblioteca de código, plug-in, etc. A questão de pesquisa **QP5** foi definida com o intuito de identificar quais métodos de avaliação utilizados pelos estudos identificados. Nesse sentido, a subquestão **QP5(a)** foi definida com o objetivo de identificar quais projetos de software tem sido utilizados como objetos de análise pelos estudos, enquanto que a subquestão **QP5(b)** tem como objetivo identificar quais estudos realizaram discussões de possíveis ameaças à validade dos resultados obtidos.

A.3.1 Processo de Busca

O processo de busca de estudos primários possui basicamente 6 (seis) etapas ou atividades. Esse processo é apresentado na Figura 23. Na primeira etapa, foram definidos os termos que seriam incluídos na *string* de pesquisa, que é basicamente um conjunto de termos e ligações entre eles, a ser utilizada para busca dos estudos primárias.

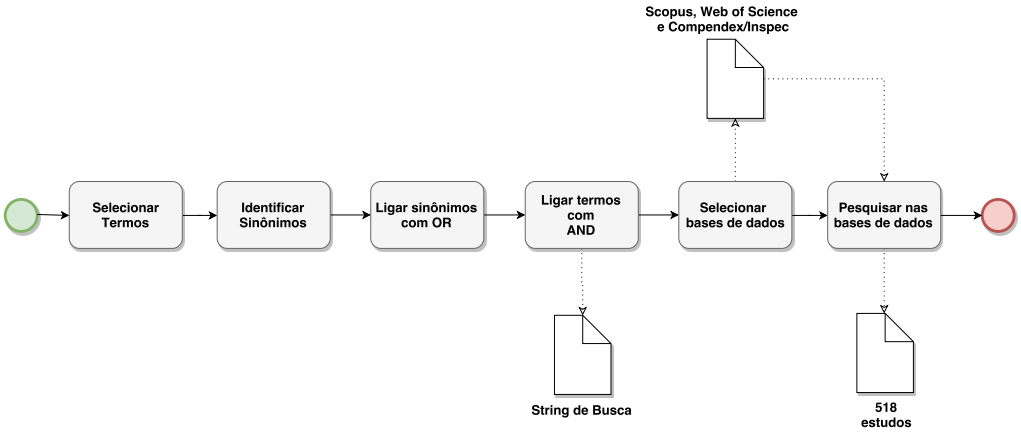


Figura 23 – Etapas do processo de busca.

Fonte – Autor.

Os principais termos incluídos foram aqueles que representam a relação entre desenvolvedor e código fonte, como "familiarity" e os termos apresentados na Seção A.2, que são: "*knowledge*", "*expertise*", "*authorship*", "*ownership*". Cada um desses termos foram precedidos pelo termo "*code*", para que pudessem ser retornados apenas estudos relacionados a desenvolvimento de software. Além disso foram inseridos os termos "*determination*", "*estimate*", "*measure*" e "*calculate*" que se relacionam com os termos anteriores.

Após a identificação dos principais termos para a busca de estudos também foram incluídos algumas palavras que poderiam substituir algum dos termos apresentados anteriormente. Os sinônimos ou palavras relacionadas que foram incluídos na *string* de pesquisa, são apresentados na Tabela 14. Os termos "*code degree of knowledge*" e "*code degree of interest*" foram extraídos com base no estudo realizado por FRITZ et al. (2014). É interessante destacar também o termo "code contribution". Esse termo é bastante relacionado com estudos que visam determinar a relação entre desenvolvedor e código fonte. A tendência é que estudos que visam inferir essa relação, levem em consideração as alterações realizadas por desenvolvedores sobre o código fonte, isto é, a sua contribuição em relação à adição, modificações ou remoções de código.

Outro termo que aparece na Tabela 14, e que pode causar curiosidade, é o termo "*truck factor*" (TF). Esse termo é definido como o número de pessoas em uma equipe de desenvolvimento de software, que têm de ser atingidas por um caminhão antes que o projeto esteja em sérios problemas (WILLIAMS; KESSLER, 2002). Em outras palavras,

Tabela 14 – Termos utilizados na *string* de pesquisa.

Termos principais	**Sinônimos/Termos relacionados**
code familiarity, *code knowledge*, *code expertise*, *code authorship*, *code ownership*	*code degree of knowledge*, *code degree of interest*, *knowledge of code*, *code knowledge*, *code contribution*, *code author*, *code owner*, *truck factor*
determination estimate, *measure*, *calculate*	*characterize*, *understand*, *evaluate*, *find*, *visualize*, *identify*, *discover*, *explore*, *mining*, *extract*, *collect*, *predict*, *forecast*, *assessment*, *analyze*, *examine*, *capture*, *locate*

Fonte – Autor.

isso significa que o *truck factor* indica quantos desenvolvedores possuem maior relação com o código fonte do projeto, ou seja, aqueles que possuem maior "domínio" sobre o software. Segundo RICCA; MARCHETTO; TORCHIANO (2011), obter ou estimar o TF é uma maneira de calcular *ownership*. Por essa razão, esse termo também foi incluído na *string* de pesquisa.

O sinônimos ou termos relacionados foram ligados aos termos principais por meio do conectivo "OR" e posteriormente, as *strings* resultantes foram ligadas por meio do conectivo "AND". A string de pesquisa final é apresentada na Tabela 15. Pode-se perceber que alguns termos foram adaptados e seguidos pelo símbolo de asterisco (*). Isso é justificado pelo fato de que alguns termos podem se apresentar com diferentes variações e com isso poderiam não ser retornados caso fosse especificado apenas uma dessas variações. Com uso do símbolo (*), estudos que contivessem, por exemplo, palavras como *estimat**e*** ou *estimat**ion*** seriam então retornados.

Tabela 15 – Estrutura da *string* final de pesquisa.

String de pesquisa
(*determinat** **OR** *estimat** **OR** *measur** **OR** *calculat** **OR** *character** **OR** *understand** **OR** *evaluat** **OR** *find** **OR** *visual** **OR** *identif** **OR** *discover** **OR** *explor** **OR** *mining* **OR** *extract** **OR** *collect** **OR** *predict** **OR** *forecast** **OR** *assessment* **OR** *analy** **OR** *examin** **OR** *captur** **OR** *locate*) **AND** ("*code familiarity*" **OR** "*code knowledge*" **OR** "*code expertise*" **OR** "*code authorship*" **OR** "*code ownership*" **OR** "*code degree of knowledge*" **OR** "*code degree of interest*" **OR** "*knowledge of code*" **OR** "*code knowledge*" **OR** "*code knowledge*" **OR** "*code contribution*" **OR** "*code contribution*" **OR** "*code author*" **OR** "*code owner*" **OR** "*truck factor*")

Fonte – Autor.

Após a formulação da *string* de pesquisa, foram selecionadas as bases de dados

para busca dos estudos primários. Foram utilizadas as seguintes bases: Scopus[2], Web of Science[3] e Engineering Village[4] (Compendex/Inspec). Essas bases foram escolhidas porque realizam a indexação das bases de dados mais relevantes no que diz respeito à Engenharia de Software, como IEEExplore[5], Springer[6], ACM[7] e Elservier[8]. A *string* foi aplicada para buscas de estudos considerando o texto do título, resumo e palavras-chave. Em todas as bases utilizadas, foram buscados estudos publicados até o início (16 de janeiro) do ano de 2017.

Tabela 16 – Resultado da busca nas bases de dados.

Base de Dados	**Resultados**
Scopus	264
Web Of Science	96
Engineering Village	158
Total	518
Total sem duplicações	282

Fonte – Autor.

Durante a realização de mapeamentos sistemáticos é comum a presença de trabalhos duplicados. Grande parte dos estudos duplicados foram identificados automaticamente por meio de umas das ferramentas utilizadas durante a realização deste mapeamento sistemático, a ferramenta *StArt*[9].

Contudo, alguns estudos duplicados não foram identificados pela ferramenta, já que ela utiliza um algoritmo de detecção *case sensitive*, isto é, estudos cujo o título estivesse com todas as letras em formato minúsculo eram diferenciadas de estudos duplicados cujo o título contivesse letras em formato maiúsculo. Com isso, alguns estudos duplicados não identificados pela ferramenta, foram discriminados manualmente.

A.3.2 Processo de Seleção

O processo de seleção de estudos primários foi composto basicamente por 5 (cinco) etapas ou atividades. Esse processo é apresentado na Figura 24.

Na primeira etapa, foram definidos os critérios de inclusão e exclusão de estudos, em conformidade com os objetivos e questões de pesquisa deste Mapeamento Sistemático. Esses critérios definem quais estudos devem ser incluídos ou excluídos, e são apresentados a seguir:

[2] https://www.scopus.com/.
[3] https://www.webofknowledge.com/.
[4] https://www.engineeringvillage.com/.
[5] http://ieeexplore.ieee.org/.
[6] http://www.springer.com/.
[7] http://dl.acm.org/.
[8] https://www.elsevier.com/.
[9] *StArt WebSite*: <http://lapes.dc.ufscar.br/tools/start_tool>

- **Critérios de Inclusão:**

  1. Apresentam estudo relacionado a mineração de dados de repositórios de software **AND**;
  2. Visam estimar o nível ou grau de familiaridade, conhecimento, autoria, propriedade, especialidade ou contribuição de código de desenvolvedores de software **AND**;
  3. São escritos em inglês **AND**;
  4. São artigos completos, não escritos em forma de abstract, apresentação, ou resumo expandido.

