Towards an Expertise-Related Metric for Preprocessor-Based Configurable Software Systems

Karolina Martins Milano

SERVIÇO DE PÓS-GRADUAÇÃO FACOM-UFMS
Data de Depósito:

Assinatura:______________________________

Towards an Expertise-Related Metric for Preprocessor-Based Configurable Software Systems

Karolina Martins Milano

Advisor: Prof. Bruno Barbieri de Pontes Cafeo

Dissertation presented to the Programa de Pós-Graduação em Ciência da Computação of the Faculdade de Computação Universidade Federal do Mato Grosso do SUL - UFMS as partial fulfillment of the requirements for the degree of Mestre em Ciência da Computação

UFMS - FACOM
May/2022

Acknowledgements

First and foremost, to Jesus for never abandoning me, even in the worst and most painful moments of my life.

I would like to thank Professor Bruno Cafeo for giving me the opportunity to advise me on my academic master's degree and for all his patience during this process.

I thank my husband, Pedro Henrique, for taking care of our family during my absence.

To my daughter, Vitória, for all the absence endured during these two years.

To my mother, Raquel, for all her prayers and help throughout my life.

I also thank my nephews, Heloisa, Davi and Sarah, and siblings, Karina and João Carlos, for praying and cheering for me during these years.

Abstract

Context: Expertise-related metrics allow us to find the best developers for a target task in a file. Configurable systems use variability as a unit of abstraction to generate different members of a program family. This misalignment between files used by expertise-related metrics and variabilities used by configurable systems may make it impossible to use them together.
Objective: The objective is twofold. The first is to explore how the work on mandatory and variable code is divided among developers and whether expertise-related metrics can indicate a developer with expertise for a task involving variable code. The second is to propose a variability-aware expertise-related metric to indicate developers with expertise in variable code.
Method: We investigate 49 preprocessor-based configurable systems. We analyzed how variabilities changes are divided between developers and whether these developers would be key developers indicated by expertise-related metrics. We use feature selection and multiple linear regression techniques to propose a variability-aware expertise-related metric. We validate our metric by comparing it with two well-known metrics.
Results: Few developers are specialists in variable code. We also identified that only a few developers concentrate the majority of changes in variable code. The results also suggested that expertise-related metrics are not a good fit to indicate experts regarding variable code. We proposed a variability-aware expertise-related metric and showed that our proposed metric outperformed well-known expertise-related metrics.
Conclusion: Even though the results show that a considerable number of developers touched variable code during the development history, such changes are only occasional. There is a concentration of work among a few developers when it comes to variable code. This uneven division may cause an unnecessary maintenance effort. We also conclude that variability-aware expertise-related metrics may better support the identification of experts in configurable systems when compared to existing metrics.

Resumo

Contexto: Métricas relacionadas à experiência dos desenvolvedores nos permitem encontrar os melhores desenvolvedores para uma tarefa específica em um arquivo. Sistemas configuráveis usam a variabilidade de código como unidade de abstração para gerar diferentes membros de uma família de programas. Esse desalinhamento entre os arquivos usados pelas métricas relacionadas à experiência e as variabilidades usadas pelos sistemas configuráveis pode impossibilitar o uso conjunto delas.

Objetivo: O objetivo é duplo. O primeiro é explorar como o trabalho em código mandatório e variável é dividido entre os desenvolvedores e se as métricas relacionadas à expertise podem indicar um desenvolvedor com expertise para uma tarefa envolvendo código variável. O segundo é propor uma métrica relacionada à experiência com conhecimento em variabilidades para indicar desenvolvedores com experiência em código variável.

Método: Foram investigados 49 sistemas configuráveis baseados em pré-processadores, sendo analisadas como as mudanças nas variabilidades são dstribuídas entre os desenvolvedores, e se esses desenvolvedores seriam os principais desenvolvedores indicados por métricas relacionadas a experiência do dosenvolvedor em arquivos de código. Foram utilizadas técnicas de feature selection e regressão linear múltipla para propor uma métrica relacionada a experiência do desenvolvedor em relação ao conhecimento de variabilidades de código. A métrica proposta foi validada comparando-a com duas métricas já conhecidas.

Resultados: Poucos desenvolvedores são especialistas em código variável. Foi identificado que poucos desenvolvedores concentram a maioria das alterações em código variável. Os resultados também sugerem que que a expertise relacionada a métricas já conhecidas não são um bom ajuste para indicar experts em relação ao código variável. Foi proposta uma métrica relacionada a experiência dos desenvolvedores em relação as variabilidades e foi mostrado que a métrica proposta superou métricas relacionadas a experiência em relação a arquivos de código, já conhecidas.

Conclusão: Embora os resultados mostrem que um número considerável de desenvolvedores realizou alterações no código variável durante o histórico de desenvolvimento, tais alterações são apenas ocasionais. Há uma concentração de trabalho entre alguns desenvolvedores quando se trata de código variável. Esta divisão desigual pode causar um esforço de manutenção desnecessário. Também concluímos que as métricas relacionadas à experiência em relação ao conhecimento das variabilidades podem apoiar melhor a identificação de especialistas em sistemas configuráveis quando comparadas às métricas existentes.

Contents

List of Figures xiii
List of Tables xv

1 Introduction 1

2 Preliminaries 5
2.1 Configurable Software Systems 5
2.2 Motivating Example 7

3 Exploratory Study 9
3.1 Research Questions 9
3.2 Subject Systems 10
3.3 Degree of Authorship (DOA) 11
3.4 Ownership 12
3.5 The Data Collection Procedure 13
3.6 The Data Evaluation Procedure 14
3.6.1 Work specialization (RQ1) 14
3.6.2 Distribution of work among developers (RQ2) 14
3.6.3 Association between file expertise and variable code (RQ3) 15

4 Results 17
4.1 Work specialization (RQ1) 17
4.2 Distribution of work among developers (RQ2) 20
4.3 Association between file expertise and variable code (RQ3) 22

5 Implications 23

6 Proposed Approach 25
6.1 Feature Selection 25
6.2 Linear Regression 27
6.2.1 Mean Absolute Error (MAE) 29

6.2.2 Mean Squared Error (MSE) 29
6.2.3 Root Mean Squared Error (RMSE) 29

7 Experiment **33**
7.1 Experimental Data Set 33
7.2 Performance Metrics 34
7.3 Approach performance (RQ4) 35
7.4 Approach comparison (RQ5) 36

8 Related Work **39**

9 Threats to Validity **41**

10 Conclusions and Future Work **43**

List of Figures

2.1 Compilation process with a preprocessor. 6
2.2 Example of use of preprocessor directive in source code. 6
2.3 Illustration of key developers in a configurable system file. 8

3.1 Procedure used to collect data for the exploratory study. 13

4.1 Work specialization over time in four systems. 19
4.2 Variabilities per developer. 20
4.3 Lorenz curve of the four representative systems. 21

6.1 Illustration of the process of linear regression. 28

List of Tables

3.1 Overview of the subject systems. 11

4.1 Summary of results. 18

6.1 Features analyzed. 27
6.2 Results of the application of three feature selection methods. . . . 28

7.1 Accuracy of the approach with individual subject systems. 35
7.2 Comparison between expertise-related metrics and our approach. 36

Lists of acronyms

DOA	Degree of atuhorship
DOA$_V$	Degree-Of-Authorship-in-Variabilities

Chapter 1
Introduction

Software development teams exchange a large amount of information about a project's code base each day. Considering a professional software development team, each developer, on average, received changes to over one thousand different source-code elements and interacted with more than one hundred and forty source-code elements per day [FMMH$^+$14]. These impressive numbers make it challenging for a developer to know about the entire code of a system. So, questions such as "Who is the most appropriate developer to perform this maintenance task?" or "Which developers would be able to review this pull request?" become usual.

Expertise in source code is an important concept used to identify developers responsible for making significant changes to the source code [FMMH$^+$14]. In large software systems with hundreds of modules, these metrics allow for coordinating developer teams by planning the overall division of work, identifying key collaborators, and finding the best developers for a target task [APHV19]. Essentially, measuring expertise aims to identify developers who made significant changes to specific system modules. The problem is that system modules are created by one developer but later changed by possibly hundreds of developers [FMMH$^+$14].

Configurable systems use variability as the unit of abstraction to implement configuration options such as features [KCH$^+$90]. In systems based on preprocessors and conditional compilation, such as the Linux Kernel or GCC, the limits of implementing variability often do not align with the system modules [AB11]. In other words, the source code of a single variability may be scattered and tangled through the source code.

While conditional compilation is highly flexible and easy to use, it leads to code that is hard to maintain. Preprocessors prepare the source code before it is handed to the actual compiler. Developers use preprocessor directives, such as #ifdef and #endif, to mark blocks of source code as optional or conditional, with the purpose of tailoring software systems to different hardware platforms, operating systems, and application scenarios. Therefore, these directives allow excluding parts of a source code file from a compilation by the C/C++ compiler if a condition is met.

Researchers and practitioners have criticized the C preprocessor because of its negative effect on code understanding and maintainability and its error proneness [LKA11, EBN02, GJ05, TOB11, MRB$^+$17]. Compile-time conditions add an entirely new layer of complexity as they are not part of the programming language, e.g., C++, but part of the preprocessor language. So, run-time and compile-time conditions are often intermixed; therefore, the reader of the code always has to keep track of the two language layers. Hence, the mental load for understanding code using preprocessor and conditional compilation is high [LKA11].

Considering a common scenario in which there are several variabilities in a single file, and several developers have modified that file throughout the development history. There is considerable difficulty in, for example, assigning a maintenance task that involves a variability for the developer who has the slightest difficulty in understanding such variability. Another example would be choosing a code reviewer to review a pull request that involves variability. Assigning the maintenance task to the most suitable developer for understanding the code units or selecting an appropriate code reviewer is essential for reducing maintenance effort and cost. Furthermore, in a scenario where the unit of abstraction (commonly used in maintenance task assignments) can be scattered and tangled in the source code, this closer to ideal selection becomes essential to develop configurable systems in a collaborative setup.

In recent years, several studies approaching expertise-related metrics have been conducted [MA00, BND$^+$09, RD11, BNM$^+$11, FMMH$^+$14, APHV19]. Many of these studies investigated the impact of expertise on software quality attributes [YL05, RD11]. However, despite the growing interest in developer expertise, to the best of our knowledge, little is known about developer expertise in configurable software systems. In most literature, configurable systems are not considered the subject of study. Moreover, these studies essentially consider code authorship at the level of files and modules.

In this context, some questions are still open, such as: how many developers are responsible for implementing variabilities in a system? Are these developers responsible for implementing both mandatory and variable code? Is a developer with a higher file level authorship likely to be associated with variability code? Are existing expertise-related metrics suitable for configurable systems? By answering these questions, we can assess expertise-related metrics in the context of preprocessor-based configurable systems, better supporting their use. In addition, if the existing expertise-related metrics are unsuitable for application in configurable systems, it is important to propose an approach that helps calculate the developer's expertise regarding variable codes.

This work provides a set of contributions to support the maintenance of preprocessor-based configurable systems. In particular, this paper has three major contributions:

1. To the best of our knowledge, this is the first work to assess expertise-related metrics in variability in the context of configurable systems.
2. We provide an extensive empirical study about code authorship to understand the implications in the context of configurable systems.
3. We provide the *Degree-Of-Authorship-in-Variabilities (DOA_V)*. A formula specifically created for preprocessor-based systems to indicate the developer's expertise regarding variabilities.

