Towards Measuring & Improving Source Code Quality

Umar Iftikhar

Department of Software Engineering
Blekinge Institute of Technology

Licentiate Dissertation Series no. 2024:02

Towards Measuring & Improving Source Code Quality

Umar Iftikhar

Blekinge Institute of Technology Licentiate Dissertation Series
No 2024:02

Towards Measuring & Improving Source Code Quality

Umar Iftikhar

Licentiate Dissertation in
Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN

2024 Umar Iftikhar

Department of Software Engineering

Publisher: Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Printed by Media-Tryck, Lund, Sweden, 2024

ISBN: 978-91-7295-474-8

ISSN: 1650-2140

urn:nbn:se:bth-25608

Abstract

Context: Software quality has a multi-faceted description encompassing several quality attributes. Central to our efforts to enhance software quality is to improve the quality of the source code. Poor source code quality impacts the quality of the delivered product. Empirical studies have investigated how to improve source code quality and how to quantify the source code improvement. However, the reported evidence linking internal code structure information and quality attributes observed by users is varied and, at times, conflicting. Furthermore, there is a further need for research to improve source code quality by understanding trends in feedback from code review comments.

Objective: This thesis contributes towards improving source code quality and synthesizes metrics to measure improvement in source code quality. Hence, our objectives are 1) To synthesize evidence of links between source code metrics and external quality attributes, & identify source code metrics, and 2) To identify areas to improve source code quality by identifying recurring code quality issues using the analysis of code review comments.

Method: We conducted a tertiary study to achieve the first objective, an archival analysis and a case study to investigate the latter two objectives.

Results: To quantify source code quality improvement, we reported a comprehensive catalog of source code metrics and a small set of source code metrics consistently linked with maintainability, reliability, and security. To improve source code quality using analysis of code review comments, our explored methodology improves the state-of-the-art with interesting results.

Conclusions: The thesis provides a promising way to analyze themes in code review comments. Researchers can use the source code metrics provided to estimate these quality attributes reliably. In future work, we aim to derive a software improvement checklist based on the analysis of trends in code review comments.

Keywords: *Source code quality, Code review analysis, Software quality improvement*

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dr. Nauman Bin Ali, Professor Jürgen Börstler and Dr. Muhammad Usman for their invaluable support, patience, guidance, discussions and motivation. I would also like to thank my colleagues at the SERL department, especially, Michael Dorner, Andreas Baeur, Xingru Chen, and Muhammad Laiq for their support.

I would also like to thank my parents, my wife, Lubna, our kids Ahmed and Khadija, and my siblings, especially Sophia Ali, for their continuous support and patience that has enabled me to achieve this humble milestone.

Overview of Publications

Publications in this Thesis

- **Chapter 2: Umar Iftikhar**, Nauman Bin Ali, Jürgen Börstler and Muhammad Usman. "A tertiary study on links between source code metrics and external quality attributes", Information and Software Technology, 165:107348, 2023.
- **Chapter 3: Umar Iftikhar**, Nauman Bin Ali, Jürgen Börstler and Muhammad Usman. "A catalog of source code metrics – a tertiary study" In Proceedings of the International Conference on Software Quality (SWQD), (pp. 87–106), Springer 2023.
- **Chapter 4: Umar Iftikhar**, Jürgen Börstler and Nauman Bin Ali. "On potential improvements in the analysis of the evolution of themes in code review comments". In Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), (pp. 340–347), 2023.
- **Chapter 5: Umar Iftikhar**, Jürgen Börstler, Nauman Bin Ali and Oliver Kopp. "Identifying prevalent quality issues in code changes by analyzing reviewers' feedback". *To be submitted.*

Contribution Statement

We used CRediT (Contributor Roles Taxonomy) [5] to lay out the individual author contributions to the included papers.

Chapter 2:

- *Umar Iftikhar:* Conceptualization, Methodology, Data curation, Writing – original draft, Writing – review and editing, Visualization
- *Nauman Bin Ali:* Conceptualization, Methodology, Validation, Writing – review and editing
- *Jürgen Börstler:* Conceptualization, Methodology, Validation, Writing – review and editing
- *Muhammad Usman:* Conceptualization, Methodology, Validation, Writing – review and editing

Chapter 3:

- *Umar Iftikhar:* Conceptualization, Methodology, Data curation, Validation, Writing – original draft, Writing – review and editing, Visualization
- *Nauman Bin Ali:* Conceptualization, Methodology, Validation, Writing – review and editing
- *Jürgen Börstler:* Conceptualization, Methodology, Writing – review and editing
- *Muhammad Usman:* Conceptualization, Methodology, Validation, Writing – review and editing

Chapter 4:

- *Umar Iftikhar:* Conceptualization, Methodology, Software, Data curation, Validation, Writing – original draft, Writing – review and editing, Visualization
- *Jürgen Börstler:* Conceptualization, Methodology, Visualization, Validation, Writing – review and editing
- *Nauman Bin Ali:* Conceptualization, Methodology, Visualization, Writing – review and editing

Chapter 5:

- *Umar Iftikhar:* Conceptualization, Methodology, Software, Data curation, Writing – original draft, Writing – review and editing, Visualization

- *Jürgen Börstler:* Conceptualization, Methodology, Visualization, Writing – review and editing, Supervision
- *Nauman Bin Ali:* Conceptualization, Methodology, Visualization, Writing – review and editing, Supervision
- *Oliver Kopp:* Validation, Writing – review and editing

Funding

This work has been supported by ELLIIT, the Strategic Area within IT and Mobile Communications, funded by the Swedish Government.

Contents

List of Abbreviations

Abbreviation	Definition
ANFIS	Adaptive Neuro Fuzzy Inference System
ANN	Artificial Neural Network
ARE	Absolute Relative Errors
AOP	Aspect Oriented Programming
AUC	Area Under the Curve
BERT	Bidirectional Encoder Representation from Transformers
CC	McCabe's cyclomatic complexity
CISQ	Consortium of Information and Software Quality
CK	Chidamber & Kemerer
CRD	Centre for Reviews and Dissemination
DARE	Database of Abstracts of Reviews of Effects
DMM	Dirichlet Multinomial Mixture
FFBN	Feed Forward Back propagation Network
FOP	Feature Oriented Programming
GMDH	Group Method of Data Handling
GRNN	General Regression Neural Network
GSDMM	Gibbs Sampling-based Dirichlet Multinomial Mixture model
GPU-PDMM	Pòlya urn Poisson-based Dirichlet Multinomial Mixture model
KN	Kohan Network
LDA	Latent Dirichlet Allocation
MARE	Mean Absolute Relative Error
MMRE	Mean Magnitude Relative Error
NLP	Natural Language Processing
OOP	Object Oriented Programming
OSS	open-source software
PNN	Probabilistic Neural Network
QGS	Quasi-gold standard
RFC	Response for Class
SMS	Systematic Mapping Study
SLR	Systematic Literature Review
STTM	Short-Text Topic Modeling
SVM	Support Vector Machine
WMC	Weighted Method per Class

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Software quality is a multidimensional concept involving different perspectives impacting various entities, including the work product, process, and resource [67]. While software quality is not defined formally [9], previous researchers have broadly described it as "conformance to requirements" [7], or in terms of "fitness for use" [8] in its given context. One of the work products during the software development is source code. Poor source code quality may lead to unwanted technical debt [16], difficulties in modifying existing source code, and adding new features, thus leading to cost overruns or delays in delivery. Improving source code quality with existing resources is a crucial initiative for practitioners, which can be achieved through various interventions, e.g., writing better software requirements [19], improving software architecture [2], conducting code reviews [162], incorporating improved testing methods [3], software process improvements [4] among others.

Software improvement during the development phase is essential as it reduces the chance of problems in source code released for later stages of development, including testing, deployment, and delivery. Among the existing practices to improve source code quality, peer code review is one of the vital quality assurance steps in software development [149, 161] where developers and code reviewers discuss the source code quality of the submitted code changes. The feedback given by code reviewers, called code review comments, contains a wealth of information addressing aspects of source code quality as experienced developers are aware of multiple essential elements of the project [148, 162, 200]. Their feedback goes beyond the capabilities of current static analysis tools, which have been reported to produce high false negatives [198, 199]. Beller et al. [159] observed that among the issues fixed during the code review, 75% of the issues related to the maintainability of the source code. In contrast, 25% of the issues discussed related to improving functionality issues. Similarly, Mäntyla et al. [155] manually analyzed discussions during code review and reported that approximately 75% of the defects highlighted in code review discussions related to the maintainability of the source code.

However, the wealth of information produced during the code review is not exploited further to benefit the project with regard to creating software improvement opportunities. Predominantly, the reviewer feedback is usually only utilized once. As reviewers may point out similar issues to the same or different developers, identifying such prevalent issues can be helpful, e.g., to propose preventive measures and find systematic improvements [153]. The categorization of code review comments to analyze common trends in review feedback that may lead to software improvement is mainly unexplored. Furthermore, manual categorization is infeasible due to the large number of changes submitted for code review. Practitioners, therefore, often have to rely on their subjective judgment of what they perceive as prevalent code quality issues and, thus, what, in their opinion, are the required improvements. To the best of our knowledge, only a few studies [152, 153, 156] have explored the automated categorization of issues in code review comments and investigated their evolution [170]. By studying and categorizing issues discussed in the code review comments, project managers can observe meaningful themes frequently highlighted by the code reviewers in code review comments and use them as the basis for improving software development.

Quantifying the software improvement derived from the analysis of code review comments is vital to underscore their effectiveness and show that such methodology may improve source code quality. Source code metrics objectively assess source code quality by measuring internal quality attributes to estimate external quality attributes Ochodek et al. [36], Nuñez-Varela et al. [117]. Measuring the quality of the source code helps take preventive or corrective steps to improve the source code quality, aid the management in predicting the quality of future releases [67], and evaluate the impact of software improvement initiatives or interventions. To reliably measure the potential software improvements using analysis of code review comments on source code quality, researchers must select source code metrics with consistent evidence of their link with external quality attributes.

Existing studies have investigated how poor source code structure, e.g., complicated code flow or dependency among classes, negatively impacts the maintainability of source code [18] and may indicate the possibility of post-release software defects [17, 141]. However, such results vary from context to context Ochodek et al. [36], and the relationship between internal quality attributes, measured by source code metrics, and external quality attributes is unclear. Various existing systematic studies have characterized empirical studies investigating the relationship between source code metrics and external quality attributes, and there is a need to provide an overview of source code metrics reported in the systematic studies and characterize the relationship between source code metrics and external quality attributes.

Thus, the licentiate thesis aims to explore potential ways to improve source code quality using the common themes highlighted in the analysis of code review comments and quantify source code quality. The overview of the thesis is shown in Figure 1.1.

To identify source code metrics and characterize the relationship between source code metrics and external quality attributes, we conduct a tertiary study reported in Chapters 2 and 3. In Chapter 2, we analyze the reported strengths of evidence between source code metrics, the measured internal quality attributes, and external quality attributes and their metrics. The outcome shows moderate strengths of evidence for certain metrics that have a consistent link with external quality attributes, including maintainability and reliability. However, there is no consistent link between source code metrics and their relationship with external quality attributes for security, functional suitability, performance efficacy, and usability. Chapter 3 reports a catalog of 423 source code metrics reported in 52 secondary studies. The outcome shows that coupling, the degree of inter-relatedness of code structure, is the most often measured internal quality attribute without executing the source code. The catalog also included complete descriptions of the source code metrics.

As mentioned, the automatic categorization of code review comments for software improvement opportunities has largely been unexplored with only a handful of studies on the subject [152, 153, 156, 170]. However, these studies [152, 153, 156] require initial training datasets, which can be resource-intensive. To our knowledge, Wen et al. [170] focused on using conventional topic modeling methods to categorize code review comments without the need for an initial training dataset. However, traditional topic modeling is unsuitable for short texts [150]. Several topic modeling methods optimized for short text have been proposed but have yet to be used to analyze code review comments. Hence, our first step is conducting an exploratory study (Chapter 4) to utilize a suitable topic model specifically designed for deriving themes in short texts. We analyze code review comments from three open-source systems and study the evolution of identified themes over 12 years. We also propose several improvements in the adopted methodology from the state-of-the-art [170] for using topic models for code review analysis. The outcome of this study is an improved method compared to the state-of-the-art in the analysis and categorization of code review comments.

Currently, practitioners must rely on their subjective assessment of the prevalent code quality issues in the repositories they are reviewing. We believe practitioners can benefit from the availability of automated data-driven methods to identify prevalent code quality issues in their code repositories. To support the practitioners, we designed a case study (Chapter 5) using the approach described in Chapter 4 as our first step. In Chapter 5, we further investigated how to support practitioners in identifying and profiling prevalent code quality issues in code review comments. We explore if it helps to identify and profile code review comments from abandoned and merged changes to demonstrate the approach's utility.

Our future work builds on Chapters 4 and 5. Both chapters utilize open-source data without measuring or quantifying the software improvement achieved. Furthermore, practitioners' perceptions and the industrial relevance of the approach are yet to be evaluated. Chapters 4 and 5 leave the following questions unanswered: 1) How effective are software improvement initiatives derived from analyzing code review comments in improving software code quality? 2) Do practitioners find analysis of code review comments relevant to software quality-related challenges? With the help of answers to these questions and the findings from the chapters mentioned above, we aim to achieve the overall goal of the Ph.D. thesis, that is, to measure and improve the source code quality using analysis of code review comments through the introduction of relevant and focused software improvements.

1.2 Background

1.2.1 Source code quality

According to Software Engineering Body of Knowledge [9], software quality is not defined formally. As stated in Section 1.1, previous researchers have broadly described it as "conformance to requirements" [7], or in terms of "fitness for use" [8] in its given context. The literature further differentiates software quality by entity in focus. Fenton et al. [55] classify the measured software entities into three broad categories, i.e., the project, the process, and the product. In this thesis, we focus on the software quality of the software product. The ISO-IEC 25010:2011 [70] describes software product quality as the "capability of the software product to satisfy stated and implied needs under specified conditions." The ISO/IEC 25010:2011 [70] further describes the software product's quality in terms of eight main quality attributes or characteristics in a hierarchical format with 32 sub-attributes. The eight main quality attributes defined by ISO/IEC-25010:2011 [70] are maintainability, reliability, security, functional suitability, performance efficacy, and usability, among others, for the entity of interest.

When focusing on source code as the entity of interest, we describe source code quality in terms of product quality attributes proposed in ISO/IEC 25010:2011 [70]. We extend the analogy Boehm et al. [52] used to describe source code quality. We ascribe source code to have high quality when it possesses the eight product quality attributes, and the absence of product quality attributes suggests source code to have low quality.

Several software models, in addition to the ISO/IEC 205010:2011 [70], have also been previously proposed [72, 74, 75], which differ in the quality attributes considered. Rashidi et al. [77] provide a good overview and comparison between the quality attributes considered by several quality models.

1.2.2 Source code quality measurement

Once software product quality is characterized into quality attributes, measuring the quality attributes reliably and objectively to support decision-making for quality management purposes and establish quality benchmarks is essential. Fenton et al. [55] categorize software quality attributes into internal software quality attributes and external quality attributes.

Internal quality attributes of the source code relate to source code characteristics without accounting for the execution environment. Examples of internal quality attributes include the size of the source code, complexity, coupling, cohesion, polymorphism, and inheritance. Intuitively, modules with large sizes, which are complex and have many dependencies on other modules, are difficult to maintain.

Source code metrics are one method to measure the software's internal quality attributes, and several source code metrics have been proposed over the years [50, 185, 186]. One of the earliest-used source code metrics is Lines of Code (LOC), which has been used since the late 1960's [20]. Other source code metrics widely used since the early days of software development include

McCabe's Cyclomatic Complexity [186] and Halstead Software Science Metrics [185], which were well suited to procedural programming languages. The introduction of object-oriented programming led to the development of new source code metrics, including the Chidamber & Kemerer metrics [50] and Li & Henry metrics [129]. Source code metrics form the basis for several important indicators of source code quality, such as heuristics for code smell detection [56], recommending refactoring candidates [21], and assessment of degree of technical debt [22]. However, certain source code metrics, e.g., size metrics, have been observed to be sensitive in different contexts [36]. Various source code measurement tools are commercially available, such as PMD[1] and SonarQube[2], to aid practitioners.

In contrast, external quality attributes relate to how the source code behaves in a specific environment. Similarly, external quality attributes can be measured using external metrics, such as the number of defects found or changes made during a given time. Since external metrics are often harder to gather and are only available once the source code is executed in its desired environment, assessing external quality attributes using internal quality attributes and their measurement has interested researchers.

1.2.3 Source code quality improvement

In this section, we present a high-level overview of some of the interventions that have been investigated to aid in improving source code quality. Intuitively, we consider such interventions to complement each other.

Existing studies have reported a positive impact of several ways to improve aspects of source code quality, including well-written and traceable requirements [27, 28], designing flexible architecture [29], improving the coverage and efficiency of tests [30, 31, 32], and incorporating improved software development process [33].

This thesis focuses on one of the software quality improvement area during the development cycle, which exploits existing resources to achieve the potential software quality improvement and is discussed below.

Modern code review

One of the processes known to be effective in enhancing software product quality is peer code review or code inspections [11]. The concept of code inspections,

[1]`https://pmd.github.io/`
[2]`https://www.sonarsource.com/products/sonarqube/`

proposed by Fagan et al. [10], involved extended group review meetings to discuss design choices and implementation details. SWEBOK [9] describes the process to be sampling-based, led by a moderator with established rule-based criteria. SWEBOK [190] also describes a similar yet less formal developer-led walkthrough approach as an alternative to code inspections.

While researchers have shown that code inspections and walkthroughs have improved software product quality by identifying defects early [12], they can be time-consuming and expensive regarding the resources required. Over the last decade, practitioners have adopted modern code review, a lightweight approach to peer code review. Bird et al. [148] define a modern code review as "(1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly in practice". Nowadays, modern code review is widely adopted in the industry [13, 14, 155, 161] and has been empirically demonstrated to improve software quality and as a medium of knowledge sharing among developers [15].

Existing literature has also discussed practices that reduce the effectiveness of the modern code review process. Doğan et al. [23] have reported a taxonomy of practices that reduce the effectiveness of the modern code review process from the literature and multiple code review smells reported in eight open source systems. Common code review smells reported [23] include poor reviews due to large pull requests and code reviews that take longer than expected. Practices at large companies with established code review cultures, such as Google, have devised recommendations to limit the size of the change set to ten files for a code review and also recommend prioritized feedback on change sets submitted for review [24].

As the modern code review process is a collaborative process involving discussions among developers, Fatima et al. [25] have also highlighted individual, personnel, and social factors that impact the modern code review process and categorized nine individual factors, four social factors, and three personnel factors.

Researchers have also systematically studied the topic of modern code review. Davila et al. [167] have summarized the research on modern code review and categorized research into foundational, proposal, and evaluatory studies. Badampudi et al. [26] systematically analyzed the researched themes in studies on modern code review and reported the alignment between research and practitioners' perceptions regarding the explored themes. Their survey results observed that 92% of the 25 practitioners agree that it is essential to investigate the impact of the code review process on source code quality.

As mentioned in Section 1.1, existing studies [155, 159, 168] have manually classified the defects found during the code review and the issues fixed by the code review process, thus supporting the hypothesis that code review repositories, code review comments in particular, contain a wealth of relevant information regarding source code quality improvement that may be further exploited to benefit the project.

1.2.4 Natural language processing

Natural language processing (NLP) is a discipline that aims to understand data in natural language or free text data and is considered one of the practical approaches for performing analytics in software engineering [154]. Over the years, machine learning-based approaches have been predominantly used for understanding text available in natural languages. The machine learning approaches can be broadly classified into supervised methods, where the ground truth is known, and unsupervised techniques, where the ground truth is unknown. Establishing the ground truth by manually analyzing code review comments can be time-intensive in our context. One of the promising unsupervised NLP methods is the Latent Dirichlet Allocation (LDA) method [147]. LDA models the entire corpus as a list of documents, where each document may be related to a topic. Each document, in turn, is modeled as a set of words. LDA hypothesizes that topics can be modeled as a group of words formed from a vocabulary of all the words in the given set of documents. Thus, text classification tasks, i.e., which document contains which topic, can be performed using a stochastic approach of iteratively assigning documents to topics based on the words contained in the document, using a prior set of topic-to-document probability (*alpha*) and word-to-document (*beta*) probability. The choice of the number of topics to look for, N, and prior probabilities need to be carefully selected for a meaningful analysis [176]. Over the years, several variants of the traditional LDA, a topic modeling method, have been proposed for specific tasks such as topic models optimized for short-texts [150].

Using NLP methods, we assume that similar issues in source code are discussed using similar vocabulary when discussed by different reviewers. Thus, in some instances, topic modeling may perform poorly when similar issues are scattered over various topics, thus limiting the interpretability and effectiveness of the approach. Additionally, the choice of the number of topics plays a significant role in deriving valuable results. We discuss measures to reduce the impact of these issues in Chapter 4.

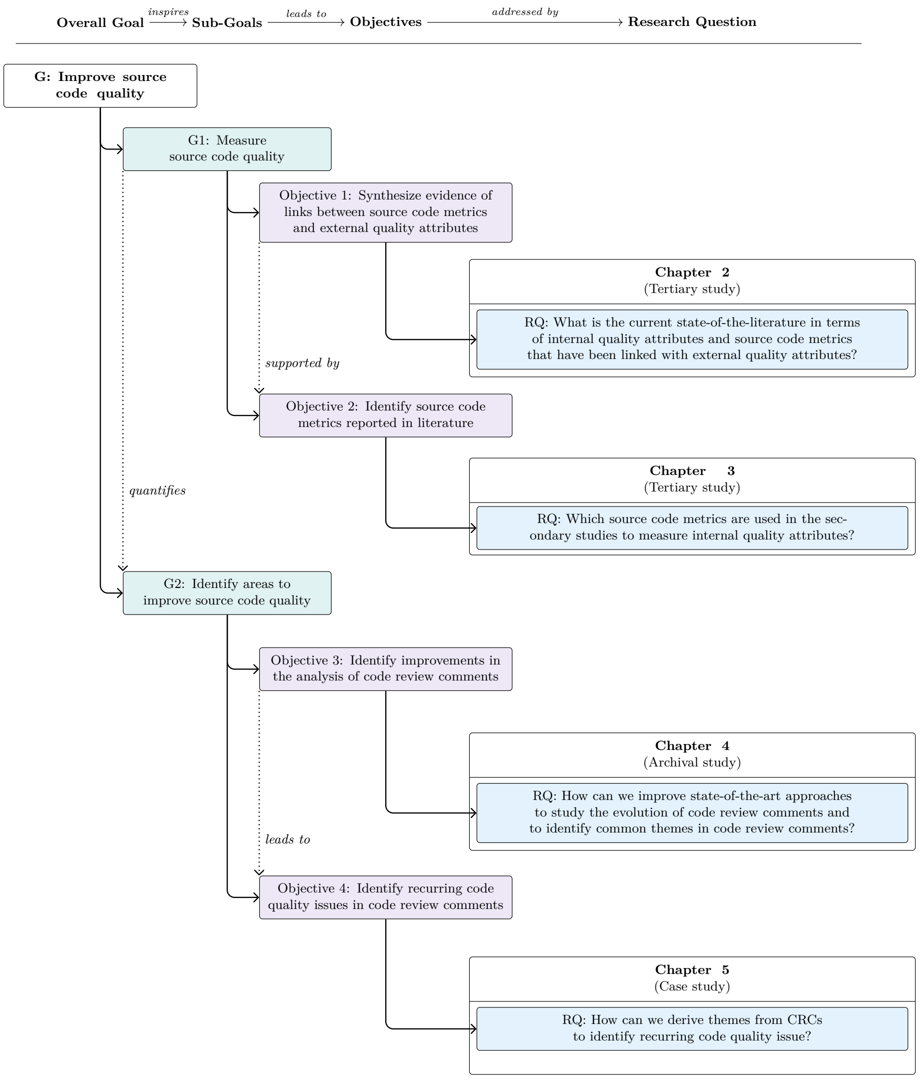

Overall Goal
inspires
Sub-Goals
leads to
Objectives
addressed by
Research Question
G: Improve source code quality
G1: Measure source code quality
Objective 1: Synthesize evidence of links between source code metrics and external quality attributes
Chapter 2 (Tertiary study)
RQ: What is the current state-of-the-literature in terms of internal quality attributes and source code metrics that have been linked with external quality attributes?
supported by
Objective 2: Identify source code metrics reported in literature
Chapter 3 (Tertiary study)
RQ: Which source code metrics are used in the secondary studies to measure internal quality attributes?
quantifies
G2: Identify areas to improve source code quality
Objective 3: Identify improvements in the analysis of code review comments
Chapter 4 (Archival study)
RQ: How can we improve state-of-the-art approaches to study the evolution of code review comments and to identify common themes in code review comments?
leads to
Objective 4: Identify recurring code quality issues in code review comments
Chapter 5 (Case study)
RQ: How can we derive themes from CRCs to identify recurring code quality issue?

Figure 1.1: Thesis overview (Goal, objective, research questions, contributions)

1.3 Research gaps and contribution

In this section, we discuss the research gaps and contributions of the different chapters in the thesis. The thesis overview is presented in Figure 1.1.

1.3.1 Chapter 2

On the topic of software measurement, existing secondary studies have investigated the relationship between source code metrics and software maintainability [61, 113, 114, 115]. Jabagangwe et al. [67] focused on studying the link between source code metrics with two external quality attributes, maintainability and reliability, for object-oriented systems. Briand et al. [68] focused on the relationship between fault-proneness and source code metrics. Similarly, several quality standards, such as the ISO-IEC 25010:2011 [70], have provided a framework to measure the quality of software products.

Research gap 1: While there are many secondary studies on the subject, the evidence on the effectiveness of source code metrics to assess or predict external quality attributes is varied and, at times, conflicting. Thus, there is a need to systematically synthesize the evidence reported in secondary studies for source code metrics linked with external quality attributes. We conducted a tertiary study to categorize the strength of evidence of links between source code metrics and external quality attributes from secondary studies.

Contribution 1: This thesis contributes by systematically synthesizing evidence of links between source code metrics and external quality attributes. The thesis further categorizes the strength of the collected evidence while considering various types of evidence. It has no limits to any particular programming paradigm or specific to any quality attribute.

1.3.2 Chapter 3

On the topic of software measurement, several systematic studies have synthesized source code metrics reported in the literature. Nunez et al. [117] mapped 300 source code metrics from four programming paradigms reported from studies between 2010 and 2015. Saraiva et al. [115] reported 67 aspect-oriented source code metrics to measure software maintainability. Hernandez-Gonzalez et al. [124] summarized 26 design-level source code metrics from 15 primary studies.

Research gap 1: These studies have specified limited scope and thus do not provide a holistic classification of the source code metrics along with their descriptions. A catalog of source code metrics reported in the systematic secondary studies is yet to be provided, along with their definitions and the measured internal quality attribute. Hence, we conducted a tertiary study to categorize source code metrics reported in secondary studies in software measurement.

Contribution 1: This thesis contributes by providing an extensive catalog of source code metrics with their descriptions and the internal quality attribute they measure with no limitation on programming paradigm and internal or external quality attributes.

1.3.3 Chapter 4

Analyzing common themes in code review comments and their evolution may provide meaningful insights that can lead to further software improvement. Existing studies have classified code review comments using machine learning methods [34, 35, 152, 153, 156]. Recently, Wen et al. [170] used traditional topic modeling to study common themes in code review comments and their evolution. However, more promising alternatives to traditional topic modeling methods are yet to be explored to study the common themes in code review comments and their evolution.

Research gap 2: It still needs to be determined how short-text topic modeling can provide better results than traditional topic modeling methods to analyze common themes in code review comments. Hence, we conducted an exploratory study to evaluate the performance of short-text topic modeling empirically and compare it with the performance of traditional topic modeling methods.

Contribution 2: This thesis empirically evaluates four improvement suggestions in analyzing common themes in code review comments and topic evolution. The thesis compares short-text and traditional topic modeling and further highlights improvement suggestions regarding the number of topic selections.

1.3.4 Chapter 5

While the exploratory study in Chapter 4 provides several improvement suggestions to the state-of-the-art [170], the study does not outline how to identify and profile prevalent code quality issues using the stated approach from CRCs. Furthermore, in Chapter 4, topic names were assigned using the software knowledge of two authors without the involvement of a domain expert, and no quality assessment of the derived themes was performed. Based on their extensive knowledge of the project, a domain expert may provide more informed names to themes, highlight the potential usefulness of the findings, and provide reflections on the quality of the derived themes.

Research gap 3: How to derive themes from code review comments to identify recurring code quality issues is yet to be performed. The analysis is yet to be used to create a profile for abandoned and merged changes. We have yet to conduct a subjective evaluation of the quality of themes derived from the approach by a domain expert. Furthermore, gathering empirical evidence to indicate the potential usefulness of the approach for the state of practice is yet to be accomplished.

Contribution 3: The thesis outlines an automation approach to identify and profile recurring code quality issues from abandoned and merged changes. The thesis empirically evaluates the quality of the themes derived from the approach proposed in Chapter 4. The thesis also demonstrates the potential avenues where the proposed approach can contribute to software improvement initiatives.

1.4 Research method

1.4.1 Tertiary literature review (Chapter 2, Chapter 3)

Chapters 2 and 3 are the outcomes of the same tertiary literature study, which is considered a systematic way of reviewing existing secondary literature studies. Chapter 2 synthesizes evidence of the link between internal quality attributes, source code metrics, and external quality attributes. In Chapter 3, the tertiary literature review focused on providing an extensive catalog of source code metrics reported in secondary studies, their descriptions, and classifications. We consider the research method appropriate for reviewing several existing secondary studies and consolidating their findings. We followed the guidelines by Kitchenham et al. [62] to design the tertiary study.

To identify relevant secondary studies, we searched three databases, namely, Scopus, ACM, and IEEE. The search string used during the search consisted of three clusters related to the artifact, i.e., source code, quality, and secondary studies, along with their synonyms. Two authors independently formulated a quasi-gold standard (QGS) to validate the search results. We used pre-defined inclusion and exclusion criteria to select relevant secondary studies. The data extraction form was piloted in light of research questions before complete data extraction. One author performed post hoc data validation on randomly selected secondary studies. The quality assessment criteria based on York University

Center for Reviews and Dissemination (CRD)[3] was used to exclude low-quality secondary studies.

To synthesize the results of evidence collected from included secondary studies, we utilized a "strength of evidence" criteria (see Chapter 2, Section 3.5.4) that consists of 11 factors, including the quality DARE score, the number of primary studies reporting the result, and the level of validation of source code metrics reported by the secondary studies. The synthesis method's result provides evidence of the link between source code metrics and external quality attributes.

1.4.2 Archival study (Chapter 4)

In Chapter 4, we conducted an archival study [1] to gain an understanding of how we can improve the design and implementation choices of analyzing themes within code review comments and their evolution using the approach by Wen et al. [170] as our foundation. The archival study provides design flexibility while retrospectively evaluating the various design improvements and their outcome. The context of our study was open-source systems. We used scripts written in Python that utilized the REST API [4] for data collection from projects on Gerrit[5] that host code review-related data for open-source systems. Using code review data from open-source systems provides us with a rich data source to validate our improvement suggestions and results. The open-source systems selected are from different application domains, with developers with varied levels of expertise contributing to the open-source systems. The OpenStack projects, Nova and Neutron, provide storage and network solutions and have been previously studied related to code review [171, 172]. Similarly, LibreOffice, an open-source office suite, has been extensively studied in several studies [173, 174]. In Chapter 4, we analyzed 209,166 code review comments. We quantitatively analyzed topic stability and coherence to select a suitable number of topics, using a method similar to thematic coding to name the topics in Chapter 4.

[3]The Centre for Reviews and Dissemination (CRD) suggests five questions to determine whether to include a systematic review in their Database of Abstracts of Reviews of Effects (DARE). `https://www.crd.york.ac.uk/CRDWeb/AboutPage.asp`

[4]`https://gerrit-review.googlesource.com/Documentation/rest-api.html`

[5]`https://gerrit-review.googlesource.com`

1.4.3 Case study (Chapter 5)

In Chapter 5, we performed an exploratory case study [54] to investigate the case of one open-source system, JabRef, to support the practitioners; we wanted to assess how to derive the common themes from code review comments to identify and profile code quality issues. We selected JabRef as it has been extensively analyzed in literature [117, 197], and a domain expert from JabRef agreed to aid in assigning representative themes to the topics. Similar to the archival study in Chapter 4, we used Python scripts implementing the REST API for data collection from GitHub[6] and analyzed 5,560 code review comments belonging to three major versions of JabRef. We designed a structured questionnaire and a separate document to collect the reflections from the practitioner. We further demonstrated how to create profiles of common themes identified from abandoned and merged changes.

1.5 Overview of chapters

1.5.1 Chapter 2: A tertiary study on links between source code metrics and external quality attributes

Several secondary studies have focused on reporting evidence for the link between source code metrics and internal and external quality attributes. Such secondary studies focus on specific external quality attributes, e.g., fault-proneness [105], or limit their scope to particular programming paradigms, e.g., object-oriented systems [67]. However, no tertiary studies exist that systematically collect and synthesize the links between source code metrics and external quality attributes from secondary studies.

From 15 secondary studies, our main findings show that source code metrics are extensively linked with maintainability and reliability for object-oriented systems. At the same time, other external quality attributes, such as security, have received less focus in the secondary studies included. We also observed that only a small set of source code metrics showed consistent results with maintainability and reliability. In contrast, several source code metrics show no consistent relationship with external quality attributes.

[6]`https://docs.github.com/en/rest/overview`

1.5.2 Chapter 3: A catalog of source code metrics – a tertiary study

Since software measurement is an essential field of study, several studies have reported source code metrics proposed over the years to measure internal quality attributes of the source code [115, 117]. However, previous studies have not categorized source code metrics reported from secondary studies on source code and related domains and their descriptions in a comprehensive catalog.

From 52 secondary studies, we reported 423 source code metrics together with their descriptions, and we categorized them based on the internal quality attributes measured. We further analyzed the unit of code used by the source code metrics to measure internal quality attributes. Additionally, we also reported the scope of the measurement made. In the source code metrics reported in the included secondary studies, *function* is the predominantly used unit of code, while more than half of the reported source code metrics utilized *class* as the scope of software measurements.