- **Critérios de Exclusão:**

  1. Não apresentam estudo relacionado a mineração de dados de repositórios de software **OR**;
  2. Não visam estimar o nível ou grau de familiaridade, conhecimento, autoria, propriedade, especialidade ou contribuição de código de desenvolvedores de software **OR**;
  3. Não são escritos em inglês **OR**;
  4. Estudos duplicados em duas ou mais bases de dados **OR**;
  5. Estar disponível somente em forma de *abstract*, apresentação, ou resumo expandido;

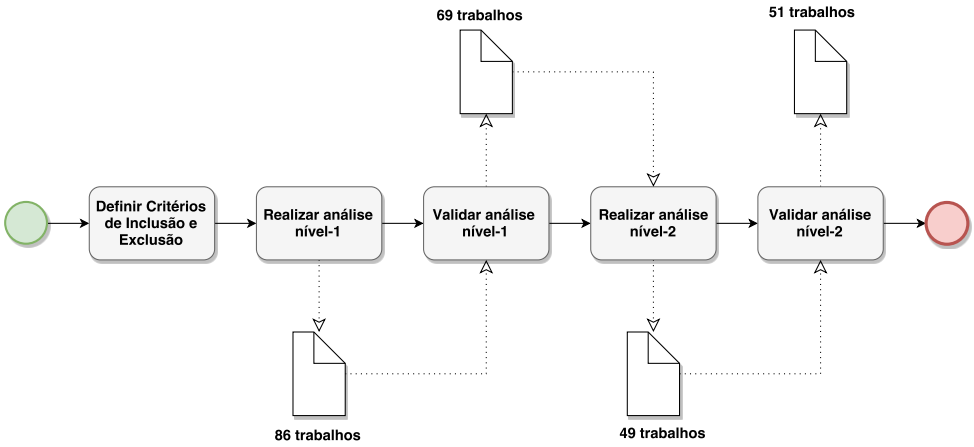


Figura 24 – Etapas do processo de seleção.

Fonte – Autor.

O processo de seleção de estudos primários foi conduzido em duas etapas de análise. Durante a primeira etapa (*nível-1*) dos estudos primários, foram considerados apenas título e resumo dos trabalhos. Na segunda etapa (análise nível-2), foram lidos os trabalhos

por completo. É importante ressaltar que durante as etapas de análise (*nível-1* e *nível-2*) foi aplicada uma abordagem inclusiva, isto é, em caso de trabalhos duvidosos decidiu-se pela inclusão destes.

Após a definição dos critérios de inclusão e exclusão de estudos, e antes de iniciar da primeira etapa de análise (*nível-1*), foi realizado um teste inicial com base na seleção de 10 (dez) estudos escolhidos aleatoriamente dentre os 282 trabalhos que seriam selecionados posteriormente, com o objetivo de avaliar o nível de concordância entre os participantes deste MS, e assim poder definir uma estratégia de seleção dos 282 estudos. Nesta etapa de teste foi utilizada com suporte a ferramenta *TheEnd*[10].

Este mapeamento foi realizado por um grupo de 4 (quatro) pesquisadores (*A*, *B*, *C* e *D*). Isso porque em um MS é fundamental que existam diferentes opiniões envolvidas, a fim de reduzir ao máximo a possibilidade de erros no decorrer da execução desse processo. Dentre os pesquisadores, o pesquisador *A* é aluno de graduação em Ciência da Computação (CC). O Pesquisador *B* e *C* são mestrandos do mesmo curso citado, enquanto o pesquisador *D* possui o grau de mestrado já concluído também no curso de CC, cujo trabalho (LIRA, 2016) está relacionada com a mesma linha de pesquisa abordada neste trabalho.

Como resultado do teste inicial de seleção com os 10 (dez) estudos escolhidos aleatoriamente, os autores *B* e *D*, estavam envolvidos a mais tempo com essa linha de pesquisa e tiverem concordância de 100%, isto é, dos 10 trabalhos, 6(seis) estudos foram aceitos por ambos, enquanto os 4 (quatro) foram rejeitados. Comparado a seleção realizada pelos outros pesquisadores (*A* e *C*), quatro trabalhos tiveram divergências relação aos estudos selecionados pelos pesquisadores *B* e *D*.

Com isso, foi definida a estratégia de divisão dos estudos a serem selecionados dentre dois grupos, cada um sendo composto então por dois pesquisadores. Os pesquisadores que tiveram maior concordância foram separados em grupos diferentes, o Grupo 1 foi formado pelos pesquisadores *B* e *C*, enquanto o Grupo 2 foi formado por *A* e *D*. Com isso, durante a primeira etapa de análise (*nível-1*) os **282 estudos** foram divididos dentre os dois grupos de pesquisadores, com uma porcentagem de 20% de trabalhos em comum entre eles, com o objetivo de aumentar a confiabilidade do processo de seleção de estudos. O processo de divisão dos estudos foi feito automaticamente por meio da ferramenta *StArt*.

Tabela 17 – Divisão de estudos entre os grupos.

Não compartilhados	113
Compartilhados	56
Total por grupo	169
Total de estudos	282

Fonte – Autor.

[10] *TheEnd WebSite*: <http://easii.ufpi.br/theend>

A Tabela 17 apresenta a quantidade de estudos por grupo, incluindo o total de artigos em comum. Os estudos em que houve divergências entre a seleção intragrupo foram reavaliados por cada grupo separadamente, e cada um dos grupos gerou como resultado um conjunto de estudos aceitos e outro contendo os trabalhos que foram rejeitados. Com isso, foi realizada uma comparação intergrupo, considerando os estudos em comum. É importante frisar que os grupos não tinham conhecimento de quais estudos foram compartilhados entre eles. A ferramenta *StArt* se encarregou de informar dentre os estudos em comum quais tiveram divergências intergrupo.

Após a seleção dos grupos, foram aceitos 44 estudos pelo Grupo 1 e 42 pelo Grupo 2, totalizando 86 trabalhos. Porém, 13 desses estudos estavam contidos nos dois grupos, e com isso foram aceitos então um total de **73 trabalhos** após a primeira etapa de análise (*nível-1*). Dentre os 56 trabalhos que foram compartilhados, apenas 5 (cinco) tiveram divergências, com 38 artigos rejeitados e 13 aceitos por ambos os grupos, obtendo um índice *Kappa* (COHEN, 1960) de 0.7771. Quando esse valor se encontra no intervalo [0.6, 0.8], significa que o nível de concordância entre os grupos foi substancial (VIERA; GARRETT et al., 2005). Após discussão entre os dois grupos (etapa de validação *nível-1*), 1 (um) dos estudos em que houve divergência passou para a próxima fase, os 4 (quatro) restantes foram rejeitados. Desconsiderando o total de artigos rejeitados após a reavaliação, e os trabalhos em comum que foram aceitos por ambos os grupos, que foi um total de 13 trabalhos, restaram então **69 estudos** aceitos após a primeira etapa de análise e validação (*nível-1*).

Na segunda etapa de análise (*nível-2*), os 69 trabalhos aceitos durante a primeira etapa foram novamente divididos entre os grupos. Como o nível de concordância entre os grupos foi aceitável, nesta etapa não foram distribuídos artigos em comum entre os grupos. O Grupo 1 ficou responsável pela leitura (separadamente, cada pesquisador do grupo avaliou todos os estudos) de 35 destes trabalhos, enquanto o Grupo 2 avaliou os outros 34 restantes.

Após a segunda etapa de análise, os pesquisadores do Grupo 1 aceitaram, sem divergência de opinião, 24 estudos, 1 (um) dos trabalhos foi reavaliado em conjunto por ambos os pesquisadores do Grupo 1. Como resultado, foi decidido a inclusão do estudo. O Grupo 2 aceitou, sem divergência, 25 trabalhos, 3 (três) estudos geraram divergência e foram reavaliados em conjunto pelos pesquisadores. Como resultado um dos trabalhos foi aceito, resultando em um total de 26 estudos.

Portanto, após a segunda etapa de análise (*nível-2*), **49 estudos** foram aceitos em primeiro momento pelos grupos 1 e 2. Após a validação da segunda etapa de análise, dois estudos foram incluídos, resultando em um total de **51 trabalhos** aceitos. Esses estudos passaram então por um processo de extração de dados, que será apresentado na Seção A.3.3. A Tabela 18 apresenta a quantidade de estudos em que cada critério de exclusão foi

utilizado, durante a etapa de seleção.

Tabela 18 – Critérios para exclusão de trabalhos

Critério	**Resultados**
Trabalhos duplicados	236
Não apresentam estudo relacionado a MSR	137
Não visam estimar a relação entre desenvolvedor e código fonte	230
Não são escritos em inglês	0
Trabalhos não completos (forma resumida)	1

Fonte – Autor.

É importante ressaltar que durante o processo de seleção não foram encontrados estudos escritos em outras linguagens, além do inglês. Isso se deve ao fato de que o processo de busca dos estudos, mais especificamente durante a aplicação da *string* de pesquisa, foi pré-definido (por meio de configurações de busca em cada uma das bases de dados utilizadas) que os tipos de estudos retornados fossem apenas aqueles escritos em inglês. Esse critério de seleção é justificado porque a língua inglesa é a linguagem oficial utilizada mundialmente, portanto, estudos escritos nessa língua possuem uma maior 'visibilidade' no meio científico.

A.3.3 Processo de Extração

O processo de extração de dados é apresentado na Figura 25 e consiste em quatro etapas principais: definir um esquema de classificação, definir um formulário de extração, extrair os dados e validar os dados extraídos.

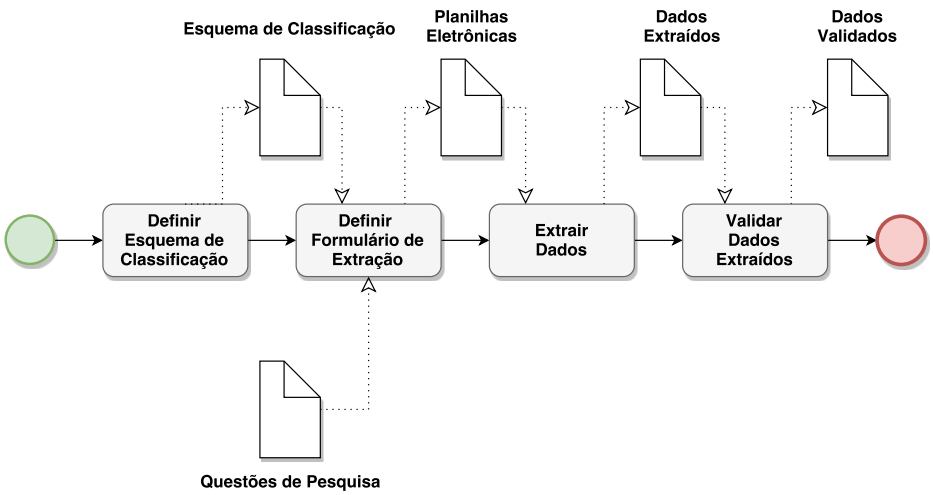


Figura 25 – Etapas do processo de extração.