We organize the remainder of the paper as follows. In Chapter 2, we introduce the basic concepts of configurable systems and show a motivating example. Chapter 3 presents the exploratory study, while Chapter 4 reports the results. Chapter 5 discusses the implications of the exploratory study. Chapter 6 details the definition of a formula to indicate expertise in variabilities in the context of configurable software systems. Chapter 7 validate our proposed approach. Chapter 8 discusses related work. Finally, Chapter 9 presents the threats to validity, and Chapter 10 concludes the paper and describes future work.

CHAPTER 2

Preliminaries

To lay a foundation for subsequent sections, we introduce the basic concepts of configurable systems and code review. We also present a motivating example to illustrate the use of code authorship in a configurable software system.

2.1 Configurable Software Systems

Systems that share a common core but also have different functionalities are referred to as configurable systems [AVRW+13]. The core implements the basic functionality presented in any member of a program family, and the different selections of configurations define the set of program variants. These commonalities and variabilities are often modeled as features, each representing additional functionality to the core software [PBvDL05].

When we consider configurable systems written in C, developers often use the C preprocessor to annotate the implementation code of variabilities. The preprocessor identifies the code that should be compiled or not based on preprocessor directives (i.e., `#ifdef` directives) along with a macro expression. Macro expressions might be composed of one or more macros, as a boolean formula, which refer to specific variabilities. Figure 2.1 shows the steps to generate an executable file with a preprocessor in the process.

Developers use preprocessor directives to wrap entire structures of code, such as functions and even single variables. Therefore, variability is a set of program elements surrounded by preprocessor directives. It is essential to highlight that the flexible granularity level of variabilities provided by conditional compilation causes a variability code to spread throughout the pro-

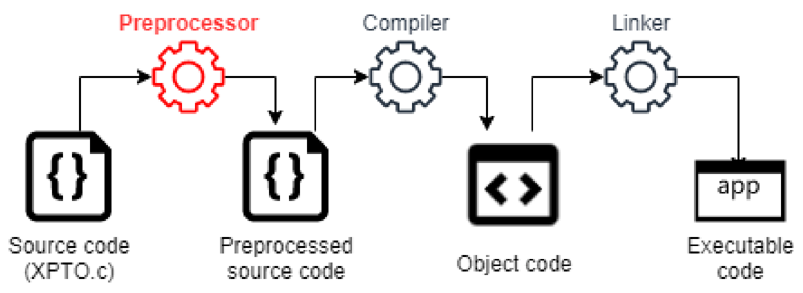
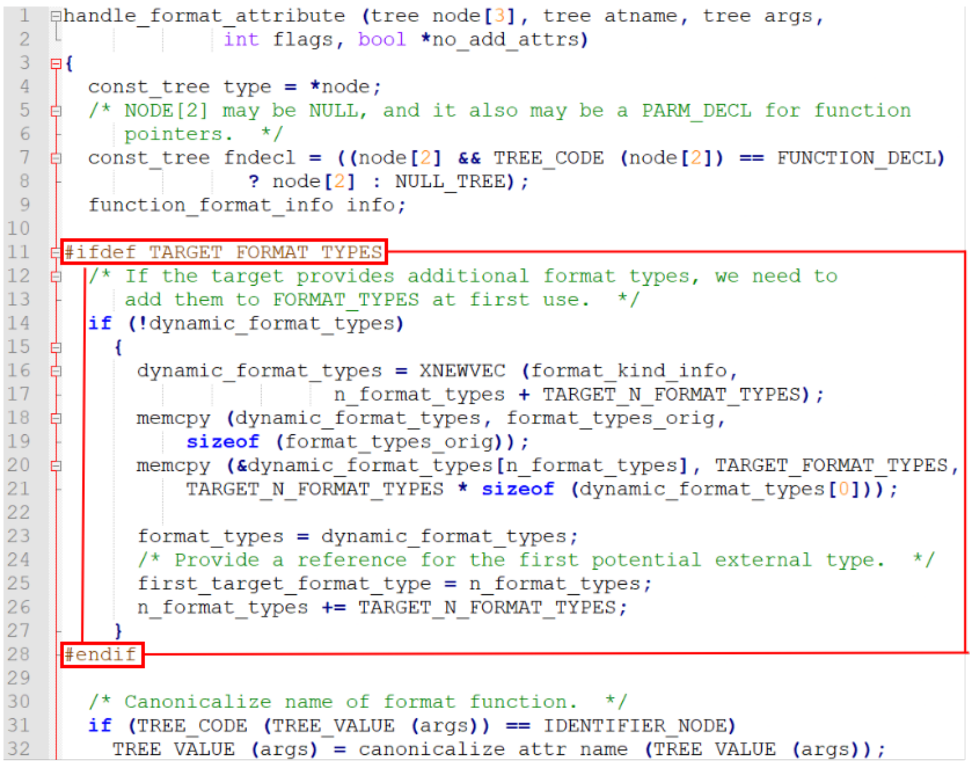


Figure 2.1: Compilation process with a preprocessor.

gram [Käs12].

We refer to the code snippet in Figure 2.2 extracted from c-format.c of the GCC project to better explain this concept.



```
handle_format_attribute (tree node[3], tree atname, tree args,
                         int flags, bool *no_add_attrs)
{
  const_tree type = *node;
  /* NODE[2] may be NULL, and it also may be a PARM_DECL for function
     pointers.  */
  const_tree fndecl = ((node[2] && TREE_CODE (node[2]) == FUNCTION_DECL)
                       ? node[2] : NULL_TREE);
  function_format_info info;

#ifdef TARGET_FORMAT_TYPES
  /* If the target provides additional format types, we need to
     add them to FORMAT_TYPES at first use.  */
  if (!dynamic_format_types)
    {
      dynamic_format_types = XNEWVEC (format_kind_info,
                                      n_format_types + TARGET_N_FORMAT_TYPES);
      memcpy (dynamic_format_types, format_types_orig,
              sizeof (format_types_orig));
      memcpy (&dynamic_format_types[n_format_types], TARGET_FORMAT_TYPES,
              TARGET_N_FORMAT_TYPES * sizeof (dynamic_format_types[0]));

      format_types = dynamic_format_types;
      /* Provide a reference for the first potential external type.  */
      first_target_format_type = n_format_types;
      n_format_types += TARGET_N_FORMAT_TYPES;
    }
#endif

  /* Canonicalize name of format function.  */
  if (TREE_CODE (TREE_VALUE (args)) == IDENTIFIER_NODE)
    TREE_VALUE (args) = canonicalize_attr_name (TREE_VALUE (args));
```

Figure 2.2: Example of use of preprocessor directive in source code.

In Figure 2.2, the code between the #ifdef and #endif directives will only be compiled if the macro TARGET_FORMAT_TYPES is defined, making the code use additional format types if provided. Otherwise, the code in the red frame will not be compiled (hence, never executed).

In many systems, this mechanism expresses variations that cannot or should not be handled at run-time in configurable systems. For example, preprocessor directives and conditional compilation support specific features by enabling/disabling them. Another example would be the support for different hardware platforms when a part of the code needs to be written differently for other hardware platforms.

2.2 Motivating Example

Preprocessor directives, such as ifdefs (#ifdef, #ifndef, #elif, and #if), are long said to be undesirable in source code [MRB+17]. Since these directives are often spread across the entire code base of configurable systems, they clutter source code, hinder program comprehension, and, consequently, complicate maintenance. Code surrounded by preprocessor directives with conditional compilation relates code fragments to corresponding variabilities. When maintaining the variabilities of the system, each related extension is a potential code fragment that has to be maintained.

Therefore, in this context, we can conclude that not all developers can keep track of the two language layers to maintain variable code. In other words, developers that never touched variable code may have difficulty during a maintenance or code review task understanding, for instance, how variable code may interact with mandatory code in some configurations (combination of variabilities). This mental load for understanding preprocessor-based configurable systems can cause an increase in maintenance effort.

Existing expertise-related metrics such as DOA [FMMH+14] and Ownership [RD11] are commonly used to indicate developers with expertise in code elements. As aforementioned, these metrics are frequently used to plan division of work and identify key collaborators. Thus, these metrics are natural candidates to be used in configurable systems. The problem is that we do not know whether the developers indicated by these metrics are the most recommended developers when a maintenance or code review task involves variabilities. We argue that existing expertise-related metrics that commonly use files as the unit of abstraction may wrongly indicate key developers in the context of configurable systems. Configurable systems use variability as the unit of abstraction. This misalignment between files and variabilities may be the root factor of the indication of a key developer who does not know about variable code under maintenance or review.

Figure 2.3 illustrates a file XPTO.c with two variabilities (Transaction Management and Security) and the mandatory code. We also note that eight developers contributed to the file over time. Of these eight developers, expertise-related metrics such as DOA and Ownership indicate three of them as key developers of the file (the ones highlighted in red). However, we also note that none contributed with variable code during the entire development history.

Expertise-related metrics consider some properties to indicate key developers, such as the number of commits and the first author of the file. However, to the best of our knowledge, none of them consider properties related to variabilities implemented with preprocessor directives. Therefore, one of our goals

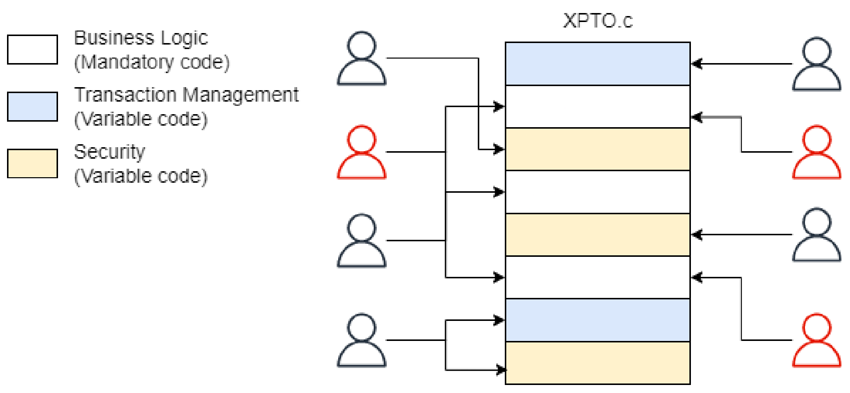


Figure 2.3: Illustration of key developers in a configurable system file.

in this work is to explore whether this type of situation occurs. In the next section, we present an exploratory study addressing this context.

CHAPTER 3

Exploratory Study

Communities working with preprocessor-based configurable systems lack guidance on managing code expertise among their developers. We argue that an empirical body of knowledge on how measures related to expertise behave in pre-processed configurable systems could mitigate this lack. In this way, we investigate the evolutionary history of 49 systems. Our main goal is to interpret expertise-related parameters. Mainly, we are interested in parameters linked to variable code. We also check whether well-known expertise metrics (Degree of Authorship [FMMH+14] and Ownership [RD11]) based on file level and from git-based version control systems are aligned to measures extracted from variable code. It is important to highlight that although the selected expertise-based metrics do not represent an exhaustive list of approaches to recommend software maintainers and code reviewers, they cover the key concepts adopted by most of them [TMHI16, BNM+11, HPSG16, JZM+17, RD11].