1.5.3 Chapter 4: On potential improvements in the analysis of the evolution of themes in code review comments

Existing studies have aimed to classify the code review comments provided by the code reviewers into meaningful themes. Previous studies have preferred manual categorization [159], while others have evaluated automated natural language processing for the categorization tasks with interesting results [153]. Other studies [34, 35, 152] have also considered using supervised machine learning to train models using multiple features, e.g., code snippets and source code metrics, in addition to code review comments to categorize code review comments and code changes automatically. To the best of our knowledge, only one study [170] used traditional topic modeling, one of the techniques for NLP, to study the topic evolution of common themes in code review comments, and none of the studies have used topic modeling methods specific to short-text. Hence, we conducted an archival study to evaluate four improvement suggestions to improve further the methodology for studying topic evolution in code review comments.

Section 1.4.2 mentioned that we utilized code review comments from three open-source systems. One of the open-source systems is shared with the state-of-the-art study on topic evolution [170]. Also, we used both traditional topic modeling and short-text topic modeling in our analysis to compare the results of our improvement suggestions.

The study's main findings are: (1) short-text modeling provides improved topic stability [176] compared to traditional topic modeling across all three open-source systems, thus can be further evaluated as a promising candidate to study topic evolution of themes within code review comments; (2) two-stage selection of the number of topics, *N*, provides a slight improvement in the average topic stability compared to single-stage selection used previously; (3) dividing the entire duration into multiple-windows may improve the analysis of common themes in code review comments; (4) topic coherence can used in tandem with topic stability when selecting suitable choice of number of topics, *N*. The study, however, did not empirically evaluate the impact of dividing the entire duration into multiple windows on the analysis of common themes in code review comments and only used a single-window approach as used by state-of-the-art [170].

1.5.4 Chapter 5: Identifying prevalent quality issues in code changes by analyzing reviewers' feedback

As a follow-up to Chapter 4 (see Section 1.5.3), which presents an improved method for analyzing common themes, we performed a case study where we aim to support practitioners by utilizing the approach proposed in Chapter 4 to identify and profile recurring code quality issues in code review comments. To demonstrate the approach, we utilized code review comments from abandoned and merged changes by using separate topic models for each set of code changes. Existing studies have manually analyzed code review comments for reasons of rejected and abandoned pull requests [6, 182]. However, they have not considered topic modeling to identify and profile recurring quality issues code changes in abandoned and merged changes. Furthermore, the quality of the identified themes has yet to be evaluated.

In Chapter 5, we identified different common themes in abandoned and merged changes, which broadly relate to the maintainability of the source code. The results from Chapter 5 demonstrate that the approach can be used to identify recurring code quality issues in code review comments. Furthermore, we observed unique code quality profiles in abandoned and merged code changes. Chapter 5 further outlines steps that can be used to implement a similar approach to other potential applications. Regarding the approach's usefulness for the state-of-the-practice, the identified themes can help improve "how-to" guidelines for developers and aid in directing discussions in developer forums.

However, further research is needed to improve the quality of the identified themes.

1.6 Synthesis and future work

In Chapter 2, we noted that source code metrics showed consistent results with external quality attributes, i.e., maintainability and reliability. Only a small set of source code metrics were related to security, and in our included secondary studies, we did not come across source code metrics linked with other external quality attributes described in ISO/IEC 25010:2011 [70].

Two of the four themes discussed in Chapter 4, component-level logic and inheritance, are also related to the maintainability of source code, which is consistent with previous taxonomies [155, 159]. The concurrency theme can be associated with performance-related issues. However, we found no themes related to security and other external quality attributes discussed in ISO/IEC 25010:2011 [70]. Similarly, in Chapter 5, the ten common themes from abandoned and merged changes broadly relate to testability, maintainability, and code quality. The common themes in Chapters 4 and 5, performed on different datasets, relate to source code quality; however, their potential for software improvement is still to be evaluated.

The overall focus of the planned doctoral thesis is to understand the subject of analyzing common themes in code review comments, operationalizing analysis of common themes for software improvement, and quantifying the software improvement achieved. Hence, future work is built upon the findings of the studies included in this licentiate thesis.

A fundamental question that needs to be answered is whether there is an industrial need to analyze code review comments and whether practitioners see value in tools or dashboards that enable such an analysis. While Wen et al.'s [170] analysis is motivated by an industrial need and the practitioner's response in Chapter 5 is largely positive for the analysis, the relevance for this analysis may be specifically evaluated with more practitioners from the industry as a basis for future work.

An interesting follow-up question to Chapters 4 and 5 studies concerns the effectiveness of the software improvement initiatives from common themes identified in code review comments in industrial settings. Here, we refer to quantitatively measuring the approach's effectiveness using source code metrics and qualitatively evaluating practitioners' perceptions regarding the proposed software improvements. While the initial qualitative evaluation in Chapter 5 suggests the proposed methodology is promising, it needs to be improved, and the results need to be generalized in industrial datasets. Among the potential applications of the proposed can be the possible optimization of existing software improvement checklists, improvement of the onboarding information, and focused discussion on current quality trends based on the common themes identified in code review comments.

As mentioned in Section 1.2.3, the quality of the code review process may impact the results we hope to achieve in the above-mentioned planned studies and the studies included in the licentiate thesis. The presence of code review smells may impact the common themes derived from the analysis performed and may be less effective in deriving software improvement initiatives. Thus, supporting and improving the code review process can improve the software improvement initiatives we aim to derive from the proposed approach. One method to enhance the code review process is to provide data-driven feedback to the code reviewer regarding the historical defects in the files under review. Currently, code reviewers only utilize their perception of historical defects to look for functionality-related issues in the files under review. For this study, we generate a list of fixed historical defects and the impacted files changed to fix these defects. This list is used to support the code reviewer when reviewing a new code change containing a file from the generated list. We hypothesize knowing the defect history of the file under review may aid the code reviewer in critically analyzing the submitted change for similar defects, thus improving the code review practice. We also collect feedback from practitioners who use this intervention and whether it helps them in improving the code review practice. We further analyze whether there is a reduction in the code review smells from the intervention we introduced.

In the above outline of future research, we aim to answer the following research question in future studies:

RQ 1: What is the perception of practitioners regarding the proposed approach to analyze common themes in code review comments to improve software development?

RQ 2: How effective are the potential quality improvements derived from the analysis of code review comments?

RQ 3: How can we support the code review practice by leveraging the defect history of files?

We intend to plan a survey to assess practitioners' perceptions regarding the usability or practical implications of the proposed approach to analyze common themes in code review comments and understand the code changes using the common themes. We aim to present the approach and results from Chapters 4 and 5 to demonstrate the effectiveness of the approach, inquire about the possible manner in which the approach can used in the industrial setting, and the perceived software improvement derived from the approach.

To answer the research question regarding the quality of the common themes identified, we plan to conduct a case study where common themes in code review comments are identified in industrial datasets, followed by interviews with developers where they evaluate the quality of the common themes identified. While the information available in code review comments is broadly known to developers, summarizing it into a representative "to-do" list may aid in improving the overall development experience. The envisaged approach from Chapters 4 and 5 above can be used after regular intervals to update the improvement suggestions, thus enhancing the chances that the identified software improvements remain relevant to the developers as the common themes in code review comments evolve.

The overall contribution of the future work is to (1) establish the need for the automation approach analyzing code review comments, (2) Evaluate the effectiveness of the software improvement initiatives derived from the analysis of code review comments, and (3) support code reviewers with historical defects data thus improving the code review practice and improving source code quality.

Challenges in future work

As mentioned in Section 1.2.3, the analysis of common themes relies on the quality of code review comments. In the scenario that the project under evaluation suffers from shallow reviews or any of the code review smells mentioned in [23], we believe that the methodology proposed can reflect such a trend. Thus, the procedure can be utilized as a data-driven approach to evaluate the quality of the code review process and the effectiveness of the code reviews. In the case of smaller repositories, there is a possibility of overrepresenting common themes from a small set of code reviewers.

The development of an effective, usable tool, such as a bot service or a dashboard, needs close interaction with practitioners to be able to capture their specific expectations and needs from the analysis method and be able to provide relevant common themes information in a manner that is both intuitive and valuable.

In the planned survey, we run the risk of biasing the practitioners' opinions by discussing the planned tool before acquiring their feedback. Thus, such a survey needs to focus on the needs of practitioners while reducing the chances of such biases. This can be both challenging and may limit the implications of the results.

1.7 Conclusion

This licentiate thesis aims to identify and synthesize the relationship between source code metrics and external quality attributes and explore ways to improve source code quality by analyzing themes in code review. We used a tertiary study methodology (Chapters 2 and 3) to identify and synthesize the relationship between source code metrics and external quality attributes from secondary studies. In Chapters 4 and 5, we used an archival and case study approach to investigate the common themes in code review comments from open-source systems.

Our main findings from the tertiary study show that a limited set of metrics from the source code metrics reported in the secondary studies report a consistent relationship with a limited set of external quality attributes, mainly maintainability and reliability. Thus, the source code metrics with consistent links with maintainability and reliability may be more beneficial for software measurement purposes than other source code metrics in the literature. Chapter 4 extends the work of Wen et al. [170] and provides specific design and analysis improvement suggestions, including a more appropriate choice of topic modeling method compared to state-of-the-art [170]. In Chapter 5, we continued the archival analysis of common themes in code review comments and used separate topic models to understand abandoned and merged code changes using common themes identified. We observed that common themes in abandoned and merged code changes discuss issues broadly related to maintainability, focusing on improvement suggestions in merged changes related to writing shorter code and using built-in tests. In contrast, abandoned changes discussed issues related to code formatting and Java code quality.

In future work, we intend to develop a tool to automate the analysis of the common themes in code review comments. The quality of the common themes identified from code review comments and the practical usability of the approach is another aspect yet to be evaluated as part of future work where we intend to involve practitioners from the industry. Lastly, we plan to evaluate the possible software improvements derived from analyzing the common themes identified.

Chapter 2

A tertiary study on links between source code metrics and external quality attributes

This chapter is based on the following paper:

Chapter 2: Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler and Muhammad Usman. “A tertiary study on links between source code metrics and external quality attributes”, Information and Software Technology, 165:107348, 2023.

2.1 Abstract

Context: Several secondary studies have investigated the relationship between internal quality attributes, source code metrics and external quality attributes. Sometimes they have contradictory results.
Objective: We synthesize evidence of the link between internal quality attributes, source code metrics and external quality attributes along with the efficacy of the prediction models used.
Method: We conducted a tertiary review to identify, evaluate and synthesize secondary studies. We used several characteristics of secondary studies as indicators for the strength of evidence and considered them when synthesizing the results.
Results: From 711 secondary studies, we identified 15 secondary studies that have investigated the link between source code and external quality. Our results show : (1) primarily, the focus has been on object-oriented systems, (2) maintainability and reliability are most often linked to internal quality attributes and source code metrics, with only one secondary study reporting evidence for security, (3) only a small set of complexity, coupling, and size-related source code metrics report a consistent positive link with maintainability and reliability, and (4) group method of data handling (GMDH) based prediction models have performed better than other prediction models for maintainability prediction.
Conclusions: Based on our results, lines of code, coupling, complexity and the cohesion metrics from Chidamber & Kemerer (CK) metrics are good indicators of maintainability with consistent evidence from high and moderate-quality secondary studies. Similarly, four CK metrics related to coupling, complexity and cohesion are good indicators of reliability, while inheritance and certain cohesion metrics show no consistent evidence of links to maintainability and reliability. Further empirical studies are needed to explore the link between internal quality attributes, source code metrics and other external quality attributes, including functionality, portability, and usability. The results will help researchers and practitioners understand the body of knowledge on the subject and identify future research directions.

2.2 Introduction

Quality evaluation of source code artifacts such as test code and source code produced during development helps in identifying future risks and judging how well the software system will perform in use. Software quality evaluation is a context dependent and multidimensional concept [63]. Depending on the context and relevant perspective of quality, various measurement methods are utilised. Several definitions of software quality have been proposed in the literature. IEEE defines software quality as *"the degree to which a system, component, or process meets specified requirements and customer or user needs or expectations"* [121]. Software quality is often described as a set of properties possessed by the software itself. The presence or absence of these properties helps differentiate levels of software quality.

Similarly, several quality models have been proposed that define the properties of software using different terminologies into a set of factors, attributes, or characteristics, e.g., McCall et al. [65], Boehm et al. [52], Dromey [66] and ISO/IEC 25010:2011 [70] (see Section 2.3.1). While the models refer to similar basic properties, they may differ on how the properties are inter-linked, along with the terminology used to refer to these properties. Some researchers have used quality characteristics [110] while others preferred quality attributes [99] when describing properties of quality. Considering that more secondary studies use *quality attributes* to refer to quality factors, we use the same terminology for consistency.

Internal quality attributes of the product relate solely to the product without considering the environment of the product. On the other hand, external quality attributes are described as properties of product behavior in relation to the product's environment [55, 64]. In the context of our tertiary study, the entity of interest is software source code. Internal quality attributes of source code can be assessed using code level metrics (e.g., lines of code) without executing the code. External quality attributes, such as how code behaves in a given environment or platform, can be measured using external metrics (e.g., number of defects), which are only available after execution. Assessing or predicting external quality attributes of software using internal quality attributes through source code metrics helps us measure product quality during development which can be invaluable considering the corrective costs related to post-deployment defects. The internal and external product attributes are classified in Figure 2.1 along with examples of external metrics and source code metrics.

The subject of using source code metrics to assess external quality attributes has been studied in various empirical studies [58, 59, 60]. Similarly, a large

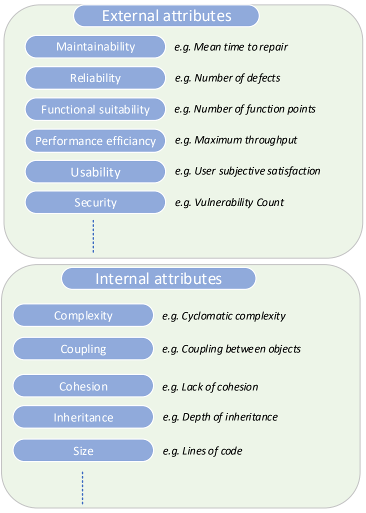


Figure 2.1: Classification of internal and external code quality attributes, according to Fenton [55]

number of secondary studies discuss utilising source code metrics as indicators of external quality attributes [61, 68, 113, 115]. While there are many secondary studies on the subject, the evidence on the effectiveness of source code metrics to assess or predict external quality attributes is varied and at times conflicting. We aim to synthesize the evidence reported in secondary studies and summarise the source code metrics that have been linked with external quality attributes.

The remainder of the paper has the following structure. Section 2.3 presents the related works in the field. Section 2.4 discusses the methodology used while Section 2.8 discusses the threats to validity, followed by Section 2.5 on conducting the review. Results and analysis of the study are covered in Section 2.6 while Section 2.7 discusses the implications of the research. Section 2.9 concludes the study along with potential future work.

2.3 Related work

This section discusses four themes of related work for this study: relevant secondary studies on source code quality, software quality models, code quality models, and closely related tertiary studies.

2.3.1 Secondary studies on source code quality

Several secondary studies discuss source code quality in the context of software maintainability. The ISO/IEC 25010:2011 defines maintainability as one of eight top-level quality attributes [1]. Baldassarre et al. [61] present the results from 28 primary studies focusing on software models used in the context of software maintainability. Riaz et al. [113] reviewed 15 primary studies on software maintainability prediction using source code metrics. Abílio et al. [114] review 11 primary studies that focus on feature-oriented and aspect-oriented metrics that aid in measuring software maintainability. Saraiva et al. [115] provide a catalog of 570 maintainability metrics for object-oriented systems. Malhotra et al. [116] synthesizes 96 primary studies focusing on software maintainability prediction in the early stages of development. Burrows et al. [111] review 12 primary studies that use coupling metrics as an indicator of maintainability in the context of aspect-oriented programming. Nuñez et al. [117] provide a review of the current state of object-oriented metrics. Briand et al. [68] show that fault proneness of software may be predicted using size, coupling, and complex-

[1]ISO/IEC 25010:2011 refers to the same as quality characteristics.

ity metrics. We evaluate and synthesize the varying claims and evidence in the secondary studies linking source code metrics and external quality attributes.

2.3.2 Software quality models

The ISO/IEC 25010:2011 [70] defines the quality of software in terms of its attributes in a revised version of the previously established ISO/IEC 9126 [69] standard. International standards are important and common reference models to ascertain quality parameters that are relevant to a broad set of products for concerned stakeholders. The ISO/IEC 25010:2011 standard [70] defines software quality in terms of eight main quality attributes, further elaborated with 32 sub-attributes.

Several other quality models have been proposed. Grady et al. [74] have identified FURPS or functionality, usability, reliability, performance, and supportability as key attributes of quality. Soto et al. [75] define quality characteristics for open-source systems as maintainability, reliability, portability, operability, performance, functional suitability, security, and compatibility. Mayr et al. [72] propose a model to operationalize the ISO-9126 quality model with 336 code level metrics for embedded software. Another attempt at bridging the gap between ISO-9126 and the needs of the industry is presented by Manet [73] which introduces the concept of *practices* or guidelines to observe a group of metrics for better assessment of underlying code. An elaborate discussion on several quality models along with analysis on overlapping attributes is presented by Rashidi et al. [77] and highlight that quality models differ in terms of definitions of attributes and assessment methods.

2.3.3 Code quality models

An alternative model that supplements ISO/IEC 25010:2011 for code quality is provided by the Consortium of Information and Software Quality (CISQ) [71]. It measures source code quality in terms of four quality attributes: reliability, maintainability, security, and performance efficiency. Each of the quality attribute is evaluated in terms of a set of properties called common weaknesses and measured in terms of aggregated weaknesses or violations.

SQALE [76] evaluates code in term of its testability, reliability, changeability, efficiency, security, maintainability, portability, reusability and is used by metrics tools such as SonarQube [2].

[2]`https://www.sonarqube.org/`

2.3.4 Tertiary studies on related topics

The tertiary study by Lacerda et al. [56] discusses code quality from the perspective of code smells and code refactoring. The study analyses 40 systematic studies and shows that code smells affect six of the 32 sub-attributes of quality, as defined by the ISO/IEC 205010:2011. The affected attributes include understandability, maintainability, testability, complexity, functionality, and reusability. The paper lists duplicated code or duplicated clones as the most reported code smell. Source code metrics based methods are reported as one of the common methods to detect the top ten code smells. The report also summarizes popular code smell detection and refactoring tools that may aid practitioners. The study, however, does not investigate the link between source code metrics and external attributes and which source code metrics are good indicators of external quality based on the evidence in the secondary studies.

In this tertiary study, we aim to synthesize the evidence from secondary studies linking external quality attributes with source code metrics. The synthesis includes both qualitative and quantitative evidence reported in the secondary studies. Additionally, we also synthesize claims reported in secondary studies on the link between external quality attributes and source code metrics. Furthermore, we evaluate the evidence with strength of evidence criterion that takes into consideration several characteristics of the secondary studies. However, for this study, we exclude secondary studies that only report the use of source code metrics to assess external quality attributes. To the best of our knowledge, there are no tertiary studies that link source code metrics with external quality attributes reported in secondary studies.

2.4 Methodology

In this tertiary study, we pose and answer the following overarching research question using the guidelines by Kitchenham et al. [62]:
What is the current state-of-the-literature in terms of internal quality attributes and source code metrics that have been linked with external quality attributes and external metrics? A significant number of secondary studies exist on source code quality. Therefore, we answer this research question by identifying, evaluating and synthesizing such secondary studies to answer the following specific questions:

RQ1: *What are the characteristics of the secondary studies that link internal quality attributes and source code metrics with external quality attributes?*

RQ2: *What is the efficacy of the relationship between internal quality attributes and source code metrics with external quality attributes and external metrics?*

RQ3: *What is the efficacy of the models that use source code metrics to predict external quality attributes?*

In RQ1, we aim to cover the characteristics of the secondary studies, the years covered by the secondary study, the aims of the secondary studies, the type of systems, the size of systems, and the programming paradigm studied. In RQ2, we aim to provide a comprehensive evaluation of the qualitative and quantitative evidence-based links that associate internal quality attributes and their metrics with various external quality attributes and their metrics. In RQ3, we consider whether particular prediction models are better than others in predicting external quality attributes using source code metrics.

2.4.1 Search strategy

This section presents the strategy used for searching relevant secondary studies and formulating the search string. The search is designed to retrieve secondary studies that report evidence linking source code metrics with external quality attributes. In order to collect as many relevant secondary studies as possible, we preferred a broader search strategy to collect various forms of evidence. In this phase, we have not differentiated between secondary studies that only report usage of source code metrics and those that report evidence.

Keyword-based search

We used keyword-based [45, 62] search in one indexing (i.e., Scopus) and two publisher databases (IEEE Xplore and ACM digital library) as recommended by Petersen et al. [57]. IEEE and ACM publish the most important journals and conferences in the software engineering field [82, 83]. Moreover, Scopus is considered as one of the largest indexing services covering papers from ACM, IEEE, Springer, Wiley and Elsevier [48, 119].

String construction

We used keywords and synonyms from ISO/IEC 25010:2011 [70], related keywords from a known set of papers (see online[3]), and our domain knowledge. The search string used in the study has the following three clusters of keywords:

Artifact: code, software program, software product, software application, software system, object-oriented, aspect-oriented, feature-oriented.

Quality: quality, smell*, pattern, functional suitability, performance, efficiency, compatibility, usability, reliability, security, maintainability, portability, analyzability, modifiability, testability, compliance, stability, comprehension, understandability, understanding, maintenance, modularity, reusability, changeability, evolvability, modification, evolution, readability, metric*, measur*, indicator, refactoring

Systematic secondary study: systematic review, systematic literature review, systematic map, systematic mapping, tertiary study, tertiary review, mapping study, multivocal literature review, multivocal literature mapping.

The search string was formed by combining the above sub-strings with a Boolean *AND* operator as depicted online[6]. We did not consider secondary studies shorter than eight pages as they are unlikely to contain necessary details related to evidence between source code metrics and external quality attributes. We restricted ourselves to papers published in peer-reviewed conferences, and journals to report the evidence widely accepted by the research community. We executed the search string in February 2021.

Moreover, in the final set of publications considered for selection, we added the publications from the validation and known sets of publications (see Section 2.4.1), and the secondary studies identified by Lacerda et al. [56].

Search validation

To evaluate the effectiveness of the search string, we used a set of secondary studies (see online[6]) as a quasi-gold standard (QGS) [62]. Two authors, not involved in the design of the search string, independently formulated the validation set which comprised 11 secondary studies [49].

Our search strategy achieved a recall of 73% which is moderate. The recall could further be improved with additional keywords. However, this led to an

[3]https://doi.org/10.5281/zenodo.7933498

Table 2.1: Inclusion/exclusion criteria used in the tertiary study

Inclusion Criteria	
C0	Publications in English language and with length of at least eight pages
C1	Peer-reviewed workshop, journal or conference publications
C2	Publications claiming to have systematically studied available literature, i.e., systematic literature studies (SLRs or SMSs) or multivocal literature studies (MLRs, MLMs)
C3	Papers that identify, define or measure internal quality attributes or determine levels of code quality (e.g., work on quality measurement or code smells)
C4	Papers that relate code metrics/quality attributes/code refactoring/code smells to external quality attributes
Exclusion Criteria	
C5	Publications that are about only external quality attributes of software product/system/service, or about the quality of other artifacts like defect reports, test code, or test cases

increase in the number of results that was impractical. Moreover, as we have supplemented the keyword-based search with relevant secondary studies from the tertiary studies by Lacerda et al. [56], we decided not to make this change.

2.4.2 Selection process

This section discusses the steps of the selection process.

Selection criteria

To identify relevant papers from the search results, we used the criteria listed in Table 2.1. Papers fulfilling the boolean expression (C0 AND C1 AND C2 AND (C3 OR C4)) were selected for full-text reading. In unclear cases, we were inclusive, i.e., when there were indications that the full-text of a paper might contain information fulfilling C3 or C4, we retained the paper for the next phase. Papers that did not fulfil C3 or C4 but fulfilled C5 were excluded.

Preliminary selection

Before applying topic specific selection criteria, the first author excluded publications with less than eight pages and not written in English.

Selection based on title and abstract

To check the objectivity of the criteria and whether we have a shared understanding of it, we performed a pilot round of the selection process [62, 78]. The piloting step involved all four authors. 12 randomly selected papers from the

Table 2.2: Decision rules in selection strategy based on Ali and Petersen [78]

		Reviewer 2		
		Relevant	Maybe	Irrelevant
Reviewer 1	Relevant	Include	Include	Resolve with discussion
	Maybe	Include	Include	Exclude
	Irrelevant	Resolve with discussion	Exclude	Exclude

search results were assessed independently by all authors as *relevant*, *irrelevant* or *maybe relevant*. Cohen's Kappa measure of inter-rater agreement was used to measure the level of agreement between authors. We aimed for an inter-rater agreement above 0.61 since it is considered a substantial agreement in literature [80, 81].

Any disagreements during piloting provide an opportunity of reviewing or updating the selection criteria in an iterative manner [57]. We resolved all disagreements through discussion and referring back to the selection criteria.

Selection based on adaptive reading

As stated in Section 2.4.2, we have been inclusive when the titles and abstracts have been insufficient to conclude the relevance of a paper. To deal with such papers in an efficient manner, we have used a modified adaptive reading method [79]. We first read the research questions of the secondary study to know more about its aims and objectives, followed by the introduction and conclusion sections to decide its relevance to our study. The selection criteria listed in Section 2.4.2 were used to ascertain the relevance. Any papers excluded during this process were reviewed by the second author to reduce the likelihood of excluding a relevant paper.

Selection based on full-text reading

In this step, the full-text of the papers was read to extract the information required to answer the research questions. During this step, an explicit attempt was made to identify overlapping papers [46], i.e., multiple papers reporting the same or extended version of a study.

Table 2.3: Data extraction traceability with research questions

Research question	**Data Extracted**
RQ1 What are the characteristics of the secondary studies that link internal quality attributes and source code metrics with external quality attributes?	– Metadata: (author, title, publication venues, publication date) – Aims: (research questions of the secondary study) – Search: (time period covered in the search, databases used). – Code quality attribute that are the secondary study's focus: (from the research questions) – Type of software system reported by the secondary study – Programming paradigm reported by the secondary study – Application domain considered by the secondary study – Software size reported by the secondary study – Type of research method used in secondary studies – Quality of the secondary study: selection criteria, search coverage, quality assessment, study description and synthesis
RQ2: What is the efficacy of the relationship between internal quality attributes and source code metrics with external quality attributes and external metrics?	– Name of external quality attribute/sub-attribute – Name and description of the external metric – Linked to which internal attribute – Name of source code metrics used to measure the external attribute – Description or definition of the source code metrics – Validation status of the source code metrics – Number of primary studies reporting the link between source code metrics and external quality attributes – Evidence for the reported link (e.g., claims, correlation results, prediction results)
RQ3: What is the efficacy of the models that use source code metrics to predict external quality attributes?	– Models used for prediction of external quality attributes – Accuracy of the prediction models – Datasets used for validation – Source code metrics used for prediction

2.4.3 Data extraction

Table 2.3 presents the data extraction form and a mapping of the data to respective research questions.

Piloting of data extraction

A piloting of the data extraction form was done to assess its reliability and completeness. The first and third author independently extracted the data from a randomly selected secondary study from the validation set to reduce the chance of selecting a secondary study that lacks sufficient data to validate the data extraction form. Any differences were resolved through a discussion.

Data extraction validation

After the complete data extraction, six secondary studies were randomly selected for validation of the extracted data by the third author. Due to the laborious nature of extracting multiple entries of measures and metrics from secondary studies, it was decided to mark the relevant section/tables for internal/external quality attributes, and reported source code metrics.

2.4.4 Quality assessment of the secondary studies

The five DARE[4] questions for the quality assessment of systematic reviews in healthcare have been widely used in software engineering research [44]. For this tertiary study, the criteria proposed by Budgen et al. [53] to answer the five DARE questions were used (see Table 2.19 in the appendix).

Piloting of the quality assessment criteria

The quality assessment form was piloted on two randomly selected secondary studies from our sample by the first and fourth authors.

Quality assessment validation

The first author performed the quality assessment for all included secondary studies. Six secondary studies were randomly selected for validation by the fourth author.

As recommended by Kitchenham et al. [62], the quality assessment score can be utilized to remove low-quality secondary studies. This approach has been used by other tertiary studies such as Hoda et al. [118], Barros-Justo et al. [119] and Curcio et al. [120]. Inspired by [118, 120], we removed secondary studies that score ≤ 1.5 (of 5), i.e., removed the papers that answered “no” to at least two questions *and* scored “partial” (0.5) on other questions.

The quality DARE score of a secondary study is also used during synthesis of results as an indicator of strength of evidence (see Section 2.4.5 and Table 2.5). The DARE score of a secondary study is classified as high, medium or low using the suggestion by Curcio et al. [120]. To critically appraise the quality assessment performed by the included secondary studies, we categorized

[4]The Centre for Reviews and Dissemination (CRD) suggests five questions to determine whether to include a systematic review in their Database of Abstracts of Reviews of Effects (DARE). `https://www.crd.york.ac.uk/CRDWeb/AboutPage.asp` accessed on: April 17, 2022

the quality assessment questions used in those secondary studies. The categorized quality assessment questions primarily assess the quality of reporting. However, focusing on methodological issues during critical quality appraisal is more beneficial [43, 44]. More recent proposals, like QAISER [47] for appraising systematic reviews, operationalize such assessments by going beyond reporting quality and assessing the risk of bias. For each included secondary study, we calculate an additional quality assessment score besides the DARE score. This additional quality assessment score is computed as follows: we count the number of categorized quality assessment questions used by the secondary study.

2.4.5 Collation process: attributes & metrics

This section reports the methodology used when collating extracted data related to internal attributes, external metrics, and source code metrics, along with the strength of evidence.

Attributes and metrics

The external quality attributes linked to the source code metrics are based on the information available in the secondary study regarding the external metric or dependent variable. Similarly, if reported by the study, the internal quality attribute is extracted. In the case where a secondary study has not assigned any internal quality attribute, and the source code metric belongs to a well-known metric suite, we then assign the internal quality attribute based on the description of the metric. If the metric is not part of a well-known suite, we then attempt to extract the description from the primary study quoted by the secondary study.

Relationship indicators quality attributes and source code metrics

The secondary studies report different types and levels of evidence when linking source code metrics to external quality attributes. We mapped the different levels of evidence reported in secondary studies as “++”, “+”, “0”, “-”, “--” and “unclear”. For the variety (e.g. correlation results, prediction accuracy results) of evidence reported, our interpret “++” to ’strong positive relationship’, “+” as ’positive relationship’, “0” as ’insignificant relationship’, “-” as ’negative relationship’, “--” as ’strong negative relationship’ and “unclear” as ’unclear relationship’ (see Table 2.4). Different types of evidence include explanatory secondary studies (i.e., studies predicting external quality attributes using

Table 2.4: Labels used indicate a relationship between external quality attributes and source code metrics

Assigned label	**Evidence reported by secondary study**
Exploratory secondary studies	
++	Strong positive correlation (with or without significance levels reported) Strong claims of positive relation
“+”	Positive correlation (significance levels reported) Positive correlation (no significance levels reported) Claims of positive relation
"0"	Insignificant results (significance levels reported) Insignificant correlation (no significance levels reported) Claims of insignificant relation
"-"	Negative correlation (significance levels reported) Negative correlation (no significance levels reported) Claims of negative relation
“--”	Strong negative correlation (significance levels reported) Strong negative correlation (no significance levels reported) Strong claims of negative relation
"Unclear"	Unclear results (significance levels reported) Unclear results (no significance levels reported) Claims of unclear relation
Explanatory studies	
“+”	Accuracy measures (e.g., TPR, FPR, Mean ARE, AUC) reported by secondary study as successful predictors Accuracy measures (e.g., MMRE) interpreted by our tertiary study as good predictors using [122, 123]
“0”	Accuracy measures reported by secondary study as insignificant predictors Accuracy measures (e.g., MMRE) interpreted by our tertiary study as insignificant predictors using [122, 123]
“-”	Accuracy measures reported by secondary study as unsuccessful predictors Accuracy measures (e.g., MMRE) interpreted by our tertiary study as unsuccessful predictors using [122, 123]
“Unclear”	Accuracy measures reported by secondary study as unclear predictors Accuracy measures (e.g., MMRE) interpreted by our tertiary study as unclear predictors using [122, 123]

source code metrics) report accuracy measures, while exploratory secondary studies (i.e., studies investigating the relationship between external quality attributes and source code metrics) report correlation co-coefficients or claims of relationship.

Double counting

Double counting can occur during results aggregation in tertiary studies when the result from the same primary study is reported by more than one secondary study and is counted more than once. If more than one secondary study reports the evidence of a link between source code metric and external quality attribute used, we also investigate if there is a risk of double counting. Thus, we consider whether these secondary studies share any primary studies when reporting their results. While processing the degree of overlap, we only considered overlap in primary studies that are used to report results in the secondary studies (thus if a primary study is found in more than one secondary study, but only one secondary study utilises it as a source for its results while the others consider it as reference is not considered an overlap).

In case there is no overlap, we report "No overlap." We note the primary study title, the author, and the year if overlap exists. We match these details within the other secondary study to see whether the other secondary study also uses results from the same primary study. If this is true, then based on how the secondary study results are presented, we try to remove the overlap before reporting the results (by counting results only once in the overlapped primary studies). If this is possible, we adjust the result to count it only once and report "X PS, Resolved" where X represents the number of removed studies. In case this is not possible, we report the overlap present and use it as a factor to downgrade the strength of evidence (see section 2.4.5).