Fonte – Autor.

Os dados extraídos dos trabalhos selecionados foram organizados em planilhas contendo alguns campos a serem preenchidos, como por exemplo: identificador do estudo, título, ano, tipo de evento, local de publicação e autores. Além disso, foram incluídos campos para as Questões de Pesquisa apresentadas na Seção A.3. Por fim, para a formação do esquema de classificação final, foram extraídas também outras informações por meio da leitura dos trabalhos, são elas:

- **Contribuição**: Esta questão foi adaptada de (SHAW, 2003) e refere-se ao tipo de contribuição proposta pelo trabalho e foi classificada nas seguintes categorias:

  a) **Modelo:** Representação de uma realidade observada por conceitos ou conceitos relacionados depois de um processo de conceptualização;

  b) **Framework/método:** Os modelos relacionaram-se à construção de software ou processos de gerenciamento de desenvolvimento;

  c) **Ferramenta:** A tecnologia, o programa ou a aplicação costumaram criar, depurar, manter ou apoiar processos de desenvolvimento;

  d) **Teoria:** Construção de uma relação causa-efeito relacionadas aos resultados de uma pesquisa;

  e) **Guidelines:** Consiste em uma lista de orientações, obtidas a partir de uma síntese dos resultados de uma pesquisa;

  f) **Lição:** Conjunto de resultados, diretamente analisados dos resultados de pesquisa obtidos;

  g) **Conselho/Implicação:** Recomendações discussivas e genéricas, consideradas a partir de opiniões pessoais.

- **Tipo de Pesquisa:** Esta questão é usada para distinguir os diferentes tipos de estudos primários (adaptada de (WIERINGA et al., 2006)) e foi classificada nas seguintes categorias:

  a) **Validação de Pesquisa:** As técnicas investigadas são novas e ainda não foram implementadas em larga escala industrial. A avaliação realizada é formal e envolve interação humana, mas os indivíduos e o ambiente utilizados são pouco representativos do contexto real. As técnicas utilizadas são, por exemplo, experimentos realizados com estudantes;

  b) **Pesquisa de Avaliação:** As técnicas foram implementadas na prática e avaliadas em larga escala industrial, experimentadas com profissionais ou aplicadas em outro contexto real. Mostra-se como a técnica é implementada na prática (a implementação da solução) e quais são as consequências da implementação em termos de vantagens e desvantagens de avaliação (implementação);

c) **Proposta de Solução:** Pode ser novo ou um aumento significativo de uma técnica existente. Os potenciais benefícios e aplicabilidade da solução é mostrada por um pequeno exemplo, ou em uma boa argumentação;

d) **Trabalhos Filosóficos:** Esses trabalhos esboçam uma nova maneira de olhar as coisas existentes, estruturando o campo na forma de uma taxonomia ou estrutura conceitual;

e) **Trabalhos de Opinião:** Esses trabalhos expressam a opinião pessoal de alguém sob determinada técnica, se é boa ou ruim, ou como as coisas deveriam ter sido feitas. Eles não dependem de metodologias de trabalho e de trabalhos relacionados;

f) **Trabalhos de Experiência:** Explica sobre o que e como algo foi feito na prática. Deve-se relatar a experiência pessoal do autor.

- **Disciplina/Sub-processo da Engenharia de Software:** Disciplinas ou sub-processos da Engenharia de Software em que cada estudo está ligado. Esta questão está classificada em:

a) **Requisitos:** Obtenção do enunciado, completo, claro e preciso dos desejos, necessidades, expectativas e restrições dos clientes em relação ao produto a ser desenvolvido;

b) **Análise:** Modelagem dos conceitos relevantes do domínio do problema, com o intuito de verificar a qualidade dos requisitos obtidos e detalhar tais requisitos em um nível adequado aos desenvolvedores;

c) **Desenho (ou projeto):** Definição de uma estrutura implementável para um produto, que atenda aos requisitos especificados;

d) **Implementação:** Codificação das partes que compõe o software, definidas no desenho, utilizando as tecnologias selecionadas;

e) **Teste:** Verificação dinâmica das partes que constituem o software, utilizando um conjunto finito de casos de teste, selecionados dentro de um domínio potencialmente infinito, contra seu comportamento esperado;

f) **Gestão de projetos:** Atividades ou processos que envolvem procedimentos de gestão de requisitos e planejamento, controle e contratação de projetos;

i) **Gestão de qualidade:** Trata de um conjunto de sub-processos que visam manter a qualidade em projetos e produtos de software;

j) **Engenharia de processo:** Trata da adaptação, da manutenção e do desenvolvimento de processos de software.

O processo de extração de dados foi realizado por 4 (quatro) pesquisadores, os mesmos que participaram da etapa de seleção dos estudos. Os estudos foram divididos entre duplas de pesquisadores. Como foram 51 estudos aceitos, uma das duplas realizou a extração de dados de 25 estudos, enquanto os 26 restantes foram analisados pela outra dupla. Cada uma das duplas realizou a extração de maneira independente. Dessa forma, cada pesquisador realizou a leitura de todos os trabalhos relacionados à sua dupla, armazenando os dados em planilhas eletrônicas contendo as questões definidas nos esquema de classificação.

A imprecisão na extração de dados e os erros de classificação referem-se a possibilidade de extração de informações de um estudo de diferentes maneiras por pesquisadores diferentes. Para aumentar a confiabilidade do processo de extração de dados, foi realizada uma etapa de validação dos dados extraídos, na qual, os resultados obtidos por cada pesquisador foram comparados. Com isso, cada dupla realizou uma discussão para solucionar divergências parciais nos dados extraídos de alguns estudos analisados, e por fim, foi gerada então uma versão final da planilha com os dados extraídos por cada dupla de pesquisadores.

A.3.4 Processo de Análise

No processo de análise foram identificados os principais conceitos de cada estudo primário, que foram organizados em forma de tabelas para permitir comparações entre os estudos e classificar categorias. A análise quantitativa foi baseada na contagem dos estudos primários que foram classificados em cada alternativa das sub-questões de pesquisa e apresentada na forma de gráfico. Por fim, são discutidos os resultados referente a cada uma das questões definidas do esquema de classificação apresentado na Seção A.3.3.

A.3.5 Resultados

Nesta seção são apresentados os resultados do MSL realizado em relação aos dados extraídos dos 51 trabalhos selecionados que satisfazem às questões de pesquisa. Além disso, são discutidos os principais pontos da área a fim de identificar lacunas deixadas pelos trabalhos disponíveis na literatura em relação a cada uma das questões de pesquisa definidas nesse MSL.

É importante ressaltar que nesse MSL é realizada uma análise quantitativa da área. Assim, os resultados são apresentados e discutidos com relação ao número de trabalhos classificados para cada uma das questões de pesquisa. A classificação individual de cada trabalho selecionado está disponível em uma Planilha Eletrônica[11]. No Apêndice B também se encontram as referências para os estudos aceitos neste trabalho.

[11] Resultados do MSL: <https://goo.gl/arfzv4>

A.3.5.1 Resultados Gerais

A Figura 26 apresenta o número de estudos selecionados nesse MSL com objetivo de estimar ou inferir a relação entre desenvolvedor e código fonte. A figura mostra que o número de pesquisas nesta área estancou entre os anos de 2001 e 2004. Apesar disso, as pesquisas na área, a partir de então, continuaram e têm crescido bastante nos últimos anos, principalmente no ano de 2016, onde aconteceu o maior número publicações. Isso sugere que interesse de pesquisadores nesta área de pesquisa cresceu recentemente.

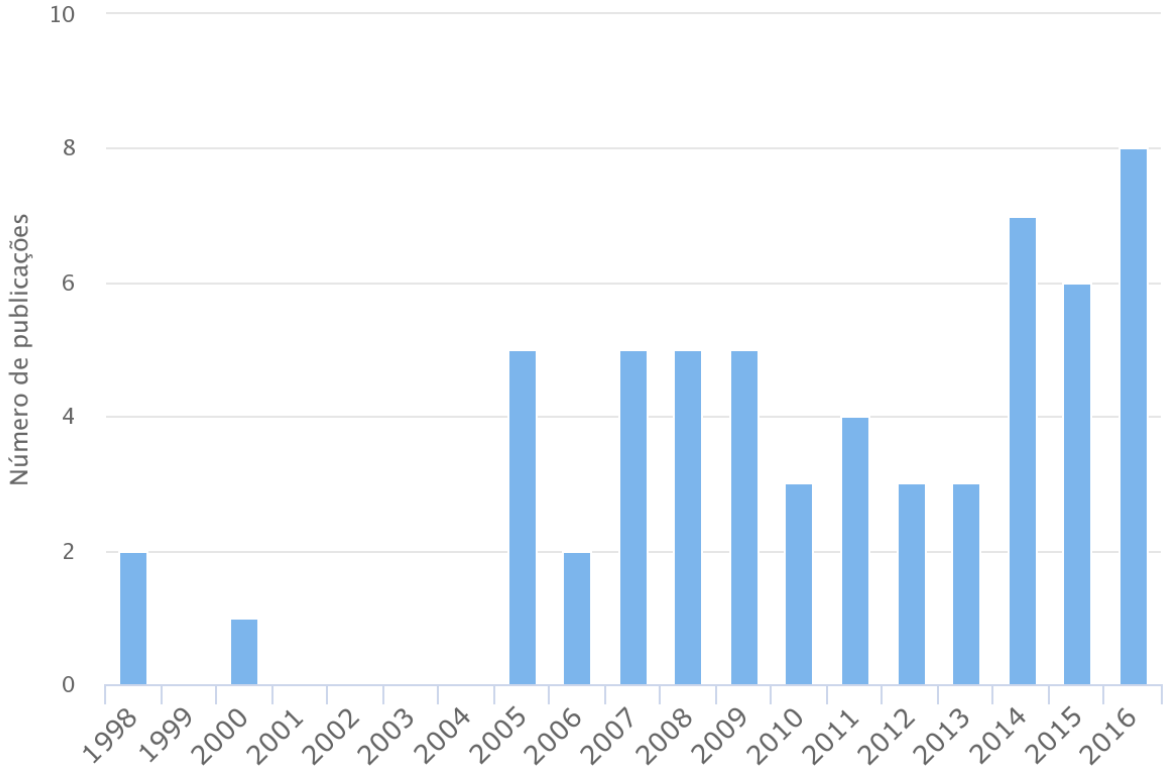


Figura 26 – Distribuição das publicações identificadas entre os anos de 1998 e 2016.