3.1 Research Questions

To achieve the primary goal of this study, we follow three research questions:

RQ1: How is the division of work between developers in terms of variable and mandatory code?

Rationale: In a collaborative setup, a considerable number of developers must know the system's source code. In the context of configurable systems, developers must have expertise in mandatory and variable parts of the source code since variable parts are tangled and scattered throughout the code. Otherwise, the list of developers who are candidates to perform a maintenance task or code review task may be reduced due to the developers' lack of expertise in essential parts of the source code, especially in parts of variable code.

RQ2: How is the distribution of variable code among developers?

Rationale: In configurable systems, it is essential that developers understand the variable parts of source code, as their combinations will generate the final products of a configurable system. So that there is no work overload on a few developers when there are, for example, maintenance tasks or code review tasks, there must be an equal distribution of variability among developers who have expertise in variable parts of code. Otherwise, few developers will be responsible for tasks involving variability, and consequently, the maintenance effort of these systems tends to increase.

RQ3: Is a developer with a higher file level expertise more likely to be associated with variable code?

Rationale: Expertise-related metrics use the file as the unit of abstraction to indicate source code authorship and source code ownership. However, in configurable systems, the unit of abstraction is not necessarily aligned with the file(s) in which it is implemented. That is, variability can be tangled with other variabilities as well as mandatory code in the same file. In addition, such variability can be scattered across different files. As one of the resources used to indicate developers to perform maintenance tasks and to suggest code reviewers in pull requests is the use of expertise-related metrics (e.g., DOA and Ownership), it is essential to understand whether developers indicated by these metrics do have expertise in variable codes within the files they are experts on. Otherwise, a developer's recommendation may not be the best when it comes to configurable systems.

3.2 Subject Systems

This study analyzes code expertise-related measures in real-world open-source configurable systems. We collected 63 preprocessor-based systems implemented in C, hosted on GitHub, and used in several studies in literature [KDP16, AMS+18, LAL+10, MRG13, MRG+17, RRM+16]. We restrict our analysis to 49 systems with more than 10 developers and 50 variabilities to filter out uninteresting systems to our scope.

Table 3.1 presents an overview of the analyzed systems in terms of number of files (Files), number of developers (Developers), number of commits (Commits), and number of variabilities (Variabilities). The most popular project is Curl (25,900 stars), while the most forked is OpenSSL (8,300 forks). The commits range from 1274 (Cherokee) to 107,049 (GCC), while contributors range from 18 (Cherokee) to 1363 (GCC). The selected systems cover distinct domains, such as graphics servers, antivirus, and utility tools.

Project	**Files (k)**	**Developers**	**Commits (k)**	**Variabilities (k)**
AMXModX	0.70	53	2.24	1.98
Angband	0.31	74	7.00	0.26
ASF	2.96	23	6.50	0.85
Bison	0.12	32	2.96	0.42
BusyBox	1.000	345	13.70	2.64
Cherokee	0.22	18	2.13	0.25
Clamav	3.36	71	7.90	4.05
Collectd	0.38	452	6.50	0.539
Curl	0.75	606	13.09	1.73
Dia	0.46	70	3.64	0.20
Emacs	0.48	334	32.81	3.22
Ethersex	0.66	95	3.33	1.29
FreeRADIUS	0.63	120	26.95	1.01
FVWM	0.23	27	3.73	0.46
GCC	48.78	1363	107.04	10.13
Glibc	7.16	249	16.37	3.53
Gnumeric	0.74	141	15.65	0.55
Gnuplot	0.14	114	6.05	0.99
Hexchat	0.16	131	1.86	0.16
Httpd	1.27	96	15.10	1.59
Irsii	0.24	88	3.85	0.09
Kerberos5	2.28	108	11.36	2.23
Libexpat	0.04	42	1.59	0.11
Libpng	0.08	31	2.85	0.78
LibSoup	0.18	138	1.82	0.08
Libssh	0.21	107	4.23	0.19
LibXML2	0.13	194	3.59	0.67
Lighttpd1.4	0.16	27	3.03	0.55
Machinekit	0.72	70	3.78	0.57
MapServer	0.31	113	7.93	0.54
Marlin	2.99	687	9.38	3.41
Mongo	0.720	495	37.90	3.21
OpenSC	0.32	181	6.26	0.38
OpenSSL	1.97	509	17.38	2.20
OpenTX	1.07	111	9.10	2.36
OpenVPN	0.14	114	1.97	0.48
OSSEC	0.34	85	2.46	0.40
Pacemaker	0.37	93	12.25	0.20
Parrot	0.32	126	12.70	0.54
Pidgin	1.00	315	25.61	0.60
RetroArch	2.76	401	41.79	3.84
SleuthKit	0.40	73	2.46	0.85
SQLite	0.38	27	16.29	1.34
syslog-ng	0.84	110	6.84	0.25
TauLabs	2.35	136	9.71	1.43
Totem	0.20	119	4.36	0.12
Uwsgi	0.24	273	5.06	0.36
WiredTiger	0.50	49	13.85	0.16
XServer	1.68	554	13.54	1.91
Average	**2.75**	**199**	**11.96**	**1.34**

Table 3.1: Overview of the subject systems.

3.3 Degree of Authorship (DOA)

One way to quantify developer expertise is to use the so-called *degree-of-authorship (DOA)* [FMMH+14]. The DOA for a file is defined in absolute terms as follows:

$$DOA(c,f) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * ln(1 + AC) \tag{3.1}$$

The DOA of a contributor c in a file f is calculated based on three parameters. The first authorship (FA) is a binary parameter related to the creation of f by c. If d creates f, FA is 1; otherwise, it is 0. The second parameter is the number of deliveries (DL) which represents the number of changes in f made by c. Finally, the number of acceptances (AC) is the number of changes in f made by other contributors than c. It is worth mentioning that our study relies on the analysis of each commit. Therefore, DL and AC values are based on the number of changes in each commit analyzed.

For the sake of simplicity, we calculate and use hereafter in this work the normalized DOA as given in Avelino et al. [APHV19]:

$$DOA_N(c,f) = DOA(c,f)/max(\{DOA(c',f) \mid c' \in changed(f)\}) \tag{3.2}$$

In the normalized DOA, *changed(f)* denotes the contributors who created or edited a file f up to a commit of interest. Therefore, $DOA_N \in [0..1]$, and values close to 1 are granted to the contributors with the highest absolute DOAs among the contributors of a file.

Based on the normalized DOA, the set of authors of a file f is calculated as given in Avelino et al. [APHV19]:

$$authors(f) = \{c \mid c \in changed(f) \wedge DOA_N(c,f) > 0.75 \wedge DOA(c,f) \geq 3.293\} \tag{3.3}$$

The interpretation of DOA results depends on specific thresholds. This work uses 0.75 and 3.293 as thresholds to normalized DOA and DOA, respectively. In other words, a developer who achieves a value higher than the aforementioned is considered an author of the file. Otherwise, this developer is viewed as a contributor. We stem those thresholds from previous work that used DOA [APHV19].

3.4 Ownership

Ownership is an expertise metric used to describe whether one person has responsibility for a software component or if there is no one responsible developer. To calculate the ownership of a developer for a particular file, one should consider the ratio of the number of commits a developer has made relative to the total number of commits for that file [RD11].

The interpretation of ownership results also depends on specific thresholds. In this work, following the threshold suggested by Bird et al. [BNM+11], a developer who has made changes to a file and whose ownership value is below 5% is considered a minor contributor to that file. A developer who has made changes to a file and whose ownership is at or above 5% is a major contributor to the file.

3.5 The Data Collection Procedure

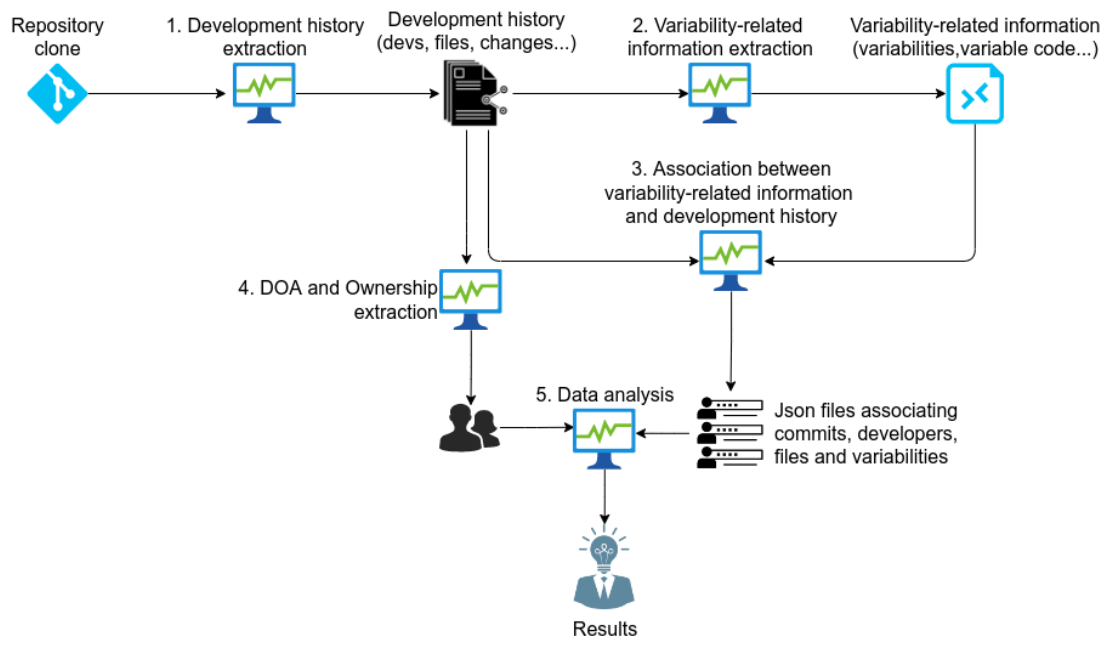

Repository clone
1. Development history extraction
Development history (devs, files, changes...)
2. Variability-related information extraction
Variability-related information (variabilities,variable code...)
3. Association between variability-related information and development history
4. DOA and Ownership extraction
5. Data analysis
Json files associating commits, developers, files and variabilities
Results

Figure 3.1: Procedure used to collect data for the exploratory study.

In this section, we describe the four main steps to collect the data used to answer the research questions in Section 3.1. Figure 3.1 illustrates the steps described below and the data evaluation procedure is presented in Section 3.6. Our source code, data, and supplementary material are available online[1].

1. Development history extraction. We extracted the development history from the repositories using a script built using pydriller [SAB18], which is a Python framework that helps developers in analyzing Git repositories. We processed each commit, extracting three pieces of information: (i) the file path; (ii) the developer who performed the change; and (iii) the type of the change—addition, modification, or deletion. Additionally, we discarded files that did not contain source code (e.g., images, documentation).

2. Variability-related information extraction. After having the development history and source code files, we modified a version of the tool pypreprocessor (a c-style macro preprocessor written in Python) to extract program elements and lines that were variable code. In other words, we identified parts of the source code that compose variabilities. We must note that we discarded files that did not contain variable code.

[1]https://www.facom.ufms.br/~cafeo/ist2022/

3. Association between variability-related information and development history. With all relevant information about development history, including lines of code changed by each developer and information about variabilities, it was possible to associate developers with variable code. Our goal was that, instead of having only an association between a developer and files changed (committed) by that developer, we also have another unit of abstraction (variability) associated with developers. In this case, we could identify developers who added, changed, and removed variabilities during the development history.