Strength of evidence

To indicate the strength of evidence for the relationship between external quality attributes and source code metrics discussed in the previous sections, we considered twelve factors: (1) the DARE quality score of the secondary study, (2) the aggregate number of primary studies (within the secondary studies) reporting the stated link between source code metric and external quality attribute, (3) whether the secondary studies have reported and described the external metric to measure the external quality attribute, (4) the degree of overlap among secondary studies reporting the same result, (5) the additional quality assessment score calculated in Subsection 2.4.4, (6) whether the secondary studies reported the source code metrics used as validated, (7) whether the secondary studies have provided a replication package or reported the extracted data in the annexure, (8) whether the authors have explicitly reported any conflict of interest, (9) whether secondary studies reporting correlations results have reported if their primary studies have adjusted the results for using multiple statistical tests. We consider three additional criteria for prediction-focused secondary studies in addition to the eight criteria, (10) whether unbiased or unskewed evaluation measures were used to report the quantitative evidence in secondary studies, (11) whether the secondary studies that focus on fault-proneness or defect prediction reported normalization by the size of modules in their primary

Table 2.5: Strength of evidence criteria

Strength of evidence	Secondary study DARE score	AQAS* score	PS in result	Reported external metrics	Number of "Yes" to criteria 6 to 12
Very High	At least one high quality secondary study	AQAS >= 10	PS >= 25	Yes	>= 2 Yes
High	At least one high quality secondary study	AQAS < 10	PS >= 25	No	>= 2 Yes
	At least one high quality secondary study	AQAS < 10	PS >= 25	No	< 2 Yes
	At least one high quality secondary study	AQAS < 10	10 < PS < 25	Yes	>= 2 Yes
Moderate	At least one high quality secondary study	AQAS < 10	PS >= 25	No	< 2 Yes
	At least one high quality secondary study	AQAS < 10	10 < PS < 25	No	>= 2 Yes
	At least one high quality secondary study	AQAS < 10	10 < PS < 25	Yes	< 2 Yes
	At least one high quality secondary study	AQAS < 10	PS <= 10	Yes	>= 2 Yes
Low	At least one high quality secondary study	AQAS < 10	10 < PS < 25	No	< 2 Yes
	At least 1 high quality secondary study	AQAS < 10	PS <= 10	No	>= 2 Yes
	One high or medium quality secondary study	AQAS < 10	PS <= 10	No	< 2 Yes
Very Low	One high or medium quality secondary study	AQAS < 10	PS <= 10	No	< 2 Yes

AQAS*:Additional quality assessment score

studies, and (12) for secondary studies that focus on prediction, if the data used in primary studies (as reported by secondary studies) is post-hoc or do the secondary study also report results of prediction of future faults/defects/change as well. The criteria used are summarised in Table 2.5.

2.5 Conducting the review

The results of the overall selection process are summarised in Figure 2.2. Individual steps of the process are described in the following sub-sections.

2.5.1 Search results

Figure 2.2 summarizes the number of results from individual databases and the total number of unique secondary studies from the keyword-based search. The publications from the validation set and the known set of publications are also included. Lacerda et al. [56] is the only tertiary study close to our topic.

Therefore, we considered including the secondary studies from their tertiary study, which may be potentially relevant. We consider this as an acceptable trade-off to supplement our search results while not duplicating the efforts of Lacerda et al. [56]. Moreover, Lacerda et al. [56] only share one secondary study with the validation set, thus improving the chances of adding potentially relevant studies missed by the automated search. After removing the duplicated publications, we have 711 unique publications.

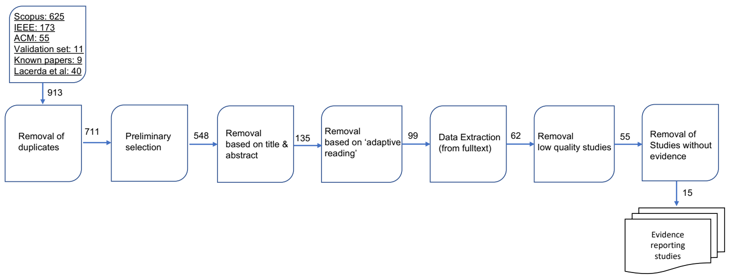

Scopus: 625
IEEE: 173
ACM: 55
Validation set: 11
Known papers: 9
Lacerda et al: 40
913
Removal of duplicates
711
Preliminary selection
548
Removal based on title & abstract
135
Removal based on 'adaptive reading'
99
Data Extraction (from fulltext)
62
Removal low quality studies
55
Removal of Studies without evidence
15
Evidence reporting studies

Figure 2.2: Selection process results (The count depicts included secondary studies at each stage)

2.5.2 Selection results

Before applying topic specific selection steps, the first author identified publications out of the 711 search results in Figure 2.2 that meet the selection criteria C0 and C1 in Table 2.1. Thus, publications with less than eight pages, not written in English or not published in a peer-reviewed venue, are excluded. A total of 163 publications that did not meet the page and language requirements were excluded leaving 548 secondary studies for the next phase of the selection. Among the 163 publications removed during this phase, four were non-English publications while the remaining were removed due to short length.

Pilot selection process

For the 12 secondary studies considered in the pilot selection, all authors agreed on seven out of twelve secondary studies while there were disagreements on the remaining five secondary studies, giving an initial agreement percentage of 58%. Only two out of six author-pairs had a substantial agreement, while two author-pairs had a moderate agreement. The disagreements were resolved in a

Table 2.6: Results after preliminary selection

S.No	Decision	Number of secondary studies	Comment	Included after full-text reading
1	Exclude-Exclude	320	Excluded	-
2	Exclude-Maybe	26	Review	0
3	Maybe-Maybe	5	Included	3
4	Include-Maybe	37	Included	14
5	Include-Include	78	Included	41
6	Include-Exclude	82	Review	4
-	Studies in preliminary selection	548	-	-

discussion and the descriptions of the inclusion/exclusion criteria were further clarified.

Complete selection process

Next, each of the remaining 548 secondary studies were reviewed by two authors. While the first author applied the selection criteria to all secondary studies, 182 randomly selected secondary studies were assigned to the second, third, and fourth author, each. Table 2.2 used in [67], was utilised during the decision making process. A secondary study was excluded if it was resolved as “irrelevant” and it was included if it was agreed upon as “maybe” or “included” by both authors. In this round, the initial agreement among the author-pairs was 73%. The average of Cohen-Kappa inter-rater agreement between author-pairs was 0.64, which is substantial agreement [80, 81].

The disagreements during this round were also resolved through discussion among authors and was utilised to further improve the clarity of the inclusion/exclusion criteria. After the study selection process, 135 secondary studies were included, while 413 secondary studies were excluded from the list of studies.

After the conclusion of the adaptive reading step, 99 secondary studies were retained while 36 secondary studies were identified as irrelevant to the topic. All secondary studies identified by the first author as irrelevant were reviewed by the second author.

The first author read the full text of 99 papers. The full text for one secondary study [84] was not available (besides our best efforts), thus it was excluded. Two papers were identified as the same secondary studies [85, 92] and the most recent of the two secondary studies [92] was retained. The first author read all the papers in this phase and further identified 37 secondary studies as irrelevant to the scope. The excluded secondary studies were reviewed by the second and fourth author and the authors agreed on excluding 35 secondary studies. Disagreements on the remaining two secondary studies were resolved through a discussion leading to inclusion of both, giving 62 secondary studies for quality assessment and data extraction.

As a reflection on the study selection process, we analysed how this final set of 62 secondary studies were initially assessed by authors after selection based on titles and abstract only. Of those 62 secondary studies, four secondary studies were initially assessed as “Include-Exclude,” three secondary studies were initially assessed as “Maybe-Maybe” and 14 secondary studies were initially assessed as “Include-Maybe”. Since a large number of secondary studies might have been excluded solely based on the decision rules mentioned in Table 2.2, we recommend that authors should include secondary studies categorized as “Include-Maybe” and “Maybe-Maybe” for full-text reading as a large percentage of these secondary studies might be relevant. The authors should also carefully review secondary studies marked as “Include-Exclude” list as immediate exclusion may result in the exclusion of relevant secondary studies.

2.5.3 Data extraction

Piloting of data extraction

In the piloting of the data extraction form, the authors agreed on 76% of the data extracted for one randomly selected paper. The differences were discussed and resolved.

Data extraction validation

In a post hoc validation of the extracted data, the third author independently did data extraction for six secondary studies. We found that all entries related to demographic data, internal/external attributes, reported metrics matched between the first and third authors. However, minor differences were observed in two fields, namely, recommendations/findings and synthesis method utilized. The differences were discussed and corrective action was taken to resolve them.

2.5.4 Quality assessment of the secondary studies

In the pilot round of quality assessment, there was agreement on the first and fifth questions (see Table 2.19) related to the inclusion and exclusion criteria and synthesis of secondary studies. The authors' results did not match questions two, three, and four regarding sufficient search coverage, quality assessment, and secondary study description. After discussing the misalignment in understanding, the differences were resolved.

The validation of a 10% randomly selected sample of secondary studies showed that there was more than 80% agreement on the first two DARE questions, 66% agreement on question 3, and 50% agreement on the last two questions. A meeting was held to discuss the differences and improve the alignment between the authors. It was observed that all disagreements were minor, e.g., one author assigned "yes" while the other author assigned "partly", or when one author has assigned "partly" the other author assigned "no" when answering questions. Based on the improved understanding, the first author reviewed all papers assigned as "yes" for Questions 4 and 5 (see Table 2.19) of the quality criteria. The fourth author reviewed the updated results, and all changes were agreed upon.

After removing secondary studies with scores less than or equal to 1.5, 55 secondary studies remained. As DARE is not designed to evaluate the quality of multi-vocal reviews, quality assessment-based selection was not applied to MLRs. The secondary studies removed due to low DARE score are listed online[6]. Detailed results of quality assessment are reported in Section 2.6.1 as part of the characteristics of the included secondary studies.

2.5.5 Secondary studies providing links between external quality attributes and source code metrics

From the list of 55 secondary studies, only 15 secondary studies provide evidence regarding a link between external quality attributes and source code metrics (see Figure 2.2). These secondary studies either provide quantitative (e.g., correlation or prediction results) or qualitative evidence (e.g., stated claims or comments on source code metrics as good or poor indicators) of the link between source code metrics and external quality attributes. Forty secondary studies[6] only report the *usage* of source code metrics for assessing or measuring external quality attributes. These 40 secondary studies have not commented on how well they measure external quality attributes, nor did they provide any qualitative or quantitative evidence exploring or explaining the link between the source code

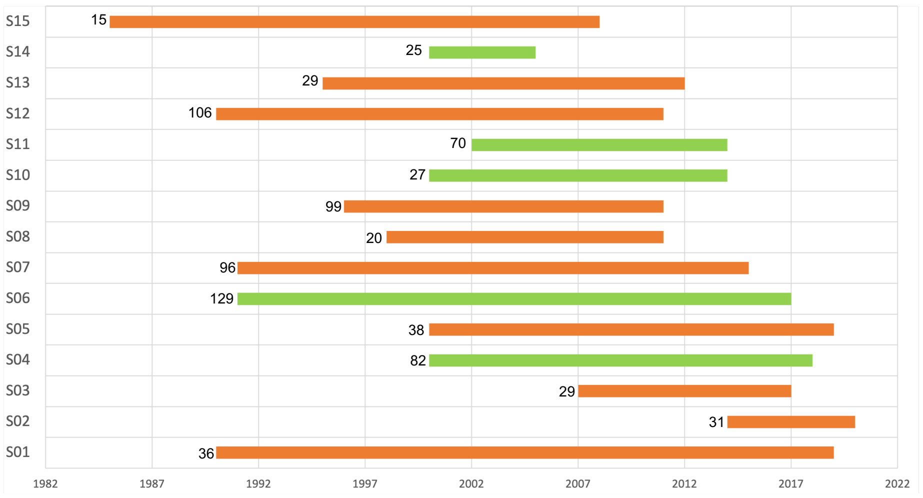


Figure 2.3: Number of primary studies and years covered by the included secondary studies

measures and the external quality attributes. While usage alone may be considered evidence, we focus on explicitly stated evidence in this tertiary study. We have thus identified 15 secondary studies reporting such evidence.

2.5.6 Removal of double-counting results in secondary studies

We resolved overlaps in primary studies reported by the secondary studies by counting duplicate primary studies only once. This was possible for the case where we could trace the evidence regarding the link between source code metrics and external quality attributes reported in the secondary study to the primary study included. We mitigated overlaps in primary studies between S06 and S12 (seven primary studies), S01 and S04 (nine primary studies), S12 and S14 (one primary study), and S06 and S14 (one primary study).

2.6 Results and analysis

2.6.1 RQ1: Characteristics of secondary studies

The 15 selected secondary studies are shown in Table 2.18. Ten of them are systematic literature reviews (SLRs) and five systematic mapping studies (SMSs). The earliest secondary study that fulfilled our selection criteria was published in 2009. The search was performed in February 2021, which may explain why no secondary studies from 2021 are included in our sample. The researcher's interest in software code metrics and reporting relationship with quality attributes has been steady over the years, with nine out of 15 of the secondary studies published between the years 2016–20.

The years covered by the secondary studies are shown in Figure 2.3. Half of the published articles cover the years between 1998 and 2014. In their search results, only three secondary studies had included the years prior to 1991 (when Chidamber and Kemerer metrics suite was introduced), indicating that researchers have focused on primary studies conducted after this year.

Secondary study aims

We have analysed the research questions and stated aims from the secondary studies based on Fenton's classification of internal and external quality attributes [55] (see also Figure 2.1). Furthermore, we classify the secondary studies in each sub-topic based on metrics utilization, i.e., whether the source code metrics have been utilized for exploring the relationship between source code metrics and external quality attributes (i.e., correlation studies) or explaining the relationship between source code metrics and external quality attributes (i.e., prediction studies).

Maintainability is the most frequently studied external attribute, with six secondary studies investigate maintainability prediction while five secondary studies investigate reliability prediction as depicted in Figure 2.4. One secondary study (though reports evidence for reliability) aimed to report influential metrics and their aggregation without specifying the internal or external attributes, which we have classified as "Unclear". Similarly, one secondary study investigates the internal attributes of code categorized into coupling and cohesion.

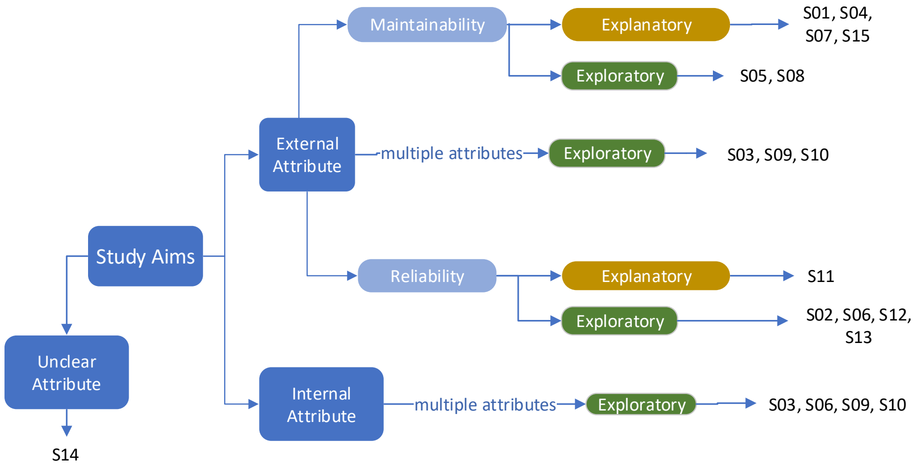


Figure 2.4: Secondary study aims

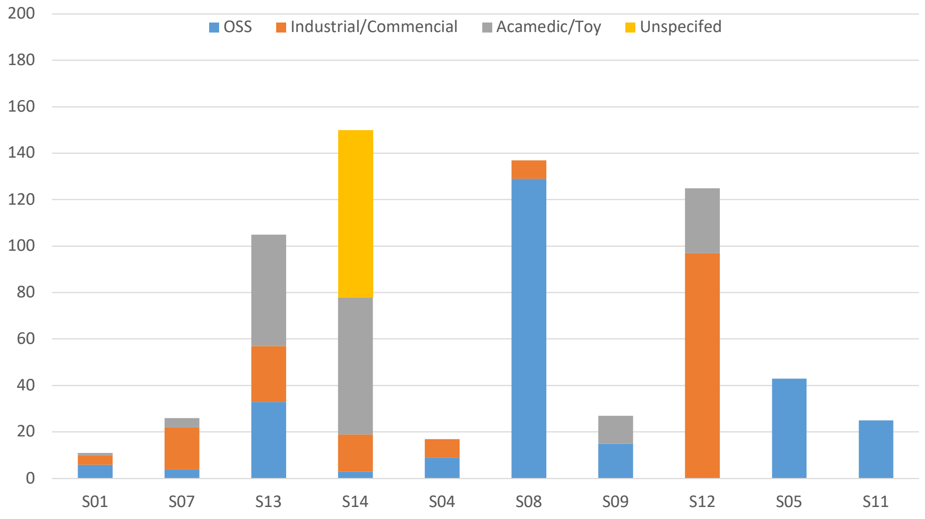


Figure 2.5: Type of systems studied

Table 2.7: Databases used for automated search by secondary studies

Databases Name	Studies
ACM Digital Library	S01, S03, S04, S05, S06, S07, S09, S10, S11, S12, S15
EI Compendex	S07, S09, S12
Google Scholar	S02, S04, S06, S07, S10, S13
IEEE Xplore	S01, S02, S04, S05, S06, S07, S09, S10, S11, S12, S15
Inspec	S09, S12, S15
Others	S01, S04, S07, S11, S12, S13
ScienceDirect – Elsevier	S01, S02, S03, S04, S06, S07, S10, S12, S15
Scopus	S01, S04, S05, S07, S09, S11, S13, S14, S15
Springer Link	S01, S02, S04, S05, S07, S12, S15
Wiley Online Library	S01, S05, S07

Systems studied

We classified the systems studied as open-source, industrial systems, or academic systems. The number of systems studied is shown in Figure 2.5. Of the 15 secondary studies, five did not mention the software systems used in their primary studies.

Open-source systems were the most popular type of software systems studied and were reported by nine out of 15 secondary studies, while seven out of 15 secondary studies utilised industrial systems. Academic systems were reported by six out of 15 secondary studies. Four secondary studies use systems from all three categories, while only two out of 15, secondary studies only use one type of system. Sixty-five unique open-source systems were reported among the included secondary studies. JEdit [5] is the most often reported open-source system used by three secondary studies followed by Mozilla [6] and Apache [7]. Among the secondary studies that used public datasets, the PROMISE [8] and NASA [9] datasets have been reported by more than three secondary studies.

Databases used

The databases used to perform an automated search by the included secondary studies are depicted in Table 2.7. The most frequently used databases for search in the included secondary studies are IEEE Xplore, ACM Digital Library, El-

[5] `http://www.jedit.org/`
[6] `https://www.mozilla.org/en-US/`
[7] `https://www.apache.org/`
[8] `http://promise.site.uottawa.ca/SERepository/`
[9] `https://data.nasa.gov/`

sevier, and Scopus. Of the 15 secondary studies, 11 searched in IEEE Xplore and ACM Digital library while nine selected Elsevier and Scopus. 11 out of 15 secondary studies searched in four or more databases. The highest number of databases searched by a secondary study was nine, reported in two secondary studies. Four secondary studies, each, searched in four databases, which is the most common choice among the selected studies and is also recommended by systematic study guidelines [53, 62]. Among the included secondary studies, three secondary studies searched in two or less than two databases, thus not fulfilling the sufficient search criteria of DARE as described by Budgen et al. [53]. Six out of 15 secondary studies searched in either three or four databases without incorporating an extra search strategy or have performed searches in a restricted set of venues.

Reported programming paradigms

The programming paradigms reported by each secondary study are categorized in Table 2.10. The programming contexts observed include object-oriented (14 secondary studies), procedural languages (8), and feature-oriented (1). Of the 14 secondary studies investigating the object-oriented paradigm, six focused solely on an object-oriented programming context. None of the secondary studies investigated links of source code metrics with external quality with focus on aspect-oriented, functional or declarative programming paradigms. This highlights an important research gap for future researchers in source code metrics. Despite the recent interest in feature-oriented development practices, only one secondary study investigated feature-oriented programming paradigm.

Quality assessment results

After applying our quality assessment criteria mentioned in Table 2.19, quality assessment results for the secondary studies are reported in Table 2.8. The range of total scores for a secondary study is 0–5. Ten studies have scored above 3.5 thus are classified as high quality secondary studies according to Curcio et al. [120] while five secondary studies are medium quality secondary studies. We excluded seven secondary studies [6] during study selection due to a quality score equal to or below 1.5, see Subsection 2.5.4. Interestingly, two out of ten SLRs do not report any quality assessment of the included primary studies. In contrast, three out of five mapping studies perform a quality assessment.

We categorized the quality assessment questions that the included secondary studies used to evaluate the quality of their included primary studies. We cat-

Table 2.8: Quality assessment of secondary studies

Type	ID	Total	AQAS*	Q1	Q2	Q3	Q4	Q5
SLR	S02	2.5	0	1	0.5	0	0.5	0.5
	S08	2.5	0	1	0	0	1	0.5
	S03	3.5	2	1	0	1	1	0.5
	S07	3.5	8	1	1	0.5	0.5	0.5
	S01	4	10	1	1	1	0.5	0.5
	S12	4	7	1	1	1	0.5	0.5
	S13	4	5	1	0.5	1	1	0.5
	S15	4	0	1	1	1	0.5	0.5
	S05	4.5	7	1	0.5	1	1	1
	S09	4.5	7	1	0.5	1	1	1
SMS	S06	2	0	1	0.5	0	0.5	0
	S11	2	0	0.5	0.5	0.5	0.5	0
	S14	2	0	1	0	0	0.5	0.5
	S04	4	4	1	1	1	0.5	0.5
	S10	4.5	2	1	1	1	1	0.5

AQAS*:Additional quality assessment score

egorized these into 18 quality assessment questions shown in Table 2.9. Seven secondary studies assessed whether the included primary studies clearly stated their objectives, study limitations, and validity threats. While, four secondary studies assessed if the primary studies justified the prediction methods used. For each secondary study, we calculated an additional quality assessment score beside the DARE score shown in Table 2.8. Eight of the 15 secondary studies considered whether their primary studies provided a detailed methodology, whether the data collection steps were clearly described, and the reproducibility of results by evaluating whether public datasets were used.

2.6.2 RQ2: Strength of evidence of link between source code metrics, internal quality attributes, external metrics and external quality attributes

This section presents the strength of evidence of the links between source code metrics (described online[6]) and external quality attributes as reported in the secondary studies. The strength of evidence is based on several factors, e.g., quality score of the secondary study, number of primary studies reporting the evidence, status of validation of metrics as reported by secondary study (see Section 2.4.5). The scheme used to interpret evidence reported in secondary studies is described in Table 2.4 and strength of evidence is classified according

Table 2.9: Categorized quality assessment questions from included secondary studies(#SS denotes the number of included secondary studies)

S.No	Quality assessment question	#SS
QA1	Whether the objectives and scope of the study are stated clearly?	7
QA2	Does the study provide relevant literature?	2
QA3	Do the research questions, purposes, or hypotheses logically flow from the introductory material?	3
QA4	Are the research questions addressed clearly?	2
QA5	Does the study have a sufficient number of citations?	2
QA6	Are the results of the study reported in a clear manner?	5
QA7	Is the programming language stated?	1
QA8	Was an analysis conducted to check for outliers in the data? (if unclear enter No)	1
QA9	Are negative findings presented?	2
QA10	Does the study discuss how the results add to the literature?	1
QA11	Does the study provide a clear description of the external quality attribute being investigated?	2
QA12	Does the study justify the prediction method used?	4
QA13	If a study deals with more than one prediction technique is the comparative analysis conducted?	3
QA14	Is there a description of limitations and threats to the validity of research?	7
QA15	Are the data analysis techniques clearly defined and described?	2
QA16	Are the predictors effectively chosen using feature selection/ dimensionality reduction techniques?	1
QA17	Did the study perform statistical hypothesis testing?	2
QA18	Whether the independent variables were clearly defined?	3

Table 2.10: Reported programming paradigms

Paradigm	Studies
Object Oriented	S01, S02, S03, S04, S05, S06, S07, S08, S09, S11, S12, S13, S14, S15
Procedural	S01, S02, S03, S04, S07, S11, S12, S15
Feature Oriented	S03

to Table 2.5. The reported evidence in the included secondary studies relates to maintainability, reliability, and security attributes and their external metrics as described in Figure 2.1.

Strength of evidence on maintainability

Nine studies report evidence on the link between source code metrics, external metrics (see Table 2.20), and maintainability. Table 2.11 depicts results from exploratory studies along with quality score, our interpretation of the reported relationship and strength of evidence. Results from explanatory studies are reported in Table 2.12. Of these nine secondary studies, two studies have medium quality while seven are of high-quality.

In the exploratory secondary studies (see Section 2.4.5), four studies (S03, S06, S08, and S09) reported correlation levels between source code metrics and maintainability. Among these, only two studies (S08, S09) reported statistical significance levels (p-values) for their results. One secondary study (S05) reported only claims without stating evidence. Change is the most often reported external metric used by two studies while one secondary study has maintenance effort. Another secondary study differentiates between changes and change-proneness, defined as the likelihood of change. Three secondary studies (S03, S09 and S15) utilise several external metrics and present aggregated results for external metrics. Thus, it is not possible to report individual external metrics used to link source code metrics with external quality attributes for these three secondary studies. Complexity is the most investigated internal quality attribute when linking source code metrics with maintainability.

From Table 2.11, exploratory studies on maintainability have varying strength of evidence ranging from moderate to very low. CBO and RFC are reported by two secondary studies to be significantly linked with maintainability with consistent moderate strength of evidence from more than 20 primary studies. Similarly, strong evidence suggests that inheritance measures NOC and DIT have insignificant relation with maintainability.

Similarly, WMC-McCabe, LCOM2 and LOC show significant links with maintainability. Interestingly, DIT has a weak positive link with maintainability when external measure is either change or effort related. Several other measures show a significant link with maintainability, including NOM, ICH, MPC, LCOM5 and DAC, though the strength of evidence for these relationships is very low.

Four explanatory studies report quantitative results linking source code metrics with maintainability. In Table 2.12, we report our interpretation of reported results (based on criteria mentioned in Section 2.4.5) linking source code metrics suites and prediction model used to explain the relationship with external metric of maintainability. While all four secondary studies achieve high DARE scores in our assessment, the overall strength of evidence of link between source code metrics and maintainability in explanatory studies is “Low” and “very low” since less than ten primary studies report evidence of the link.

Three accuracy measures are reported in the secondary studies: Mean Magnitude Relative Error (MMRE) or Mean Absolute Relative Error (MARE), Absolute Relative Error (ARE), and Ratio of true positive.[10] CK metrics and Li & Henry metrics are the most commonly used sets of metrics in explanatory studies and are used by three secondary studies (S01, S04, S07).

The combination of CK and Li & Henry metric suites used along with prediction models are largely classified as “Unclear” for sixteen different prediction techniques. One possible explanation could be the varying effectiveness of prediction models that they used in the secondary studies and the dependency on the datasets used.

Strength of evidence on reliability

Eight secondary studies report strength of evidence on the link between source code metrics, external metrics and reliability. Table 2.16 depicts results from exploratory studies along with quality score, our interpretation of the reported relationship and strength of evidence. Table 2.17 reports results from explanatory studies on reliability. Four out of the eight secondary studies have a medium DARE score while the other four secondary studies have high DARE score. Similar to the previous section, complexity is the most linked internal attribute to reliability. Among the external measures reported, fault-proneness and number of defects are the most associated measures.

In the exploratory studies, seven secondary studies report correlation results for the link between source code metrics and reliability. Among these secondary studies, only one secondary study (S09) reported confidence intervals for their results. The other secondary studies reported the correlation results without providing confidence intervals. Two secondary studies S02 and S14 state claims without giving any evidence of results. Fault-proneness is the most utilised external metric for reliability. S09 reports multiple external metrics and thus is not possible to report individual external metric used to link source code metrics with reliability.

[10]We contend that MMRE and MARE are both summary statistics based on taking numerical average of a set of Absolute Relative Errors (ARE).

Table 2.11: Strength of evidence: maintainability linked to metrics in exploratory studies

External Metrics	Internal Attribute	Source code metric	Secondary studies	Overlap in PS	DARE score	Relationship Indication					Strength of evidence
						"++"	"+"	"0"	Unclear	Sig*	
Not reported in SS	Coupling	CBO	S03V, S09	No	H, H	-	-	2	9	20	Moderate
	Inheritance	DIT	S03V, S09	No	H, H	-	-	24	8	6	Moderate
	Inheritance	NOC	S03V, S09	No	H, H	-	-	34	8	5	Moderate
	Complexity	RFC	S03V, S09	No	H, H	-	-	3	8	26	Moderate
	Complexity	WMC-McCabe	S09	-	H	-	-	1	8	28	Moderate
	Size	LOC	S09	-	H	-	-	-	-	28	Moderate
	Cohesion	LCOM2	S09	-	H	-	-	-	8	13	Moderate
	Size	NC	S09	-	H	-	-	-	1	1	Moderate
Change-proneness	Coupling	CBO	S05	-	H	-	-	-	-	11	Moderate
	Inheritance	DIT	S05	-	H	-	-	-	-	3	Low
	Cohesion	LCOM	S05	-	H	-	-	-	-	5	Low
	Complexity	RFC	S05	-	H	-	-	-	-	8	Low
	Size	SLOC	S05	-	H	-	-	-	-	14	Moderate
	Complexity	WMC	S05	-	H	-	-	-	-	10	Moderate
Change	Coupling	CBO	S06V, S08	No	M, M	2	1	1	-	-	Low
	Inheritance	NOC	S06V, S08	No	M, M	-	1	2	1	-	Low
Maintainenance Effort	Coupling	CBO	S06V	-	M	-	-	-	1	-	Low
	Inheritance	DIT	S06V	-	M	-	1	-	-	-	Low
	Cohesion	LCOM	S06V	-	M	-	1	-	-	-	Low
	Inheritance	NOC	S06V	-	M	-	1	-	-	-	Low
	Complexity	RFC	S06V	-	M	-	1	-	-	-	Low
	Complexity	WMC	S06V	-	M	-	1	-	-	-	Low
Not reported in SS	Cohesion	CAMC	S09	-	H	-	-	-	-	3	Very Low
	Stability	CoV	S03V	-	H	-	1	-	-	-	Very Low
	Complexity	DAC	S09	-	H	-	-	1	1	7	Very Low
	Inheritance	DDC	S03V	-	H	-	1	-	-	-	Very Low
	Coupling	EC	S09	-	H	-	-	1	1	-	Very Low
	-	GoS	S03V	-	H	-	1	-	-	-	Very Low
	Cohesion	ICH	S09	-	H	-	-	-	-	9	Very Low
	-	LoS	S03V	-	H	-	1	-	-	-	Very Low
	Cohesion	LCOM1	S09	-	H	-	-	3	-	6	Very Low
	Cohesion	LCOM5	S09	-	H	-	-	1	-	6	Very Low
	Coupling	MPC	S09	-	H	-	-	1	-	7	Very Low
	Inheritance	NMO	S09	-	H	-	-	4	-	4	Very Low
	Complexity	NOM	S09	-	H	-	-	-	1	7	Very Low
	Cohesion	TCC	S09	-	H	-	-	3	-	5	Very Low
	Complexity	WMC	S03V	-	M	-	-	1	-	-	Very Low
Change	Inheritance	DIT	S08	-	M	2	-	1	-	-	Very Low
	Cohesion	LCOM	S08	-	M	-	2	1	-	-	Very Low
	Complexity	RFC	S08	-	M	-	1	2	-	-	Very Low
	Size	LOC	S08	-	M	-	2	-	-	-	Very Low
	Complexity	WMC	S08	-	M	1	-	1	-	-	Very Low

Sig*: Results do not distinguish between positive or negative correlation when reporting significantly linked source code metrics

Table 2.12: Strength of evidence: maintainability linked to source code metrics in explanatory studies

External metrics	Source code metrics	Prediction model	Secondary studies	DARE Score	Relationship indicator		Strength of evidence
					"+"	Unclear	
-	Halstead metrics, LCOM, DAC & misc metrics	SFs	S15	H	1	-	Very Low
		SFSf	S15	H	1	-	Very Low
	CK & LH & Li suite	ANFIS	S07	H	1	-	Low
Change	CK & LH suite	GRNN	S01	H	-	9	Low
		ANN	S01	H	-	3	Low
		FFBN	S01	H	-	3	Low
		MLP	S01	H	-	4	Low
		PNN	S01	H	-	7	Low
		KN	S01	H	-	7	Low
		GMDH	S01	H	-	9	Low
		RT	S01	H	-	5	Low
		M5P	S01	H	-	4	Low
		BN	S01	H	-	4	Low
		SVM	S01	H	-	9	Low
		Kstar	S01	H	-	9	Low
		ELM	S04	H	-	1	Low
		MLP	S04	H	-	1	Low
		SVM	S04	H	-	1	Low
		Neuro-GA	S04	H	-	1	Low
		Neuro-Fuzzy	S04	H	-	1	Low
		GMDH	S04	H	1	-	Low
		GA	S04	H	1	-	Low
		PNN	S04	H	1	-	Low
		ANN	S07	H	-	1	Low
	CK & Tang suite	GRNN	S07	H	-	1	Low
	CK Suite	ANN	S07	H	-	1	Low
		BPN	S07	H	-	1	Low
		PNN	S07	H	1	-	Low
Maintainability Index	Halstead 'E', Halstead 'V'	Polynomial model	S15	H	1	-	Very Low
	OSAVG, CSO, CSA, SNOC	Univariate Linear Regression Analysis	S15	H	1	-	Very Low

Table 2.13: Metrics that are good indicators of external quality attributes

External Attribute	Internal Attribute	Source code metrics	Secondary studies	Relationship Indicator			Strength of Evidence
				"++"	"+"	Sig*	
Maintainability	Cohesion	LCOM2	S09	-	-	13	Moderate
	Complexity	RFC	S03V, S09	-	-	26	Moderate
		WMC-McCabe	S09	-	-	28	Moderate
	Coupling	CBO	S03V, S09	-	-	20	Moderate
	Size	LOC	S09	-	-	28	Moderate
Reliability	Code Churn	Changes	S12	16	-	-	Moderate
		Churn	S12	15	-	-	Moderate
		Age	S12	10	-	-	Moderate
	Cohesion	LCOM	S13V	-	-	15	Moderate
		LCOM5	S09	-	-	12	Moderate
		LCOM2	S09	-	-	21	Moderate
	Complexity	RFC	S06V, S12	51	-	5	High
		WMC	S06V, S12	44	-	4	High
		Cyclomatic Complexity	S12	-	43	-	High
		RFC	S13V	-	-	21	Moderate
		VG (McCabe)	S09	-	-	32	Moderate
		WMC	S13V		-	21	Moderate
		AMC	S09	-	-	35	Moderate
		WMC-McCabe	S09	-	-	62	Moderate
		NOM	S09	-	-	16	Moderate
		RFC	S09	-	-	50	Moderate
	Coupling	CBO	S06V, S12	48	-	4	High
		OMMIC	S09	-	-	10	Moderate
		CBO	S13V		-	23	Moderate
		OCAIC	S09	-	-	10	Moderate
		CBO	S09	-	-	60	Moderate
	Maturity	Past Faults	S12	-	10	-	Moderate
	Size	LOC	S12, S14V	-	61	-	High
		SLOC	S13V	-	-	16	Moderate
		LOC	S09	-	-	58	Moderate

Exploratory studies on reliability have varied strength of evidence ranging from “High” to “Very low” (due to space limitations, results graded as “Very low” are only included in the online supplement[6]). CBO, RFC, LOC and WMC show positive links with reliability with consistently high strengths of evidence from two high quality secondary studies and more than 40 primary studies. At the same time, strong evidence suggests that LCOM, DIT, and NOC have insignificant relationship with reliability.