Fonte – Autor.

A Tabela 19 apresenta os principais locais de publicação de pesquisas científicas relacionadas à estimação da relação entre desenvolvedor e código fonte. Dentre os estudos aceitos, 63% foram publicados em conferências, 13% em *journals*, 12% foram publicados em simpósios e outros 12% dos trabalhos foram publicados em workshops.

Tabela 19 – Locais de publicação com trabalhos aceitos no MSL.

Tipo de Evento	**Número de trabalhos**	**Porcentagem**
Conferência	32	63%
Journal	7	13%
Simpósio	6	12%
Workshop	6	12%

Fonte – Autor.

A.3.5.2 Tipos de Pesquisa e Contribuição dos Estudos

Nesta seção são apresentados os resultados correspondentes à questões adaptadas para este estudo. A Figura 27 apresenta os resultados da classificação dos trabalhos quanto ao **Tipo de Pesquisa** (Eixo X) (WIERINGA et al., 2006) e **Contribuição** (Eixo Y) (SHAW, 2003). É importante destacar que não foram identificados trabalhos referentes aos tópicos: **Filosófico**, **Experiência** ou **Opinião**, **Teoria**, ***Guidelines*** e **Conselhos**.

Analisando a Figura 27 pode-se perceber que 65% dos estudos primários (33 trabalhos) foram classificados como **Proposta de Solução** e ***Framework*/Método**, isto é, apresentam uma nova solução para o problema de estimar a relação entre desenvolvedor e código fonte, por meio da criação de um método ou abordagem.

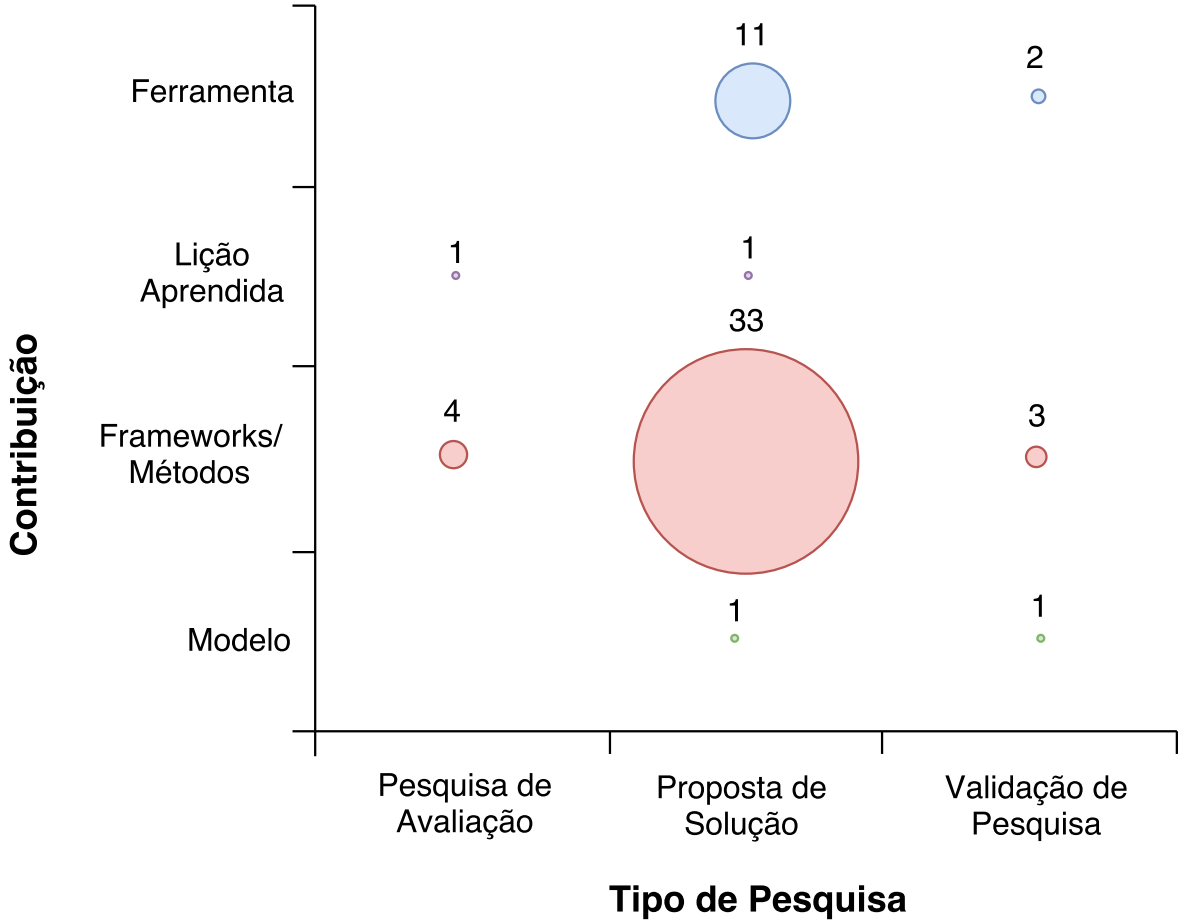


Figura 27 – Contribuição *vs.* Tipo de Pesquisa

Fonte – Autor.

Pode-se perceber também, que apenas 21% dos estudos (11 trabalhos) visam propor uma ferramenta para apoiar no processo de estimação da relação entre desenvolvedor e código fonte. Esse resultado é muito importante, já que a disponibilização de uma ferramenta tende a facilitar que pesquisadores ou mesmos profissionais da área possam utilizar a solução proposta. Os resultados mostram então que poucos estudos se preocuparam com

essa questão.

Além disso, dentre os estudos que apresentam alguma ferramenta, nenhum deles realiza **Pesquisa de Avaliação** e apenas 2 estudos realizam uma **Validação de Pesquisa** em ferramentas propostas. De forma análoga, apenas 4 estudos realizam uma pesquisa de avaliação e 3 apresentam validação de pesquisa sobre métodos propostos. Dessa forma, é possível perceber que grande parte das pesquisas está voltada ao desenvolvimento de novos métodos e ferramentas estimação ou inferência da relação entre desenvolvedor e código fonte. Entretanto, tais métodos ou ferramentas ainda foram pouco avaliados na indústria.

A.3.5.3 QP1 - Qual a relação entre desenvolvedor e código fonte o estudo visa inferir? Familiaridade, conhecimento, autoria, propriedade, especialidade ou contribuição de código?

A Figura 28 apresenta a quantidade de estudos que visam inferir cada tipo de relação entre desenvolvedor e código fonte. É importante ressaltar que existem estudos que inferem ou tratam a respeito de uma ou mais relações apresentadas.

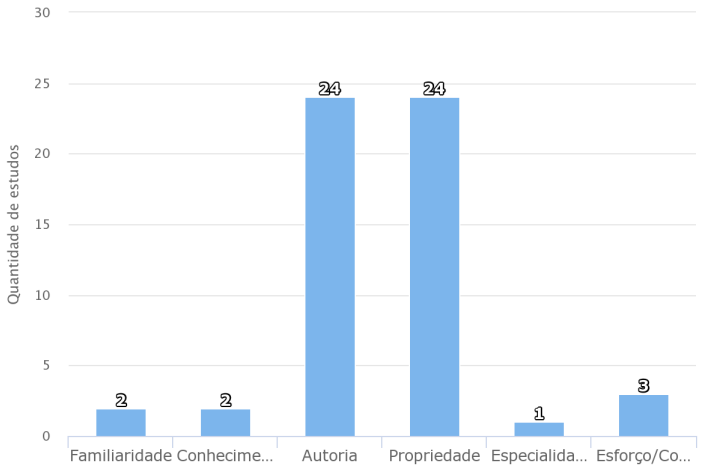


Figura 28 – Quantidade de estudos por relação entre desenvolvedor e código fonte.

Fonte – Autor.

Pode-se perceber que a maioria dos estudos aceitos abordam **Authorship** e **Ownership** como sendo a relação inferida entre desenvolvedor e código fonte. A Tabela 20 apresenta quais relações entre desenvolver e código fonte cada um dos estudos aborda.

Os resultados para as subquestões pertencentes à questão de pesquisa **QP1** são apresentadas a seguir.

QP1(a) Em que se baseiam as métricas utilizadas no processo de inferência?

QP1(b) Qual a granularidade (linhas de código, arquivos, módulos ou funcionalidades) da relação inferida entre desenvolvedor e código fonte?

Tabela 20 – Estudos por relação entre desenvolvedor e código fonte.