4. DOA and Ownership extraction. This step consisted in calculating the expertise-related metrics DOA and Ownership. To do so, we developed scripts that processed all files considering the entire development history of all subject systems. After processing the source code files, we generated a file containing the expertise-related value for every developer in every source code file.

3.6 The Data Evaluation Procedure

This section describes our evaluation procedures to answer the research questions presented in Section 3.1.

3.6.1 Work specialization (RQ1)

To understand the work specialization between developers in terms of variable and mandatory code, we produced, for each month, a cumulative list of all developers that contributed to an analyzed system. The idea was to identify whether these developers worked on variable, mandatory code, or both. Each month we updated this list, tracking which developers worked in which parts of the source code (mandatory and/or variable). To categorize developers, we used the terms *generalist* to developers who only changed mandatory code considering the entire development history; *specialist* to developers who only changed variable code considering the whole development history; and *mixed* to developers who changed both variable and mandatory code.

3.6.2 Distribution of work among developers (RQ2)

To address RQ2, we applied data aggregation using concentration statistics [Gas72]. The idea is to analyze the equality of the distribution of work between developers on variable code. This analysis allows us to make statements such as "10% of the developers are responsible for over 70% of the variable code".

As a statistic concentration, we adopt a method to analyze and visualize income inequalities in a country's population called Lorenz inequality or Lorenz curve [Gas72]. This paper uses it to explore the concentration of changes in variabilities per developer. For a more in-depth description of Lorenz inequality, the reader may refer to the original work of Lorenz [Gas72]. To compare the Lorenz concentration between different subject systems, we compress it to one number named Gini coefficient [Dor79]. The Gini coefficient indicates the degree of distributional inequality of variability changes per developer in a subject system. The Gini coefficient takes a value between 0 and 1, with $g = 0$ denoting perfect equality, meaning that x percent of the developers are responsible for x percent of the variabilities changes. Conversely, $g = 1$ indicates perfect inequality with only one developer responsible for 100 percent of the cumulative changes in variabilities. Using the Gini coefficient, we can compare different concentrations since such coefficient represents a concentration in a scalar value.

3.6.3 *Association between file expertise and variable code (RQ3)*

To answer RQ3, we first extracted expertise-related metrics to identify authors (DOA) and major contributors (Ownership) of all files of the analyzed systems. In addition, to each file analyzed, we identified a list of developers who changed variable code in this specific file. After that, we investigated whether authors (DOA) and/or major contributors (Ownership) were on the list of developers who changed variable code.

CHAPTER

4

Results

This Chapter presents the results of the research questions raised in Section 3.1. Section 4.1 shows the results regarding the division of work of developers in configurable systems. We classify them as generalists, specialists, and mixed developers to analyze this division. Section 4.2 describes the results of the distribution of work among developers by using, for example, concentration statistics. Finally, Section 4.3 shows the results of the comparison between file experts and variable code developers.

Table 4.1 shows the results extracted from the data to answer the research questions (Section 3.1).

4.1 Work specialization (RQ1)

To assess work specialization, we introduce three developer profiles. We classify a developer as a generalist if (s)he only worked in mandatory code considering the entire development history. A developer is classified as a specialist if (s)he only worked with variable code. A developer is classified as mixed if (s)he worked both in mandatory and variable code considering the entire development history.

Table 4.1 shows the percentage of generalists, specialists, and mixed developers for each subject system in the last commit analyzed. Proportionally, any system has at least 25% of generalist developers, with an average of 61.53% and a maximum of 82.30%. No more than 2.75% are specialists on average. Considering mixed developers, we can note an average of 35.71% of mixed developers. However, we notice that the proportion of generalists, specialists, and mixed developers appears to vary across the entire data set of systems.

Project	**RQ1**			**RQ2**	**RQ3**			
	Specialists	**Generalists**	**Mixed**	**Gini**	**Authors who never touched variabilities (%)**	**Majors who never touched variabilities (%)**	**Variabilities that have never been changed by Authors (%)**	**Variabilities that have never been changed by Majors (%)**
AMXModX	0.00	64.40	35.59	0.70	35.84	54.72	1.50	0.00
Angband	1.19	66.66	32.14	0.71	32.43	54.05	11.56	2.98
ASF	0.00	53.12	46.87	0.81	78.26	91.3	3.38	0.58
Bison	0.00	66.66	33.33	0.74	31.25	40.63	14.36	3.79
BusyBox	2.15	76.81	21.02	0.88	15.94	24.06	19.23	2.72
Cherokee	0.00	69.56	30.43	0.78	44.44	44.44	5.88	2.26
Clamav	1.26	67.08	31.64	0.81	23.94	36.62	3.42	0.19
Collectd	1.88	77.61	20.50	0.76	21.01	29.87	28.38	10.76
Curl	2.95	62.83	34.21	0.84	11.55	14.52	28.15	13.58
Dia	10.25	55.12	34.61	0.73	24.28	34.29	16.09	1.95
Emacs	1.31	62.10	36.57	0.81	9.88	13.77	37.47	14.29
Ethersex	3.73	37.38	58.87	0.80	33.68	40.00	12.51	0.46
FreeRADIUS	2.25	56.39	41.35	0.84	17.50	25.00	11.19	1.37
FVWM	0.00	25.00	75.00	0.59	51.85	77.78	16.30	4.78
GCC	2.33	63.15	34.51	0.82	48.49	55.39	31.36	10.86
Glibc	4.23	60.91	34.85	0.89	42.16	59.44	13.46	1.35
Gnumeric	1.84	74.23	23.92	0.86	26.95	33.33	7.16	0.35
Gnuplot	0.00	65.32	34.67	0.86	7.01	11.40	20.33	3.49
Hexchat	4.44	72.59	22.96	0.80	6.87	20.61	19.51	1.82
Httpd	2.54	37.28	60.16	0.75	53.12	61.45	16.69	6.25
Irsii	1.05	81.05	17.89	0.71	15.90	39.77	17.70	9.37
Kerberos5	3.96	52.38	43.65	0.80	46.29	58.33	19.23	1.43
Libexpat	6.97	60.46	32.55	0.61	14.28	16.67	24.32	3.60
Libpng	6.25	46.87	46.87	0.83	9.67	19.35	4.18	1.39
LibSoup	5.26	73.02	21.71	0.72	13.04	17.39	9.09	2.27
Libssh	2.41	67.74	29.83	0.75	13.08	22.43	14.50	4.50
LibXML2	8.04	73.86	18.09	0.81	4.12	5.15	18.28	5.16
Lighttpd1.4	10.71	39.28	50.00	0.82	18.51	37.04	7.73	0.54
Machinekit	1.23	53.08	45.67	0.75	34.28	47.14	6.08	0.52
MapServer	0.00	63.11	36.88	0.77	19.46	20.35	11.07	0.92
Marlin	7.67	53.83	38.49	0.83	18.34	26.35	17.59	8.38
Mongo	0.83	72.12	27.04	0.88	51.91	64.24	4.88	0.65
OpenSC	1.99	67.66	30.34	0.68	15.47	30.39	37.69	6.02
OpenSSL	3.92	71.47	24.59	0.87	12.96	20.24	17.86	3.13
OpenTX	4.06	42.27	53.65	0.85	34.23	37.84	11.10	1.43
OpenVPN	2.54	62.71	34.74	0.77	10.52	12.28	23.90	3.53
OSSEC	1.03	62.88	36.08	0.79	12.94	32.94	18.61	11.66
Pacemaker	3.60	70.27	26.12	0.75	21.50	25.81	15.45	2.41
Parrot	0.63	54.43	44.93	0.63	45.23	66.67	20.70	11.27
Pidgin	0.82	71.15	28.02	0.70	22.54	31.11	26.15	7.78
RetroArch	4.22	53.33	42.44	0.89	17.70	21.7	6.26	2.52
SleuthKit	1.17	70.58	28.23	0.89	28.76	38.36	1.17	0.35
SQLite	0.00	43.33	56.66	0.77	22.22	18.52	11.49	2.38
syslog-ng	3.87	69.76	26.35	0.75	40.00	60.91	20.40	2.40
TauLabs	1.87	50.00	48.12	0.73	51.47	70.59	8.14	0.83
Totem	2.30	82.30	15.38	0.77	19.32	27.73	5.73	0.00
Uwsgi	2.72	63.60	33.67	0.71	8.42	19.05	36.85	13.00
WiredTiger	0.00	61.66	38.33	0.70	48.98	65.31	10.71	4.16
XServer	3.20	66.40	30.40	0.85	11.37	15.16	19.70	4.79
Average	**2.75**	**61.53**	**35.71**	**0.78**	**26.51**	**36.56**	**15.60**	**4.09**

Table 4.1: Summary of results.

These proportions naturally depend on characteristics like the number of variabilities and developers. Nevertheless, we can say that specialists are always the minority in all subject systems.

Since the subject systems present a wide range of systems in terms of the number of developers and variability, we chose four systems representative of the data of the entire system set to perform a more in-depth analysis of this division over time. Figure 4.1 presents the work specialization over time considering the following systems: GCC, Ethersex, LibSoup, and Machinekit.

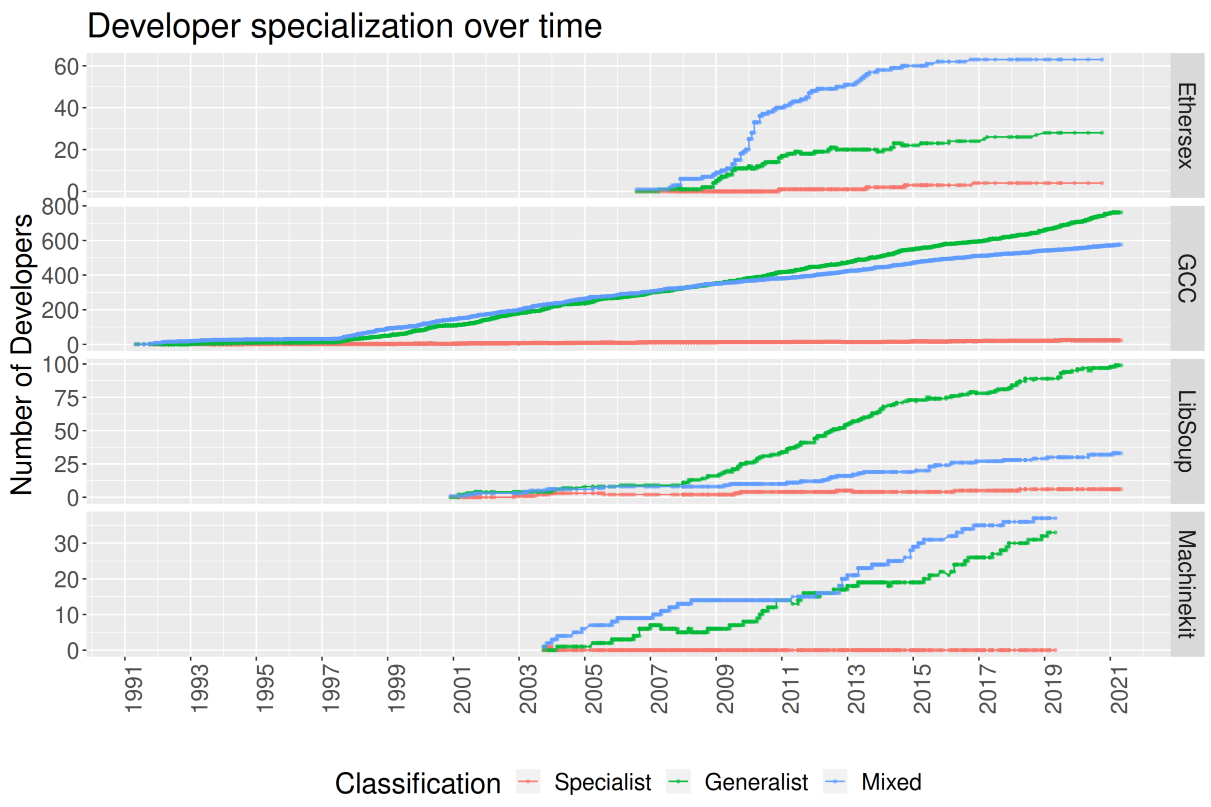


Figure 4.1: Work specialization over time in four systems.