Among the secondary studies with moderate strength of evidence, several metrics including AMC, WMC-McCabe, OCAIC, LCOM2, cyclomatic complexity and LOC have been significantly linked with reliability. However, TCC and NOC, are shown to be insignificantly correlated with reliability. Thus, we recommend that TCC and NOC may not be used when assessing reliability. One secondary study of medium quality only provides strength of evidence of a link between code metrics, error-proneness, and bug-prediction which may be considered weak compared to the results from high quality studies.

Only one explanatory secondary study, S11, reports accuracy measures as evidence of the relationship between source code metrics and the number of faults as the external metric for reliability in Table 2.17. S11 reports several different accuracy measures to describe the link between metrics and reliability. Due to the medium DARE score and fewer primary studies reporting results, the overall quantitative evidence is assigned as “low”.

Strength of evidence on security

One secondary study S03 with a high DARE score reports seven metrics correlated with security. The relationship is summarized in Table 2.14. Four metrics are positively linked with vulnerabilities as the sub-attribute of security. The description of these metrics is available online[6]. This seems to be an intuitive result as software code with several external configuration options may be easier to compromise or changing the externally configurable options may lower a module’s security settings.

2.6.3 RQ3: Efficacy of prediction models

Seven out of 15 secondary studies (S01, S02, S04, S05, S07, S11 and S15) reported models used to predict external quality attributes using source code metrics. Among these, two secondary studies S11 and S15 do not compare their effectiveness in terms of prediction accuracy due to the different focus of the secondary study. Among the studies that compare different prediction techniques

Table 2.14: Strength of evidence: security linked with source code metrics in exploratory studies

Ext sub-attribute	Internal attribute	Source code metric	Secondary study	PS in Overlap	DARE score	Relationship indicator			Strength of evidence
						"+"	"0"	"-"	
Vulnerability	Complexity	Degree Centrality (outgoing)	S03^V	-	H	1	-	-	Low
Vulnerability	Complexity	Eigenvector Centrality	S03^V	-	H	1	-	-	Low
Vulnerability	Complexity	Internal Configuration Options	S03^V	-	H	1	-	-	Low
Vulnerability	Complexity	Number of Internal #ifDefs	S03^V	-	H	1	-	-	Low
Vulnerability	Complexity	Betweenness Centrality	S03^V	-	H	-	1	-	Low
Vulnerability	Complexity	Degree Centrality (ingoing)	S03^V	-	H	-	1	-	Low
Vulnerability	Complexity	External Configurations Options	S03^V	-	H	-	-	1	Low

(for description see online [6]), only S02 reports the comparison for reliability, while other secondary studies report a comparison for maintainability prediction models. The results of the comparisons are shown in Table 2.15. In our sample, PNN, GA, ANFIS, and GMDH are superior prediction techniques for maintainability.

S01 compares the effectiveness of statistical models with machine learning based models using UIMS and QUES data-sets along with proprietary software and open source system. The secondary study notes that machine learning based techniques outperform statistical models based on linear regression and multiple linear regression. Among the machine learning techniques, Group Method of Data Handling (GMDH) based models outperform other machine learning techniques including General Regression Neural Network (GRNN), Feed Forward Back propagation Network (FFBN), Probabilistic Neural Network (PNN), KStar, Kohan Network(KN) and Support Vector Machine (SVM). S01 further observes that the effectiveness of hybrid techniques that combine statistical and machine learning models is inconclusive.

One secondary study, S07, also reported machine learning algorithms to be superior to statistical methods. Based on the results, Adaptive Neuro Fuzzy Inference System (ANFIS) based models perform slightly better than Artificial

Neural Network (ANN) and other methods reported Back Propagation Network (BPN), PNN, and GRNN.

Another secondary study, S04, investigated machine learning based prediction models and point out that the effectiveness of models is dependent on the data-set used as well as the independent variables utilised. Among the secondary studies, changing the data-set revealed different techniques to be superior. In the secondary studies that use UIMS and QUES datasets, GMDH is the superior technique with better MMRE results compared to Genetic Algorithm (GA) and PNN. Among the secondary studies that only used QUES dataset, Kstar and Mamdani-based Fuzzy Logic (MFL) perform better than other methods.

Similarly, S05 notes that search-based machine learning algorithms such as Bagging (BG) have better mean accuracy than SVM and k-means. In terms of the mean area under the curve (AUC), BG, random forests (RF), and adaptive boosting (AB) techniques are suggested to be superior.

S02 notes there is no explicit agreement on the best prediction technique for reliability among the six different prediction techniques used. The compared prediction techniques are Bayesian Network(BN), Neural Network(NN), Support Vector Machine (SVM), Clustering, Feature Selection (FS), and Ensemble Learning (EL).

When comparing the prediction accuracy of ANFIS from S04 and GMDH, GA and PNN results from S07 reporting the same combination of metrics, GMDH achieves lower MMRE values in all the four modelling techniques, thus can be suggested to be more suited for prediction studies. However, when comparing different algorithms based on the varied results in our analysed secondary studies, we can suggest that the effectiveness of prediction techniques is context-dependent and data-sensitive, and more analysis is needed in evaluating and comparing the different techniques using similar datasets and independent variables. Given the large variety of machine learning techniques, statistical methods, nature-inspired techniques and hybrid methods, this can be an interesting dimension for future analysis.

2.7 Discussion

The focus of our tertiary study is to provide a comprehensive evaluation of research on source code quality by synthesizing and categorizing the state-of-the-art literature that reports evidence-based links between source code metrics and external quality attributes in secondary studies.

Table 2.15: Comparison of prediction models reported

Superior model	**Models compared**	**Secondary study**
GMDH	GRNN, FFBN, PNN, Kstar, KNN, SVM	S01
ANFIS	ANN, BPN, PNN, GRNN	S07
GMDH	GA, PNN	S04
BG, RF	SVM, K-means	S05
None	BN, NN, FS, SVM, EL, Clustering	S02

In this section, we reflect on our results and analyses in the light of related literature.

Out of 55 secondary studies using source code metrics to study external attributes, only 15 secondary studies of sufficient quality reported evidence on the relationship between source code metrics and external quality attributes. To better understand the relationship between source code metrics and external quality attributes, more secondary studies are needed that synthesize available empirical evidence (e.g., meta-analyses). The results of the present tertiary study provides a starting point and provides information about source code metrics that show consistent relationships with two external quality attributes; maintainability and reliability.

2.7.1 External quality attributes

A small subset of quality attributes, namely, maintainability, reliability, and security from ISO-IEC 25010 [70] have been reported with link of evidence with source code metrics in our included secondary studies. All 15 secondary studies in our sample have focused on reliability or maintainability, while security has received considerably less focus (one secondary study). We have not found evidence linking source code metrics with functional suitability, compatibility, usability and other external quality attributes outlined in Figure 2.1. Colakoglu et al. [138] also observe maintainability and reliability to be the most studied external quality attributes.

Intuitively, we may argue that other external quality attributes such as usability are less related to source code metrics and more to the user interface or the ease of interaction. Thus, we have no secondary studies linking source code with usability in our included studies.

However, in other secondary studies considered during the study selection process, source code metrics have been linked with functional correctness[11] [109]. While highly relevant to our scope, this secondary study was excluded due to a low DARE score. We suggest conducting high-quality secondary studies on evaluating the link between source code metrics and other external quality attributes by extending secondary studies such as [97, 109].

2.7.2 Internal quality attributes

Complexity, coupling and size are most popular among internal quality attributes and have consistently predicted or assessed maintainability. Our results are in agreement with Arvanitoua et al. [99] who also report complexity, cohesion, and coupling to be the most frequently studied internal attributes. We also note the lack of evidence for other internal attributes such as polymorphism, abstraction, code churn, encapsulation, and hierarchies. Arvanitoua et al. [99] also observe abstraction to be a less studied attribute. While polymorphism has not been linked to any external quality attributes in our included secondary studies, Briand et al. [68] report polymorphism to be positively linked to reliability. Our results show that inheritance has a weak positive link with maintainability. One possible explanation for this could be that software systems used to study the link between DIT and maintainability have little less variation in class hierarchies. Our results agree with Saxena et al. [142], who also noted that inheritance metrics, namely, DIT and NOC, were noted to have inconsistent results for fault-proneness. Catal et al. [141] noted that complexity metrics evaluated at the method level have been predominantly used for fault prediction. We observed a similar trend in our results, where complexity metrics from Halstead and McCabe have been evaluated with multiple prediction models for fault prediction.

2.7.3 External metrics

Among the secondary studies that reported a description of the external metric, change is described as "Lines of code added, deleted, and modified" in S01. In contrast, S04 described it as "The number of changes made to the source code (changes in LOC, change in modules, change in class)." Since adding comments to a source code may also result in changes in LOC, there is a minor difference

[11]The secondary study defined functional correctness as "the degree to which a product or system provides the correct results with the needed degree of precision" and quantified functional correctness in terms of *fault count*, *fault proneness*, and *fault density*.

between the two descriptions of the external metric. Similarly, S06 describes fault-proneness as “The probability of detecting faulty classes”, whereas S14 describes it as “The probability of exhibiting a fault” without specifying the scope of the evaluation.

Since the maintainability index [126] uses complexity and size metrics to interpret maintainability, it is intuitively correlated to certain source code metrics. Thus results from prediction models that use it as a dependent variable may be misleading. However, only one of the earliest secondary study (S15) reports using the maintainability index as the external metric; we still feel that the results in the tertiary study can be relied upon and utilized. This shows that in recent years, researchers have preferred to use more accurate external metrics instead of the maintainability index.

While the accuracy of prediction of bugs and defects is an important element in assessing software product quality, the severity of bugs and defects also impacts software quality. In our tertiary study, we have only found evidence of the effectiveness of prediction. As suggested in S02, using prediction capability and bug severity into account gives a better view of software product quality.

2.7.4 Good indicators for external attributes

In this section, we synthesize the results from the previous Sections 2.6.2 and 2.6.2. We synthesized the results with “High” or “Moderate” strength of evidence and have been significantly linked with quality attributes in exploratory and explanatory studies in Table 2.11, 2.12, 2.14, 2.16, and 2.17. Table 2.13 shows several metrics to be significantly linked with reliability and maintainability. In our included secondary studies, there are no high-quality secondary studies that report the link between source code metrics and security.

Based on the results, we can suggest using this selective set of metrics over others as they may be better indicators of related external attributes. We note that certain source code metrics related to cohesion, complexity, coupling, and size can be observed to be good indicators of maintainability with consistent strength of evidence from two high-quality secondary studies reporting results from more than ten primary studies.

The strength of evidence for reliability is considerably richer as compared to maintainability. Source code metrics related to code churn, cohesion, complexity, coupling, maturity, and size are good indicators of reliability with consistent strength of evidence from four high-quality secondary studies reporting results from more than ten primary studies.

It is interesting to note that a similar set of source code metrics are suggested to be good indicators of both maintainability and reliability. These source code metrics do not belong to any particular set of metric suites. Individual source code metrics from widely popular CK, QMOOD, and Li & Henry metric suites have been shown to have varied reported evidence. Inheritance metrics DIT and NOC from CK metric suites have been shown to be poorly linked with maintainability and reliability. Variants of LCOM such as LCOM2 and LCOM5 have been noted to be more effective indicators of cohesion attributes. Complexity metrics can be considered as consistent indicators of quality attributes as we note that complexity metrics have shown strong strength of evidence for both maintainability and reliability. Code churn and past faults are also reported as good indicators for reliability but have not shown a similar link with maintainability. Also, in our results, there are no source code metrics that have a predominantly negative relation with maintainability and reliability.

2.7.5 Beyond object-oriented & static metrics

Most of the evidence available in the included studies links source code metrics with external attributes in the object-oriented and procedural programming paradigms. Only one secondary study investigated feature-oriented programming, while none of the secondary studies focused on aspect-oriented, functional or declarative programming. This limits the applicability of our results to other programming paradigms, which need to be evaluated further.

Another interesting trend is that all of the evidence presented in the previous sections is from static source code metrics. Dynamic source code metrics provide useful information regarding the software's run-time characteristics and software quality during execution. Our analysis has not reported empirical evidence of the link between dynamic source code metrics and external attributes. As suggested by Malhotra et al. in S01 and S07, the combined use of static and dynamic source code metrics to assess quality attributes may aid the source code community in improving the assessment of quality attributes using source code metrics. Tahir et al. [107] have investigated challenges in the selection and implementation of dynamic metrics, which may explain why dynamic metrics have been used less frequently. Several studies have investigated the effectiveness of using in-process metrics collected during testing [139, 140] to predict post-release software quality with promising results. The included secondary studies have not considered using in-process metrics during testing. In-process metrics may be good indicators of external quality attributes during testing compared to internal code quality metrics.

In S12, Radjenović et al. point out that in highly iterative development environments with frequent code changes, object-oriented metrics are noted to be less effective in assessing external quality attributes. In contrast to product metrics, process metrics are recommended instead of object-oriented metrics and are considered more suitable for highly iterative development environments. Malhotra et al. (S05) also recommend using process metrics in combination with object-oriented metrics such as CK metrics to improve the prediction ability of the independent variables.

The ISO-IEC 25023 standard [143] has recommended several metrics for product quality attributes identified in ISO-IEC 25010. However, one of the challenges of using the recommended metrics from ISO-IEC 25023 is that some are abstract and lack direct application to source code [144]. This may explain why none of the included secondary studies reported any of the recommended metrics from ISO-IEC 25023. The results have focused on popular metric suites such as CK, Halstead, Li & Henry metrics for reliability and maintainability. S02 suggests that other metrics not part of metric suites, such as LOCQ [134] have been noted to be effective in fault-prediction and may be explored in further studies.

2.7.6 Metrics validation

Since we have chosen to include the validity status mentioned by the secondary studies only, the validation status of source code metrics presents only the perspective available through the secondary studies. We acknowledge that using the validation status from other sources will reflect and change the results presented in Table 2.11, 2.12, 2.16, and 2.17. Metric suites such as CK metrics [50], Li & Henry metrics [131], MOOD metrics [130] and QMOOD metrics [51] have been empirically validated [81, 127, 128, 132, 133]. However, all the good indicators metrics pointed out in the results (Table 2.13) remain the same as we have included both "high" and "moderate" strength of evidence. Incidentally, all source code metrics mentioned in Table 2.13 are validated based on the information available in literature.

Certain cohesion metrics, such as LCOM, have been highlighted as theoretically invalid metric for cohesion by Kitchenham et al. (S14) and thus may be considered less effective in capturing the cohesiveness of source code. Based on our findings, we suggest LCOM1 and LCOM2 are more suitable cohesion metrics and indicators of reliability.

2.7.7 Accuracy measures

The mean magnitude of residual error (MMRE) has been predominantly used in the secondary studies reporting explanatory results related to maintainability and reliability. The MMRE accuracy measure has been highlighted as biased in the literature [135, 136]. Additionally, if primary studies use MMRE as a goodness of fit criteria for model evaluation, as well as to report the prediction accuracy, this bias may increase [145, 146]. Another reason for MMRE being biased is that the measure does not define an upper bound for MMRE values greater than one, while MMRE values lower than one cannot become negative; thus, results of MMRE from models that underestimate seem more promising. Foss et al. [137] recommend using accuracy measures after considering the characteristics of the data. Since we can only report the accuracy measures provided in the secondary studies, we acknowledge that data related to data-distribution is not reported in our tertiary study.

2.7.8 Prediction models

In S01, Malhotra et al. note that machine learning models outperform statistical models and recommend exploring hybrid prediction models for maintainability prediction. Hybrid prediction models and ensemble learners are also recommended in S07 as well. Elmidaoui et al. (S04) further note that the accuracy of prediction models has been reported to be highly dependent on datasets used in addition to the independent variables used. Two secondary studies have investigated changing the datasets while using the same source code metrics as independent variables and modeling techniques report dependency on the datasets used. Prediction models that are accurate for a variety of datasets may be considered more robust. According to S05, the selection of source code metrics as independent variables for prediction models can be based on feature selection methods to utilize only those source code metrics which contribute to the improvement of the model and can be considered to be effective predictors.

2.7.9 Datasets used

In our included secondary studies, the datasets used in explanatory studies have been predominantly UIMS, and QUES datasets (e.g., S01 and S05) among other datasets. S05 point out that validation of prediction models on industrial datasets may further improve the validity of the prediction models and may be more relevant for practitioners. Large datasets based on languages used in practice may also increase the relevance of the maintainability prediction research. In this regard, S04 highlights the need for curated datasets (such as PROMISE dataset). Additionally, Radjenović et al. (S12) suggest that realistic datasets are often unbalanced, with faults occurring randomly among the software modules. Similarly, major restructuring between two software versions may change the distribution of faults. To produce accurate fault-prediction models, the availability of realistic unbalanced datasets is needed for model training and validation.

2.7.10 Consistent tool support

Measurement tools used across secondary studies are not the same. Thus, measurement error is possible when different tools are used to evaluate the same datasets across studies. In S06, Tiwari et al. [98] point out that consistent and dependable tools for coupling and cohesion metrics are needed for reliable evaluation of source code metrics.

2.7.11 Conducting tertiary review challenges

We also want to report challenges faced during data extraction phase which led to exclusion of relevant studies. While some secondary studies report the evidence between external attributes and source code metrics, the method of aggregation utilised in the secondary study makes it difficult to trace how many primary studies have reported the relationship between specific code metrics and external measures. Hosseini et al. [93] provide a synthesis of cross project defect prediction studies. The meta-analysis performed is seminal work in our opinion. However, the accuracy measures reported for different metrics could not be traced to individual metrics along with primary studies reporting the results. Similarly, Arvanitoua et al. [99] suggest certain metrics to be effective in measuring maintainability, stability and changeability though do not provide number of primary studies reporting the trend for individual metrics. Since, one of the criterion used to establish strength of evidence was primary studies reporting individual metrics, these secondary studies were excluded from our analysis.

Another challenge faced during metrics aggregation is the non-standard method used when naming source code metrics. The inconsistency in naming of code metrics leads to several issues where the same metric is described by two acronyms or when same acronyms are assigned to two different metrics. This observation is shared by other researchers as well including Malhotra et al. (S07), Sharkawy et al. (S03), and Saraiva et al. [108]. To ameliorate the inconsistency, we also suggest an online catalog of source code metrics can be created which consistently reports metric descriptions, validation levels, along with alternate acronyms, and related internal attributes. This can be a difficult task since several secondary studies such as [117] have reported 300 metrics related to different programming paradigms and [108] report 575 object-oriented metrics for maintainability alone. Though, it could be a very effective tool for researchers, software tool developers and practitioners in the field.

While performing the quality assessment for the secondary studies, we identified several instances that are not explicitly covered by the DARE quality assessment criteria. To assess the search coverage sufficiency, the criteria assigns "yes" to secondary studies that have "searched four or more digital libraries and included additional search strategies OR identified and referenced all journals addressing the topic of interest" and "partly" to "searched three or four digital libraries with no extra strategies OR searched a defined but restricted set of journals and conference proceedings". The cases where a secondary study has searched in three or four digital libraries with extra search strategies (e.g., contacting the authors, snowballing) or when secondary studies have searched in five databases without any extra search strategy is open to interpretation and subject to the researcher conducting the assessment. Similarly, during assessment of quality of the secondary study, we noticed that some authors have explicitly defined the quality criteria and reported the results whereas others only defined the criteria and not reported any results, such as, S07, possibly due to page limitations. Since it is often the case that more than one author is involved in the quality assessment, such instances where subject assessment needs to be used, may lead to confusion and disagreements between the authors. We suggest that such issues maybe updated in the DARE criteria and software engineering community may evaluate other quality assessment tools derived from mature fields such as medicine, e.g., [47], to improve the quality assessment of secondary and tertiary studies. We also suggest that tertiary studies should not solely rely on DARE for assessing the quality of included secondary studies. DARE does not consider the strength of evidence of the results reported in secondary studies, since it does not critically appraise the quality assessment performed by the secondary studies. To perform an in-depth evaluation of evidence from secondary studies, future tertiary studies could use a similar method to our strength of evidence criteria (see Subsection 2.4.5), to complement DARE.

2.7.12 Future works

Other researchers can utilize different criteria for the strength of evidence on the data available online[6]. They can customize the results to specific usage scenarios. Additionally, researchers can extend the same schema of evaluating the strength of evidence to other secondary studies that have not been considered in our tertiary study.

We have also highlighted that no secondary studies reported links between source code metrics and external quality attributes in the aspect-oriented programming paradigm which is a research gap that needs to be investigated. The link between dynamic metrics and external attributes has not been reported in the secondary studies considered. Investigating the link between dynamic metrics and external attributes can be an interesting theme for future research. Similarly, the use of process metrics in addition to object-oriented metrics may be explored for suitability in assessing quality attributes in highly iterative development environments.

We further suggest that a catalog of source code metrics with consistent terminology and description can benefit practitioners, tool developers, and researchers. Researchers can utilize the list of secondary studies that only report using source code metrics and external quality attributes as a starting point for creating a catalog of source code metrics. Keeping in view the limitations raised in Section 2.8, we suggest further work to bridge the gap between syntactic structural information and semantic assessment of internal quality attributes. Fregnan et al. [39] have summarized semantic coupling metrics using machine learning models to link similar words and documents. This may help develop semantic cohesion metrics.

2.8 Limitations

We follow suggestions by Kitchenham et al. [43] and discuss the intentional deviations from the systematic review guidelines [31,33] followed in the design of the tertiary study. We also discuss possible implications of these deviations on the study outcomes. In addition, we discuss the threat of double counting due to overlapping primary studies (see Subsection 2.8.3).

2.8.1 Study selection

We have aimed to cover a large corpus of knowledge, however, there is a possibility that we have missed a small percentage of relevant secondary studies that were not in English language, or due to limitations of keywords in our search string and no snowballing on the final set of secondary studies. We believe that such number of secondary studies are less compared to the included set of secondary studies and unlikely to significantly alter the results presented.

2.8.2 Research validity

We acknowledge the limitation of using the DARE quality score as a measure contributing to the strength of evidence. These scores represent the quality of a secondary study and will abstract away the individual primary studies' quality. A deeper analysis can be performed where we utilise the quality of the primary studies reporting the link between code metrics and quality attributes.

In Table 2.13, we have suggested source code metrics that are good indicators of external quality attributes. Several previous studies have highlighted that source code metrics measuring the same internal quality attribute can be correlated [41]. Source code metrics that measure different internal quality attributes have also been reported as correlated [40, 42].

Source code metrics capture syntactical structural information and thus do not capture the source code's semantical complexity, which limits their applicability in assessing program comprehension.

Similarly, we note that the semantic cohesion of source code may differ from syntactic structure information related to cohesion and has not been discussed in the included secondary studies. While developers in practice may perceive semantic cohesion differently in given contexts, they use the concept of logical cohesion by preferring to place classes and functions with similar functionality in the same software packages [55]. The relationship between syntactic cohesion and external quality attributes needs to be characterized for specific contexts. As an example, while designing libraries with particular functionality may be considered good design practice, accessing attributes or methods from a library may lower the overall cohesion of the calling class, thus giving a wrongful impression of cohesion and external quality attribute being measured. Therefore, we suggest utilizing the evidence presented in our study in light of such limitations of source code metrics.

When using source code metrics for coupling, the coupling direction needs to be considered for a meaningful utilization of the coupling metrics. Arisholm et al. [37] discussed how "control classes," i.e., large classes with the bulk of functionality that depend on several smaller classes for ancillary functionality, may be easier to modify for inexperienced developers. Kitchenham et al. [38] suggested CBO to be invalid as it treats inward and outward coupling similarly.

In practice, large classes may become a “bottleneck” when they have both high outward coupling and high inward coupling. The coupling metrics reported in Table [13] do not capture such nuances. Practitioners should prioritize classes with high outward and high inward coupling for refactoring or redesign, as small changes in them may lead to multiple changes in classes linked to them.

2.8.3 Double counting

When conducting tertiary studies, there is a risk of double counting when the same primary studies are included in more than one secondary study [46]. To address the threat of double counting, we identified the secondary studies with overlap of primary studies. For such cases, we ensured that the duplicate primary studies were considered only once so that the results are not inflated due to double counting.

2.9 Conclusions

In this tertiary study, we have performed a systematic review of secondary studies that have evidence for the link between source code metrics and external quality attributes as classified by prevailing software quality standards. After an extensive search, we identified 15 exploratory and explanatory secondary studies of moderate and high quality on the subject. The linked evidence considered includes qualitative and quantitative results of the link between source code metrics and external quality attributes. We excluded secondary studies that did not report any stated evidence of a relationship between source code metrics and external quality attributes.

Results from moderate and high-quality secondary studies show that evidence of a link is only available for reliability, maintainability, and security. In contrast, other external quality attributes have not been linked with source code metrics in the included secondary studies. Only one secondary study reported a link with a security sub-attribute, highlighting the need for further studies exploring how source code metrics are linked with security and other external quality attributes.

The evidence shows that source code metrics have a varied link with external attributes depending on the external metric used as the dependent variable for external quality attributes. After evaluating individual results from the perspective of the overall strength of evidence, using the quality of secondary studies and the number of primary studies reporting the results, we report consistent results for a limited set of source code metrics and internal quality attributes. Good indicators of external attributes are provided in Table 2.13. Several source code metrics have been observed to have an insignificant or unclear link with both maintainability and reliability.

Our results aid in developing confidence in the metrics reported as good indicators, which may be useful for future studies on the subject. Source code metrics categorized as consistent good indicators of external quality attributes can aid practitioners in focusing on specific source code metrics to assess the external quality attributes of interest. Source code metrics that are insignificantly linked may be investigated for a link with other external attributes of quality.

In our analyzed secondary studies, the data sets used for validation impact the effectiveness of code metrics-based prediction models. However, GMDH-based prediction models have performed better than other models considered on several data sets.

Software development is essentially a human-centric activity. An interesting future research direction can be evaluating the link between source code metrics and external quality attributes under different development processes, e.g., continuous integration where source code undergoes regular updates due to a deployment-oriented outlook. We also note that evidence of the link has been context-sensitive. Future studies can aim to solidify evidence of the link for specific development contexts, e.g., web development, databases development, or mobile development. Thus, we can compare structurally and semantically similar source code and use the source code metrics to assess product quality objectively.

Acknowledgment

This work has been supported by ELLIIT, a Strategic Area within IT and Mobile Communications, funded by the Swedish Government. The work has also been supported by research grant for the OSIR project (reference number 20190081) from the Knowledge Foundation in Sweden.

Appendix

Table 2.16: Strength of evidence: reliability linked with code metrics in exploratory studies

External metric	Internal attribute	Metrics	Studies	Overlap in PS	DARE score	Relationship indicator							SOE
						“++”	“+”	“0”	“-”	“--”	Unclr	Sig*	
	Coupling	CBO	S06V, S12	7 PS, Re-solved	M, H	48	-	-	-	-	-	4	High
Fault-proneness	Inheritance	DIT	S06V, S12	7 PS, Re-solved	M, H	0	-	52	-	-	-	3	High
	Cohesion	LCOM	S06V, S12	7 PS, Re-solved	M, H	0	-	66	-	-	-	4	High
	Size	LOC	S12, S14V	1 PS, Re-solved	H, M	-	61	-	-	-	-	-	High
	Inheritance	NOC	S06V, S12	7 PS, Re-solved	M, H	-	-	53	-	-	-	3	High
	Complexity	RFC	S06V, S12	7 PS, Re-solved	M, H	51	-	-	-	-	-	5	High
	Complexity	WMC	S06V, S12	7 PS, Re-solved	M, H	44	-	-	-	-	-	4	High
	Complexity	Cyclomatic Complexity	S12	-	H	-	43	-	-	-	-	-	High
	Code Churn	Age	S12	-	H	10	-	-	-	-	-	-	Moderate
	Coupling	CBO	S13V	-	H	-	-	1	-	-	-	23	Moderate
	Code Churn	Changes	S12	-	H	16	-	-	-	-	-	-	Moderate
	Code Churn	Churn	S12	-	H	15	-	-	-	-	-	-	Moderate
Fault-proneness	Inheritance	DIT	S13V	-	H	-	-	15	-	-	-	9	Moderate
	Size	Halstead N1	S12	-	H	-	-	12	-	-	-	-	Moderate
	Size	Halstead n1	S12	-	H	-	-	12	-	-	-	-	Moderate
	Size	Halstead N2	S12	-	H	-	-	12	-	-	-	-	Moderate
	Size	Halstead n2	S12	-	H	-	-	12	-	-	-	-	Moderate
	Cohesion	LCOM	S13V	-	H	-	-	4	-	-	-	15	Moderate
	Inheritance	NOC	S13V	-	H	-	-	15	-	-	-	3	Moderate
	Maturity	Past Faults	S12	-	H	-	10	-	-	-	-	-	Moderate
	Complexity	RFC	S13V	-	H	-	-	-	-	-	-	24	Moderate
	Size	SLOC	S13V	-	H	-	-	-	-	-	-	16	Moderate
	Complexity	WMC	S13V	-	H	-	-	1	-	-	-	21	Moderate
	Complexity	AMC	S09	-	H	-	-	3	-	-	-	35	Moderate
	Coupling	CBO	S09	-	H	-	-	3	-	-	20	60	Moderate

Not Reported in SS

	Inheritance	DIT	S09	-	H	-	-	34	-	-	25	20	Moderate
	Cohesion	LCOM1	S09	-	H	-	-	17	-	-	21	33	Moderate
	Cohesion	LCOM2	S09	-	H	-	-	10	-	-	9	21	Moderate
	Cohesion	LCOM3	S09	-	H	-	-	8	2	-	-	10	Moderate
	Cohesion	LCOM4	S09	-	H	-	-	9	1	-	2	5	Moderate
	Cohesion	LCOM5	S09	-	H	-	-	2	3	-	2	12	Moderate
	Size	LOC	S09	-	H	-	-	0	1	-	-	58	Moderate
	Inheritance	NOC	S09	-	H	-	-	31	1	-	25	22	Moderate
	Complexity	NOM	S09	-	H	-	-	5	-	-	1	16	Moderate
	Size	NPM	S09	-	H	-	-	5	-	-	2	6	Moderate
	Coupling	OCAEC	S09	-	H	-	-	6	1	-	-	3	Moderate
	Coupling	OCAIC	S09	-	H	-	-	1	-	-	-	10	Moderate
	Coupling	OCMEC	S09	-	H	-	-	7	-	-	-	4	Moderate
	Coupling	OCMIC	S09	-	H	-	-	6	-	-	-	5	Moderate
	Coupling	OMMEC	S09	-	H	-	-	6	-	-	-	4	Moderate
	Coupling	OMMIC	S09	-	H	-	-	-	-	-	-	10	Moderate
	Complexity	RFC	S09	-	H	-	-	5	-	-	21	50	Moderate
	Cohesion	TCC	S09	-	H	-	-	11	7	-	2	1	Moderate
	Complexity	VG (Mc-Cabe)	S09	-	H	-	-	-	-	-	1	32	Moderate
	Complexity	WMC-McCabe	S09	-	H	-	-	1	1	-	19	62	Moderate
Defects	Complexity	Cyclomatic Com-plexity	S10	-	H	-	1	-	-	-	-	-	Low
Fault-proneness	Code Churn	Change Set	S12	-	H	8	-	-	-	-	-	-	Low
Not Reported in SS	Coupling	MPC	S09	-	H	-	8	0	-	-	1	-	Low

Sig*: Study S13 does not distinguish between positive or negative significance when reporting significantly linked source code metrics

Table 2.17: Strength of evidence: reliability linked with code metrics in explanatory studies

External metric	Metrics sets	Prediction model	Studies	DARE Score	Relationship indicator		Strength of evidence
					"+"	Unclear	
	6 Halstead, McCabe, LOC and Branch	CCA, NB, BAG, LR and BO	S11	M	1		Low
	BRS, EBS, EBC, BP, BS and BR	RF, SVM and MP	S11	M	1		Low
	CBO, DEPTH, LCOM and WMC	BN	S11	M	1		Low
	CK metrics	NB and J48	S11	M		1	Low
	CK & QMOOD metrics	LR,UR, MR, NN, DT, SVM, BST, RF, BAG and MP	S11	M	1		Low
	CK metrics	J48, NB.	S11	M	1		Low
	Halstead, McCabe, LOC, CK-suite & Misc	MOPSO-N, NN, BN, NB, SVM and DT(C45)	S11	M	1		Low
		AFP and ALS	S11	M	1		Low
		AR, DT, k-NN, NBC, SVM, BAG and BST	S11	M		1	Low
Fault-prediction		CC, NN-filter and TNB.	S11	M	1		Low
		DT and K-means.	S11	M	1		Low
		DT(C4.5), RF, AdaCost, Adc2, Csb2, MetaCost, Weighting, and RUS.	S11	M	1		Low
		J48 Tree, DWT and PCA.	S11	M	1		Low
		K-Means and X Means.	S11	M	1		Low
		KNN and SVM.	S11	M	1		Low
	Halstead, McCabe & LOC	LR, PR, SVR, NN, SVLR, NB and J48 Tree.	S11	M	1		Low
		MLR.	S11	M	1		Low
		NB and DT(C4.5)	S11	M	1		Low
		NB and DT(J48).	S11	M	1		Low
		NB, J48 and OneR	S11	M	1		Low
		NB, SVM and NN.	S11	M	1		Low
		RANQ,NB,MLP,K-NN and LR	S11	M	1		Low
		RF	S11	M	1		Low
		RF, BAG, LR, BST and NB	S11	M	1		Low
		RF, BAG, LR, BST and NB.	S11	M	1		Low
		RF.	S11	M	1		Low
		SA, ACP, SVM and ANN.	S11	M	1		Low
		Clustering(SK-means, Clustering, MDBC, RF, NB and J48 Tree.	S11	M		1	Low
		NB and RF	S11	M		1	Low
		NB, DT(C4.5-J48), SVM and LR.	S11	M		1	Low
		RF, BAG, LR, BST, NB, Jrip, Ibk, J48, Decorate and AODE	S11	M		1	Low
	Halstead, McCabe, LOC and Branch	RF, LDF and LR.	S11	M	1		Low
	Halstead, McCabe, LOC & Misc	ANN, SVM and DT and CCN	S11	M	1		Low
	Halstead, McCabe, LOC & Misc	CA,LR,J48 and NB	S11	M		1	Low
	LOC, CBO, LOCQ, WMC, RFC, LCOM, LCOM3, DIT & NOC	BN	S11	M	1		Low
	OO & McCabe	RF, LR, NB and DT	S11	M		1	Low

Table 2.18: List of selected secondary studies

#	Title	Study Type	Venue	Year	Ref
S01	A systematic literature review on empirical studies towards prediction of software maintainability	SLR	Soft Computing	2020	Malhotra and Lata
S02	Machine Learning Techniques for Software Bug Prediction: A Systematic Review	SLR	Journal of Computer Science	2020	Saharudin et al.
S03	Metrics for analyzing variability and its implementation in software product lines: A systematic literature review	SLR	Information & Software Technology	2019	El-Sharkawy et al.
S04	Empirical studies on software product maintainability prediction: A systematic mapping and review	SMS	E-Informatica	2019	Elmidaoui et al.
S05	Software change prediction: A systematic review and future guidelines	SLR	E-Informatica	2019	Malhotra and Khanna
S06	Coupling and cohesion metrics for object-oriented software: A systematic mapping study	SMS	Innovations in Software Engineering Conference	2018	Tiwari and Rathore
S07	Software Maintainability: Systematic Literature Review and Current Trends	SLR	International Journal of Software Engineering & Knowledge Engineering	2016	Malhotra and Chug
S08	Software change prediction: A literature review	SLR	International Journal of Computer Applications in Technology	2016	Malhotra and Bansal
S09	Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review	SLR	Empirical Software Engineering	2015	Jabangwe et al.
S10	How have we evaluated software pattern application? A systematic mapping study of research design practices	SMS	Information & Software Technology	2015	Riaz et al.
S11	Software fault prediction: A systematic mapping study	SMS	Ibero-American Conference on Software Engineering	2016	Murillo-Morera et al.
S12	Software fault prediction metrics: A systematic literature review	SLR	Information & Software Technology	2013	Radjenović et al.
S13	A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction	SLR	International Journal of Software Engineering & Knowledge Engineering	2013	Isong and Obeten
S14	What's up with software metrics? – A preliminary mapping study	SMS	Journal of Systems & Software	2010	Kitchenham
S15	A systematic review of software maintainability prediction and metrics	SLR	International Symposium on Empirical Software Engineering & Measurement	2009	Riaz et al.