Relação	**Estudos**
Familiarity	E01, E04
Knowledge	E12, E42
Authorship	E01, E04, E12, E13, E14, E15, E16, E17, E19, E20, E22, E23, E24, E25, E26, E27, E28, E29, E30, E31, E35, E36, E46, E48
Ownership	E03, E05, E06, E07, E08, E09, E10, E16, E18, E21, E32, E34, E37, E38, E39, E40, E41, E43, E44, E45, E47, E49, E50, E51
Expertise	E35
Effort/Contribuition	E02, E11, E33

Fonte – Autor.

QP1(c) Existem restrições em relação a linguagem de programação?

A subquestão **QP1(a)** visa obter informações das métricas utilizadas, mais especificamente do quê é considerado no cálculo dessas métricas. Para isso foram criadas algumas categorias que representam cada elemento considerado no cálculo das métricas utilizadas pelos estudos aceitos. Esses elementos são apresentados na Tabela 21.

Tabela 21 – Elementos considerados para cálculo de métricas.

Categoria	**Descrição**
1	Quantidade de *commits*.
2	Alterações de código.
3	Estilo de código de desenvolvedores.
4	Identificação de autores por meio de comentários no código fonte.
5	Dados obtidos a partir de *Pull-Requests* (YING et al., 2016), que são requisições de mudanças de código feitas em projetos de software *open source*.

Fonte – Autor.

A Figura 29 apresenta o número de estudos que utilizam como base cada elemento apresentado na Tabela **??**. Pode-se perceber que grande parte dos estudos, ou 56,8%, utilizam como base, as alterações realizadas pelos desenvolvedores sobre o código fonte (**Categoria 2**). Outra grande parte dos estudos, ou 27,4% deles, utilizam como base o perfil de desenvolvedores na escrita de código (**Categoria 3**), isto é, o estilo de código fonte desenvolvido pelos desenvolvedores. O estilo de código envolve, por exemplo: identação, tamanho dos nomes de variáveis, tamanho das linhas de código, etc.

Uma parte considerável dos estudos, cerca de 9,9%, levam em consideração o número de *commits* realizados pelos desenvolvedores (**Categoria 1**). Outros 4% levam em consideração a assinatura de desenvolvedores em comentários feitos no código fonte

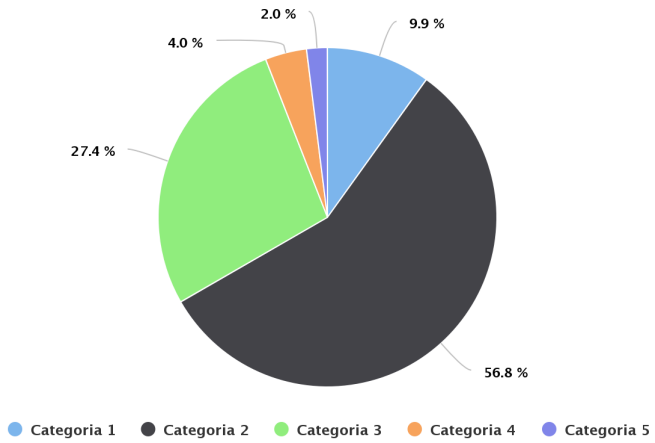


Figura 29 – Quantidade de estudos por elementos utilizados na definição de métricas.

Fonte – Autor.

(**Categoria 4**), e os outros 2% dos trabalhos consideram requisições para alterações de código realizadas pelos desenvolvedores a repositórios de software *open source*.

As categorias 1 e 4 tendem a sofrer uma maior ameaça à validade. O número de *commits* (Categoria 1) pode ocultar o caso de desenvolvedores que costumam realizar muitos *commits* porém com poucas alterações de código. A assinatura de desenvolvedores em comentários de código (Categoria 4) pode ser pouco aplicável, pelo fato de que obrigatoriamente deve existir comentários de código com a identificação de desenvolvedores. Além disso, um arquivo assinado por um desenvolvedor pode sofrer alterações por outros membros da equipe de desenvolvimento e pode não representar mais precisamente o nível de relação entre desenvolvedor e código fonte. A Tabela 21 apresenta quais elementos são levados em consideração por cada um dos estudos.

Tabela 22 – Estudos por elementos utilizados na definição de métricas.

Elemento	**Estudos**
Quantidade de *commits*	E01, E21, E34, E38, E40
Alterações de código	E03, E04, E05, E06, E07, E08, E09, E10, E11, E12, E14, E16, E18, E33, E36, E37, E43, E44, E45, E46, E47, E48, E49, E50, E51
Estilo de código de desenvolvedores	E13, E17, E19, E20, E22, E23, E24, E25, E26, E27, E28, E29, E30, E31
Comentários no código fonte	E01, E15
Dados obtidos a partir de *Pull-Requests*	E35

Fonte – Autor.

A subquestão **QP1(b)** tem como objetivo identificar qual a granularidade da relação inferida entre desenvolvedor e código fonte, ou seja, se essa relação é dada considerando linhas de código (ou LOC, do inglês *Lines Of Code*), arquivos, módulos ou funcionalidades. A Figura 30 apresenta o número de estudos com base em cada granularidade.

Pode-se perceber que grande parte dos estudos visam estimar a relação entre desenvolvedor e código fonte considerando a granularidade por arquivo. Em contrapartida, a menor parte dos trabalhos consideram a granularidade por módulo e funcionalidade, que podem trazer um grande ganho de informações. Apenas 1 (um) dos estudos visa inferir a relação entre desenvolvedor e código fonte considerando a granularidade por funcionalidade, que foi proposto por GREEVY; GIRBA; DUCASSE 2007.

A Tabela 23 quais estudos abordam cada nível de granularidade da relação entre desenvolvedor e código fonte.

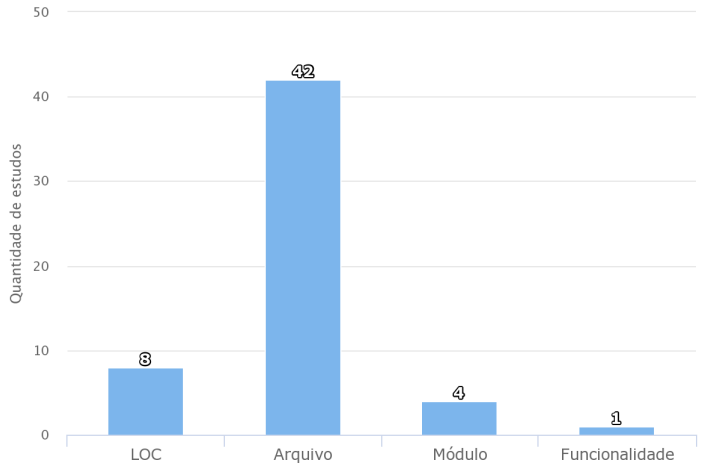


Figura 30 – Quantidade de estudos por granularidade de relação de código.

Fonte – Autor.

Tabela 23 – Estudos por nível de granularidade de relação de código.

Granularidade	**Estudos**
LOC	E08, E10, E14, E16, E32, E33, E36, E45
Arquivo	E01, E02, E04, E05, E06, E07, E08, E09, E10, E12, E13, E15, E16, E17, E18, E19, E20, E21, E22, E23, E24, E25, E26, E27, E28, E29, E30, E31, E34, E37, E38, E39, E40, E41, E42, E43, E44, E46, E47, E48, E49, E50, E51
Módulo	E11, E21, E25, E35
Funcionalidade	E03

Fonte – Autor.

A subquestão **QP1(c)** tem por objetivo identificar se o método utilizado para inferência da relação entre desenvolvedor e código fonte, possui restrições quanto a linguagens

de programação.

A Figura 31 apresenta a quantidade de estudos por restrição de linguagem de programação. Aqueles estudos em que não foi possível identificar se o método apresentado possui ou não restrição de linguagem são representados por **N/I** (Não Informado). Os estudos que não possuem restrições de linguagem são representados por **S/R** (Sem Restrições).

Pode-se perceber que grande parte dos estudos não possuem restrições quanto a linguagem de programação. Dentre aqueles que apresentam método dependentes de linguagem, a maioria aborda Java (GOSLING; MCGILTON, 1995) como objeto de estudo. Dentre as outras linguagens identificadas, as mais abordadas são C (KERNIGHAN et al., 1988) e C++ (STROUSTRUP, 1995), respectivamente.

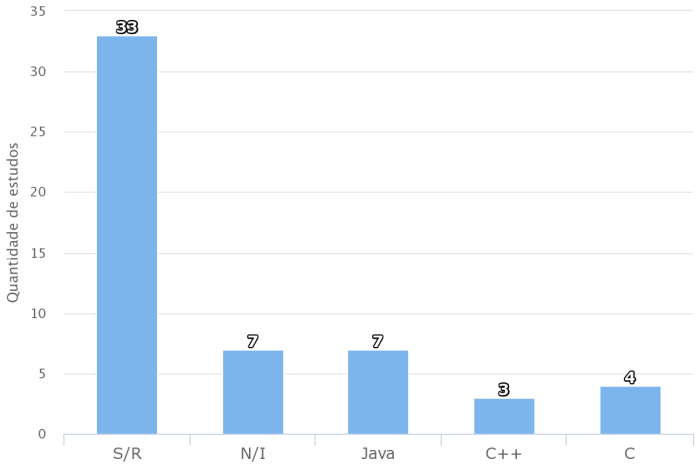


Figura 31 – Quantidade de estudos por restrições de linguagem de programação.

Fonte – Autor.

A Tabela 24 apresenta cada estudo indicando quais deles apresentam ou não métodos com restrições de linguagem de programação, além daqueles que são dependentes de uma linguagem específica.

Tabela 24 – Estudos por restrições quanto à linguagem de programação.