Analyzing the graphs of the four systems (Figure 4.1), we can see a growing trend in the number of developers. We can also notice that the growth of generalist and mixed developers is much higher than that of specialists in all four systems. It is interesting to note that Ethersex and Machinekit have more mixed developers than generalists. Both systems also have fewer developers compared to GCC and LibSoup, which have more generalists. That is, there seems to be a tendency for developers to end up working more on mandatory code than on variable code.

Considering the results of all subject systems and the four analyzed systems, depending on the division of variable code between mixed and expert developers, the number of developers who can perform tasks in variable code may be small. If this division is not equal, a few developers can have a concentration of variable code knowledge, which is harmful to any system, including configurable systems.

Specialization of work like the one we saw in the results was expected. However, the considerable difference in relative terms between these divisions can be a problem in many of the analyzed systems.

Lessons Learned 1: Specialization among developers of configurable systems is uneven when considering mandatory and variable code. Specialization is more even with few developers at the beginning of the development history. Still, as the system evolves, there is a tendency for new developers to focus more on mandatory code.

4.2 Distribution of work among developers (RQ2)

As mentioned in Section 4.1, it is essential to understand how variability is divided among developers working with variability. There are a small number of specialist developers and many mixed developers. Therefore, this division can be pretty uneven.

In a first analysis, it is possible to notice in the violin chart with boxplot (Figure 4.2) the division of variabilities by developer (mixed or specialist) in the four analyzed systems with representatives of the subject systems (GCC, Ethersex, LibSoup , and Machinekit).

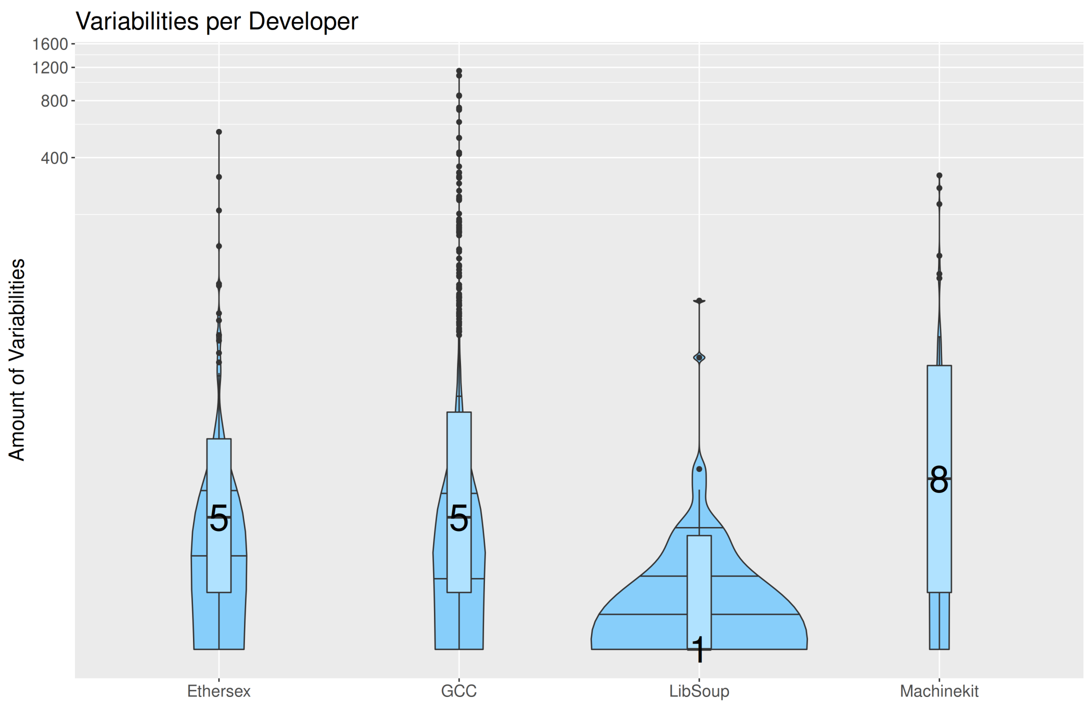


Figure 4.2: Variabilities per developer.

In Figure 4.2, it is noted that there is a relatively equal distribution among developers for most developers, with a median of 5 variabilities per developer in the Ethersex and GCC systems; 8 variabilities per developer in Machinekit; and one variability per developer in LibSoup.

Despite the apparent equality of variabilities between developers, there are several outliers in all systems, which may indicate a concentration of variability in a few developers. Table 4.1 shows the Gini coefficient for each subject system. The Gini coefficient takes a value between 0 and 1, with 0 denoting perfect equality and 1 denoting perfect inequality.

We can notice that there is considerable inequality in all subject systems. In other words, few developers worked on variable code during the entire development history. This concentration limits developers who have expertise in variable code and, therefore, who are best suited to perform maintenance tasks and code reviews in variabilities.

We can see this concentration when we plot the Lorenz curve for the four representative systems (Figure 4.3). The Lorenz curve is complementary to the Gini coefficient to represent inequality. The black line represents complete equality in the distribution of variabilities by developers. The further the curve is from the line of equality, the more uneven the distribution. The Lorenz curves of the four systems corroborate with what was found in all subject systems and with the findings mentioned above.

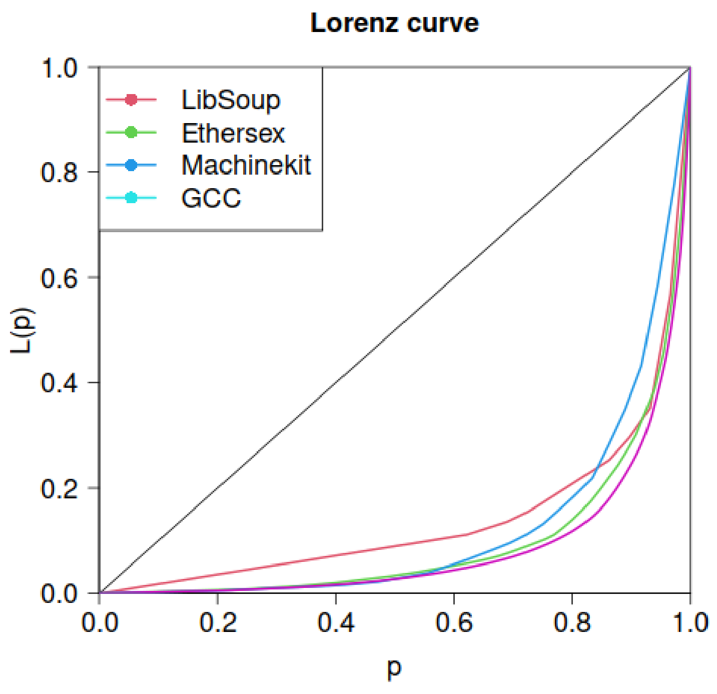


Figure 4.3: Lorenz curve of the four representative systems.

Lessons Learned 2: Few developers accumulate considerable expertise over system variabilities. Only a small portion of developers working with variable code accumulate the most changes over their entire development history.

4.3 Association between file expertise and variable code (RQ3)

According to the results of RQ2 (Section 4.2), few developers know most of the variable code. Expertise-related metrics try to identify developers with expertise in parts of the code to assign them to specific tasks in those parts. However, expertise-related metrics known in the literature do not consider variability to indicate developers.

Table 4.1 shows the percentage of developers nominated by DOA and Ownership who have never worked on variable code. In addition, it is shown the rate of variability that has never been changed by the developers indicated by such metrics.

It is possible to notice from the data presented in Table 4.1 that a considerable number of developers indicated by the expertise-related metrics never work on any variability of the system, even though they are experts developers in the file in which the implementation of one or more variabilities is found. It is also possible to notice the low participation of these developers in the implementation of most of the variability. A lot of variabilities were never changed by such developers. Therefore, considering the analyzed systems, it is possible to say that the analyzed expertise-related metrics may not be right in the indication of developers' expertise when it comes to configurable systems.

Lessons Learned 3: Expertise-related metrics may not correctly indicate the expertise of developers in most cases when configurable systems, even when variabilities are implemented in the file in which the developer is an expert.

CHAPTER 5

Implications

Be average at everything = be good at nothing. The data from our analysis showed that there are a lot of developers who either work with only the mandatory part of the code (generalist) or with mandatory parts and few variable parts (mixed developer). In configurable systems, it is of paramount importance that developers know variable code, as they are tangled with mandatory code. Furthermore, combining these variable codes will generate different final system configurations. Due to the mental load to understand combinations of variabilities, developers tend to avoid changing variable code. Thus, it is likely that there are few developers able to maintain or perform variable code review tasks.

Knowledge is power. Sharing knowledge is the key to unlocking that power. Our data showed that in addition to the fact that few developers work with variable code, only a few work with most of the variabilities of the systems. Several studies claim that the concentration of knowledge is harmful to any software system [APHV16, RMT11]. It can be even more critical in configurable systems, given that essential parts of the system (i.e., variable code) are concentrated in a few developers. These developers may have a considerable backlog of maintenance tasks and code reviews. As a result, the time for carrying out these activities may be compromised. An alternative would be to randomly assign such activities to developers with no knowledge of variable code. In this case, the maintenance effort is likely more significant than usual. Thus, it is essential to try to share knowledge of variable code and introduce developers, especially the new ones, to familiarize themselves with variable code to reduce the concentration of knowledge and increase the number of developers capable of working with variability.

Wrong does not cease to be wrong because the majority share in it. Expertise-related metrics are extensively used to plan the overall division of work, identify key collaborators, and find the best developers for a target task. However, in configurable systems, it is natural that variabilities are used as units of abstraction, for example, in maintenance tasks. Expertise-related metrics generally use the file as a modular unit for indicating developers. Our results showed that this misalignment between the unit of abstraction used by configurable systems and the unit used by expertise-related metrics causes such metrics to be wrong in the indication of developers if this indication is for the performance of a task that involves variability. Metrics used in preprocessor-based configurable systems must be variability-aware so that suggestions are more likely to be correct regarding variable code.