Table 2.19: Interpretation of the DARE criteria by Budgen et al. [53]

Criterion	**Score**	**Interpretation**
Inclusion and exclusion	yes	The criteria used are explicitly defined in the paper
	partly	The inclusion/exclusion criteria are implicit
	no	The criteria are not defined and cannot be readily inferred.
Search coverage	yes	The authors have searched four or more digital libraries and included additional search strategies OR identified and referenced all journals addressing the topic of interest.
	partly	Searched three or four digital libraries with no extra search strategies OR searched a defined but restricted set of journals and conference proceedings.
	no	Searched up to two digital libraries or an extremely restricted set of journals.
Assessment of quality	yes	The authors have explicitly defined quality criteria and extracted them from each primary study
	partly	The research question involved quality issues that are addressed by the study
	no	No explicit quality assessment of individual papers has been attempted
Study description	yes	Detailed information is presented about each study
	partly	Only summary information is presented about the studies
	no	Details of the studies are not provided
Synthesis of studies	yes	The authors have performed a meta-analysis or used another form of synthesis for all the data of the study
	partly	Synthesis has been performed for some of the data from some of the primary studies
	no	No explicit synthesis has been performed (as in a mapping study)

Table 2.20: External quality attributes and external metrics used in the secondary studies

Study ID	**External quality attribute**	**External metric**	**Description**
S01	Maintainability	Change	Lines of code added, deleted, and modified
S02	Reliability	Bugs	Not provided in the secondary study
S03	Maintainability	Change impact	Not provided in the secondary study
S03	Vulnerability	Not Reported	Number of internal #ifdef-blocks, Number of internal configurations options, Number of external configuration options, degree centrality, eigenvector centrality, betweenness centrality
S04	Maintainability	Change	The number of changes made to the source code (changes in LOC, change in modules, change in class)
S05	Maintainability	Change-proneness	Change-proneness is defined as the likelihood that a class would change across different versions of a software product
S06	Reliability	Fault proneness	The probability of detecting faulty classes
S06	Maintainability	Maintenance Effort	The maintenance effort is measured by the number of lines changed per class
S07	Maintainability	Change	Not defined by secondary study. Only referred to as 'change' in the inclusion/exclusion criteria
S08	Maintainability	Change	Not defined in the secondary study. Only referred to as "change prediction" in the introduction
S09	Maintainability & Reliability	Multiple external metrics	The secondary study combines evidence for link between external quality attribute and source code metrics from primary studies that use multiple external metrics to measure the external quality attribute. Also, the definition of the external metric are not provided.
S10	Reliability	Defects	Not provided in the secondary study
S11	Reliability	Fault-prediction	Estimating the number of defects remaining in software systems
S12	Reliability	Fault proneness	Fault prediction models are used to improve software quality and to assist software inspection by locating possible faults. A correct service is delivered when the service implements the system function. A service failure is an event that occurs when the delivered service deviates from the correct/expected service. The deviation is called an error. The adjudged or hypothesized cause of an error is called a fault
S13	Reliability	Fault proneness	The paper does not define fault proneness, however, mentions "Defects during development are inevitable and the earlier they are found and fixed, the lesser it costs and the higher the quality of the products delivered"
S14	Reliability	Fault proneness	The probability of exhibiting a fault
S15	Maintainability	Maintainability Index	Not defined in the secondary study. In the primary studies of S15 it is defined as MI = 171 - 5.2 ln(average Halstead volume) - 0.23 (average extended cyclomatic complexity per module) - 16.2 ln(average count of lines of source code per module) + 50 sin(sqrt (2.4*(average percentage of lines of comments per module)))

Chapter 3

A catalog of source code metrics – a tertiary study

This chapter is based on the following paper:

> **Umar Iftikhar**, Nauman Bin Ali, Jürgen Börstler and Muhammad Usman. "A catalog of source code metrics – a tertiary study" In Proceedings of the International Conference on Software Quality, (pp. 87-106), Springer 2023.

3.1 Abstract

Context: A large number of source code metrics are reported in the literature. It is necessary to systematically collect, describe and classify source code metrics to support research and practice.
Objective: We aim to utilize existing secondary studies to develop a catalog of source code metrics together with their descriptions. The catalog will also provide information about which units of code (e.g., operators, operands, lines of code, variables, parameters, code blocks, or functions) are used to measure the internal quality attributes and the scope on which they are collected.
Method: We conducted a tertiary study to identify secondary studies reporting source code metrics. We have classified the source code metrics according to the measured internal quality attributes, the units of code used in the measures, and the scope at which the source code metrics are collected.
Results: From 711 secondary studies, we identified 52 relevant secondary studies. We reported 423 source code metrics together with their descriptions and the internal quality attributes they measure. Source code metrics predominantly incorporate *function* as a unit of code to measure internal quality attributes. In contrast, several source code metrics use more than one unit of code when measuring internal quality attributes. Nearly 51% of the source code metrics are collected at the *class* scope, while almost 12% and 15% of source code metrics are collected at *module* and *application* levels, respectively.
Conclusions: Researchers and practitioners can use the extensive catalog to assess which source code metrics meet their individual needs based on the description and classification scheme presented.

3.2 Introduction

During software development or evaluating open-source components before incorporating them into the codebase, measuring the quality of the software product is essential. One of the objective methods to measure the quality of a software product is through source code metrics.

Fenton and Bieman [55] classify quality attributes of a software product into internal and external quality attributes. Internal quality attributes of the source code relate to source code characteristics without accounting for the execution environment. In contrast, external quality attributes relate to how the source code behaves in the context of a specific environment. Several studies have shown a link between internal quality attributes and underlying issues in source code, such as code smells [95] and code decay [104]. Similarly, studies have also measured internal quality attributes to investigate the impact of code refactoring [90]. By assessing the internal attributes of the codebase regularly, practitioners can avoid introducing anti-patterns and incurring technical debt.

Source code metrics are often used to measure the internal quality attributes of the software. Several source code metrics have been proposed over the years. Some of the popular metric suites include Halstead metrics [185], McCabe complexity metric [186], Chidamber & Kemerer (CK) metrics [50] and Li & Henry metrics [129]. Source code metrics are utilized in several cases, e.g., defect proneness [93], bug prediction [89], assessing domain-specific software [88], and in evaluating the implementation of software product lines [86].

Source code metrics use information regarding a software product's structure and size and provide numerical values mapped to quality attributes [184]. While measuring, source code metrics target various units of code. These units of code are measured at different scope levels (e.g., at application, class, module, or function level) to gain insight into specific aspects and areas of code. As an illustrated example, *number of methods (NOM)* is described as "count of all the methods defined in a class" [51]. In this case, we measure the size of the source code by measuring a unit of code *method*, and the scope of the measurement is at the *class* level.

The large number of secondary studies reporting source code metrics provides an opportunity to collect and categorize source code metrics. Through a tertiary study, we aim to provide an extensive catalog of source code metrics reported in secondary studies, their descriptions, and classifications. The catalog of source code metrics, along with definitions and measured internal attributes, the scope of measurement can be a starting point in identifying and selecting suitable source code metrics for the specific measurement needs of researchers and practitioners.

In our previous work [191], we investigated the strength of the evidence linking source code metrics with internal and external quality attributes from 15 secondary studies. The aim of the current tertiary study is to provide an extensive catalog of the source code metrics reported in secondary studies.

The paper is structured as follows. Section 3.3 presents the related work, followed by Section 3.4 on methodology. We discuss the threats to validity in Section 3.5 and the results in Section 3.6. Section 3.7 summarizes our reflections on the results while Section 3.8 concludes the review.

3.3 Related work

Several systematic studies have synthesized source code metrics reported in the literature. Nunez et al. [117] conducted a mapping study that classified more than 300 source code metrics according to four programming paradigms, supported extraction tools, systems used for benchmarking and topics studied from primary studies between 2010 and 2015. However, the study does not report descriptions of the source code metrics.

Saraiva et al. [108] identified 67 aspect-oriented source code metrics to measure software maintainability and reported 15 aspect-oriented metrics reported by at least two primary studies. The study is limited to only one quality attribute, i.e., maintainability, and does not report aspect-oriented metrics for other quality attributes. Hernandez-Gonzalez et al. [124] focused only on design-level metrics and summarized 26 design-level source code metrics from 15 primary studies. Caulo et al. [187] proposed a taxonomy of 512 metrics that can be used for software fault prediction. These studies have specified limited scope and thus do not provide a holistic classification of the source code metrics along with their descriptions.

Arisholm et al. [189] proposed a classification of dynamic coupling metrics based on granularity, entity, and scope, though their study is limited to dynamic coupling metrics only.

In contrast, several studies provide descriptions of the frequently used source code metrics, including Briand et al. [68], Sharma et al. [188], Kaur et al. [192] but only include descriptions for the source code metrics which are part of a source code metric suite.

Lacerda et al. [56] have conducted a tertiary study on a closely related topic of code smells and refactoring. While the tertiary study does not report any source code metrics, the secondary studies included several source code metrics for code smell detection and comparing refactoring improvements. As mentioned in Section 3.4.1, we have included the secondary studies reported by Lacerda et al. [56] in the list of publications considered for selection criteria.

To our knowledge, no systematic study reports a catalog of source code metrics and classifies them by units of code and scope. We report source code metrics aggregated in secondary studies, with no limitations on the years a secondary study was published and without limiting the scope to a particular programming paradigm. We also report descriptions of all the source code metrics extracted in these studies. A comparison of the secondary and tertiary studies on the subject is provided in Table 3.1.

Table 3.1: Comparison of secondary and tertiary studies on source code metrics

Studies	Source	Years covered	Focus	Limitations
Saraiva et al. [115]	138 primary studies	1992–2011	Aspect-oriented metrics for maintainability	Other quality attributes, e.g., reliability are not in the scope
Nunez et al. [117]	226 primary studies	2010–2015	Source code metrics for AOP, OOP, FOP, tools used, datasets used	Component-based metrics are not reported, source code metrics definitions are not provided
Hernandez-Gonzalez et al. [124]	15 primary studies	1997–2016	Design level metrics	Scope focused on design level metrics; search years covered, primary studies used are not reported
Caulo et al. [187]	196 primary studies	1991–2017	Metrics for fault prediction	Scope focused on fault-prediction source code metrics only
Our earlier study [191]	15 secondary studies	1985–2020 (based on the included primary studies)	Strength of evidence linking source code metrics and quality attributes	Only investigate reported link between source code metrics and external quality attributes
Present study	52 secondary studies on source code	1976–2020 (based on the included primary studies)	Catalog of source code metrics to measure quality attributes, report various uses of source code metrics, e.g., bad smells detection	Secondary studies used as the source

3.4 Methodology

We used the guidelines by Kitchenham et al. [62] in this tertiary study to answer the following research question:

RQ 1: ***Which source code metrics are used in the secondary studies to measure internal (code quality) attributes?***

RQ 1.1: ***Which units of code are used to measure the internal (code quality) attributes?***

RQ 1.2: ***At which scope are the internal (code quality) attributes measured?***

3.4.1 Search strategy

We followed the guidelines by Petersen et al. [57] and searched in one indexing (Scopus) and two publisher databases (IEEE Xplore and ACM digital library). ACM and IEEE are among the most relevant publishers of research in software engineering [82, 83] while Scopus is one of the largest indexing services covering published articles from several publishers [48, 119]. Source code metrics are often reported in the context of measuring quality attributes. Thus, we utilized a keyword-based search [45, 62] as our primary search strategy. The search string consisted of six blocks; the first block contains synonyms for source code, the second block focuses on quality attributes measured, and the third block restricts the search results to systematic studies. The remaining three blocks limit the search results to articles and conference papers in the area of computer science written in English. We also incorporated synonyms for metrics, such as "measure" and "indicator" to improve the search string.

As depicted in Figure 3.1, we identified keywords from ISO/IEC 25010:2011 [70] and a set of 14 relevant papers already known to the authors due to their domain expertise (see *KnownSetOfPapers* in the online supplement [193]) to formulate our search string given in Table 3.2. The search string in Table 3.2 was also adapted to ACM and IEEE.

We used a set of 11 secondary studies (see *Validation set(QGS)* in the online supplement [193]) as a quasi-gold standard (QGS) [62] to evaluate the effectiveness of the search string. Two authors independently formulated the QGS, which included of 11 secondary studies [49]. We executed the search string in February 2021, which captured eight of the 11 (precision 1.46% and recall 73%) studies mentioned in the QGS. To improve the search coverage, we supplemented our search results with the secondary studies covered by Lacerda et al. [56] as they are relevant to our topic (these studies are italicized in Table 3.8). After removing duplicates, we found 711 unique publications (see Figure 3.2).

3.4.2 Selection process

We used the criteria described in Table 3.3 to select relevant papers from the search results. Papers fulfilling the Boolean expression (C0 AND C1 AND C2 AND (C3 OR C4)) were selected for full-text reading. We retained papers for the next phase if there were indications that the full text of a paper might contain relevant information. Papers that only fulfilled C5 were excluded.

As a first step, the first author excluded publications with less than eight pages and not written in English. We excluded systematic studies with less than

Table 3.2: Search string used for automated search in the study

Search string
TITLE-ABS-KEY ((("code" OR "software program" OR "software product" OR "software application" OR "software system" OR "object oriented" OR "aspect oriented" OR "feature oriented")
AND
("quality" OR "smell*" OR "pattern" OR "functional suitability" OR "performance" OR "efficiency" OR "compatibility" OR "usability" OR "reliability" OR "security" OR "maintainability" OR "portability" OR "analyzability" OR "modifiability" OR "testability" OR "compliance" OR "stability" OR "comprehension" OR "understandability" OR "understanding" OR "maintenance" OR "modularity" OR "reusability" OR "changeability" OR "evolvability" OR "modification" OR "testability" OR "evolution" OR "readability" OR "metric*" OR "measur*" OR "indicator" OR "refactoring"))
AND
("systematic review" OR "systematic literature review" OR "systematic map" OR "systematic mapping" OR "tertiary study" OR "tertiary review" OR "mapping study" OR "multivocal literature review" OR "multivocal literature mapping"))
AND
(LIMIT-TO (DOCTYPE , "re") OR LIMIT-TO (DOCTYPE , "ar") OR LIMIT-TO (DOCTYPE , "cp"))
AND
(LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "ENGI"))
AND
(LIMIT-TO (LANGUAGE , "English"))

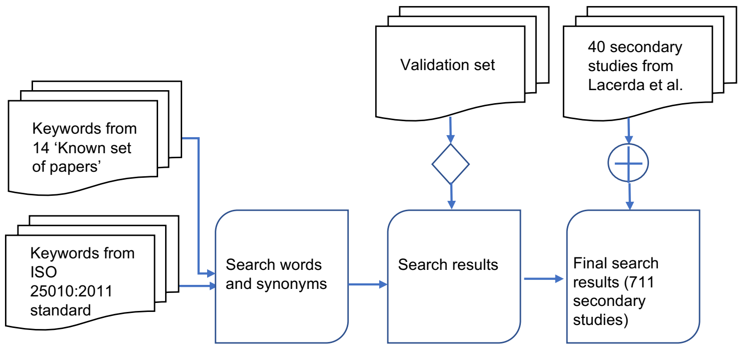

Validation set
40 secondary studies from Lacerda et al.
Keywords from 14 'Known set of papers'
Keywords from ISO 25010:2011 standard
Search words and synonyms
Search results
Final search results (711 secondary studies)

Figure 3.1: Search string generation and validation steps

Table 3.3: Inclusion/exclusion criteria used in the tertiary study

Inclusion Criteria	
C0	Publications in English language and with length of at least eight pages
C1	Peer-reviewed workshop, journal or conference publications
C2	Publications claiming to have systematically studied available literature, i.e., systematic literature studies (SLRs or SMSs) or multivocal literature studies (MLRs, MLMs)
C3	Papers that identify, describe source code metrics to measure internal quality attributes or determine levels of code quality (e.g., work on quality measurement or code smells)
C4	Papers that relate source code metrics/quality attributes/code refactoring/code smells to external quality attributes
Exclusion Criteria	
C5	Publications that are about only external quality attributes of software product/system/service, or about the quality of other artifacts like defect reports, test code, or test cases i.e., studies not related to source code metrics

eight pages as such studies are unlikely to report sufficiently detailed literature review methods and results. Out of the 711 search results in Figure 3.2, the first author identified 163 publications that did not meet the page and language requirements according to Table 3.3 and were excluded.

We conducted a pilot round of the selection process [62, 78] to improve its objectivity and to develop a shared understanding of the topic. The piloting step involved all four authors and 12 randomly selected papers from the search results, which were assessed independently by all authors as *relevant*, *irrelevant*, or *maybe relevant*. An initial agreement percentage of 58% was achieved, which is moderate. To reduce the chances of misalignment between authors and to improve the moderate initial agreement, the selection criteria were discussed during a meeting to improve the shared understanding.

From the remaining 548 secondary studies, the first author applied the selection criteria to all secondary studies, while the second, third, and fourth author were randomly assigned 182 secondary studies each, thus ensuring that each publication is reviewed by two authors. Decision making process suggested by [67, 78], was utilised. A secondary study was excluded if it was resolved as "irrelevant" and it was included if it was agreed upon as "maybe" or "included" by both authors. The initial agreement among the author-pairs was 73%. The average Cohen-Kappa inter-rater agreement between author-pairs was 0.64, which is substantial agreement [80, 81]. The disagreements during this round were resolved through discussion. After the study selection based on title and abstract, 413 secondary studies were excluded.

We have used a modified adaptive reading method [79] to conclude the relevance of papers included in the previous step. We read the paper's research questions, introduction, and conclusion sections to decide its relevance. The selection criteria listed in Table 3.3 were used to ascertain the relevance. The second author reviewed all papers excluded in this stage to reduce the likelihood of excluding a relevant publication. During the adaptive reading of the secondary studies, 36 secondary studies were further identified as not meeting the selection criteria. These excluded studies were reviewed by the second author leading to 99 secondary studies being retained for full-text reading.

During the full-text reading stage, the full text for one secondary study [84] was not available (besides our best efforts), thus, it was excluded. Two papers were identified as the same secondary studies [85, 92], and the most recent of the two secondary studies [92] was retained. The first author further identified 40 secondary studies as irrelevant to the scope, which the second and fourth authors reviewed. The authors agreed on excluding 38 secondary studies. After discussion, the remaining two secondary studies were included giving 59 secondary studies for quality assessment and data extraction.

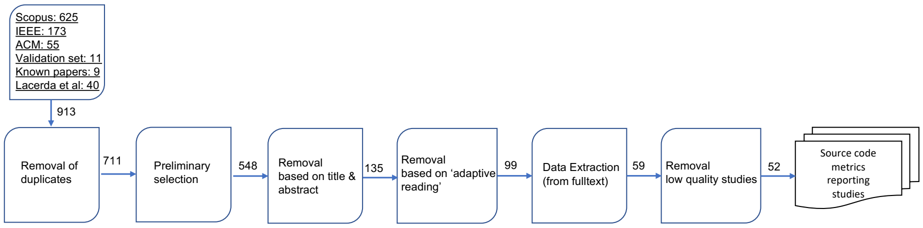

Scopus: 625
IEEE: 173
ACM: 55
Validation set: 11
Known papers: 9
Lacerda et al: 40
913
Removal of duplicates
711
Preliminary selection
548
Removal based on title & abstract
135
Removal based on 'adaptive reading'
99
Data Extraction (from fulltext)
59
Removal low quality studies
52
Source code metrics reporting studies

Figure 3.2: Selection process results (The count depicts included secondary studies at each stage)

3.4.3 Data extraction

Table 3.4 presents the data extraction form used.

Piloting of the data extraction.

To validate the data extraction form, the first and third authors independently extracted the data from a randomly selected secondary study from the validation set [193]. The authors agreed on 76% of the data extracted for one randomly

[1]when no description of the source code metric was available in the secondary study, we searched in the referenced primary studies.

Table 3.4: Data extraction form used in the study

Data Extracted
– Metadata: (author, title, publication venues, publication date)
– Search: (time period covered in the search).
– Source code quality attribute that are the secondary study's focus: (from the research questions)
– Name and acronym of the source code metric (any metrics for which the measured entity is source code or its attributes).
– Description of the source code metric[1].
– Name of the external quality attribute/sub-attribute (i.e., maintainability, reliability, security, functionality, performance, compatibility, usability, or portability [55, 70]) measured by the source code metric.
– Name of the internal quality attribute (i.e., coupling, cohesion, complexity, inheritance, or size [55]) measured by the source code metric.
– Programming paradigm
– Application domain

selected paper. The differences were discussed and resolved. The threats related to data validity are further discussed in Section 3.5.

Validation of the data extraction.

After data extraction by the first author on all included secondary studies, the fourth author randomly reviewed 5% source code metrics, internal quality attributes, and classifications assigned. The fourth author agreed with 55% of the data entries, while there were "minor issues" with 25% and 20% data entries highlighted as "major issues." The authors discussed the issues in a meeting, and the first author took remedial action to resolve the highlighted minor and major issues throughout the dataset.

3.4.4 Quality assessment of the secondary studies.

For this tertiary study, the criteria proposed by Budgen et al. [53] to answer the five DARE [194] questions were used (see online [193]). After piloting the quality assessment criteria on one study to improve shared understanding, the first author applied the DARE quality criteria on all studies followed by post-hoc validation on 10% secondary studies by the fourth author. We used the quality assessment score to remove low-quality secondary studies [62]. Inspired by [118, 120], we removed secondary studies that score 1.5 (of 5).

After removing secondary studies with scores less than or equal to 1.5, 52 secondary studies remained. As DARE is not designed to evaluate the quality of multi-vocal reviews, quality assessment-based selection was not applied to

Table 3.5: Descriptions of units of code used for categorization

Name	Description
Operators	This includes mathematical, assignment and logical operation.
Operands	This includes inputs and variables needed to perform a mathematical, logical or assignment operation
Variables	For our classification, they include attributes or variable declarations
Lines of code	A single source code statement. This include composite code statements, logical lines of code, executable lines of code, commands, point cut declarations
Comments	Comments that are part of the source code files
Parameters	Parameters include the parameters declared in method declaration, definition and its implementation
Code Blocks	Code block which span more than a line of code. It could be several lines of code inside a function, code expressions, conditional blocks of code, switch statements and variation point that span several lines of code
Functions	The methods (public, private, protected, abstract, virtual, setters, getters) or operations in a class, procedures or routines (procedural programming languages), advices (aspect oriented programming), refined/constant/base features (feature oriented programming (FOP))
Function Calls	This includes the different method calls, message requests between classes, modules, packages or components.
Classes	We include sub-classes, super classes, classes that use instances of other classes, inherited classes, parent classes, children classes, cross-cutting concerns (aspect oriented programming (AOP)), base/constant/refined features classes (feature oriented programming (FOP))
Modules	We use this terminology to loosely classify collection of classes, components, packages, libraries, sub-packages, sub-systems
Others	In the case where the software construct being measured is not clearly stated, or when stated construct is a feature or concern.

MLRs. The secondary studies removed due to low DARE scores are listed online [193]. Detailed results of quality assessment are also reported online [193].

3.4.5 Categorization of source code metrics.

We read all the source code metric names and their descriptions to identify the units of code measured and the scope at which the values of source code metrics are reported. We used a bottom-up approach to identify the units of code stated in the source code metric descriptions. The definitions of the units of code are shown in Table 3.5 while the definition of scope are available online [193]. To identify unique source code metrics, we referred to the descriptions of the source code metric. Source code metrics with the same descriptions are treated as duplicates and are combined.

3.5 Threats to validity

In the discussion below, we use the classification of threats by Ampatzoglou et al. [125].

Study selection.

During study selection, we included steps to improve the objectivity of the process. We carefully designed the inclusion/exclusion criteria before the selection process. All authors participated in the pilot rounds, and at least two authors evaluated the relevance of each secondary study. The inter-rater agreement was calculated and reported for all author pairs. All secondary studies that were excluded in the adaptive reading and full-text reading phases by the first author were reviewed by the second author. Since we excluded secondary studies with less than eight pages, some source code metrics may be excluded from our catalog. However, we believe the number of excluded source code metrics to be small and unlikely to change the overall results significantly.

Data validity.

The third author validated the data extraction form designed after discussion. We also piloted the data extraction as recommended by Kitchenham et al. [62] on 10% of the secondary studies. A post-hoc data validation was performed on randomly selected 5% secondary studies with corrective actions taken to resolve the differences. As the data extraction from secondary studies is a manual process, there is a possibility of errors in data extraction given the large data extracted for the given study.

Research validity.

We have reported the search string used, databases used, and the inclusion/exclusion criteria to improve the repeatability of the tertiary study. We regularly updated the design document of the tertiary study and recorded all intermediate results in the protocol document.

Double counting.

Double counting of extracted data can occur in a tertiary study when included secondary studies use the same primary study as their source of information. It may lead to overstating a particular result when a tertiary study aggregates findings from multiple secondary studies that utilized the same primary studies. To avoid double counting, we preferred not to perform any quantitative aggregations of results across secondary studies.

3.6 Results and analysis

Of the 52 included secondary studies reporting source code metrics (see Table 3.8), 31 are systematic literature reviews (SLRs), 20 are systematic mapping studies (SMSs), and one is a multi-vocal literature review (MLM). The earliest study was published in 2009, and 69% (36 out of 52) were published 2015–2020.

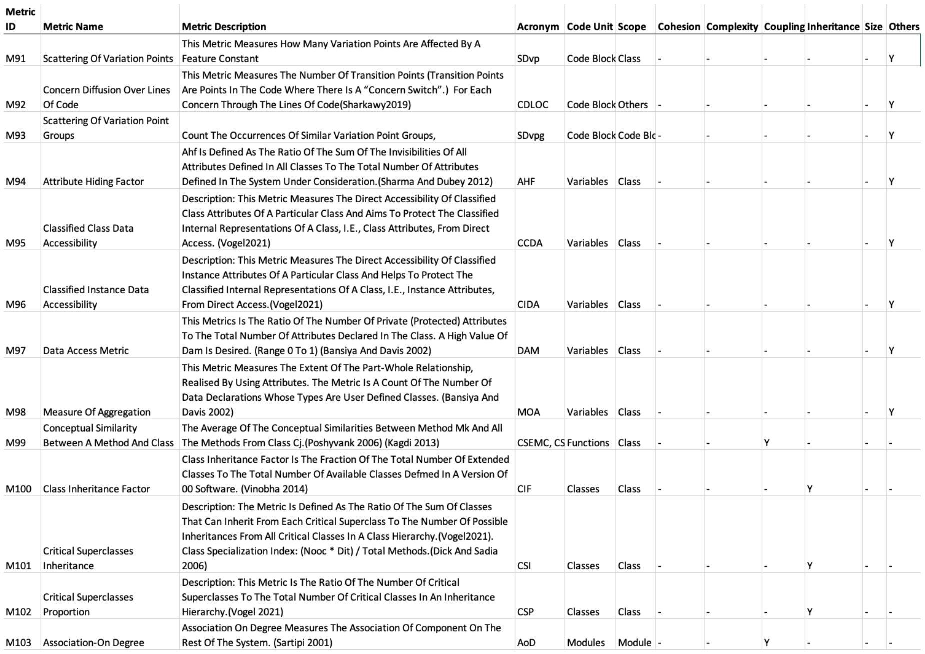
The included secondary studies report 423 source code metrics which measure internal quality attributes at different scopes. Due to space limitations, the complete list is available online [193]. Figure 3.3 shows a screenshot of the online catalog. CK metrics are among the most commonly reported source code metrics. Apart from the CK metrics suite, other frequently reported metric suites include McCabe, and QMOOD metrics.

Among the included secondary studies, 59% (31 out of 52) secondary studies report source code metrics for specific programming paradigms such as aspect-oriented (AOP), feature-oriented (FOP), procedural, and object-oriented (OOP). Source code metrics used in OOP are reported in 46% (24 out of 52) of the secondary studies, while source code metrics used in AOP are reported in 12% (six out of 52) of the secondary studies. Eight (15%) secondary studies report source code metrics used in the procedural paradigm, while source code metrics used in FOP are reported in three secondary studies. Over 50 source code metrics are reported for more than one programming paradigm.

The secondary studies use source code metrics to assess external quality attributes (27 secondary studies), evaluate software-product line implementations (four secondary studies), measure the impact of code refactoring (two secondary studies), and detect source code smells (five secondary studies).