Restrição	**Estudos**
S/R	E02, E05, E07, E08, E09, E10, E11, E14, E16, E17, E19, E20, E21, E24, E25, E26, E27, E28, E30, E31, E32, E35, E37, E38, E40, E41, E44, E45, E46, E47, E49, E50, E51
N/I	E01, E04, E12, E13, E33, E34, E36
Java	E03, E06, E15, E18, E23, E39, E42
C++	E15, E22, E23
C	E15, E22, E43, E48

Fonte – Autor.

A.3.5.4 QP2 - O estudo busca solucionar algum problema ligado ao processo de desenvolvimento de software, a partir da inferência da relação entre desenvolvedor e código fonte? Qual?

A segunda questão de pesquisa (**QP2**) foi definida com o intuito de identificar qual o objetivo em inferir a relação entre desenvolvedor e código fonte, no ponto de vista de pesquisadores ou profissionais para o processo de desenvolvimento de software. Como mencionado na Seção A.3, essa informação a respeito da relação entre desenvolvedor e código fonte, pode ser utilizada para auxiliar ou solucionar problemas ligados ao processo de desenvolvimento de software. Pode auxiliar, por exemplo, durante a etapa de manutenção de software, mais especificamente na etapa de recomendação de desenvolvedores para solucionar problemas ou adicionar novas funcionalidades aos sistemas.

Durante a análise dos estudos, foram identificados trabalhos que não buscavam solucionar problemas ligados ao processo de desenvolvimento de software, mas que visavam entender ou obter outros tipos de informações secundárias. Dentre essas informações estão: cálculo de *Truck Factor* (TF) (RICCA; MARCHETTO; TORCHIANO, 2011) e a associação entre desenvolvedores e a qualidade do software (JONES, 1997).

A Figura 32 apresenta a porcentagem de estudos que utilizam a informação de relação entre desenvolvedor e código fonte para solucionar problemas ligados ao processo de desenvolvimento de software ou mesmo obter novas informações.

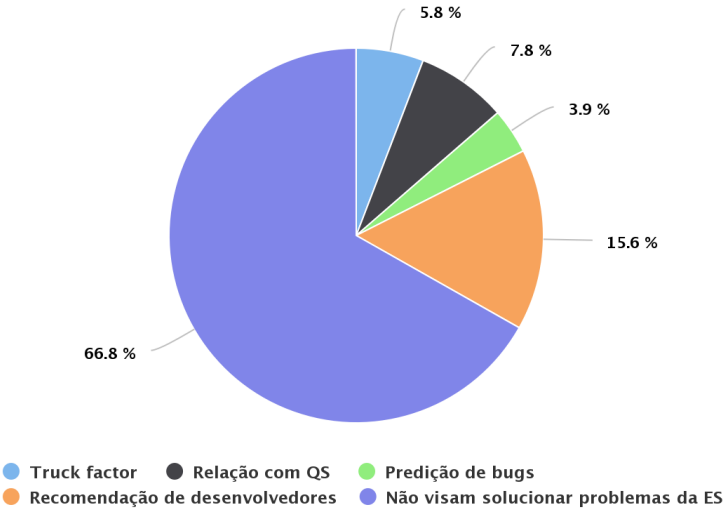


Figura 32 – Porcentagem de estudos por objetivo em inferir a relação entre desenvolvedor e código.

Fonte – Autor.

A Tabela 25 apresenta a classificação dos estudos quanto aos objetivos.

Pode-se perceber que apenas 19,5% dos estudos visam solucionar algum tipo de problema relacionado ao desenvolvimento de software. Sendo que cerca de 15,6% dos trabalhos propõem abordagens para a recomendação de desenvolvedores para tarefas

Tabela 25 – Estudos por objetivo em inferir a relação entre desenvolvedor e código.

Objetivo	**Estudos**
Estimar *Truck Factor*	E07, E41, E46
Associação com QS	E16, E32, E39, E40
Predição de *bugs*	E14, E43
Recomendação de desenvolvedores	E01, E04, E15, E35, E37, E44, E45, E48
Nenhum	E02, E03, E05, E06, E08, E09, E10, E11, E12, E13, E17, E18, E19, E20, E21, E22, E23, E24, E25, E26, E27, E28, E29, E30, E31, E33, E34, E36, E38, E42, E47, E49, E50, E51

Fonte – Autor.

ligadas a manutenção de software, incluindo por exemplo, resolução de defeitos. Outros 3,9% propõem abordagens para prever a ocorrência de erros em projetos de software.

Com isso, é possível também classificar os estudos quanto à **Disciplina/Subprocesso** da Engenharia de Software em que os estudos estão ligados. Esse modelo de classificação foi apresentado na Seção A.3.3. Pode-se perceber então que 15,6% dos trabalhos estão ligados à disciplina de **Implementação**, que compõem os estudos que propõem métodos para recomendação de desenvolvedores, enquanto 3,9% dos estudos estão ligados à disciplina de **Gestão de Qualidade**, que compõe os estudos que propõem métodos para predição de *bugs*. Os demais estudos não visavam solucionar algum tipo de problema da ES, e com isso não houve trabalhos classificadas nas demais disciplinas de ES.

A.3.5.5 QP3 - Quais tipos de repositórios de software foram utilizados no processo de mineração?

Essa questão de pesquisa tem como objetivo identificar quais tipos de repositórios software foram utilizados como fonte de extração de dados durante processo de mineração. A Tabela 26 apresenta os estudos por tipo de repositório de software utilizado. É importante ressaltar que a discriminação de tipos de repositórios foi feita com base no que foi definido por HASSAN (2008).

Pode-se perceber que a maior parte dos estudos (25 no total) utilizam como fonte de extração de dados, repositórios de software do tipo **sistema de controle de versionamento**. Outra grande parte dos estudos (18 no total) também utilizam como fonte de dados para o processo de mineração, *repositórios de código*, que geralmente oferecem, além de armazenamento, o recurso de versionamento de código. Pode-se perceber também que, dentre os estudos, existem aqueles que utilizam como fonte dados no processo de mineração, um ou mais tipos de repositórios de software.

Os resultados para as subquestões pertencentes à questão de pesquisa **QP3** são apresentadas a seguir.

Tabela 26 – Estudos por tipos de repositórios de software.

Tipo de Repositório	**Estudos**	**Total**
Issue Track System (ITS)	E01, E47	2
Bug Track System BTS	E04, E12, E15, E16, E21, E33, E43	7
Sistema de arquivo local	E18, E19, E24, E37, E42, E47, E48	7
Lista de *E-mail*	E34, E44	2
Documentos	E25, E29	2
Repositório de código	E07, E09, E11, E13, E17, E20, E22, E23, E26, E27, E28, E29, E30, E31, E35, E36, E41, E46	18
Sistema de Controle de Versão (SCV)	E01, E02, E03, E04, E05, E06, E08, E10, E12, E14, E15, E16, E21, E32, E33, E34, E38, E39, E40, E43, E44, E45, E49, E50, E51	25

Fonte – Autor.

QP3(a) Qual o repositório de software?

QP3(b) Os dados obtidos a partir do repositório de software são públicos ou privados?

QP3(c) Qual o objeto de análise obtido (extraído) a partir do repositório?

Com o objetivo de identificar também, além do tipo de repositório de software, qual repositório foi utilizado, foi definida a subquestão de pesquisa **QP3(a)**. A Figura 33 apresenta o número de estudos que utilizam cada repositório de software identificado.

Os repositórios de software identificados são referentes a 4 (quatro) tipos de repositórios de software: sistema de controle de versão (**CVS**, **Git**, **SVN**); repositório de código (**Google Code**, **GitHub**, **Sourceforge**, **Google Code Jam**, **Plant Source Code**, **Freshmeat.net**); *bug track system* (**Bugzilla**, **Clear Quest**) e documentos (**ApreessInc Books**). Alguns repositórios não puderam ser identificados nos estudos, apenas os tipos.

Pode-se perceber os tipos de repositórios mais utilizados no processo de mineração foram os Sistemas de Controle de Versão (SCV), como já identificado pelos resultados apresentados na questão de pesquisa **QP3**. Além disso, pode perceber também que dentre esse tipo de repositório de software, os mais utilizados são: **Git**, **SVN** e **CVS**, respectivamente.

Alguns tipos de repositórios de software oferecem tanto serviços gratuitos quanto privados. Por essa razão, a subquestão **QP3(b)** tem por objetivo identificar se os dados obtidos a partir do repositório de software são públicos ou privados. A Tabela 27 apresenta os estudos que realizam a extração de dados públicos ou privados. Para alguns dos estudos

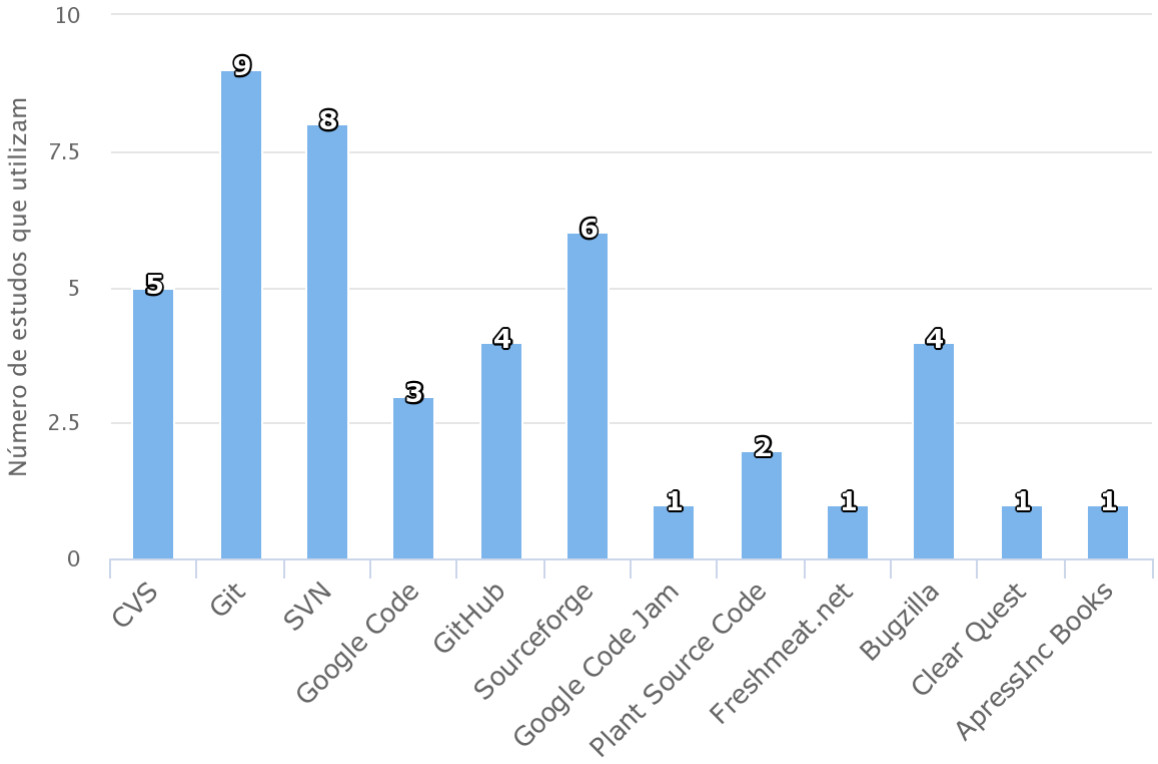


Figura 33 – Quantidade de estudos por repositórios identificados.