CHAPTER 6

Proposed Approach

Since findings from the exploratory study (Chapter 3) surges that (i) the minority of the developers have touched variable code considering the entire development history and (ii) there is a concentration of expertise about variable code among few developers, we exploit such information in our proposed approach. In our approach, we analyze important properties of source code repositories, developers, and source code to propose a formula that can indicate the expertise level of a particular developer regarding a specific variability. Using this formula, we can generate a list of expert developers for a particular variability that may be reviewer candidates. Our goal is to support the choice of an appropriate reviewer for a pull request involving variabilities. This section describes the concepts, steps, and details used in the definition of the proposed approach.

6.1 Feature Selection

The first step of our approach is to understand which properties can be used as indicators of expertise of developers regarding variabilities. The idea is to use these indicators to propose a formula that calculates the level of authorship of a particular developer considering a specific variability. To do so, we use the well-known machine learning and statistics technique called feature selection.

Feature selection is the process of detecting relevant features and removing irrelevant, redundant, or noisy data for use in model construction [KM14]. This process improves accuracy while accelerating the training of the algorithms by removing features that do not provide helpful information or do not provide more information than the currently selected features [DL97, GE03, XZBY15, FPHK94].

The features selection algorithms are often divided into three groups: Filter approaches, Wrapper approaches, and Embedded approaches [XZBY15, KM14].

Filter approaches. Filter approaches are computationally efficient methods that evaluate features without a learning algorithm, applying an independent measure to each subset of features. The filter approach allows the algorithms to have a straightforward structure, with a straightforward search strategy and a feature evaluation criterion [KM14, LMSZ10].

Wrapper approaches. The main difference between the filter and wrapper approaches resides in the evaluation criteria. The wrapper approach uses a learning algorithm to evaluate each subset of features and identify relevant features [KM14, LMSZ10].

Embedded approaches. Embedded approach incorporate the learning algorithm, with lower computational cost than the wrapper approach, with the search element from the filter approach, by searching for the optimal feature subset for a known cardinality. Then, it uses the learning algorithm to select the optimal subset among the optimal subsets across different cardinalities [KM14, LMSZ10].

In our solution, to find the features that best fit the problem, we applied one method from each group using the implementation given by the tool scikit-learn [1]. The following methods were applied to select relevant features:

Univariate Selection (US). Univariate Selection is a feature selection method that selects the k highest scoring features based on univariate statistical tests, comparing each feature with the dependent variable [JBB15]. For regression, the only test available is the cross-correlation test that generates an F-score, which is recommended as a feature selection criterion to identify potentially predictive features and the mutual information gain test that measures the dependence between features using entropy and k-nearest neighbors [JBB15].

Recursive Feature Elimination (RFE). Recursive Feature Elimination is a feature selection method that starts with all the features and iteratively removes those with the weakest relationship with the output until the desired number of features is reached [CJ07].

Decision Tree (DT). The Decision Tree is a graph where every node represents the event or choice, and the edges represent the decision rules or conditions. With this structure, the graph ends up taking the form of a tree, where, for each decision made, one level is descended in the tree [DBW$^+$02].

[1] https://scikit-learn.org/

We selected candidate features based on properties used in the studies proposing DOA and Ownership. However, since these expertise-related metrics use files as the unit to calculate the expertise, we used to adapt these properties to consider the variability as the unit of interest in our approach. Moreover, we also included properties related to files in the feature selection process that are relevant to defining the boundaries of a variability. Table 6.1 shows all the candidate features and a brief description of these features.

Feature	**Description**
fa_variability	First author of the variability
n_commits_variability	Number of commits involving a particular variability
dl_variability	Number of commits done by a developer touching a particular variability
ac_variability	Number of commits done by others than a particular developer touching a specific variability
n_files_variability	Number of files used to implement a particular variability
n_commits_files_variability	Number of commits involving files with code of a particular variability
fa_files_variability	First author of a file that contains code of a particular variability
n_r_commit_variability	Relative number of commits of a developer considering the total number of commits in a variability

Table 6.1: Features analyzed.

We executed scripts to analyze the development history of all subject systems (Section 3.2) and calculate the values of the candidate features. With these values, we applied the three methods mentioned earlier to select the relevant features to our approach based on developers who changed variabilities. Since we used three methods, our criteria to select a feature was that a feature should be selected by the majority of the methods (i.e., at least two methods) to be considered in our approach. Table 6.2 shows the results of the feature selection for each method. The value True means that a feature was selected by a method. The value False means that a feature was not selected by a method. The features considered to be used in our approach (two or three true values) are represented in bold.

6.2 Linear Regression

With relevant features, the next step was to use them to create a formula that could indicate the expertise of a developer regarding a specific variability. One of the most used techniques to do so in related studies is linear regression.

Linear Regression can be used to predict, or classify, a set of labeled data.

Feature	**US**	**RFE**	**DT**
fa_variability	TRUE	TRUE	FALSE
n_commits_variability	TRUE	FALSE	TRUE
dl_variability	TRUE	TRUE	TRUE
ac_variability	TRUE	TRUE	TRUE
n_files_variability	TRUE	FALSE	TRUE
n_commits_files_variability	FALSE	FALSE	FALSE
fa_files_variability	FALSE	TRUE	FALSE
n_r_commit_variability	FALSE	TRUE	FALSE

Table 6.2: Results of the application of three feature selection methods.

This labeled data, composed of input (x) and predicted output (y), is used to find a function that relates the input to the predicted output [MA20]. The function that can infer the relationship between the dependent variable y and the independent variable x can be defined as:

$$y = \beta_0 + \beta_1 * x + \varepsilon \tag{6.1}$$

with β_0, β_1 and ε as constants. Linear regression provides a sloped straight line representing the relationship between the variables (Figure 6.1).

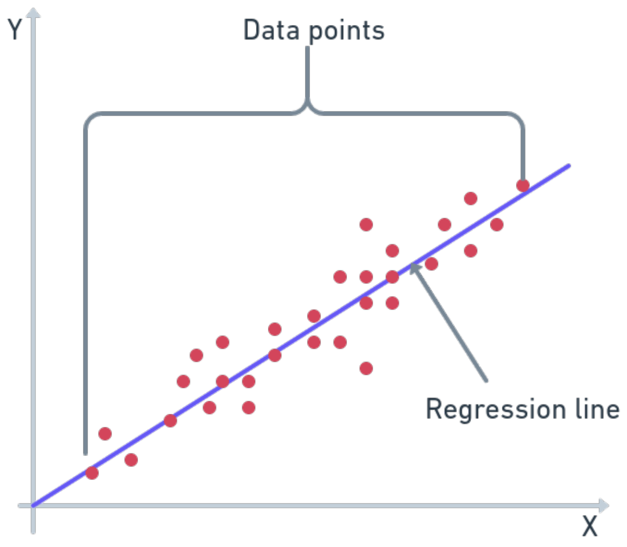


Figure 6.1: Illustration of the process of linear regression.

If more than one independent variable is used to predict the value of the dependent variable, we can use Multiple Linear Regression (MLR). Similarly to linear regression, MLR attempts to find a linear equation that best predicts the relationship between a scalar result Y and n explanatory variables $X_1, X_2, X_3, \ldots, X_n$, based on the values of the features multiplied by the weights calculated during the regression run [Job91]. Given a sample of m observations as the matrix:

$$\begin{matrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1m} & y_1 \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2m} & y_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \cdots & x_{nm} & y_n \end{matrix}$$

We can describe this sample as $(x_{i1}, x_{i2}, x_{i3}, \ldots, x_{im}, y_i)$, for $i = 1, 2, 3, \ldots, m$ and, therefore, we can describe this as a linear model:

$$y_i = \beta_0 + \beta_1 * x_{i1} + \beta_2 * x_{i2} + \beta_3 * x_{i3} + \cdots + \beta_m * x_{im} \tag{6.2}$$

for $i = 1, 2, 3, \ldots, n$, where $\beta_0, \beta_1, \beta_2, \beta_3, \ldots, \beta_m$ are constants.

To find the best fit line, the error between the predicted values and the obtained values must be minimized. This is calculated through a cost function that will choose the best values for β_0 and β_1. The cost function can be Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE).

6.2.1 Mean Absolute Error (MAE)

Mean Absolute Error (MAE) is the average of the difference between the estimated values and the predicted values and can be described as:

$$MAE = \frac{1}{n} \sum_{j=1}^{n} |y_j - \hat{y}_j| \tag{6.3}$$

where y_j is the estimated value, $\hat{y}_j$ is the predicted value from the regression model, and m is the number of data points.

6.2.2 Mean Squared Error (MSE)

Mean Squared Error (MSE) finds the average of the squared difference between the estimated value and the value predicted by the regression model.

$$MSE = \frac{1}{n} \sum_{j=1}^{n} (y_j - \hat{y}_j)^2 \tag{6.4}$$

where y_j is the estimated value, $\hat{y}_j$ is the predicted value from the regression model, and m is the number of data points.

6.2.3 Root Mean Squared Error (RMSE)

Root Mean Squared Error (RMSE) corresponds to the square root of the average squared difference between the estimated value and the value predicted. Mathematically, it can be represented as:

$$RMSE = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2} \qquad (6.5)$$

where y_j is the estimated value, $\hat{y}_j$ is the predicted value from the regression model, and m is the number of data points.

Once the error function is defined, the execution of linear regression takes place in 2 stages: training and test. In the training stage, the data set is divided into two parts. One part of the data set is used to train the regression in a series of iterations, called epochs. At each epoch, the weights applied to each feature are updated using a previously defined fixed value, called the learning rate. The value of the cost function and the accuracy of the model are also calculated in this stage. The number of epochs and the learning rate are empirically defined values, differing for different applications. The linear regression test stage is performed at the end of the training stage. In this step, the function is applied to the other part of the data set, and the accuracy of the regression is measured. More details related to this stage are presented in Section 7.

To generate the function to calculate the authorship of variability, we used MLR with the most relevant features (Section 6.1) to create a formula to indicate the degree of authorship of a particular developer regarding a variability.

We used fa_variability as (x_1), n_commits_variability as (x_2), dl_variability as (x_3), ac_variability as (x_4), n_files_variability as (x_5), and the developers that touched the variability during the development history as the y. Multiple linear regression is suitable for our data, even though changes in variabilities are ordinal because we are attempting to find an approximation, not a certain class, for expertise. We use the following linear combination as an initial starting point to determine the *Degree-Of-Authorship-in-Variabilities (DOA_V)*:

$$DOA_V(d,v) = \alpha_0 + \beta_1 * x_1 + \beta_2 * x_2 + \beta_3 * x_3 + \beta_4 * x_4 + \beta_5 * x_5 \qquad (6.6)$$

being d a particular developer and v a particular variability.

To determine appropriate weightings for the factors contributing to our formula, the first step was to select a subset of systems to train the linear regression algorithm, the number of epochs, and the learning rate. We trained the linear regression with 20 and 32 systems, using 1000 and 2000 epochs and learning rates of 0.001, 0.0001, and 0.00001. The best solution was found when training the linear regression with 32 systems, 1000 epochs, and learning rate of 0.00001, and the resulting equation to the *Degree-Of-Authorship-in-Variabilities (DOA_V)* is as follows:

$$DOA_V(d,v) = 1.446 + 0.364 * x_1 - 0.007 * x_2 - 0.326 * x_3 + 0.633 * x_4 - 0.004 * x_5 \tag{6.7}$$

Chapter 7
Experiment

One of the most effective ways to evaluate an expertise-related metric is to consult with actual code maintainers and code reviewers assigned from a code base. We evaluate our technique using the actual commits and code reviews data of a subset of 17 from the subject system. In particular, we use 201,707 commits involving variabilities as our oracle in evaluating our approach against a popular performance metric. To further validate our findings and demonstrate its superiority, we experiment our approach using 45,955 pull requests and their code review details and compare them with the state-of-the-art expertise-related metrics. We mainly answer the following research questions through our conducted experiments:

RQ4: How does DOA_V accurately suggest maintainers of variabilities?