3.6.1 Internal quality attributes

The secondary studies report 14 quality attributes measured by source code metrics which we mapped into six internal quality attributes, as shown in Table 3.6. The descriptions of the internal quality attributes [193] are based on Fenton and Bieman [55] and Bansiya and Davis [51]. Coupling, size, and complexity are the most frequently reported internal quality attributes, with 161 source code metrics reporting coupling and 78 source code metrics reporting



Metric ID	Metric Name	Metric Description	Acronym	Code Unit	Scope	Cohesion	Complexity	Coupling	Inheritance	Size	Others
M91	Scattering Of Variation Points	This Metric Measures How Many Variation Points Are Affected By A Feature Constant	SDvp	Code Block	Class	-	-	-	-	-	Y
M92	Concern Diffusion Over Lines Of Code	This Metric Measures The Number Of Transition Points (Transition Points Are Points In The Code Where There Is A "Concern Switch".) For Each Concern Through The Lines Of Code(Sharkawy2019)	CDLOC	Code Block	Others	-	-	-	-	-	Y
M93	Scattering Of Variation Point Groups	Count The Occurrences Of Similar Variation Point Groups,	SDvpg	Code Block	Code Blc	-	-	-	-	-	Y
M94	Attribute Hiding Factor	Ahf Is Defined As The Ratio Of The Sum Of The Invisibilities Of All Attributes Defined In All Classes To The Total Number Of Attributes Defined In The System Under Consideration.(Sharma And Dubey 2012)	AHF	Variables	Class	-	-	-	-	-	Y
M95	Classified Class Data Accessibility	Description: This Metric Measures The Direct Accessibility Of Classified Class Attributes Of A Particular Class And Aims To Protect The Classified Internal Representations Of A Class, I.E., Class Attributes, From Direct Access. (Vogel2021)	CCDA	Variables	Class	-	-	-	-	-	Y
M96	Classified Instance Data Accessibility	Description: This Metric Measures The Direct Accessibility Of Classified Instance Attributes Of A Particular Class And Helps To Protect The Classified Internal Representations Of A Class, I.E., Instance Attributes, From Direct Access.(Vogel2021)	CIDA	Variables	Class	-	-	-	-	-	Y
M97	Data Access Metric	This Metrics Is The Ratio Of The Number Of Private (Protected) Attributes To The Total Number Of Attributes Declared In The Class. A High Value Of Dam Is Desired. (Range 0 To 1) (Bansiya And Davis 2002)	DAM	Variables	Class	-	-	-	-	-	Y
M98	Measure Of Aggregation	This Metric Measures The Extent Of The Part-Whole Relationship, Realised By Using Attributes. The Metric Is A Count Of The Number Of Data Declarations Whose Types Are User Defined Classes. (Bansiya And Davis 2002)	MOA	Variables	Class	-	-	-	-	-	Y
M99	Conceptual Similarity Between A Method And Class	The Average Of The Conceptual Similarities Between Method Mk And All The Methods From Class Cj.(Poshyvank 2006) (Kagdi 2013)	CSEMC, CS	Functions	Class	-	-	Y	-	-	-
M100	Class Inheritance Factor	Class Inheritance Factor Is The Fraction Of The Total Number Of Extended Classes To The Total Number Of Available Classes Defmed In A Version Of 00 Software. (Vinobha 2014)	CIF	Classes	Class	-	-	-	Y	-	-
M101	Critical Superclasses Inheritance	Description: The Metric Is Defined As The Ratio Of The Sum Of Classes That Can Inherit From Each Critical Superclass To The Number Of Possible Inheritances From All Critical Classes In A Class Hierarchy.(Vogel2021). Class Specialization Index: (Nooc * Dit) / Total Methods.(Dick And Sadia 2006)	CSI	Classes	Class	-	-	-	Y	-	-
M102	Critical Superclasses Proportion	Description: This Metric Is The Ratio Of The Number Of Critical Superclasses To The Total Number Of Critical Classes In An Inheritance Hierarchy.(Vogel 2021)	CSP	Classes	Class	-	-	-	Y	-	-
M103	Association-On Degree	Association On Degree Measures The Association Of Component On The Rest Of The System. (Sartipi 2001)	AoD	Modules	Module	-	-	Y	-	-	-

Figure 3.3: Screenshot of the online catalog of source code metrics

Table 3.6: Number of unique source code metrics (column *Metrics*) reported in included secondary studies, categorized by commonly referred internal quality attributes (column *Attribute*)

Attribute	**Metrics**	**Studies**
Cohesion	56	S01, S02, S03, S05, S06, S07, S08, S09, S10, S11, S13, S14, S15, S16, S17, S18, S19, S20, S22, S24, S25, S26, S27, S28, S29, S30, S31, S32, S34, S36, S37, S38, S41, S42, S44, S46, S47, S49, S50, S51
Complexity	78	S01, S02, S03, S05, S06, S07, S08, S09, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S36, S37, S38, S40, S41, S42, S43, S46, S47, S48, S49, S50, S51
Coupling	161	S01, S02, S03, S05, S06, S07, S08, S09, S10, S11, S13, S14, S15, S16, S17, S18, S19, S20, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S34, S36, S37, S38, S40, S41, S43, S44, S45, S46, S47, S49, S50, S51
Inheritance	34	S01, S02, S03, S05, S06, S07, S08, S09, S10, S11, S13, S14, S15, S16, S17, S19, S20, S22, S24, S25, S26, S27, S28, S29, S30, S31, S34, S36, S37, S38, S41, S44, S46, S47, S48, S49, S50, S51
Size	61	S01, S02, S03, S05, S06, S08, S09, S10, S11, S12, S13, S15, S16, S17, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S40, S41, S42, S43, S44, S46, S47, S48, S49, S51, S52
Others	33	S01, S02, S05, S06, S08, S09, S11, S16, S17, S19, S20, S24, S25, S26, S27, S29, S31, S34, S36, S37, S38, S45, S49, S51

the complexity of the source code. Complexity is the most frequently reported internal quality attribute, with 88% (46 out of 52) secondary studies reporting source code metrics for it. Certain source code metrics are commonly reported. Frequently reported inheritance metrics include *Depth of inheritance tree (DIT)* and *Number of Children (NOC)*. Similarly, frequently reported complexity metrics include *Weighted method per class (WMC)*, *McCabe's cyclomatic complexity (CC)*, and *Response for a class (RFC)*.

3.6.2 Units of code in source code metrics to measure internal quality attributes

We identified 26 units of code utilized in the source code metrics descriptions, which we mapped to 13 categories of units of code [193]. Source code metrics either use a single unit of code or a combination of two or more units of code to measure the reported internal quality attribute. Out of the 423 source code metrics that measure an internal quality attribute, 25% (107 out of 423) of the source code metrics incorporate multiple units of code to measure internal quality attributes. Standalone units of code are more frequently used than composite units of code, with 75% of source code metrics using standalone units of code. *functions* (106 source code metrics) and *classes* (69 source code metrics) are among the most frequently used standalone units of code. Among the frequently used composite units of code, *classes & functions* (25 source code metrics), and *functions & variables* (34 source code metrics) are used together. The frequently used units of code vary for different internal quality attributes. Source code metrics for coupling predominantly use *classes*, and *functions & variables*. In contrast, size-related source code metrics rely equally on *lines of code*, *classes*, in addition to *functions*. Complexity-focused source code metrics analyze the *code blocks*, *operators & operands*, and *functions* to measure source code's complexity.

3.6.3 Scope of source code metric evaluation

The identified scope [193] categorize the source code at six levels of abstraction: *application - module - class - function - code-block - lines of code*. The results of the scope are depicted in Table 3.7. Source code metrics are most frequently evaluated at the *class* level, followed by *module* and *application* levels. Among source code metrics that report internal quality attributes, 216 evaluate source code metrics at the *class* level. Evaluation of source code metrics at the *class* level is the predominant trend when the scope of individual internal quality attributes is analyzed, followed by evaluation at the *others* and *application* level. Coupling metrics have the highest percentage, 102 out of 161 (63%), among reported internal quality attributes to be evaluated at the *class* level. Intuitively, none of the source code metrics are evaluated at *lines of code*. Only a small subset of source code metrics (three source code metrics) are assessed below the *function* level, suggesting that the lowest meaningful scope is at the *function* level.

One method to utilize the catalog is filtering the source code metrics list using the internal quality attribute of interest, required scope, and unit of code. The results can act as a good starting point for determining source code metrics available for the specific needs of the catalog user. As an example, selecting complexity as the internal quality attribute of choice, scope as function, and unit of code as code blocks provides 21 source code metrics and their descriptions.

3.7 Discussion

Our tertiary review provides a catalog of source code metrics and their descriptions for researchers and practitioners. We classified the source code metrics based on units of code used to measure internal quality attributes and the

Table 3.7: Scope identified for source code metrics in secondary studies

Scope	Cohesion	Complexity	Coupling	Inheritance	Size	Others	Total
Application	6	16	16	6	15	3	62
Module	10	8	22	2	4	3	49
Class	30	23	102	20	26	15	216
Functions	4	11	1	1	4	0	21
Code Blocks	0	1	0	0	1	1	3
Others	6	19	20	5	11	11	72
Total	56	78	161	34	61	33	423

scope at which the measured values are reported. We have reported six internal quality attributes measured by source code metrics in the included studies. However, we did not find any source code metrics for internal quality attributes such as messaging and hierarchies, as defined in [51]. It suggests that the two internal quality attributes are less relevant to the included secondary studies' scope, and the two internal quality attributes have received less focus in the literature. Our results show that almost 38% of the reported source code metrics relate to coupling and nearly 18% measure complexity. Arvanitoue et al. [99] also observe complexity and coupling as the most studied internal quality attributes.

Our results show that the CK metrics suite [50] is one of the most frequently used metric suites, which is consistent with other studies (e.g. [117]). Compared to Nunez et al.'s SMS [117], we report more unique source code metrics (423 in comparison to 300) and provide descriptions for source code metrics that may aid researchers and practitioners alike.

One of the challenges in source code metrics is the lack of standardization of names and descriptions. Several studies [91, 115, 116] have highlighted the inconsistency of metrics' names and acronyms, which may lead to a proliferation of source code metrics. We report 61 unique source code metrics referred to in the literature with more than one acronym (e.g., *cyclomatic complexity* is assigned several acronyms such as *CC, cyclo, MVG*, and *V(G)*). In the cases where the metric's name is not specified along with the acronym, it may mislead the audience. Using metrics' names and descriptions, we further identified 150 source code metrics that use similar units of code while aggregating the units of code at different scope levels (e.g., lines of code, lines of feature code, lines of concern code). We considered these as essentially similar source code metrics and reported them as similar source code metrics accordingly. However, we observed that the lack of standardization of names of source code metrics remains an open issue. This affects the utility provided by various source code metrics.

We observe that the units of code and scope vary when a particular programming paradigm is considered. Intuitively, such a variation is expected as different programming paradigms focus more on certain scopes than others. We note that source code metrics for feature-oriented programming are predominantly measured at the feature level or concern level, which we have classified as *others*. The most often measured units of code for procedural languages are *operators & operands*, which are more frequently assessed at the *application* level. Source code metrics reported for the object-oriented programming paradigm measure *functions* as units of code and predominantly collect metrics at the *class* level of scope. One possible reason for the difference is that applications written in procedural languages have different code structure compared to object-oriented applications, and the size of the application being investigated may also vary.

In the included studies, we observed a lack of source code metrics explicitly designed for contemporary programming languages, such as Python, Go, and Kotlin. While several open-source measurement tools exist, summarising these source code metrics may improve the utilization of appropriate source code metrics for contemporary programming languages.

Please note that the catalog currently does not provide information about which reported measurement tools also support source code metrics. Future work can report the available tool support for the reported source code metrics to improve the usability of the catalog for practitioners.

3.8 Conclusions

We analyzed 52 systematic studies reporting 423 unique source code metrics, which we have compiled into a catalog. We have intentionally excluded metrics related to change, architecture, and testing for the catalog. We have categorized the source code metrics in the catalog according to the units of code and the scope.

Our results show that source code metrics predominantly measure function-level units of code such as methods, advices, procedures, and routines. Furthermore, source code metrics frequently report values at the *class* level instead of higher scope levels, such as at the *module* or *application* level.

When reporting the catalog of source code metrics, we have not considered the validation status of the presented source code metrics. One of the future works can supplement the catalog to include the validation status of the reported source code metrics, thus improving the usability of the catalog.

Acknowledgment.

This work has been supported by ELLIIT, a Strategic Area within IT and Mobile Communications, funded by the Swedish Government. The work has also been supported by the OSIR project funded by the Swedish Knowledge Foundation (grant number 20190081).

Appendix

Table 3.8: List of included secondary studies (PS: No. of primary studies, QS: Quality score)

#	Title	Study type	Publ year	PS	Start year	End year	QS	Focus
S01	*A Systematic Literature Review on Bad Smells-5 W's: Which, When, What, Who, Where*	*SLR*	*2021*	*351*	*1990*	*2017*	*2.5*	*Bad Smells*
S02	Evolution of quality assessment in SPL: A systematic mapping	SMS	2020	63	2000	2019	2.5	Design Approach Evaluation
S03	A systematic literature review on empirical studies towards prediction of software maintainability	SLR	2020	36	1990	2019	4	Maintainability
S04	Evaluating code readability and legibility: An examination of human-centric studies	SLR	2020	54	2016	2019	3	Maintainability
S05	Software smell detection techniques: A systematic literature review	SLR	2020	145	1993	2018	3	Bad Smells
S06	A Tool-Based perspective on software code maintainability metrics: A Systematic Literature Review	SLR	2020	43	2000	2019	3	
S07	A systematic review of software usability studies	SLR	2020	150	1990	2016	4	Usability
S08	Metrics in automotive software development: A systematic literature review	SLR	2020	38	1990	2018	3	Source code metrics
S09	Machine learning techniques for software bug prediction: A systematic review	SLR	2020	31	2014	2020	2.5	Reliability
S10	How does object-oriented code refactoring influence software quality? Research landscape and challenges	SMS	2019	142	2000	2017	4.5	Refactoring
S11	Metrics for analyzing variability and its implementation in software product lines: A systematic literature review	SLR	2019	29	2007	2017	3.5	Source code metrics
S12	Software quality assessment model: a systematic mapping study	SMS	2019	31	1998	2015	3	Quality assessment models and measurement
S13	A survey on software testability	SMS	2019	208	1982	2017	2	Maintainability

Continued on next page

Table 3.8 – *Continued from previous page*

#	Title	Study type	Publ year	PS	Start year	End year	QS	Focus
S14	A survey on software coupling relations and tools	SLR	2019	136	2002	2017	2.5	Internal quality attributes
S15	Software quality measurement in software engineering project: A systematic literature review	SLR	2019	38	1984	2005	2	Quality assessment models and measurement
S16	A systematic literature review and meta-analysis on cross project defect prediction	SLR	2019	30	2008	2015	4	Reliability
S17	Empirical studies on software product maintainability prediction: A systematic mapping and review	SMS	2019	82	2000	2018	4	Maintainability
S18	A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems	SLR	2019	78	2000	2017	4	Bad Smells
S19	A Systematic Literature Review on empirical analysis of the relationship between code smells and software quality attributes	SLR	2019	74	1997	2018	5	Bad Smells
S20	Software change prediction: A systematic review and future guidelines	SLR	2019	38	2000	2019	4.5	Maintainability
S21	The impact of code smells on software bugs: A systematic literature review	SLR	2018	18	2007	2017	2.5	Bad Smells
S22	Mapping the field of software life cycle security metrics	SMS	2018	71	2000	2017	3	Security
S23	Smells in software test code: A survey of knowledge in industry and academia	MLM	2019	166	2001	2016	-	Bad Smells
S24	Coupling and cohesion metrics for object-oriented software: A systematic mapping study	SMS	2018	129	1991	2017	2	Internal quality attributes
S25	Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review	SLR	2018	76	2001	2015	4.5	Refactoring
S26	A systematic review on search-based refactoring	SLR	2017	71	2000	2016	2.5	Refactoring
S27	Software maintainability: Systematic literature review and current trends	SLR	2016	96	1991	2015	3.5	Maintainability

Continued on next page

Table 3.8 – *Continued from previous page*

#	Title	Study type	Publ year	PS	Start year	End year	QS	Focus
S28	Metrics and statistical techniques used to evaluate internal quality of object-oriented software: A systematic mapping	SMS	2016	79	2004	2013	2	Internal quality attributes
S29	Software change prediction: A literature review	SLR	2016	20	1998	2011	2.5	Maintainability
S30	Open source software evolution: A systematic literature review (part 1 2)	SLR	2016	190	1997	2016	2	Software Evolution
S31	Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review	SLR	2015	99	1996	2011	4.5	Muliple External Attributes
S32	Software metrics for measuring the understandability of architectural structures - A systematic mapping study	SMS	2015	25	1990	2013	4	Maintainability
S33	How have we evaluated software pattern application? A systematic mapping study of research design practices	SMS	2015	27	2000	2014	4.5	Design Patterns
S34	Software fault prediction: A systematic mapping study	SMS	2016	70	2002	2014	2	Reliability
S35	Software product size measurement methods: A systematic mapping study	SMS	2014	208	1982	2014	2	Internal quality attributes
S36	*Empirical evidence of code decay: A systematic mapping study*	*SMS*	*2013*	*30*	*1999*	*2013*	*4*	*Bad Smells*
S37	A systematic mapping study on software product line evolution: From legacy system re-engineering to product line refactoring	SMS	2013	74	1997	2012	2.5	Software Evolution
S38	Software fault prediction metrics: A systematic literature review	SLR	2013	106	1990	2011	4	Reliability
S39	Software clone detection: A systematic review	SLR	2013	213	1997	2011	3.5	Bad Smells
S40	A mapping study to investigate component-based software system metrics	SMS	2013	36	2000	2010	3.5	Source code metrics
S41	A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction	SLR	2013	29	1995	2012	4	Reliability
S42	A systematic review of quality attributes and measures for software product lines	SLR	2012	35	1996	2012	3	Source code metrics
S43	A systematic review of studies of open source software evolution	SLR	2010	41	1976	2009	2.5	Software Evolution

Continued on next page

Table 3.8 – *Continued from previous page*

#	Title	Study type	Publ year	PS	Start year	End year	QS	Focus
S44	A systematic review of comparative evidence of aspect-oriented programming	SLR	2010	22	1997	2008	4	Source code metrics
S45	Software architecture degradation in open source software: A systematic literature review	SLR	2020	74	2000	2019	4	Bad Smells
S46	A mapping study on design-time quality attributes and metrics	SMS	2017	154	1976	2015	2.5	Source code metrics
S47	What's up with software metrics? – A preliminary mapping study	SMS	2010	100	2000	2005	2	Source code metrics
S48	A systematic review of software maintainability prediction and metrics	SLR	2009	15	1985	2008	4	Maintainability
S49	Source code metrics: A systematic mapping study	SMS	2017	226	2010	2015	4	Source code metrics
S50	A survey of search-based refactoring for software maintenance	SMS	2018	50	1999	2016	3	Refactoring
S51	A review of code smell mining techniques	SLR	2015	46	1999	2015	3	Bad Smells
S52	*Software design smell detection: a systematic mapping study*	*SMS*	*2018*	*395*	*2000*	*2017*	*3*	*Bad Smells*

Chapter 4

On potential improvements in the analysis of the evolution of themes in code review comments

This chapter is based on the following paper:

Chapter 4: Umar Iftikhar, Jürgen Börstler and Nauman Bin Ali. “On potential improvements in the analysis of the evolution of themes in code review comments”. In Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), (pp. 340-347), 2023

4.1 Abstract

Context: The modern code review process is considered an essential quality assurance step in software development. The code review comments generated can provide insights regarding source code quality and development practices. However, the large number of code review comments makes it challenging to identify interesting patterns manually. In a recent study, Wen et al. used traditional topic modeling to analyze the evolution of code review comments. Their approach could identify interesting patterns that may lead to improved development practices.

Objective: In this study, we investigate potential improvements to Wen et al.'s state-of-the-art approach to analyze the evolution of code review comments.

Method: We used 209,166 code review comments from three open-source systems to explore and empirically analyze alternative design and implementation choices and demonstrate their impact.

Results: We identified the following potential improvements to the current state-of-the-art as described by Wen et al.: 1) utilize a topic modeling method that is optimized for short texts, 2) a refined approach for identifying a suitable number of topics, and 3) a more elaborate approach for analyzing topic evolution. Our results indicate that the proposed changes have quantitatively different results than the current approach. The qualitative interpretation of the topics generated with our changes indicates their usefulness.

Conclusions: Our results indicate the potential usefulness of changes to state-of-the-art approaches to analyzing the evolution of code review comments, with practical implications for researchers and practitioners. However, further research is required to compare the effectiveness of both approaches.

4.2 Introduction

Modern code reviews are a common step in software quality assurance [149, 158, 161]. Apart from improving software quality, modern code reviews also support sharing knowledge with developers new to a code base [148, 162]. The reviewer comments provided in code reviews may provide useful information for system development and evolution. Studies have shown that approximately 75% of the issues discussed in code review comments relate to improving the maintainability of the software and 25% of the feedback relates to improving its functionality [155, 159]. From the perspective of project and quality managers, it would be interesting to identify common themes within the code review comments and to analyze how these themes evolve over time. Such an analysis could help project and quality managers to understand theme evolution and aid them in introducing systematic improvements, improving company-wide development guidelines [153], or identifying training needs for developers.

The number of code commits and associated code reviews in medium to large projects can lead to hundreds of code review comments [162]; thus, manually analyzing the evolution of common themes in code review comments is infeasible. Previous studies have manually identified quality-related themes in a small set of code review comments [155, 159]. Automatic classification of code review comments is an open research question, and recent studies have also used machine learning approaches with promising results [153, 160, 163, 170]. Recently, Wen et al. [170] demonstrated that traditional topic modeling approaches can be used to identify common themes within code review comments and to analyze their evolution over time.

In this study, we analyze the design and implementation choices in Wen et al.'s approach to studying the evolution of common themes in code review comments. We started by replicating the research design used by Wen et al. to propose potential alternatives to their approach. We then empirically analyzed each alternative to demonstrate its potential impact.

The remainder of the paper is organized as follows. In Section 4.3, we discuss related works, whereas in Section 4.4, we cover the adopted methodology. Section 4.5 presents our findings, followed by a discussion of the results (Section 4.6) and a discussion of threats to validity (Section 4.7). Section 4.8 concludes the paper.

4.3 Related work

Several studies have manually analyzed code review comments. Mäntlya et al. [155] manually analyzed nine industrial and 23 academic systems, classified code defects discussed in code review, and proposed a taxonomy of defects discovered in code reviews. Beller et al. [159] manually analyzed two open-source systems and studied 1400 code changes in code review to identify fixed code issues, thus demonstrating the practical benefits of the code review process. Gunawardena et al. [168] provided a fine-grained taxonomy of defects discussed in code review comments by manually analyzing 417 code review comments. They further identify which code defects can be resolved using existing static analysis tools to reduce the overall effort required in the modern code review process. However, none of these studies utilize an automated method to analyze code review comments and thus are time intensive to implement.

In contrast, other studies have explored the feasibility of automating code review comments. Tufano et al. [163] utilized deep learning to select candidate code review comments from the repository of code reviews for a given code commit with up to 31% accuracy. Hong et al. [166] improved accuracy to 42% using a retrieval-based code review comments tool. This was further improved by Zampetti et al. [169], who used cosine similarity between code review comments and CheckStyle warning descriptions. They presented an automated approach for configuring the CheckStyle [1] static analysis tool, achieving a precision of 61% and a recall of 52%. However, the scope of these studies is towards automation using code review comments for automatic code review comment generation or configuring static analysis tools, which differs from our goal of analyzing code review comments.

Arafat et al. [156] used supervised machine learning algorithms to categorize and predict the topics in code review comments from six closed-source systems and reported 63% accuracy for Support Vector Machine (SVM) method. However, their approach needs a manually labeled dataset for the initial training.

Ochodek et al. [153] utilized the Bidirectional Encoder Representation from Transformers (BERT) language model to automatically classify 2,672 code review comments from three open-source systems to classify discussed topics within code review comments and achieved an average accuracy of over 80% when compared to manually classified code review comments. However, their study only considers a small set of 2,672 code review comments, and its effectiveness on a large dataset is yet to be evaluated.

Wen et al. [170] investigated how community and personal feedback trends evolve as the community matures using topic modeling. They utilized Latent Dirichlet Allocation (LDA) on one open-source system, Nova, and one closed-sourced system to study the evolution of themes in code review comments from 2011 and 2018. They considered topic stability [176] over five runs of LDA to select a suitable number of topics and assessed values between N=[10..50] as the range to search for a suitable number of topics. Their results show that the context-specific and technical feedback increases with the community's maturity and improved reviewer experience. Our work extends their study by identifying decision points where other alternatives may lead to better results. Each potential alternative proposed in this study is also demonstrated by using code review comments from three open-source systems.

[1] `https://checkstyle.org/`

Silva et al. [154] surveyed how different topic modeling methods have been used for various tasks in software engineering. Qiang et al. [150] provided a taxonomy of short-text topic modeling approaches along with an open-source Java implementation of the studied topic modeling methods and compared the performance of eight topic modeling methods on six datasets. Their results show that Dirichlet multinomial mixture (DMM) based models perform best on all considered datasets. We used their survey to identify candidate techniques to consider instead of LDA as used by Wen et al. [170].

4.4 Methodology

As stated in Section 4.2, our study explores potential improvements of the promising approach developed by Wen et al. [170] to analyze common themes discussed by reviewers using topic modeling. To achieve the stated study aims, we define the following research question:

RQ1: ***How can we improve state-of-the-art approaches to study the evolution of code review comments?***

RQ2: ***Which common themes in code review comments can we identify using the suggested modifications to Wen et al.'s approach and how do these themes evolve over time?***

Using Wen et al.'s [170] research design as a baseline, we investigate possible improvements in the choice of algorithm, the strategy of selecting a suitable number of topics, and the approach for analyzing topic evolution. Specifically, we compare the topic stability of topics generated by traditional topic modeling used by Wen et al. [170] and the short-text topic modeling method summarized by Qiang et al. [150]. Within Wen et al.'s [170] approach, we suggest alternate strategies for selecting a suitable number of topics and an alternate method to analyze the evolution of code review comments. An overview of the steps followed in our study is depicted in Figure 4.1. In the figure, the approach by Wen et al. [170] is depicted in blue, and our changes are in green.

4.4.1 Datasets

We utilized code review comments from three open-source software (OSS), two OpenStack projects, Nova and Neutron, and LibreOffice. We included the same open-source system, Nova, as Wen et al. [170] to compare the performance of

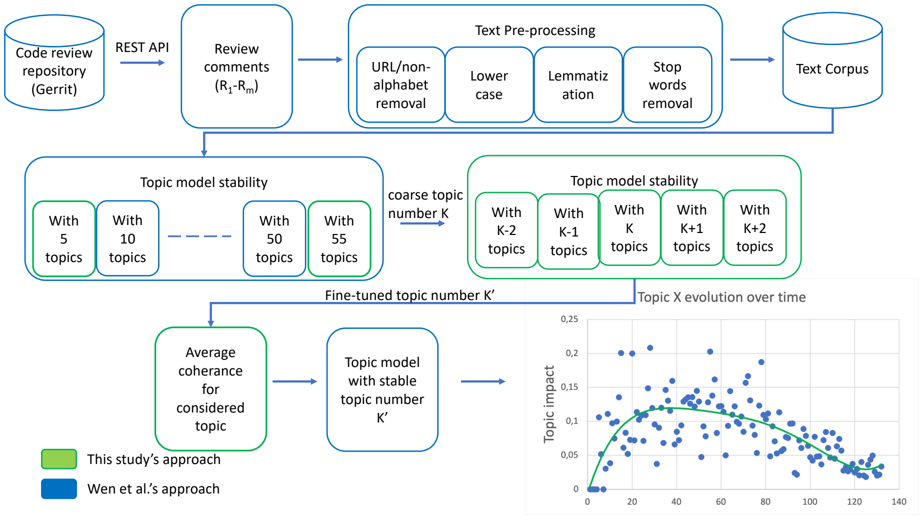


Figure 4.1: Data extraction, pre-processing and topic modeling process, based on Wen et al. [170]

the short-text modeling method and LDA. The OpenStack community develops various storage and networking solutions and has been previously investigated in the studies on the code review process [171, 172]. The code review process of LibreOffice[2], an open-source office suite, also has been studied in the literature by several researchers [173, 174].

We used the REST API[3] provided by Gerrit to extract the code review comments from all three OSS. To study the evolution of code review themes over an extensive period, we extracted the code review comments between September 2011 till February 2023. We also only considered code review comments with more than two words.

4.4.2 Natural language processing model selection

One of the natural choices for the unsupervised classification of text data is LDA method [147]. LDA assumes that each document consists of a set of latent topics, whereas each latent topic consists of a group of words. LDA is shown to have

[2]https://www.libreoffice.org/discover/libreoffice/
[3]`https://gerrit-review.googlesource.com/Documentation/rest-api.html`

degraded performance for short text [151] due to reduced word co-occurrence in short texts for topic extraction. Code review comments are relatively short pieces of text [152]. Therefore, we considered topic modeling models suitable for short text, as suggested by Qiang et al. [150].

Qiang et al. [150] compared several short-text topic modeling (STTM) methods and traditional LDA on six datasets. They found that STTM outperforms LDA in all six cases regarding classification accuracy and purity. In particular, Dirichlet multinomial mixture (DMM) based models outperformed global word co-occurrence-based short-text topic models and self-aggregation-based short-text topic models. We selected the Gibbs Sampling-based Dirichlet Multinomial Mixture model (GSDMM) [175] from among the DMM-based models due to its comparable classification accuracy and superior execution performance [150].

4.4.3 Data preprocessing

We replicate the pre-processing steps taken by Wen et al. [170] as depicted in Figure 4.1. After data extraction, we removed responses from the authors of the code commit to focusing only on code review comments from the reviewers. We also removed all brackets and punctuation to clean the code review comments and converted all code review comments to lowercase. We further removed URLs and words containing numbers as they did not contribute to the classification of themes in the code review comments. Next, we removed stop words using the built-in preprocessing library in the Gensim natural language processing toolkit. We also lemmatized all code review comments and removed any null strings in the code review comments.

Table 4.1: Overview of code review comments in example systems.

OSS Name	**Total code review comments**	**Average comment length (in words)**	**Stable number of topics**
Nova	102,642	28.2	24
Neutron	78,196	24.3	4
LibreOffice	28,328	26.8	11
Total	209,166	26.4	-

4.4.4 Parameter selection

The topic modeling algorithm's hyperparameters, such as the topic-to-document probability (*alpha*), word-to-topic probability (*beta*), and the number of topics (N), impact the performance of topic classification. A larger value for N may lead to fragmentation of topics, while a smaller value may tangle topics, thus compromising the semantic meaning of generated topics. A lower value for (*alpha*) limits the model to fewer topics per document, while higher (*beta*) results in a higher number of terms per topic. Generally, lower hyperparameter values lead to a more decisive model [181].

As suggested by Agarwal et al. [176] and used by Wen et al. [170], we use topic stability as a measure to identify suitable values for N for each corpus independently. Topic stability, a modified measure of cross-run similarity of topics based on Jaccard similarity [176], is the median number of word-terms occurrences in all considered runs for a given topic number. Extending the possible values of N used by Wen et al. [170], we considered N=[5..55] (in steps of five) for the number of topics to analyze topic stability and topic coherence for each of our datasets. We trained the GSDMM models for five runs, with the selected corpus sorted in a different order and varied choices for hyperparameters, *alpha*, and *beta*. We used the 10 top words from five runs for the GSDMM model to calculate topic stability for each dataset and identified the most stable number of topics. While Wen et al. [170] only evaluated a suitable choice of N in steps of five, we propose a two-stage approach for selecting a suitable N. As a first step, we select the most stable choice of N in steps of five (as did Wen et al.); we then iterate in steps of one in the neighborhood to find a more stable value for N.

In addition to using topic stability as suggested by Wen et al. [170], we also evaluate average coherence for all considered values of N. Using topic stability and average coherence value to select a suitable number of topics, N, in the previous step, we generate topics from the corpus and store the topic membership probabilities for each code review comment in the corpus. To compare the topic stability of GSDMM with LDA, we also repeated the above process for the LDA model.

4.4.5 Topic naming

Several methods have been utilized in the literature to assign an appropriate name to a topic. Silva et al. [154] classified these approaches as manual, automated, and a combination of manual and automated procedures. The manual naming approach has been frequently used in software engineering [170, 177, 178]. To identify a topic name, two authors read the top 20 words associated with a topic and the 20 unprocessed code review comments that have the highest topic membership to the topic. We read the code review comments belonging to each topic and assigned a label that captured the central theme within each code review comment. Then we reviewed all the labels belonging to a topic and assigned an appropriate name that captured most of the themes within the topic.

4.4.6 Topic impact evolution

Like Wen et al. [170], we study a given topic's evolution by plotting the topic's impact for each month, along with a five-degree polynomial-based regression line. A topic's impact is defined as the proportion of code review comments belonging to a specific topic within a month [179]. As suggested by Wen et al. [170], we used a low cut-off score for topic membership and included only code review comments with a topic membership probability ≥ 0.1 when calculating a given topic's impact.

In addition to Wen et al. [170], we also consider top code review comments belonging to months where the topic impact for a given topic has an interesting pattern. This may provide insight into how themes change over time and whether their sub-themes can be analyzed.

4.5 Results and analysis

After preprocessing the data as described in Section 4.4.3, we got 209,166 code review comments from three OSS after removing approximately 84K code review comments due to short length. Their details are depicted in Table 5.1. The distribution of code review comments by length for the three OSS is depicted in Figure 4.2. The evolution of the topic impact for all generated topics is provided in the replication package[4]. Here we first discuss our findings regarding the possible improvement areas. We also discuss themes for the system Neutron as a demonstration of the modified approach.

[4]`https://doi.org/10.5281/zenodo.7836738`

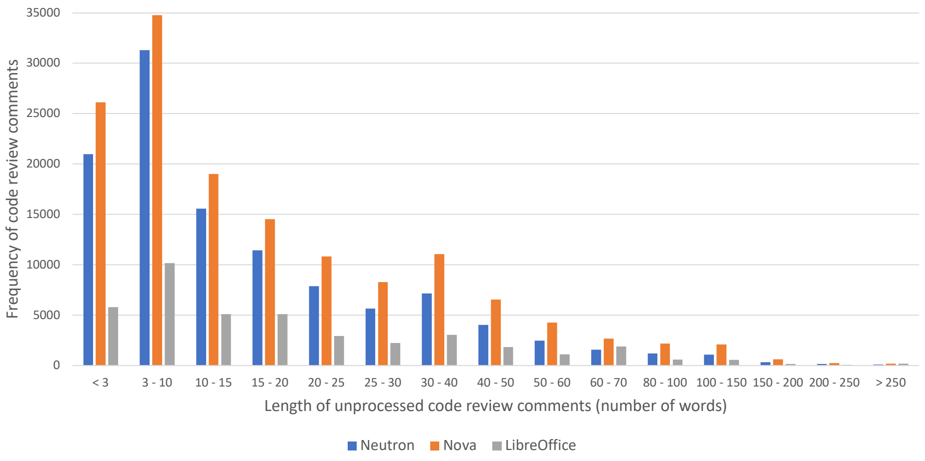


Figure 4.2: Distribution of code review comments by length for the three example systems

4.5.1 Improvement suggestions

Two-stage vs single-stage selection of N

As described in Section 4.4.4, we utilized a two-stage process for selecting a suitable N, improving the average topic stability for the considered systems. Figure 4.3 depicts the average topic stability for the most stable choice of the number of topics for the single and two-stage topic selection. We observe that LibreOffice shows the highest improvement when using the two-stage topic selection, with average topic stability improving from 0.82 to 0.86.