Fonte – Autor.

não foi possível identificar a disponibilidade dos dados utilizados no processo de mineração. É importante ressaltar que há estudos que extraem dados públicos e também privados durante o processo de mineração.

Tabela 27 – Estudos e disponibilidades dos dados extraídos.

Disponibilidade de dados	**Estudos**	**Total**
Público	E01, E02, E03, E05, E06, E07, E08, E09, E10, E11, E13, E14, E15, E16, E17, E20, E22, E26, E27, E28, E29, E30, E31, E32, E33, E34, E35, E36, E38, E39, E40, E41, E43, E44, E45, E46, E47, E49	38
Privado	E08, E12, E18, E19, E20, E21, E23, E28, E31, E33, E37, E42, E45, E47, E48	15
Não identificado (N/I)	E03, E04, E24, E25, E51	5

Fonte – Autor.

A Tabela 28 apresenta os tipos de dados extraídos a partir dos repositórios de software, cuja a identificação foi objetivada pela subquestão de pesquisa **QP3(c)**.

Pode-se perceber que os tipos de dados mais comumente extraídos são dados de *commits*, reforçando o fato de que os repositórios mais utilizados no processo de mineração

Tabela 28 – Estudos por tipos de dados extraídos.

Dados Extraídos	**Estudos**	**Total**
E-mails	E34, E44	2
Relatórios de questões (*issues*)	E01, E12, E18	1
Comentários de código	E01, E35, E50	5
Relatórios de *bugs*	E04, E12, E15, E21, E33, E43	6
Dados de *commits*	E02, E03, E04, E05, E06, E07, E08, E09, E10, E11, E12, E14, E16, E21, E33, E34, E35, E38, E39, E40, E41, E43, E44, E45, E46, E47, E49, E50, E51	29
Metadados de mudanças de código	E01, E11, E12, E14, E16, E18, E32, E36, E37, E48	10
Código fonte	E01, E11, E13, E15, E17, E18, E19, E20, E22, E23	10

Fonte – Autor.

são os sistemas de controle de versão.

A.3.5.6 QP4 - Qual mecanismo utilizado para o processo de extração de dados de repositórios?

A questão de pesquisa **QP4** tem como objetivo identificar que tipo de artifício foi utilizado para extrair dados de repositórios, como ferramenta ou algum tipo de *framework*, por exemplo. A Tabela 29 apresenta os mecanismos de extração de dados utilizados por cada um dos estudos.

Tabela 29 – Mecanismos de extração de dados utilizados.

Mecanismo de Extração	**Estudos**	**Total**
Plug-in	E18, E37, E42, E48	4
Framework	E39	1
Ferramenta	E02, E05, E07, E08, E09, E10, E11, E14, E21, E24, E33, E36, E38, E41, E43, E46, E47	17
Manual	E13, E25, E27, E28, E29, E30, E31, E32	8
Mecanismo não identificado	E01, E03, E04, E06, E12, E15, E16, E17, E19, E20, E22, E23, E26, E34, E35, E40, E44, E45, E49, E50, E51	21

Fonte – Autor.

Pode-se perceber que grande parte dos mecanismos de extração de dados utilizados pelos estudos é composta por ferramentas. O mecanismo de extração **manual** envolve, por exemplo, o *download* de dados feito manualmente a partir de repositórios de código, como **GitHub** e **Google Code**.

A.3.5.7 QP5 - Qual o método de avaliação utilizado por cada estudo?

A questão de pesquisa *QP5* foi elaborada com o intuito de identificar quais métodos de avaliação empírica têm sido utilizados para avaliar os trabalhos que visam estimar ou inferir a relação entre desenvolvedor e código fonte, e assim, fornecer uma visão da área que facilite a escolha de métodos ou abordagens com base em avaliações realizadas em cada estudo.

Os métodos de avaliação empírica foram classificados com base em WOHLIN et al. 2012, que define três métodos formais de avaliação empírica: **Survey**, **Estudo de Caso** e **Estudo Experimental**. Além disso, experimentos realizados com o intuito de provar a eficácia do método proposto, foram classificados como **Prova de Conceito**. Aqueles estudos que apresentam ou visam explorar novas soluções foram classificados como **Estudo Exploratório**. Quanto aos estudos que visam avaliar métodos propostos por meio por avaliação comparativa com outros abordagens, foram classificados como **Estudo de Comparação**.

A Figura 34 apresenta a porcentagem de métodos de avaliação utilizados pelos estudos. Pode-se perceber que a maior (53,1%) parte dos trabalhos utilizam **Prova de Conceito** como método de avaliação, seguido por **Estudo de Comparação** (19,7%), **Estudo Exploratório** (11,7%), **Estudo de Caso** (7,8%) e **Survey** (1,9%). Não foram identificados estudos que utilizaram **Experimento Controlado** como método de avaliação. Pode-se perceber também, que cerca de 5,8% dos trabalhos não utilização algum tipo de avaliação.

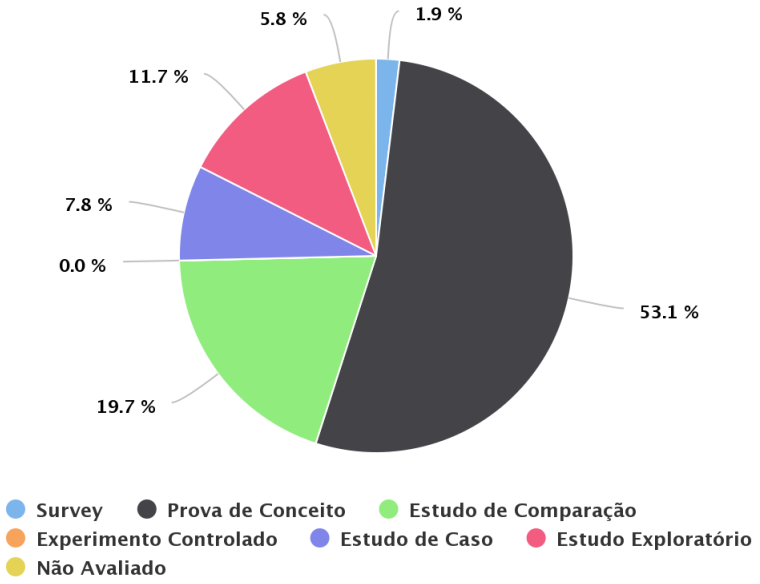


Figura 34 – Métodos de avaliação utilizados pelos estudos.

Fonte – Autor.

A Tabela 30 apresenta os métodos de avaliação utilizados pelos estudos durante a

análise de seus métodos propostos.

Tabela 30 – Estudos por métodos de avaliação utilizados.

Tipo de Avaliação	**Estudos**	**Total**
Survey	E32	1
Prova de Conceito	E02, E05, E07, E08, E10, E11, E16, E21, E23, E25, E26, E29, E30, E31, E33, E34, E35, E36, E39, E40, E43, E44, E45, E46, E47, E49, E50	27
Estudo de Comparação	E01, E06, E13, E14, E15, E17, E19, E20, E22, E48	10
Experimento Controlado	-	0
Estudo de Caso	E12, E18, E37, E42	4
Estudo Exploratório	E03, E04, E09, E27, E38, E41	6
Não Avaliado	E24, E28, E51	3

Fonte – Autor.

Além dos métodos de avaliação utilizados, objetivou-se também identificar quais foram os objetos avaliados pelos estudos, mais especificamente se os projetos de avaliação são abertos (*open source*), privados (projetos comerciais) ou mesmo projetos de pequeno porte, como projetos acadêmicos ou estudantis por exemplo, e também conjuntos de código fonte extraídos de documentos ou livros. Com esse objetivo foi definida então a subquestão de pesquisa **QP5(a)**.

A Tabela 31 apresenta os tipos de objetos de avaliação utilizados por cada estudo. É importante ressaltar que aqueles estudos que não realizaram algum tipo de avaliação, foram desconsiderados.

Tabela 31 – Objetos de avaliação utilizados pelos estudos.

Objeto de avaliação	**Estudos**	**Total**
Projetos *open-source*	E01, E02, E03, E05, E06, E07, E08, E09, E10, E11, E13, E14, E15, E16, E17, E20, E26, E27, E28, E29, E30, E32, E33, E34, E35, E36, E38, E39, E40, E41, E43, E44, E45, E46, E47, E49, E50	37
Projetos comerciais	E04, E08, E12, E18, E21, E31, E33, E37, E42, E47, E48	11
Projetos Estudantis	E03, E19, E20, E22, E23	5
Códigos de livros	E13, E25, E28, E31	0

Fonte – Autor.