RQ5: Does DOA_V outperform widely used expertise-related metrics for reviewer recommendation in preprocessor-based configurable software systems?

7.1 Experimental Data Set

We used 17 projects from the subject systems (Section 3.2) for our experiments. It should be noted that the experiments involve the actual recommendation of code reviewers, and the choice of the same systems does not impact the evaluation since there is no direct relationship with the exploratory study. Instead, choosing the same systems confirms the findings of the exploratory study and vice versa.

We identify the developers who added, changed, or deleted variable code from the development history and are appended to the maintainer set of the variability. We use such a set as the *gold maintainer set* for each of the variabilities in our experiment to answer RQ4. In addition, besides some constraints the subset of subject systems met as mentioned in Section 3.2, they also have explicit reviewers identified in pull requests. In other words, these systems present pull requests with code reviewers assigned explicitly in a GitHub feature to assign code reviewers. We collect developer references (i.e., recommended reviewers) and make a reviewer set for each variability touched, considering all pull requests. We use such a set as the *gold reviewer set* for each of the variabilities in our experiment to answer RQ5. Table 3.1 shows the summary statistics of the selected projects (in bold).

7.2 Performance Metrics

Since our technique recommends a list of developers with expertise regarding a particular variability, we choose two relevant performance metrics for evaluation from the corresponding literature [FMMH+14, BNM+11]. These metrics were adapted to the context of each research question. We also choose these metrics from the information retrieval domain due to the inclination of this technique to this domain.

Maintainer Accuracy: It refers to the percentage of developers for which at least one maintainer is correctly recommended within the results given by DOA_V considering the gold maintainer set of each variability. $M_{Accuracy}(V)$ can be defined as follows:

$$M_{Accuracy}(V) = \left(\frac{\sum_{variabilities \in V} isCorrect(variabilities, Maintainers)}{|V|} * 100 \right) \% \tag{7.1}$$

$isCorrect(variabilities, Maintainers)$ returns a value of 1 if at least one developer exists from the gold set in the Maintainers results generated by DOA_V and returns 0 otherwise. V denotes the set of all variabilities of a particular system. The higher the accuracy, the better our approach. It is worth mentioning that the list of Maintainers depends on a cut-off based on the value of DOA_V.

Jaccard distance of reviewers: It refers to the dissimilarity between the gold reviewers set and the indicated reviewers set given by DOA_V. $d_J(A,B)$ can be defined as follows:

$$d_J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{7.2}$$

Here, A is the list of reviewers indicated by our approach with a predefined cut-off. B is the gold reviewer's list. The value of Jaccard distance is bound between 0 and 1. Values closer to 1 mean a higher dissimilarity between sets. Values closer to 0 mean lower dissimilarity between sets. The lower the distance, the better our approach compared to other approaches. It is worth mentioning that we used cut-off values in the compared expertise-related metrics that were recommended in their respective papers.

7.3 Approach performance (RQ4)

In this section, we discuss our evaluation results, and answer RQ4. We evaluate our technique using a collection of 226,287 commits from 17 subject systems comprising 23,761 touched variabilities. We collected all developers during the entire development history. Then, the developers were ranked by our approach. We then compare the ranked developer's list with the *gold developer set* containing developers that touched variable code of a particular variability. This procedure is made for each variability of each analyzed system. Table 7.1 summarizes the performance details of our approach.

	Accuracy		
	cut off of 0.1	cut off of 0.15	cut off of 0.2
Clamav	90.33	90.15	89.99
Collectd	40.62	40.62	40.62
Curl	60.08	59.91	59.77
FreeRADIUS	58.57	57.99	57.89
Httpd	82.28	81.79	81.68
Irssi	69.82	69.29	69.29
Libexpat	60.90	60.90	60.90
Lighttpd1.4	69.12	69.03	68.52
Marlin	76.44	75.94	75.65
Mongo	94.70	93.71	93.04
OpenSC	66.15	66.02	65.58
OpenSSL	79.89	79.68	79.51
OSSEC	74.70	72.84	72.42
Pacemaker	73.00	72.11	71.86
RetroArch	85.16	84.35	82.28
syslog-ng	78.71	78.71	78.71
WiredTiger	71.50	71.50	70.50
Average	**72.47**	**72.03**	**71.66**

Table 7.1: Accuracy of the approach with individual subject systems.

Table 7.1 shows the performance of our approach for each of the individual subject system analyzed in this stage (test stage). We consider three different cut-off values to evaluate the performance of our approach: We noted that the approach provides a recommendation accuracy of around 72.5% with a cut-off of 0.1, 72.0% with a cut-off of 0.15, and 71.7% with a cut-off of 0.2. We also note that the cut-off value of 0.1 returns an average accuracy of 72.5%, which suggests its more significant potential for a recommendation. Therefore, the cut-off value of 0.1 was chosen as the recommended cut-off value in our approach. It is also worth mentioning that our technique performs well with the performance metric with all the cut-off values tested, and the findings answer our first research question (RQ4).

7.4 Approach comparison (RQ5)

To further validate the performance of our approach, we compare it with DOA and Ownership, two widely used expertise-related metrics. We evaluate the techniques using a collection of 45,955 pull requests from 17 subject systems.

We collected the reviewers of each GitHub pull request that had a reviewer assigned explicitly in the reviewer assignment feature. It is important to mention that we only considered pull requests involving changes in variable code. We created a list of reviewers for each variability, thus creating our *gold reviewer set*. We also mapped files with variable code in pull requests to use in the target expertise-related metrics. After that, we generate a list of reviewers for each variability (DOA_V) or file (DOA and Ownership) and calculate the Jaccard distance between those lists and the gold reviewer set. Table 7.2 summarizes the comparison details for three different cut-offs in our approach. Values in bold represent the best values of an approach for a particular system.

	Cut-off of 0.1			**Cut-off of 0.15**			**Cut-off of 0.2**		
	DOA	**Ownership**	DOA_V	**DOA**	**Ownership**	DOA_V	**DOA**	**Ownership**	DOA_V
Clamav	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Collectd	0.92	0.91	0.83	0.92	0.91	0.77	0.92	0.91	0.77
Curl	0.84	0.86	0.82	0.84	0.86	0.84	0.84	0.86	0.85
FreeRADIUS	1.00	0.96	0.50	1.00	0.96	0.50	1.00	0.96	0.50
Httpd	1.00	0.98	0.50	1.00	0.98	0.50	1.00	0.98	0.50
Irssi	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Libexpat	0.70	0.76	0.25	0.70	0.76	0.25	0.70	0.76	0.50
Lighttpd1.4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Marlin	0.98	0.97	0.88	0.98	0.97	0.87	0.98	0.97	0.86
Mongo	1.00	0.99	1.00	1.00	0.99	1.00	1.00	0.99	1.00
OpenSC	0.90	0.87	0.69	0.90	0.87	0.69	0.90	0.87	0.69
OpenSSL	0.89	0.90	0.87	0.89	0.90	0.86	0.89	0.90	0.85
OSSEC	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Pacemaker	0.93	0.84	0.87	0.93	0.84	0.87	0.93	0.84	0.87
RetroArch	0.96	0.98	1.00	0.96	0.98	1.00	0.96	0.98	1.00
syslog-ng	0.86	0.90	0.86	0.86	0.90	0.86	0.86	0.90	0.88
WiredTiger	0.73	0.78	0.84	0.73	0.78	0.84	0.73	0.78	0.84
Average	**0.92**	**0.92**	**0.82**	**0.92**	**0.92**	**0.81**	**0.92**	**0.92**	**0.83**

Table 7.2: Comparison between expertise-related metrics and our approach.

In Table 7.2, we can observe that our approach outperforms the competing metrics for most of the analyzed systems using the *gold reviewer set*. The distance of our reviewer's list is closer to the *gold reviewer set* than the competing metrics in 8 out of 17 systems, with an average distance of 0.82 with a cut-off of 0.1. We also noticed that our metric outperformed the competing metrics even using other cut-off values. For example, using 0.15 as the cut-off, our metric outperformed DOA and Ownership in 8 systems presenting an average distance of 0.81. Using 0.2 as the cut-off, our metric also outperformed the competing metrics in 8 systems showing an average distance of 0.83.

Thus, each of the analyses above shows that our approach outperforms widely used expertise-related metrics answering RQ5.

CHAPTER 8

Related Work

C preprocessor. In literature, the C preprocessor is often heavily criticized. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability [AVRGC08, EBN02, Fav95, Fav97, KS94, SC92]. There are attempts to extract structures from the source code (e.g., nesting, dependencies, and include hierarchies) and visualize them in a separate view [KS94, PO97, SC92]. Views on configurations have been explored, which show only part of the feature code and hence reduce complexity [ABGM02, CCWY03, HEB$^+$10, KAK08, SGC07]. In addition, there are also the idea of using colors to support a developer to work with variable code [FKA$^+$13, Ram86, ON92, Wuu94]. Similarly to our work, these studies try to support the use of C preprocessor. However, our work focus on using C preprocessor properties to indicate key developers that probably will not have difficulties to maitain variable code. Another problem related to our work is related to modularity. Variability modularity has been a long-standing goal of feature-oriented software development such as configurable systems [ABKS13]. While some researchers view variabilities as modular unit of behavior and composition, others pointed out that, at the source-code level, most implementation mechanisms provide merely syntactic compositions, and thus lack proper interface abstractions and modular reasoning. Kästner et al. [KAO11] pinpoint two different notions of modularity: one based on locality and cohesion, and another based on information hiding and interfaces. Examples of approaches for improving modularity include architecture-based product lines (based on frameworks or components) [BCK03], feature-oriented programming [Pre97, BSR04], aspectual feature modules [ALMK08], feature cohesion [AB11], and superimposition [AKL13]. Despite the improvement of modularity in those cases, simple solutions like conditional compilation prevail in practice [GA01, EBN02, KAK08]. We argue that, independent from the notion of modularity, the identification of key developers to deal with variabilities remains important. Moreover, differently from their work, our goal is not improve modularity regarding variability. We use existing variabilities properties to propose a metric that fit to the solution that prevail in practice.