Topic modeling method

To evaluate the variation in topic stability for the three systems for different values of N, we plot the average topic stability for all five runs for both GSDMM and LDA in Figure 4.4. Apart from system Neutron for N=4, GSDMM produces substantially more stable topics when compared to LDA for the considered systems. GSDMM, on average, produces 38% more stable topics than LDA for system Nova. For the system Neutron, the average topic stability improvement is 32%. LibreOffice showed a similar pattern apart from the topic stability ratio for N=5. Based on the empirical results, short-text topic models such as

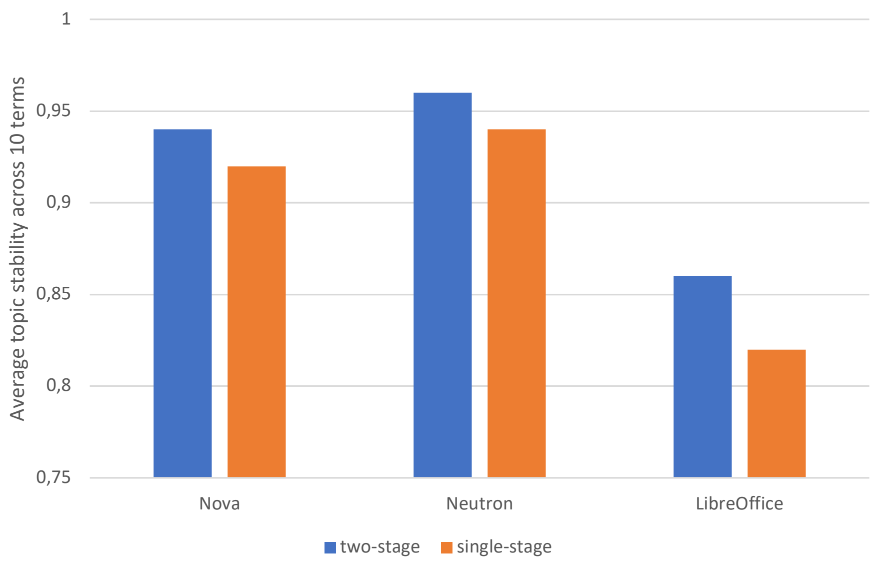


Figure 4.3: Average topic stability for top 10 terms for two-stage (blue) and single-stage (orange) topic selection

GSDMM may be further evaluated when studying the evolution of code review comments.

We also observe that the most stable number of topics, N, may be less than 10, as in the case of Neutron, regardless of the topic modeling method used. There might be systems with most stable topic beyond N=55, thus plotting average topic stability for such systems may aid in selecting the upper and lower bounds for N.

Topic stability and coherence

We also evaluated the average coherence of the topics for different values of N for all five runs. The results for both LDA and GSDMM for the three example systems are depicted in Figure 4.5. Compared to GSDMM, LDA achieves a slightly higher average coherence for all considered systems. The average coherence gradually increases with N for the considered systems and both algorithms, except for LibreOffice, where the average coherence declines after a peak around N=25. The highest average coherence for Neutron and Nova is

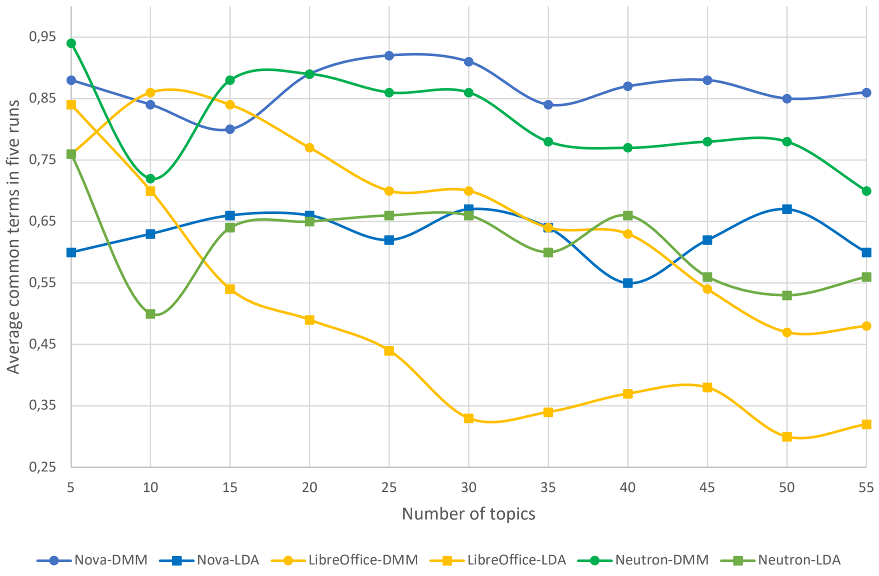


Figure 4.4: Topic stability comparison using GSDMM and LDA for the three example systems

0.60 and 0.63, respectively, for N=55. In comparison, the trend for average topic stability is more varied for the increase in N. The contrasting results need further investigation.

The average coherence for the most stable number of topics is only slightly less than the most coherent topic for all considered systems. We preferred that the minor decrease in average coherence is an acceptable trade-off in selecting a suitable choice of N to study the evolution of the prevalent themes in code review comments.

Evolution analysis: one vs multiple windows

We hypothesize that some themes observed in code review comments may become obsolete with time as systems, processes, and methods to maintain the code base evolve. In contrast, new themes may replace them as code reviewers focus on current concerns within the submitted code. This raises the question regarding the timeframe to select for the analysis of themes. Choosing a very long duration may make it challenging to find relevant representative themes

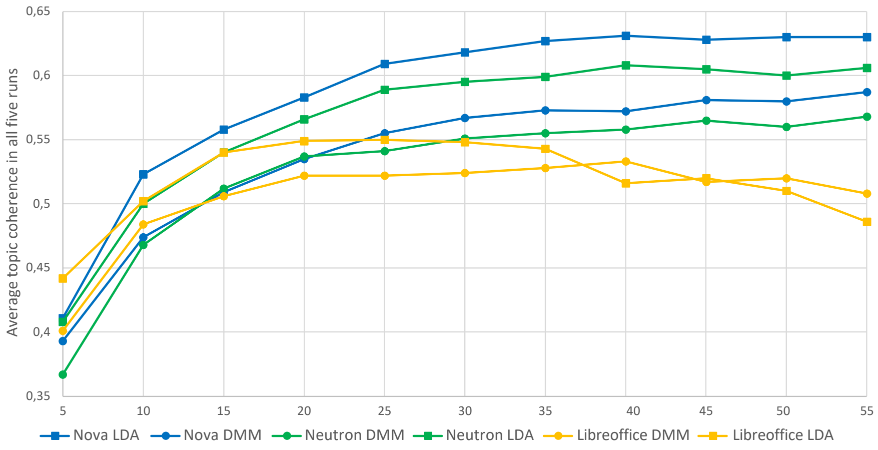


Figure 4.5: Average coherence graph using LDA and GSDMM for the three example systems

for all months considered. Similar to the approach by Wen et al. [170], we used a single window for the duration considered. In contrast, multiple windows approach divides the entire duration into sub-parts for a more refined analysis of the themes in code review comments.

As an exploration of the multiple windows approach, we analyzed the theme related to "guidelines" given its interesting evolution (see Figure 4.8). After generating the overall theme, we analyzed the top 20 code review comments from months when the trend changed significantly, along with code review comments from beginning to end, to evaluate if the issues discussed in the code review comments belonging to the topic evolved. We considered code review comments from November 2011, February 2020, and January 2023.

In the initial months, code review comments related to the code review process appear more frequently in the guidelines theme (e.g., "*...I think this would usually be a separate commit...*"). In February 2020, the guidelines theme focused more on code styling suggestions (e.g., "*L.28-38 belongs to L.26 so it looks better to indent these lines.*") while in the ending month, the guidelines theme considered code review style-related suggestions to be more critical (e.g., "*The patch is ok please remove the second line in the commit message.*"). One possible explanation for this trend is that guidelines provided by Neutron's core reviewers evolved with the change in contributing developer behavior.

Considering the pace of development technologies, processes, and practices improvements, we suggest generating topics in shorter windows, e.g., four months, rather than the entire duration. Using a "windowed" approach may refine the quality of themes identified as we can assess the impact of newly introduced interventions, e.g., company-wide procedural changes, in code review comments.

4.5.2 Named topics

In this section, we discuss the themes identified after generating topics from the Neutron system using the fine-tuned topic choice discussed above. The plots for topic evolution from the other two systems are provided in the replication package. The depicted scatter plots show the topic impact (see Section 4.4.6) over the months considered. We also draw a polynomial-based trend line to represent the overall evolution of the topic impact across months. For Neutron, if all four topics were equally divided, we would expect an average topic impact of around 0.25 for each topic. We represent this with a continuous horizontal line at 0.25 in the generated plots.

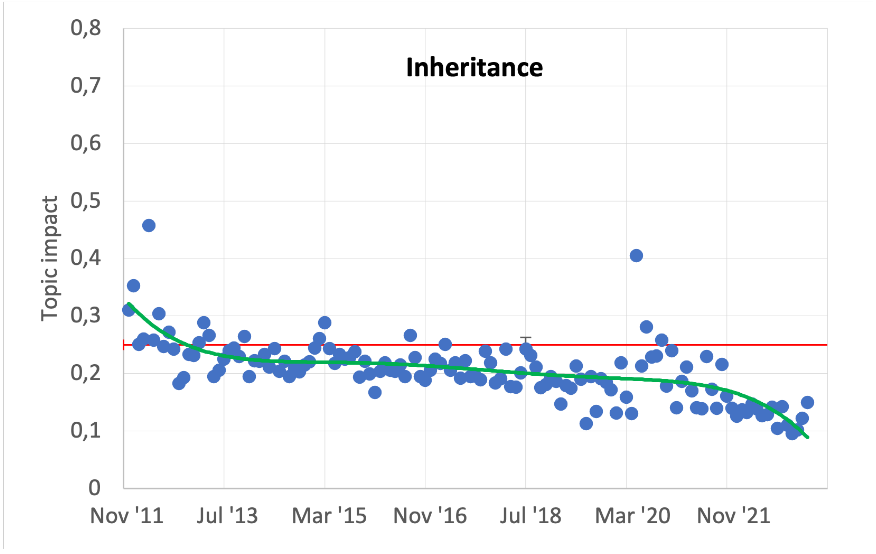


Figure 4.6: Topic evolution for theme Inheritance in system Neutron.

Theme 1: Inheritance

We define inheritance-related themes as code review comments that discuss how classes inherit other classes, shared attributes and methods in multiple inheritance instances, and misuse of inheritance, among others. The ratio of code review comments belonging to the inheritance-related theme or topic share [179] is 0.33. The regression line shows that while the reviewers discussed inheritance-related issues more frequently in the initial months, the theme evolved with a gradual decrease in the topic impact over the duration considered. We reproduce an example of a code review comment discussing an inheritance-related theme.

> "Althrough it's done everywhere is this class it's not pythonic to use the following (java?) pattern: *class MyClass: @staticmethod def method1(): pass @staticmethod def method2(): MyClass.method1()* because it breaks python inheritance principle: if a subclass *SubClass* of *MyClass* overloads method1 *SubClass* method2 will still use *MyClass.method1*! ..." (Italics added to improve readability)

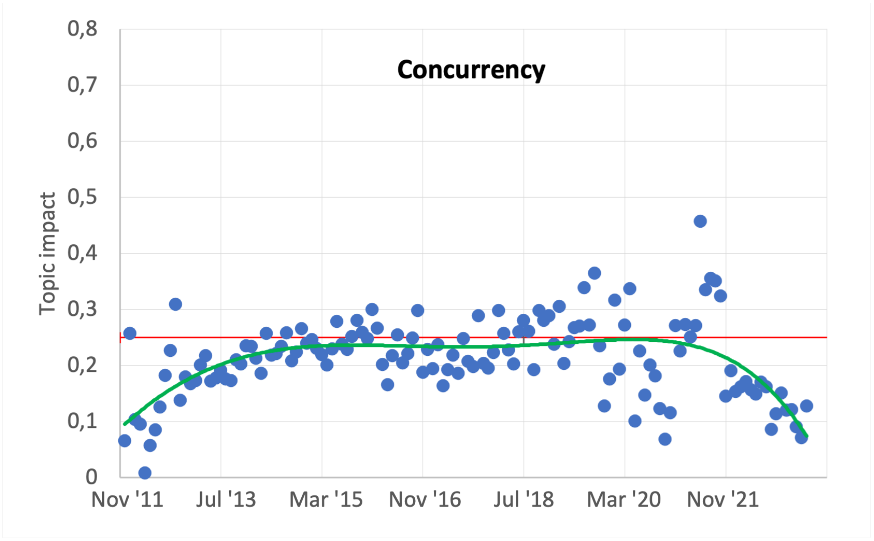


Figure 4.7: Topic evolution for theme concurrency in system Neutron

Theme 2: Concurrency

We define concurrency-related themes as code review comments that discuss timing issues in a multi-threaded application, event handling, and network performance management. The topic share for concurrency-related code review comments is 0.39, making it the most prominent theme for Neutron. The evolution of this theme shows that topic impact has been steady between March 2015 till March 2020, after which it gradually declined to the levels observed before this period. An example of a code review comment discussing a concurrency theme is reproduced as follows.

> “Not sure this is a good idea... what happens if another thread creates a new floating IP on this router between the time that another thread called this function and we get here? ...”

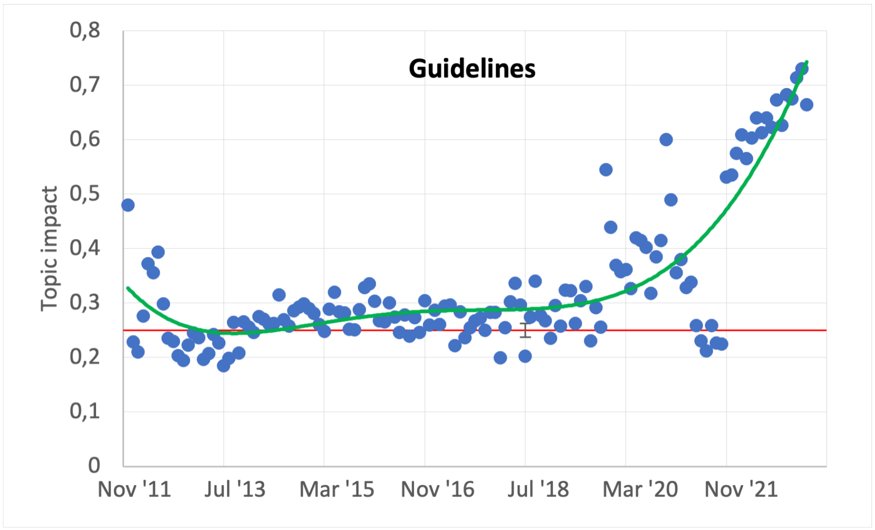


Figure 4.8: Topic evolution for theme guidelines in system Neutron

Theme 3: Guidelines

We define the guidelines theme as code review comments focusing on issues related to formatting code commit messages, patch-related procedural guidelines, code styling suggestions, recommendations of what should be included in a commit message for clarity, and explaining why it was required. We observe that the guidelines theme undergoes a significant change during its evolution over the months considered. While the code review comments related to the theme remained steady between July 2013 till March 2020, there is a steady increase in code review comments related to guidelines. It would be interesting to analyze further the reasons for the sharp increase in the guidelines theme, which may lead to process-level improvement in how code reviews are performed in the Neutron community.

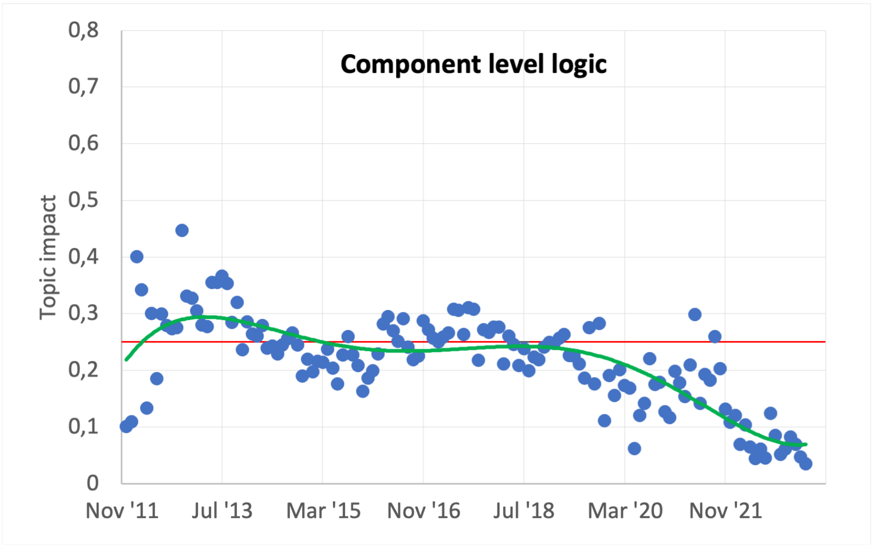


Figure 4.9: Topic evolution for theme component level logic in system Neutron

Theme 4: Component level logic

We define component-level logic themes as code review comments discussing logic-related issues at the method or class level, such as conditional logic, suggestions for additional logic, tips to replace loops with alternate implementation, and recommendations for introducing alternate parameters in the implemented method. The percentage of code review comments belonging to this theme is only 17% making it the least discussed theme in Neutron. We observe that the topic impact for component-level logic theme remains steady till July 2018, after which it gradually declines below 0.1. We reproduce the following code review comment belonging to the theme.

> "You can fold lines 130–133 into the for loop as *ichain_name* an *ochain_name* and then remove 137 138 141 142 So pull those 8 lines and add these two after *if* clause ending on line 142. ..." (Italics added to improve readability)

4.6 Discussion

We suggested three potential improvements to the approach of Wen et al. [170]. The two-stage selection of a suitable number for N improves topic stability compared to the single-stage approach. GSDMM produces substantially more stable topics as compared to LDA. Similarly, using average coherence in addition to topic stability provides a comparison between the two measures. Using the modified approach, we were able to identify several themes.

Themes similar to those we identified using our modified approach have also been discussed in the literature. The theme *component-level logic* is synonymous with *Logic* issues identified by Mäntyla et al. [155] and *code_logic* by Ochodek et al. [153]. The theme *concurrency* bears similarity with *Timing* issues identified in Mäntyla et al. [155]; however, we classify event handling and network-related performance in *concurrency.* The *guidelines* theme includes *code_style* issues from Ochodek et al. [153], *visual representation* issues [155], as well as *code review process* [170]. The theme *inheritance* covers similar concepts as *structure* [155] and *code_design* [153] though it takes a fine-grained view of design-related code review comments. Intuitively, the topic share [179] for the themes can vary across the systems studied in the previous studies.

We achieved a different number of N for topic stability using the modified approach compared to Wen et al. [170] for Nova, possibly due to the difference in the duration considered and the topic modeling method used. Our modified approach produced a 60% more average topic stability than the state-of-the-art for Nova; however, we could not compare the topic stability for other systems considered. We also did not perform a comparison between the themes identified.

Several Dirichlet Multinomial Mixture (DMM) based topic modeling models have been reported in the literature. GSDMM assumes only one topic for each code review comment, which we consider a valid assumption given the short length of code review comments, especially for in-line code review comments.

We selected GSDMM over the generalized Pòlya urn Poisson-based Dirichlet Multinomial Mixture model (GPU-PDMM), which had better classification accuracy than GSDMM in the four out of six datasets considered [150]. However, we selected GSDMM for its fast execution time and slightly lower classification accuracy than GPU-PDMM. GPU-PDMM may improve topic stability and average coherence in similar studies investigating topic evolution in code review comments. Similarly, several studies have proposed bi-directional transformer-based (BERT) topic modeling methods [164, 165]; however, a qualitative comparison between the performance of BERT and STTM on code review comments may improve the evolution analysis of code review comments.

There can be different approaches for using multiple windows to analyze the evolution of themes in code review comments. We only demonstrated analyzing selective months based on the trend. A sliding window approach can also be used to analyze the evolution of themes for code review comments in a given set of months; then, we update the window by adding newer months while removing the oldest months. However, further research is needed to qualitatively evaluate these approaches for the quality of the generated themes.

The identified themes and an analysis of how the themes with high topic impact evolve can lead to crucial changes in how teams approach development tasks, company development guidelines, and process-level improvements. As an illustration, an analysis of the *guidelines* theme may provide a checklist for developers to self-check before submitting source code. As the theme evolves, the checklist may be updated and thus remains relevant for the contributing developers. The updated analysis provides an initial step toward developing data-driven dashboards for practitioners to aid in the study of important themes which can be utilized to suggest improvement directions.

We have demonstrated that incorporating the suggestions leads to quantitative improvements in topic stability. However, we have yet to evaluate if these design suggestions lead to qualitative improvement in the quality of the themes identified.

Wen et al. [170] demonstrated that their approach was suitable for analyzing one closed-source system. Our results indicate that the updated approach may also be well-suited for analyzing closed-source systems. The effectiveness of the analysis of common themes and their evolution is intuitively dependent on the quality of the code review comments. Previous studies have observed that experienced reviewers with in-depth knowledge of the project provide context-specific feedback that may lead to more meaningful common themes using our approach. As a future study, we intend to use the updated approach and interview code reviewers regarding the quality of the themes produced.

4.7 Threats to validity

4.7.1 Data validity

We utilized the three open-source datasets and used random sampling from the dataset to evaluate the quality of the individual dataset. We removed the code review comments with an entry in “in reply to” to remove discussion replies from developers. We used an embedding model [180] designed from posts in StackOverflow [5], a platform to discuss software code issues, to further improve the quality of the generated topic from short-text modeling methods. While an embedding model designed from code review comments may improve results, we believe our selected word embedding model captures essential information related to software development and is a suitable option. Only one author was involved in the data extraction and topic stability evaluation. However, we used automated tools and scripts where possible in these stages to keep the possibility of human error to a minimum. We also did not consider removing highly frequent words during pre-processing, which may impact the results presented.

4.7.2 Research validity

To improve the repeatability of our study, we have shared the datasets and Python scripts online as part of our replication package. We have also described our steps in preprocessing data, and the topic selection process. Moreover, to reduce the chances of researcher bias, two authors were involved in assigning names to generated topics.

4.7.3 External validity

The empirical study presented may have a few threats relating to its external validity and limit the generalizability of the results. Since we selected only open-source systems, the language used in the code review comments may vary for other open-source and industrial systems. Intuitively, the language can inherently differ from one reviewer to another; thus, the number of reviewers involved in the review also impacts the language in code review comments. Further studies are needed using varied datasets from both open-source and industry to assess the generalizability of our approach.

[5]https://stackoverflow.com/

4.8 Conclusions

Building on the recent study by Wen et al. [170], we performed an exploratory study to evaluate possible design improvements in the study of the evolution of common themes in code review comments. Among other design and analysis improvements, we observed that the short-text modeling method leads to more stable topics than traditional topic modeling. Studying the evolution of common themes in code review comments is a promising field, with practical implications for research and practice that may lead to suggestions that help improve the development and process-related practices. By extending their work and proposing new approaches for topic selection and analysis of topic evolution, we highlight that the choice of modeling technique is essential as it may lead to different results. Further studies are needed in code review comments evolution and analysis to investigate the suggestions made. In future work, we aim to use industrial datasets along with interviews with reviewers to investigate the reasons behind the changes in the interesting themes as well as their reflections on using the identified themes to create data-driven dashboards and interventions at the development or process level that may aid in improving the issues highlighted in the derived themes.

Acknowledgments

This work has been supported by ELLIIT, the Strategic Research Area within IT and Mobile Communications, funded by the Swedish Government.

Chapter 5

Identifying prevalent quality issues in code changes by analyzing reviewers' feedback

This chapter is based on the following paper:

Chapter 5: Umar Iftikhar, Jürgen Börstler, Nauman Bin Ali and Oliver Kopp. "Identifying prevalent quality issues in code changes by analyzing reviewers' feedback". *To be submitted.*

5.1 Abstract

Context: Code reviewers provide valuable feedback during the code review. Identifying common issues described in the reviewers' feedback can provide input for context-specific software improvement opportunities. However, the use of reviewer feedback for this purpose is currently less explored.
Objective: Assessing if and how automation can derive themes in reviewers' feedback and whether these themes help to identify recurring quality-related issues in code changes.
Method: We conducted a case study using the JabRef system to distinguish reviewers' feedback on merged and abandoned code changes for the analysis. We used topic modeling to identify themes in 5,560 code review comments. The resulting themes were analyzed and named by a domain expert from JabRef.
Results: The domain expert considered the identified themes from the proposed automation approach to represent quality-related issues. We found that different quality issues are pointed out in code reviews for merged and abandoned code changes.
Conclusions: The results indicate the usefulness of our proposed automation approach in utilizing code review comments for understanding the prevalent code quality issues that can help derive targeted and context-bound improvement actions.

5.2 Introduction

In modern code review, experienced developers and architects review code changes and give feedback as code review comments (CRCs) to improve source code quality [149, 158, 161]. In addition to assessing the submitted code change, the expert feedback is also cognizant of important factors such as the system architecture [200], domain [148], technology and organizational [162], as well as team aspects like contribution guidelines (e.g., styling, documentation, or test coverage requirements). Such feedback is often beyond what static code analyzers can provide today. Moreover, static code analyzers have been criticized for reporting relatively high numbers of false positives [198, 199] or trivial issues.

The feedback provided in CRCs is typically only utilized once when developers implement a specific corrective action. As reviewers may point out similar issues to the same or different developers, identifying such prevalent issues can be helpful, e.g., to propose preventive measures and find systematic improvements [153, 210]. Such preventive measures may include more precise guidelines for project contributors, focused training on a particular technology, or reconfiguring the static analysis tool [168], thus helping to avoid similar issues in the future.

It is infeasible to aggregate quality issues in individual CRCs without qualitatively analyzing individual comments. The large number of review comments [162] makes their manual analysis impractical beyond research studies. Practitioners, therefore, often have to rely on their subjective judgment of what they perceive as prevalent code quality issues and, thus, what, in their opinion, are the required improvements. Therefore, automated classification approaches are needed to more objectively assess prevalent code quality issues from large code review repositories. Automated classification approaches based on machine-learning methods have recently been shown to provide promising results [153, 156, 170]. Iftikhar et al. [210], have improved on Wen et al. [170]'s topic modeling approach to achieve more stable topics and a refined topic selection approach.

In this study, using the approach by Iftikhar et al. [210], we investigate how to support practitioners in identifying and profiling prevalent quality issues, as pointed out in code review comments. To demonstrate the approach's utility, we explore if it helps identify code quality issues in code review comments for merged or abandoned code changes. We expect that the feedback on abandoned and merged code changes will be different, and our approach will reveal those differences.

We organized the remainder of the paper as follows. We discuss related work in Section 5.3 and the methodology is covered in Section 5.4. Section 5.5 presents our results, followed by a discussion of the results in Section 5.6 and threats to validity in Section 5.7. Conclusions can be found in Section 5.8.

5.3 Related work

This section summarizes related work on three themes, (1) manual categorization of CRCs, (2) automated analysis of CRCs, (3) investigations of factors that impact the outcome of submitted code changes.

5.3.1 Manual categorization of CRCs

Among the existing studies that manually categorized issues in code reviews, Mäntylä and Lassenius [155] manually analyzed nine industrial and 23 academic systems to categorize defects identified during code review discussions. They further classified the identified defects and observed that 75% of the defects relate to evolvability while 25% of the defects relate to functionality. Beller et al. [159] adopted a similar approach to analyze 1400 code changes from two open-source systems to investigate issues fixed during code review. Their results corroborated Mäntylä and Lassenius [155], where approximately 75% of the issues fixed were related to maintainability and 20% of the issues fixed were related to functionality. Gunawardena et al. [168] manually analyzed 417 CRCs to propose a fine-grained taxonomy of 117 defects discussed in CRCs. They further mapped the proposed defects taxonomy to static analysis tools, where 38% identified defects could be resolved using static analysis tools, thus demonstrating that categorization of issues in CRCs may have implications for practice.

While these studies demonstrate that categorizing CRCs can lead to identifying quality-related issues, which can assist in software improvement tasks, the manual approach limits their application to new datasets.

5.3.2 Automation to support CRCs analyis

Given the large code review repositories in practice, existing studies have approached automated categorizing of CRCs with different methodologies. Arafat and Shamma [156] categorized and predicted the topics in CRCs from six closed-source systems using supervised machine learning algorithms using a manually labeled dataset. They achieved 63% accuracy with the Support Vector Machine (SVM) method. Ochodek et al. [153] classified 2,672 CRCs from three open-source systems using the Bidirectional Encoder Representation from Transformers (BERT) [215] language model. They achieved an average accuracy of over 80% compared to manually classified CRCs. The studies have promising results but require labeled datasets, thus limiting the generalizability of their approach.

To investigate how community and personal feedback trends evolve as the community matures, Wen et al. [170] utilized Latent Dirichlet Allocation (LDA) on CRCs from one open-source system, Nova, and one closed-sourced system. Their results show that as reviewers accrue experience, the feedback provided to code changes is more context-specific and technical.

Iftikhar et al. [210] extended the work of Wen et al. [170] and evaluated several potential improvements in the design by Wen et al. [170]. Among the proposed improvements, they found that short-text topic modeling leads to more stable topics than traditional topic modeling. Similarly, among the alternative methods for selecting the number of topics, their two-stage topic selection approach slightly improved topic stability over the single-stage topic selection used by Wen et al. [170]. However, both studies [170, 210] do not demonstrate how to derive and profile code quality issues using common themes identified from CRCs. In this study, we address this gap by involving a domain expert from Jabref.

5.3.3 Investigations of factors behind abandoned and merged code changes

Existing studies have investigated reasons for the rejection of code submissions. Gottigundala et al. [208] reported that reasons for rejection of pull requests include implementing unnecessary functionality, conflicting pull requests, pull request reattempts, and inactivity. Kononenko et al. [209] reported that pull requests that are large and do not address a single purpose are likely to be not merged. They further observed that the experience of a pull request author is significantly linked with the merge decision for a pull request. Papadakis et al. [6] found that source code management issues, lack of understanding of project functionality, and poor understanding of reviewer expectations and project guidelines were among the reasons for the rejection of pull requests.

Wang et al. [182] identified 12 reasons for abandoned code changes. Duplicate code changes, i.e., similar to other code changes and code changes with a lack of reviewer feedback, were among the frequent reasons for abandoned code changes. While researchers have identified several factors impacting the likelihood of whether a code change will be merged or abandoned, to the best of our knowledge, identifying the recurring quality issues in abandoned and merged code changes through an analysis of the CRCs has not been done.

As discussed in Section 5.2, code review comments are potentially a very relevant source of information to mine insights regarding code quality issues. Thus, in this study, we explore if CRCs can be analyzed to identify and develop a better profile of quality issues identified in merged or abandoned code changes.

5.4 Methodology

In this case study, we pose and answer the following research question:

RQ: How can we derive themes from CRCs to identify recurring code quality issues?

To answer the RQ, we used Iftikhar et al. [210]'s approach to identify themes in code review comments automatically. These themes were analyzed and named by a domain expert. If the named themes are indeed about quality issues, that would provide evidence that the proposed approach can help identify prevalent code quality issues.

Furthermore, in this study, we analyzed code review comments for both abandoned and merged code changes to demonstrate the effectiveness of our approach in profiling code quality issues. We expect that there are differences in the reviewer feedback for the two classes of code changes. The named themes identified using our approach and named by the expert will help to profile the code quality issues the reviewers are pointing out for merged and abandoned code changes.

5.4.1 Case project and domain expert

The case is JabRef[1], an open-source, cross-platform, citations and reference management tool developed in Java [212] to help students, academics, and researchers to collect and maintain bibliographic information. It covers over 15 reference formats and supports 23 languages. At the time of the study, JabRef version 5.12 was released. It is maintained by a core team of researchers and students with 581 active contributors[2]. Two reviewers review each submitted code change[3]. A proposed code change may undergo several iterations of code review before merging till the code reviewers are satisfied with the code quality. Code changes to specific modules, e.g., JavaFX and BibTex, are reviewed by module experts, while anyone in the core team can review the generic code changes. The source code quality of JabRef has been extensively analyzed in literature [117, 197].

Another reason for selecting JabRef was the availability of a long-term maintainer and a domain expert for assigning representative names to identified topics. The domain expert was not part of the research team that conceived, designed, or analyzed the results. The research team collected, analyzed, performed member checking, and completed the first draft of the paper before inviting the domain expert to be a co-author. In the writing phase, the domain expert contributed mainly with a deeper contextual understanding of Jabref, checking the results and reflections in the paper and thoroughly reviewing several versions of the paper.

[1]`https://www.jabref.org/`
[2]https://github.com/jabref/jabref/
[3]`https://devdocs.jabref.org/teaching.html`

5.4.2 Datasets

Table 5.1 shows an overview of the dataset. We used the REST API provided for GitHub[4] to extract CRCs and code change outcome status. Since we are only interested in code changes that are either abandoned or merged, we did not consider CRCs from code changes that are still open or under discussion. We extracted CRCs from May 2014 till September 2023 as CRCs from earlier were unavailable. We used the *pull request number* to relate CRCs and code change status.

We replicated the pre-processing steps in previous studies [210]. We converted all CRCs to lowercase and removed all brackets, punctuations, URLs, and words containing numbers. We removed stop words using the standard pre-processing library in the Gensim natural language processing toolkit[5]. We further removed null strings and lemmatized all CRCs to create a document entry for each code review comment, forming two lists of documents, one from abandoned changes and one from merged changes.

After data extraction and pre-processing of the CRCs, we got 5,560 CRCs from 426 code changes belonging to three major releases of JabRef. We did not consider code changes without CRCs for the analysis. We did not consider 60 CRCs from version 2 as the release only contained CRCs from merged changes. We also removed 124 CRCs with zero length after pre-processing. The extracted CRCs are provided in the replication package online[6].

Table 5.1: Overview of data from JabRef (PR=pull request).

PR status	Total number of PRs	Number of PRs with CRCs	Total CRCs	Average CRC length in words
Abandoned	717	38	535	26.5
Merged	4,862	388	5,025	22.8
Total	5,579	426	5,560	23.2

5.4.3 Natural language processing model selection

Short text topic models (STTM) have been demonstrated by Qiang et al. [150] to have superior performance than Latent Dirichlet Allocation (LDA) for short

[4]`https://docs.github.com/en/rest/overview`
[5]`https://www.nltk.org/`
[6]`https://doi.org/10.5281/zenodo.10408930`

texts in terms of classification accuracy and purity [175]. CRCs are short pieces of text [152]. We selected the Gibbs Sampling-based Dirichlet Multinomial Mixture model (GSDMM) [175] implementation by Qiang et al. [150]. GSDMM has also improved average topic stability compared to LDA [210]. Since we are interested in analyzing CRCs from abandoned and merged changes, we generate separate topic models for CRCs from abandoned and merged changes.

5.4.4 Parameter selection

The performance of topic classification depends on the choice of hyperparameters, topic-to-document probability *(alpha)*, word-to-topic probability *(beta)*, and number of topics *(N)* [210]. Biggers et al. [181] suggest that lower values for the hyperparameters lead to a more decisive model with less overlap among generated topics.