Com o objetivo de identificar se os estudos realizaram discussões que relatam ameaças à validade dos resultados apresentados, foi definida então a subquestão de

pesquisa **QP5(b)**. A Tabela 32 apresenta a os trabalhos que apresentam discussões sobre ameaças à validade do estudo.

Tabela 32 – Estudos que realizam discussão sobre ameaças à validade.

Discussão à ameaças	**Estudos**	**Total**
Sim	E01, E04, E09, E11, E12, E15, E16, E18, E21, E32, E33, E34, E35, E37, E38, E39, E40, E42, E45, E46, E48, E49, E50.	23
Não	E02, E03, E05, E07, E08, E10, E13, E14, E17, E19, E20, E22, E23, E25, E26, E27, E28, E29, E30, E31, E36, E41, E43, E44, E47	25

Fonte – Autor.

É importante ressaltar também, que foram considerados apenas os estudos que realizaram algum tipo de avaliação empírica do método proposto para inferência da relação entre desenvolvedor e código fonte. Portanto, pode-se perceber que dentre aqueles trabalhos que apresentaram algum tipo de avaliação, cerca de 94%, a maioria (52%) desses não apresentam discussões sobre ameaças à validade de seus métodos.

A.4 Ameaças à Validade

Nesta seção apresenta algumas ameaças à validade do Mapeamento Sistemático conduzido apresentado neste capítulo. Normalmente, durante a condução de um MSL surgem fatores que podem vir a trazer ameaças à validade do estudo. Com isso, a seguir, são apresentados alguns desses fatores, assim como a estratégia utilizada para atenuar as influências que essas ameaças podem oferecer ao trabalho realizado.

- **Cobertura da *string* de pesquisa:** Essa ameaça refere-se à eficácia da *string* utilizada para busca de trabalhos relacionados à área de pesquisa desta trabalho. Para atenuar os riscos oriundos dessa ameaça, a *string* de pesquisa utilizada foi construída de forma à abranger o maior número de estudos possíveis. Para isso, foram utilizados sinônimos para os diferentes termos que inicialmente foram utilizados na composição da *string*. Além disso, foi realizada uma pesquisa preliminar em diferentes bases de dados, com o objetivo de verificar a eficiência e cobertura da *string* utilizada.

- **Seleção de Estudos:** Essa ameaça está relacionada ao processo de seleção de trabalhos relevantes retornados após a busca ou aplicação da *string* de pesquisa. Isso se deve ao fato de que o resultado da seleção de estudos pode apresentar diferença devido à opinião individual de pesquisadores que participaram do processo

de seleção. Para atenuar essa ameaça, a seleção de estudos foi realizada por grupos de pesquisadores de maneira independente. Além disso, foram realizadas análises de validade após a realização de cada etapa de seleção para verificar a concordância entre os pesquisadores, conforme descritas na Seção A.3.2.

- **Extração de Dados:** Durante a etapa de extração de dados dos estudos aceitos, é comum a possibilidade de pesquisadores interpretarem as informações de trabalhos de maneiras diferentes. Inicialmente, para atenuar essa ameaça, foi desenvolvida uma planilha como todos os campos referentes ao esquema de classificação (apresentado na Seção A.3.3) dos estudos, e foi realizado um processo de seleção piloto para identificar e discutir os principais conflitos encontrados, a fim de esclarecer melhor possíveis dúvidas. Além disso, durante o processo de extração, aqueles trabalhos que apresentaram discordâncias quanto às informações extraídas, foram novamente analisados e discutidos até que houvesse consenso entre os pesquisadores.

A.5 Considerações Finais

Este apêndice apresentou um Mapeamento Sistemático realizado como o objetivo identificar, analisar e sumarizar estado da arte sobre estudos que apresentam técnicas baseadas em mineração de repositórios de software para inferir a relação entre desenvolvedor e código fonte considerando diferentes perspectivas. Esse estudo fornece uma visão geral dessa linha de pesquisa, identificando os principais tópicos abordados por trabalhos anteriores e destacando lacunas passíveis de investigação.

APÊNDICE B – Referências para trabalhos relacionados ao Mapeamento Sistemático

Tabela 33 – Referências para estudos aceitos.

ID	Título	Referência
E01	Amalgamating Source Code Authors, Maintainers, and Change Proneness to Triage Change Requests	(HOSSEN; KAGDI; POSHYVANYK, 2014)
E02	Fractal figures: Visualizing development effort for cvs entities	(D'AMBROS; LANZA; GALL, 2005)
E03	How developers develop features	(GREEVY; GIRBA; DUCASSE, 2007)
E04	A Degree-of-Knowledge Model to Capture Source Code Familiarity	(FRITZ et al., 2010)
E05	OvERVIeW: Ownership visualization word cloud	(FRONZA; TREBESCHI, 2016)
E06	Impact of Version History Metrics on Maintainability	(FARAGÓ et al., 2015)
E07	Are Heroes common in FLOSS projects?	(RICCA; MARCHETTO, 2010a)
E08	How Developers Drive Software Evolution	(GIRBA et al., 2005)
E09	Heroes in FLOSS projects: an explorative study	(RICCA; MARCHETTO, 2010b)
S10	An Analysis of Author Contribution Patterns in Eclipse Foundation Project Source Code	(TAYLOR et al., 2011)
E11	Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects	(YAMASHITA et al., 2015)
E12	Degree-of-Knowledge: Modeling a Developer's Knowledge of Code	(FRITZ et al., 2014)
E13	A Bayesian Ensemble Classifier for Source Code Authorship Attribution	(TENNYSON; MITROPOULOS, 2014a)
E14	Mining Software Repositories for Accurate Authorship	(MENG et al., 2013)

E15	Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship?	(LINARES-VÁSQUEZ et al., 2012)
E16	Ownership, Experience and Defects: A Fine-Grained Study of Authorship	(RAHMAN; DEVANBU, 2011)
E17	On the Use of Discretized Source Code Metrics for Author Identification	(SHEVERTALOV et al., 2009)
E18	Mining the History of Synchronous Changes to Refine Code Ownership Lile	(HATTORI; LANZA, 2009)
E19	Source Code Authorship Attribution using n-grams	(BURROWS; TAHAGHOGHI, 2007)
E20	Effective identification of source code authors using byte-level information	(FRANTZESKOU et al., 2006a)
E21	Code Ownership and Software Quality: A Replication Study	(GREILER; HERZIG; CZERWONKA, 2015)
E22	OBA2: An Onion approach to Binary code Authorship Attribution	(ALRABAEE et al., 2014)
E23	Application of Information Retrieval Techniques for Source Code Authorship Attribution	(BURROWS; UITDENBOGERD; TURPIN, 2009)
E24	Identified: A dictionary-based system for extracting source code metrics for software forensics	(GRAY; SALLIS; MACDONELL, 1998)
E25	A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis	(KILGOUR et al., 1998)
E26	A Probabilistic Approach to Source Code Authorship Identification	(KOTHARI et al., 2007)
E27	Examining the Significance of High-Level Programming Features in Source Code Author Classification	(FRANTZESKOU et al., 2008)
E28	Choosing a profile length in the SCAP method of source code authorship attribution	(TENNYSON; MITROPOULOS, 2014b)
E29	Deep neural networks for source code author identification	(BANDARA; WIJAYARATHNA, 2013a)
E30	Source code author identification with unsupervised feature learning	(BANDARA; WIJAYARATHNA, 2013b)

E31	Source code author identification based on N-gram author profiles	(FRANTZESKOU et al., 2006b)
E32	On the usefulness of ownership metrics in open-source software projects	(FOUCAULT et al., 2015)
E33	Unveiling Developers Contributions Behind Code Commits: An Exploratory Study	(COSTA et al., 2014)
E34	Stochastic actor-oriented modeling for studying homophily and social influence in OSS projects	(KAVALER; FILKOV, 2017)
E35	EARec: Leveraging Expertise and Authority for Pull-Request Reviewer Recommendation in GitHub	(YING et al., 2016)
E36	Fine-Grained Binary Code Authorship Identification	(MENG, 2016)
E37	Refining Code Ownership With Synchronous Changes	(HATTORI; LANZA; ROBBES, 2012)
E38	A Case Study on the Relationship between Code Ownership and Refactoring Activities in a Java Software System	(ORRÚ; MARCHESI, 2016)
E39	Code Ownership in Open-Source Software	(FOUCAULT; FALLERI; BLANC, 2014)
E40	Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review	(THONGTANUNAM et al., 2016)
E41	On the difficulty of computing the truck factor	(RICCA; MARCHETTO; TORCHIANO, 2011)
S42	Does a Programmer's Activity Indicate Knowledge of Code?	(FRITZ; MURPHY; HILL, 2007)
E43	Using the Gini Coefficient for Bug Prediction in Eclipse	(GIGER; PINZGER; GALL, 2011)
E44	A Case Study of Open Source Software Development: The Apache Server	(MOCKUS; FIELDING; HERBSLEB, 2000)
E45	Quantifying and Mitigating Turnover-Induced Knowledge Loss: Case Studies of Chrome and a project at Avaya	(RIGBY et al., 2016)

E46	A Novel Approach for Estimating Truck Factors	(AVELINO et al., 2016b)
E47	In-Situ Visualisation of Fractional Code Ownership over Time	(MÜLLER; REINA; ERTL, 2015)
E48	Using Developer-Interaction Trails to Triage Change Requests	(ZANJANI; KAGDI; BIRD, 2015a)
E49	Using Code Ownership to Improve IR-Based Traceability Link Recovery	(ZANJANI; KAGDI; BIRD, 2015b)
E50	Who are Source Code Contributors and How do they Change?	(PENTA; GERMAN, 2009)
E51	Succession: Measuring Transfer of Code and Developer Productivity	(MOCKUS, 2009)