Expert recommendation. A number of prior studies have examined the effect of developer contribution on software quality. McDonald and Ackerman propose the "Line 10 Rule", one of the first and most used heuristics for expertise recommendation [MA00]. The heuristic considers that the last person who changes a file is most likely to be the expert. Expertise Browser [MH02] and Emergent Expertise Locator [MM07] are alternative implementations to the "Line 10 Rule". Our study relies on the Degree-of-Authorship (DOA) metric [FMMH$^+$14] and Ownership [BNM$^+$11] to identify experts. DOA and Ownership considers the whole version history to indicate developers with respect to a given code element. Rahman and Devanbu [RD11] examined the effects of ownership and experience on quality in several open-source projects. Similarly to our work, they analyze experience of developers in open-source software. However, they operationalize ownership differently from the ownership used in our work and they do not consider preprocessor-based configurable systems. Similarly, Meneely and Williams [MW09] examined the relationship of the number of developers working on parts of the Linux kernel with security vulnerabilities. They found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability. Avelino et al. [APHV19] also analyze code authorship on Linux kernel and other systems. They found that a small portion of developers makes the most significant contribution to the code base and that the number of files per developer is highly skewed. Similarly to our work, both of these studies consider configurable systems. However, none of them explore whether the expertise-related metrics they used are appropriate to that type of system. There are also studies that analyze code authorship through machine learning algorithms [ARA$^+$19, KKG$^+$19]. Differently from our work, these studies take into account characteristics related to code style to identify developers. Thongtanunam and colleagues focus on proposing models to select reviewers using neural networks [TTK$^+$15, TMHI16]. In Ye et al. [YZAM21], pull request title, commit message, and code change data are extracted to predict the likelihood that a candidate reviewer will be the appropriate reviewer. However, none of these studies focus on preprocessor-based configurable systems.

CHAPTER
9

Threats to Validity

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [WRH+12]. Each category has a set of possible threats to the validity of an experiment. We identified these potential threats to our study within each category, which are discussed in the following with the measures we took to reduce each risk.

Conclusion validity. It concerns the relationship between the treatment and the outcome. In this work, potential threats arise from *violated assumptions of statistical tests*: the statistical tests used to support our conclusions may have been inappropriately chosen. To mitigate this threat wherever possible, we used statistical tests obeying the characteristics of our data. More specifically, we used non-parametric tests, which do not make any assumption on the underlying data distribution regarding variances and types.

Internal validity. It is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. A critical threat to the internal validity is related to *historical events:* a past reviewer/maintainer may be the natural reviewer/maintainer of variability, thus concentrating knowledge compared to other developers. We analyze the systems with many developers and variabilities to alleviate this threat.

Construct validity. It refers to the degree to which inferences can legitimately be made from the operationalizations in your study to the theoretical constructs on which those operationalizations were based. We detected a possible threat related to the *restricted generalizability across constructs*: C/C++ might present specific source code characteristics compared to other programming languages and affects our study. This risk cannot be avoided since we analyzed only source code implemented in C/C++. However, we argue that C/C++ is an important programming language with preprocessor integrated and comprises most preprocessor-based configurable systems in the GitHub repository.

External validity. Threats associated with external validity concern the degree to which the findings can be generalized to the broader classes of subjects from which the experimental work has drawn a sample. We identified a risk related to *the interaction between selection and treatment*: the use of open source systems might present specific aspects when compared to proprietary systems. This risk cannot be avoided because our focus was open source systems with a development history available. However, we argue that they are relevant worldwide adopted systems with millions of end-users. Therefore, we believe the results extracted can be a first step toward generalizing the results.

CHAPTER

10

Conclusions and Future Work

Expertise-related metrics allow for coordinating developer teams by planning the overall division of work, identifying key collaborators, and finding the best developers for a target task. However, misalignment between files and variabilities of configurable systems may make it impossible to use such metrics in preprocessor-based configurable software systems.

In this context, this work was the first to explore how the work on mandatory code and variable code is divided between project developers and whether expertise-related metrics can help to indicate a developer with expertise for a task involving variable code. The results showed that few developers are specialists in variable code. We also identified that, among developers who changed variable code during the development history, only a few concentrate the majority of changes in variable code. We also explored whether expertise-related metrics can help to indicate a developer with expertise for a task involving variable code. The results suggest that expertise-related metrics are not a good fit to point to experts regarding variable code.

Given the results obtained with the exploratory study, we have had enough clues to conclude that a variability-aware expertise-related metric would be interesting in this context. To propose this metric, we used techniques such as feature selection and multiple linear regression to propose a variability-aware expertise-related metric based on historical data. We validate this metric by analyzing maintenance tasks made on variable code. We also compare our metric with two well-known expertise-related metrics - DOA and Ownership - in 45,955 pull requests involving variable code. The results showed that our metric outperformed both metrics in the subject systems.

Despite the contributions of this work, there are many other directions for future work. For example, it is important to conduct a qualitative study to understand how difficult it is for a developer to maintain variable code without knowledge. Another future work would be using different techniques such as neural networks and deep learning to propose a more precise set of developers. Finally, we believe it is important to expand the exploratory study with more systems and different research questions to have a broader view of this and other problems, as well as possible solutions.

References

[AB11] Sven Apel and Dirk Beyer. Feature cohesion in software product lines: an exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 421–430, 2011.

[ABGM02] David L Atkins, Thomas Ball, Todd L Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, 2002.

[ABKS13] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. Feature-oriented software product lines: Concepts and implementation, 2013.

[AKL13] Sven Apel, Christian Kastner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In *International Conference on Algebraic Methodology and Software Technology*, pages 36–50. Springer, 2008.

[AMS$^+$18] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability bugs in highly configurable systems: a qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–34, 2018.

[APHV16] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[APHV19] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. Measuring and analyzing code authorship in 1+ 118 open source projects. *Science of Computer Programming*, 176:14–32, 2019.

[ARA$^+$19] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[AVRGC08] Bram Adams, Bart Van Rompaey, Celina Gibbs, and Yvonne Coady. Aspect mining in the presence of the c preprocessor. In *Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, pages 1–6, 2008.

[AVRW$^+$13] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[BND$^+$09] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *2009 IEEE 31st International Conference on Software Engineering*, pages 518–528. IEEE, 2009.

[BNM$^+$11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[BSR04] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[CCWY03] Mark C Chu-Carroll, James Wright, and Annie TT Ying. Visual separation of concerns through multidimensional program storage. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197, 2003.

[CJ07] Xue-wen Chen and Jong Cheol Jeong. Enhanced recursive feature elimination. In *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*, pages 429–435. IEEE, 2007.

[DBW+02] Wlodzislaw Duch, Jacek Biesiada, Tomasz Winiarski, Karol Grudzinski, Krzysztof Grabczewski, Krzysztof Gr, et al. Feature selection based on information theory filters. In *In Proceedings of the International Conference on Neural Networks and Soft Computing (ICNNSC 2002), Advances in Soft Computing*. Citeseer, 2002.

[DL97] Manoranjan Dash and Huan Liu. Feature selection for classification. *Intelligent data analysis*, 1(1-4):131–156, 1997.

[Dor79] Robert Dorfman. A formula for the gini coefficient. *The review of economics and statistics*, pages 146–149, 1979.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[Fav95] Jean-Marie Favre. The cpp paradox. In *Proc. European workshop on software maintenance*. Citeseer, 1995.

[Fav97] J-M Favre. Understanding-in-the-large. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 29–38. IEEE, 1997.

[FKA+13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the# ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.

[FMMH+14] Thomas Fritz, Gail C Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):1–42, 2014.

[FPHK94] Francesc J Ferri, Pavel Pudil, Mohamad Hatef, and Josef Kittler. Comparative study of techniques for large-scale feature selection. In *Machine Intelligence and Pattern Recognition*, volume 16, pages 403–413. Elsevier, 1994.

[GA01] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 symposium on*

Software reusability: putting software reuse in context, pages 109–117, 2001.

[Gas72] Joseph L Gastwirth. The estimation of the lorenz curve and gini index. *The review of economics and statistics*, pages 306–316, 1972.

[GE03] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[GJ05] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a c program. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 379–388. IEEE, 2005.

[HEB+10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Toolchain-independent variant management with the leviathan filesystem. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, page 18–24, New York, NY, USA, 2010. Association for Computing Machinery.

[HPSG16] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 99–110, 2016.

[JBB15] A. Jović, K. Brkić, and N. Bogunović. A review of feature selection methods with applications. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1200–1205, 2015.

[Job91] JD Jobson. Multiple linear regression. In *Applied multivariate data analysis*, pages 219–398. Springer, 1991.

[JZM+17] He Jiang, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 60(1):1–16, 2017.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 311–320. IEEE, 2008.

[KAO11] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 1–8, 2011.

[Käs12] Christian Kästner. Virtual separation of concerns: toward preprocessors 2.0. 2012.

[KCH$^+$90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KDP16] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in github projects. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 291–302. IEEE, 2016.

[KKG$^+$19] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. Code authorship attribution: Methods and challenges. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019.

[KM14] Vipin Kumar and Sonajharia Minz. Feature selection: a literature review. *SmartCR*, 4(3):211–229, 2014.

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of 16th International Conference on Software Engineering*, pages 49–57. IEEE, 1994.

[LAL$^+$10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114, 2010.

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202, 2011.

[LMSZ10] Huan Liu, Hiroshi Motoda, Rudy Setiono, and Zheng Zhao. Feature selection: An ever evolving frontier in data mining. In *Feature selection in data mining*, pages 4–13. PMLR, 2010.

[MA00] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240, 2000.

[MA20] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(4):140–147, 2020.

[MH02] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 503–512. IEEE, 2002.

[MM07] Shawn Minto and Gail C Murphy. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 5–5. IEEE, 2007.

[MRB$^+$17] Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. The discipline of preprocessor-based annotations-does# ifdef tag n't# endif matter. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 297–307. IEEE, 2017.

[MRG13] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 75–84, 2013.

[MRG$^+$17] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2017.

[MW09] Andrew Meneely and Laurie Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462, 2009.

[ON92] Bruce Oberg and David Notkin. Error reporting with graduated color. *IEEE Software*, 9(6):33–38, 1992.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[PO97] T Troy Pearse and Paul W Oman. Experiences developing and maintaining software in a multi-platform environment. In *1997 Proceedings International Conference on Software Maintenance*, pages 270–277. IEEE, 1997.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[Ram86] Gerard K Rambally. The influence of color on program readability and comprehensibility. In *Proceedings of the seventeenth SIGCSE technical symposium on Computer science education*, pages 173–181, 1986.

[RD11] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, 2011.

[RMT11] Filippo Ricca, Alessandro Marchetto, and Marco Torchiano. On the difficulty of computing the truck factor. In *International Conference on Product Focused Software Process Improvement*, pages 337–351. Springer, 2011.

[RRM$^+$16] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Baldoino Fonseca, and Rohit Gheyi. Assessing fine-grained feature dependencies. *Information and Software Technology*, 78:27–52, 2016.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories*. 2018.

[SC92] Henry Spencer and Geoff Collyer. # ifdef considered harmful, or portability experience with c news. In *USENIX Summer 1992 Technical Conference (USENIX Summer 1992 Technical Conference)*, 1992.

[SGC07] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-clr: A tool for navigating highly configurable system software. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, pages 9–es, 2007.

[TMHI16] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050, 2016.

[TOB11] Christian Kästner Paolo G Giarrusso Tillmann, Rendel Sebastian Erdweg Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. 2011.

[TTK$^+$15] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[Wuu94] Yang Wuu. How to merge program texts. *Journal of Systems and Software*, 27(2):129–135, 1994.

[XZBY15] Bing Xue, Mengjie Zhang, Will N Browne, and Xin Yao. A survey on evolutionary computation approaches to feature selection. *IEEE Transactions on Evolutionary Computation*, 20(4):606–626, 2015.

[YL05] Huilin Ye and Hanchang Liu. Approach to modelling feature variability and dependencies in software product lines. *IEE Proceedings-Software*, 152(3):101–109, 2005.

[YZAM21] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.