Topic stability, an adapted measure of cross-run similarity of topics [176], is defined as the median number of word-terms occurrences in all considered runs for a given topic number while varying the hyperparameters *alpha* and *beta* between 0 and 1 for each considered run. While Panichella [205] observes that no fitness measure consistently leads to superior quality topics in their dataset, we chose topic stability as it has been used in existing studies [170, 210].

In the first stage, we considered N=[5..55] (in steps of five) for the number of topics to analyze topic stability. We trained five GSDMM models for each dataset, sorted in a different order for each run, with the choice of hyperparameters varied for each run. We used the 10 top words from five runs of GSDMM models to identify the most stable choice of N in this first stage. In the second stage, we iterate in steps of one in the neighborhood of N from the first stage to select the most stable topic. Next, to choose the most suitable values for hyperparameters, we also calculate the coherence for the five combinations of *alpha* and *beta* used in the topic stability stages and select the combination that provides the highest coherence.

5.4.5 Analysis of topic similarity

To objectively evaluate the hypothesis that the feedback in CRCs from abandoned and merged changes focuses on different issues and themes, we analyzed the similarity of the generated topics using cosine similarity [204]. As input to the comparison, we used the 20 top terms and their frequencies for each topic. The cosine similarity value belongs to the interval [0,1]. A cosine similarity score of 1 indicates identical documents [201].

We also separately evaluated cosine similarity within the groups of abandoned and merged topics. The maximum within-groups cosine similarity was then used as a baseline for interpreting the between-groups cosine similarity.

5.4.6 Data collection

We designed a structured questionnaire[6] that was shared with the domain expert for data interpretation tasks. The structured questionnaire consisted of the top 20 unprocessed CRCs belonging to each topic, the top 20 terms representing the topic [170], and meta information such as the PR number, PR URL, and PR title, among other important information that may help in assigning a suitable theme to CRCs.

In addition to the structured questionnaire, we provided a separate document containing the study aims, instructions for steps to be performed during the interpretation task. We used the same document to collect the overall reflections regarding ease of naming themes, reflections on the naming process, difficulties faced when interpreting topic evolution, and potential implications of the identified themes for process improvements within in the project.

5.4.7 Topic naming

Current naming approaches can be categorized as manual, automated, and a combination of manual and automated steps [154]. Manual topic-naming has been used in existing studies [170, 177, 178]. We provided the domain expert with the top 20 unprocessed CRCs belonging to each topic and the top 20 terms representing the topic [170]. We asked the domain expert to suggest a representative name for each topic.

To triangulate the common themes interpreted by the domain expert, we utilized the publicly available large language model, ChatGPT [216], for the topic naming process. We prompted ChatGPT to suggest a representative name for each topic by utilizing each topic's top 20 CRCs. We then compared the names assigned by ChatGPT to the names by the domain expert.

5.5 Results and analysis

In this section, we describe the characteristics of the data before presenting the identified themes and the profiles for abandoned and merged changes.

5.5.1 Data characteristics

The distribution of unprocessed CRC lengths from abandoned and merged changes shows that both are predominantly short texts and follow a similar trend (see Figure 5.1). Of the CRCs in abandoned and merged changes, 75% are at most 28 and 29 words long, respectively, and only 5% and 3%, respectively, are 80 words or longer in abandoned and merged changes. This confirms our choice of topic model in Section 5.4.3.

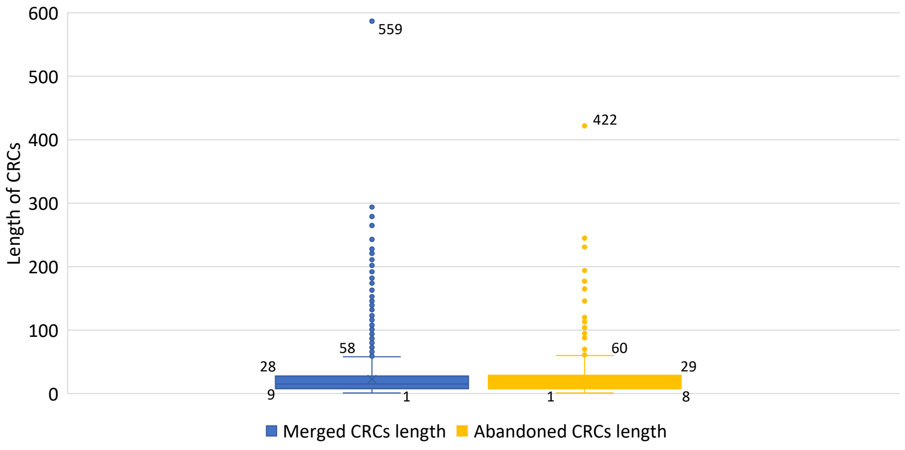


Figure 5.1: Distribution of lengths (in number of words) of CRCs for abandoned and merged changes.

Figure 5.2 shows the number of code commits contributed by different groups of reviewers. Approximately 71% (3,937 out of 5,560) of the CRCs are from three reviewers who have each contributed more than 500 PRs. Eight reviewers contributed between 100 and 500 PRs each and provided 18% (1,013 out of 5,560) of the CRCs. The remaining 33 out of 44 reviewers have only contributed approximately 9% (523 out of 5,560) of the CRCs. The above patterns show that reviewing is important in Jabref development, as the main contributors also extensively review the code.

Figure 5.3 shows the distribution of the percentage of total PRs contributed by different developers. The figure further depicts the percentage of each developer's abandoned and merged PRs. The data showed that for only 11 out of 630 developers, their individual PR contributions constituted 1% or more of

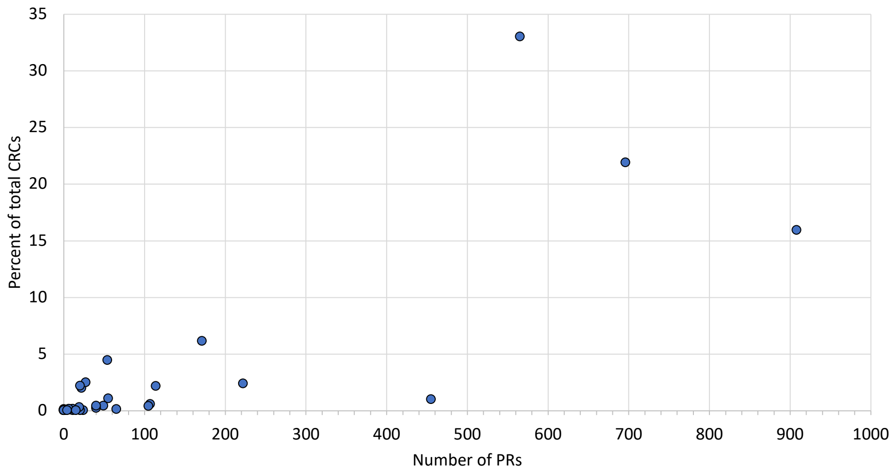


Figure 5.2: Distribution of the number of PRs and percentage of CRCs by code reviewers.

the total PRs in Jabref. Therefore, in Figure 5.3, we show the contributions of 11 developers individually and group the rest as 'Others'. Nearly 67% (3,780 out of 5,579) of the total PRs are contributed by 11 developers. The 'Others' in the figure, who individually have contributed to less than one percent of total PRs, have collectively contributed approximately 33% (1,799 out of 5,579) of the total PRs. The top 11 active developers, on average, had approximately 7% PRs abandoned, while other developers, on average, had 30% PRs abandoned.

5.5.2 Manually labeled themes

As described in Section 5.4.4, we generated five themes each for the abandoned (T_a1–T_a5) and merged (T_m1–T_m5) CRCs, respectively. Our domain expert then named these topics as described in Section 5.4.7.

Table 5.2 summarizes the theme names assigned to the generated topics and the corresponding topic share [179] for each theme. The topic share is the ratio of the number of CRCs for a topic (or theme) compared to the total number of CRCs and indicates the relative size of a theme. As can be seen from Table 5.2, there is a larger variation in the topic share for the themes from abandoned PRs.

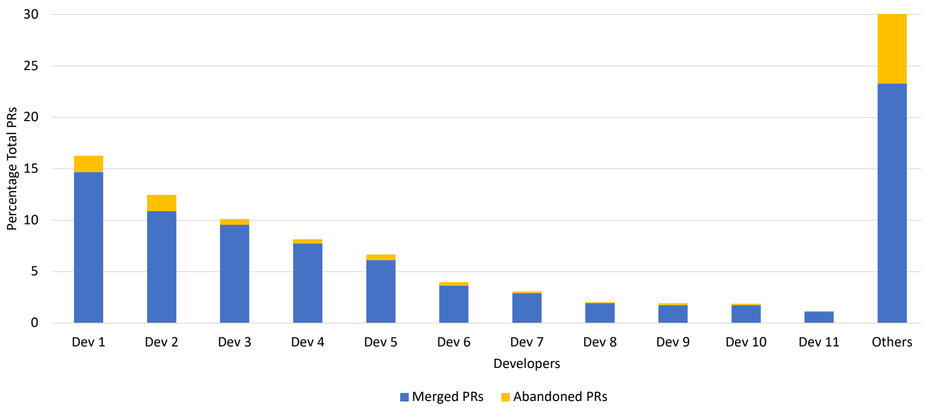


Figure 5.3: Distribution of the number of PRs by the developers who contributed 1% or more of the total PRs (Dev 1–Dev 11) and all others (Other).

As described in Section 5.4.5, we calculated between-groups and within-groups cosine similarity. Within the groups of abandoned and merged topics, the similarity scores range between *0.08* and *0.34*. Table 5.3 summarizes the cosine similarity scores between the topics generated for the abandoned CRCs (rows) versus the topics generated for the merged CRCs (columns). Similarity scores greater than *0.34* are highlighted in bold. The results show that the between-groups similarity is mainly comparable to within-groups similarity. Only four out of 25 between-groups topic pairs have larger similarities than the maximum within-groups similarity.

In the following subsections, we go through the results provided by the domain expert.

5.5.3 Manually labeled themes for abandoned CRCs

T_a1: Java code quality

With a topic share of 0.38, *Java code quality* is the most prevalent theme for CRCs of abandoned PRs. The CRCs in this theme are related to the conformance with common principles and patterns to foster maintainability. An example of a CRC illustrating this theme is shown below (pull request 3418).

Table 5.2: Manually assigned themes for the CRCs from abandoned (T_a1–T_a5) and merged (T_m1–T_m5) PRs together with their topic share.

Topic	**Manually assigned theme**	**Topic share**
T_a1	Java code quality	0.38
T_a2	Preferences localization	0.15
T_a3	External resources & path handling	0.24
T_a4	Code architecture	0.10
T_a5	Code formatting	0.14
T_m1	Shorter code	0.21
T_m2	Modern test style	0.31
T_m3	Simplify control flow	0.14
T_m4	JavaFX architecture	0.18
T_m5	UX	0.17

Table 5.3: Cosine similarities between the topics of CRCs from merged changes (T_m1–T_m5) and abandoned changes (T_a1–T_a5).

	T_m1	T_m2	T_m3	T_m4	T_m5
T_a1	0.21	0.32	0.23	0.23	0.28
T_a2	0.13	0.15	0.15	0.24	**0.46**
T_a3	0.19	0.31	0.14	**0.44**	**0.53**
T_a4	0.09	0.33	0.12	**0.64**	0.24
T_a5	0.21	0.20	**0.36**	0.22	0.12

PR 3418: “I would propose to make this class non-static / not a singleton. You then have a (public) constructor that accepts the current version string and a default constructor that uses *‘java_version’*. In this way, you can also easily write a test to verify the methods in this class.” (Italics added for readability)

T_a2: Preferences localization

The CRCs in theme *Preferences localization* address issues related to JabRef’s customization code, which deals with preferences related to the user interface, language, and localizations. Such preferences and localizations also need to be reflected in the coding style. An example of a CRC illustrating this theme is shown below.

PR 7181: "I wouldn't set a maximum width for year and type columns. If users prefer to give more space to these columns, why not? Simply setting *'prefWidth'* should be fine." (Italics added for readability)

T_a3: External resources & path handling

With a topic share of 0.24, *External resources& path handling* is the second largest theme for CRCs of abandoned PRs. The CRCs in this theme deal with the adequate handling of external resources such as file paths, resources identified by URLs, user and global directories, as well as exception handling. An example of a CRC illustrating this theme is shown below.

PR 7728: "I don't think this comment should be here. If you give this method a relative path it is implementation-dependent I don't think it is always going to be JabRef's home directory. I guess that in practice we should avoid giving it an absolute path."

T_a4: Code architecture

Code architecture is the smallest theme for CRCs of abandoned PRs (topic share=0.1). Its CRCs deal with issues related to placing functionality in appropriate classes and methods. An example of CRC illustrating this theme is shown below.

PR 557: "When a utility method is only used once, it should not be in a utility class. In that case, it would only be used for this call. What is more, I do not like utility classes for domain-specific things like external file types. It is OK for IO methods like reading from a file, interfacing with the file system or for type conversion such as String to int."

T_a5: Code formatting

Code formatting is another relatively small theme (topic share=0.14). The CRCs in this theme are related to code styling, check-style issues, and setting up the workspace. An example of a CRC illustrating this theme is shown below.

PR 7172: "Please set up checkstyle configuration according to our workspace set up guide. Wildcard imports are not allowed."

5.5.4 Manually labeled themes for merged CRCs

T_m1: Shorter code

The CRCs in theme *Shorter code* are related to recommendations for using built-in methods. The primary motivation for such recommendations is that shorter code is perceived to be more maintainable. An example of CRC illustrating this theme is shown below.

> *PR 6434:* “One of the *Date.parse* method overloads accepts already *Optional* as parameter so you can make your code a bit easier and remove the *ifPressent* checks for Year and month...” (Italics added for readability)

T_m2: Modern test style

With a topic share of 0.31, *Modern test style* is the most prevalent theme for CRCs of merged PRs. The CRCs in this theme are related to recommendations for using JUnit’s parametrized test functionality instead of duplicating test code. The size of the theme indicates that contributing developers often have to be asked to reuse test code. The domain expert observed that groups of contributing developers focused on improving the project’s test code, which may explain the increase in testing-related discussions in CRCs. Similarly, when PRs contribute to specific modules related to the core logic and user interface, reviewers often ask them to provide test cases to evaluate the contribution. An example of a CRC illustrating this theme is shown below.

> *PR 6479:* “Thanks for adding so many tests. This is really good. I would propose to split them a bit into two categories: tests for parsing and test for representation. The former should take a string and test against a *‘AuthorList‘*. The latter should take a *‘AuthorList‘* and test against a string...” (Italics added for readability)

T_m3: Simplify control flow

The CRCs in this theme focus on suggestions to simplify the code flow, e.g., by reducing the number of code branches and lines of code in a code branch to improve maintainability. An example of a CRC illustrating this theme is shown below.

PR 379: "I think it would be better to find out the exact Exception (should be easy with the test) and then write a multi-catch block..."

T_m4: JavaFX architecture

The CRCs in this theme discuss the JavaFX[7] architecture that is used in JabRef for user interface design. An example of a CRC illustrating this theme is shown below.

PR 4227: "This works because the dialog is very simple but goes against the usual strategy of *JavaFX / MVVM*. You should add properties in the *ViewModel* class... Please have a look at the other *JavaFX* dialogs to see how this is done..." (Italics added for readability)

T_m5: UX

The CRCs in this theme deal with issues related to meeting the needs of the user experience expected from intermediate users of JabRef and the behavior of JabRef's user interface on different operating systems. An example of a CRC illustrating this theme is shown below.

PR 1390: "I find it counterintuitive that the same button sometimes resets only a few bindings and sometimes all. Proposal: add a third column to the table which contains a small reset button (only icon light gray by default dark gray on hovering the row)..."

5.5.5 Automatically labeled themes

As mentioned in Section 5.4.7, we triangulated the manually assigned themes by the domain expert using ChatGPT. The results from ChatGPT and the corresponding theme name from manual labeling are provided in Table 5.4, showing that, in our case, automatically labeled themes by LLM closely resemble the manually generated themes. Some examples of closely resembling themes include *Java code quality* and *Code architecture*, which the LLM identified as *Refinement & documentation for code quality* and *Refinement & architecture enhancement*, respectively. In contrast, themes with low resemblance include *External resources & path handling*, which the LLM identified as *Refinement & precision in code development.*

[7] `https://openjfx.io/`

This result also opens up further opportunities for automation in our approach to identifying prevalent quality issues.

Table 5.4: Comparison of manually and automatically assigned themes for the CRCs from abandoned (top) and merged (bottom) changes.

Manual	**Automatic**
Java Code Quality	Refinement & Documentation for Code Quality
Preferences localization	User Interface Enhancement & Preference Handling
External resources & path handling	Refinement & Precision in Code Development
Code architecture	Refinement & Architecture Enhancement
Code formatting	Code Standards Adherence and Refactoring
Shorter Code	Code Optimization & Best Practices
Modern test style	Test Refinement & Maintenance
Simplify control flow	Refinement for Enhanced Code Quality & Readability
Javafx architecture	Improving GUI Architecture & Responsiveness
UX	User Experience Enhancement & Functionality Optimization

5.5.6 Understanding abandoned and merged changes

The themes presented in Section 5.5.2 are derived from the analysis of individual CRCs. On average, abandoned code changes in JabRef have 14 CRCs per code change, while merged code changes have 13 CRCs per code change on average. The highest value of CRCs per code change for abandoned and merged changes are 28 and 29, respectively. Since identified themes can belong to different code changes, we aggregated identified themes at the code change using the *pull request number* to understand better how code changes differ regarding the themes of their related CRCs. The results are shown in Figure 5.4 for abandoned changes, and Figure 5.5 for merged changes.

We observed nine profiles for the abandoned code changes shown in Figure 5.4. The most frequently discussed combination of themes, *Java code quality* and *External resources & path handling*, is considered in five out of nine profiles containing 60% (23 out of 38) code changes. While we noted in Table 5.2, 38% CRCs relate to *Java code quality*, interestingly it is discussed in eight of nine profiles, suggesting that the theme is frequently highlighted across distinct code changes. *Preference locatlization* with only one percent more CRCs compared to *Code formatting* is discussed in twice as many profiles as the latter, thus indi-

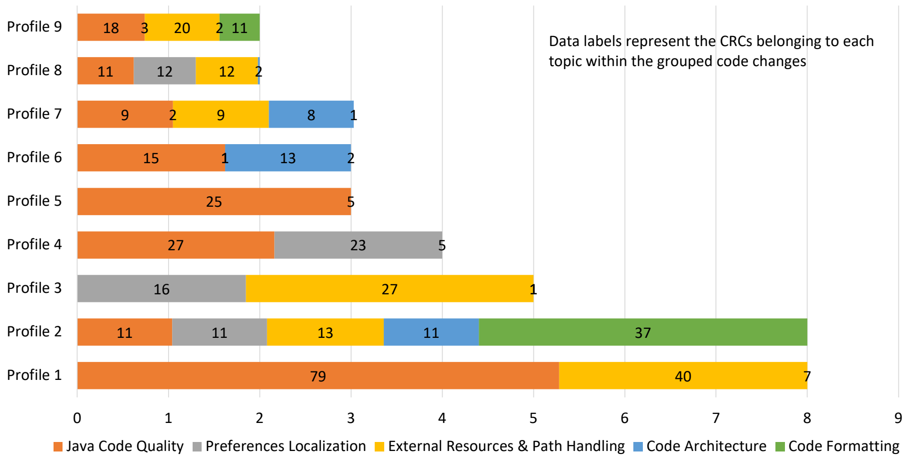


Figure 5.4: Profiles for abandoned changes

cating that the topic impact of a theme and the number of profiles discussing the theme is a non-linear and may differ for themes. In Figure 5.4, we also report the CRCs belonging to each theme within the profile to depict the relationship between profiles and CRCs.

We identified 23 profiles among the merged changes depicted in Figure 5.5. The frequently discussed pair of themes, *Shorter code* with *Modern test style* and *Modern test style* with *Simplify control flow*, is considered in eight out of 23 profiles. While we noted in Table 5.2, 31% CRCs relate to *Modern test style*, it is highlighted in 70% (16 out of 23) profiles. suggesting that the theme is significantly emphasized across distinct code changes. Furthermore, Profile 1, with the highest number of code changes, focuses on only two themes, *Shorter code* and *Modern test style*, with 15% (57 out of 381) of the code changes belonging to the profile. *Modern test style* theme is most often discussed in combination with other themes and is discussed across 16 profiles. Incidentally, the other four themes are highlighted across 11 out of 23 profiles each, despite varied topic share. Figure 5.5, we further report the CRCs for each theme within the profile to highlight the relationship between profiles and CRCs.

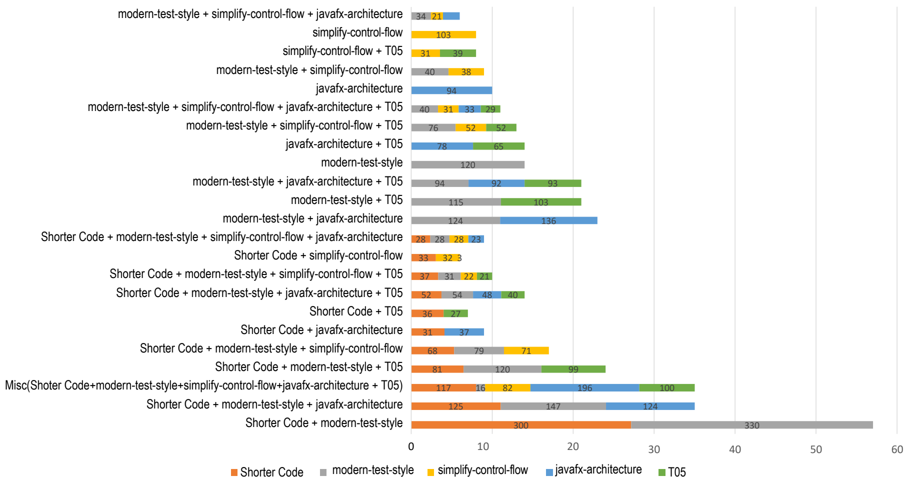


Figure 5.5: Profiles for merged changes

5.6 Discussion

Using the approach by Iftikhar et al. [210], we identified recurring quality issues in code changes by analyzing the CRCs. In this section, we discuss the results of using the approach and the implications for software development practice.

5.6.1 Themes related to code quality

The common themes indicate a focus on long-term readability (such as T_m1, T_a5), testability (for example, T_m2), and code structure (see T_m3, T_m4). The common themes in abandoned and merged changes can be broadly categorized as related to maintainability, e.g., by focusing on code structure and utilizing well-tested built-in functions, which is aligned with existing taxonomies of issues found in CRCs [155, 159]. The identified themes in abandoned changes highlight issues related to the code architecture, formatting of code, and code quality. In contrast, the themes in merged changes emphasize alternate suggestions that help to improve the issues related to implementation choices related to JavaFX architecture, testing options, and built-in functions.

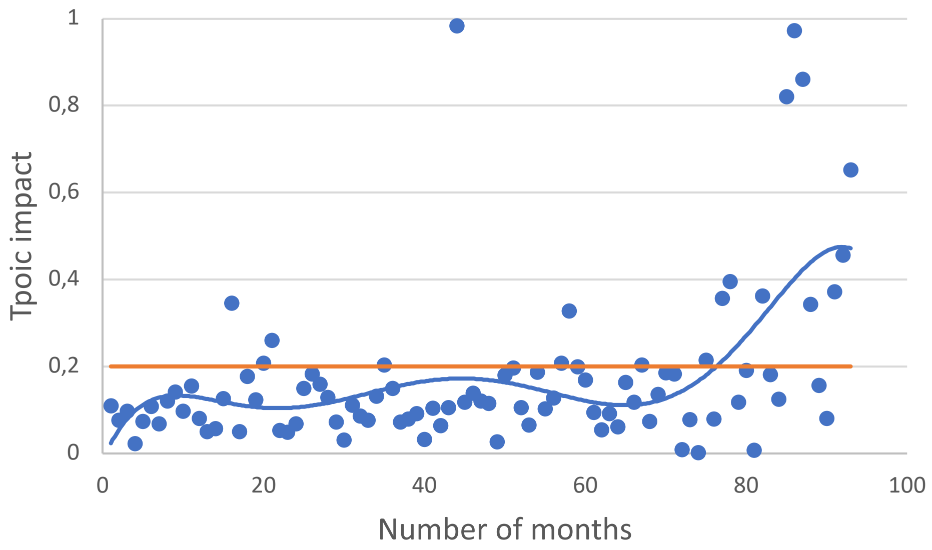


Figure 5.6: Topic evolution for merged theme T_m3 *Simplify control flow* in JabRef

5.6.2 Quality of identified themes

When evaluating the quality of the current approach for identifying common themes, the domain expert noted that three-fourths of the themes were easy to provide a name. In contrast, one-fourth of the identified themes needed to be more coherent. However, after reading CRCs from other themes, it was also possible to name such themes. The feedback suggests that further research is needed to design improved topic modeling approaches to study common themes in CRCs. Recently, Udupa et al. [211] reported that the transformer-based (BERT) [215] topic modeling method for short-text provided superior coherence compared to GSDMM and may lead to better performance for common themes analysis. Future studies can explore BERT and promising variants, e.g., RoBERTa [214] in this regard.

5.6.3 Automatic theme labels

The identified themes need to be assigned an appropriate name, which is currently a manual process requiring practitioners' input, which limits the approach. An automated method that aids in naming the identified themes, e.g., by using large language models [216], may improve the efficiency of the topic naming steps.

5.6.4 Implications for state-of-practice

The domain expert observed two potential areas where the proposed analysis could improve the state of the practice for JabRef. The identified themes can provide ideas to initiate focused discussions on specific aspects for the JabRef discussion forums. By creating Q&As forms for each identified theme, the discussion forums can be used to acquire important feedback on identified themes, thus improving the interactivity of the discussion forums. Additionally, JabRef provides guidelines to help newcomers.[8] The identified themes can give concrete ideas on which content to update in the guidelines to improve their effectiveness and address some of the challenges faced by newcomers [213].

5.6.5 Theme evolution

To analyze the evolution of common themes over time, we depicted scatter plots to show the topic impact [179] over the months considered used by existing

[8] `https://devdocs.jabref.org/getting-into-the-code/guidelines-for-setting-up-a-local-workspace/`

studies [170, 210]. While the evolution graphs may be helpful to visualize the overall trend for a theme, the domain expert found it challenging to explain specific trends in the topic evolution graphs, Figure 5.6 depicts the evolution of theme T_m3 *Simplify control flow* as an example. For a meaningful analysis of the topic evolution graphs, the domain expert observed that more data is needed, e.g., which pull requests are active in those months and who are the top contributors for the given month. The main reason is that different contributors with different preferences were active in different periods of the project. Due to the relatively small dataset, we chose not to empirically evaluate the themes version-wise as suggested previously [210], as version-wise analysis for larger datasets may lead to interesting results

The data characteristics discussed in Section 5.5.1 indicate that there is potential link between the number of PRs submitted by developers and the outcome of the code review, thus indicating that there may be other non-technical factors that impact the outcome of the code review process, which is aligned with previous results [183, 207].

5.6.6 Potential applications for presented approach

The potential application of the presented approach includes analyzing the code review comments in various scenarios, e.g., studying prevailing code quality issues, trend analysis of code quality issues, and the impact of an intervention on code quality. Similar to the current study's setup, we can use the presented approach to compare the quality issues in two groups of code changes based on the code review comments, e.g., studying quality issues from code changes that take longer to merge compared to code changes that are merged quickly, or studying differences in feedback given by less experienced reviewers compared to more experienced reviewers. For practitioners and researchers, we briefly summarize the approach we followed in this study and discuss further applications.

Step 1: Collect CRCs; if one is interested in comparing two or more groups, split the CRCs into separate subsets. For example, in this study, we compared prevalent quality issues in merged and abandoned code changes. Therefore, we split the CRCs into comments made by reviewers on merged code changes and comments on abandoned code changes.

Step 2: Using the scripts available in the replication package, pre-process CRCs and generate topics for each dataset.

Step 3: A domain expert reads each topic's top CRCs and top terms to assign a suitable theme (see Section 5.4.7).

Step 4: Using the named themes, create profiles of prevalent quality issues for each dataset.

5.7 Threats to validity

We use the classification suggested by Runeson and Höst [202] to discuss the validity threats.

5.7.1 Reliability

We used automated tools and scripts to reduce the possibility of human error during data curation to a minimum. To ensure that we only used CRCs as input from the extracted data, we removed discussion replies from developers by removing the discussion comments where the comment's author was the same as the change author. We incorporated an embedding model [180] derived from posts in StackOverflow,[9] a platform to discuss software code issues, to further improve the quality of the generated topics. Our selected word embedding model is based on software development terminologies, which we believe is suitable. While studies have suggested removing highly frequent words to aid in creating distinct topics [195], this can remove important words [196]. We chose not to remove highly frequent words and short CRCs during preprocessing, as we consider frequent words by reviewers and short CRCs relevant to the analysis performed in the study. To support the repeatability of the study, we have provided a replication package containing the extracted datasets and Python scripts used for the data extraction, preprocessing, and topic modeling.

5.7.2 Internal validity

The extracted CRC data from the GitHub platform may only partially capture the recurring quality issues. Some quality issues may be discussed using other modes of communication, e.g., the discussion forums, which are not reflected using the approach used in the study. However, since open-source communities extensively use GitHub for collaboration among contributors, we sufficiently capture the recurring quality issues in JabRef. The structured questionnaire shared with the domain expert was curated by the first author and reviewed by the second author for content validity and clarity. However, the order of the data shared with the domain expert for topic naming, e.g., the order of 20

[9]`https://stackoverflow.com/`

CRCs and the order of the topics presented, may introduce a response bias. Due to practical considerations, we could only involve a single practitioner from JabRef. Involving more practitioners could further reduce potential biases while interpreting topics.

5.7.3 Construct validity

While we have selected only topic stability [176] to select the appropriate number of topics, other fitness measures, such as coherence [206] and silhouette coefficient [157], may lead to different topics.

Our study only considered cosine scores (see Section 5.4.5) for topic similarity analysis. To assess whether other similarity measures might produce different results, we considered Okapi BM25 [203], which led to similar results.

5.7.4 External validity

Since we utilized only JabRef, other systems with different developers, reviewers, and review practices may lead to distinct results. Thus, further studies with varied datasets are needed to establish the generalizability of the approach.

5.8 Conclusions

We performed an exploratory case study to support practitioners through automation in identifying and profiling prevalent quality issues, using common themes in CRCs from abandoned and merged changes. We followed the approach from an existing study [210].

The common themes named by the domain expert demonstrate that the approach can help identify recurring code quality issues in CRCs. We identified different themes from CRCs in abandoned and merged changes. The prevalent code quality issues broadly aim to address the maintainability-focused issues in JabRef. Furthermore, we observed unique profiles for code quality issues in pull requests from abandoned and merged changes. Additionally, we outlined the steps required to apply a similar approach to other potential applications.

The results derived from the analysis of CRCs can help in improving the guidelines for new developers. They can assist in directing focused discussions in the developer forums, thus potentially enhancing the current practices for JabRef. While many identified themes were easy to assign a name to, further research is needed to improve the quality of the common themes identified using our approach.

In future studies, we plan to explore variants of BERT [214] and large language models [216] to improve the topic model generated to improve the quality of the themes. To evaluate the generalizability and effectiveness of the approach, we also plan to use industrial datasets to improve the development guidelines and data-driven discussion aimed at improving development practices.

Acknowledgment

Commented out for double-blind peer-reviews.

Supplementary material

The Python scripts and datasets are provided online[6].

References

[1] Claes Wohlin. Case study research in software engineering—it is a case, and it is a study, but is it a case study? *Information and Software Technology*, 133:106514, 2021.

[2] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012.

[3] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[4] Noushin Ashrafi. The impact of software process improvement on quality: in theory and practice. *Information & Management*, 40(7):677–690, 2003.

[5] Liz Allen, Alison O'Connell, and Veronique Kiermer. How can we ensure visibility and diversity in research contributions? how the contributor role taxonomy (credit) is helping the shift from authorship to contributorship. *Learned Publishing*, 32(1):71–74, 2019.

[6] Nick Papadakis, Ayan Patel, Tanay Gottigundala, Alexandra Garro, Xavier Graham, and Bruno Da Silva. Why Did your PR Get Rejected?: Defining Guidelines for Avoiding PR Rejection in Open Source Projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 165–168, 2020.

[7] Philip B Crosby. *Quality is free: The art of making quality certain.* McGraw-Hil, NY, USA, 1979.

[8] Watts S Humphrey. *Managing the software process.* Addison-Wesley Longman Publishing Co., Inc. Reading, MA, USA, 1989.

[9] Alain Abran, James W Moore, Pierre Bourque, Robert Dupuis, and L Tripp. Software engineering body of knowledge. *IEEE Computer Society*, 25, 2004.

[10] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[11] A Frank Ackerman, Priscilla J Fowler, and Robert G Ebenau. Software inspections and the industrial production of software. In *Proceeding of a Symposium on Software Validation: Inspection-testing-verification-alternatives*, pages 13–40, 1984.

[12] Forrest Shull and Carolyn Seaman. Inspecting the history of inspections: An example of evidence-based technology diffusion. *IEEE Software*, 25 (1):88–90, 2008.

[13] Barry Boehm and Victor R Basili. Top 10 list [software development]. *Computer*, 34(1):135–137, 2001.

[14] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Processings of 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144, 2015.

[15] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016.

[16] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[17] Santosh Singh Rathore and Atul Gupta. Validating the effectiveness of object-oriented metrics over multiple releases for predicting fault proneness. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 350–355, 2012.

[18] Siim Karus and Marlon Dumas. Code churn estimation using organisational and code metrics: An experimental comparison. *Information and Software Technology*, 54(2):203–211, 2012.

[19] Daniel Méndez Fernández and Stefan Wagner. Naming the pain in requirements engineering: design of a global family of surveys and first results from germany. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 183–194, 2013.

[20] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[21] Ally S Nyamawe, Hui Liu, Zhendong Niu, Wentao Wang, and Nan Niu. Recommending refactoring solutions based on traceability and code metrics. *IEEE Access*, 6:460–475, 2018.

[22] Dimitrios Tsoukalas, Nikolaos Mittas, Alexander Chatzigeorgiou, Dionysios Kehagias, Apostolos Ampatzoglou, Theodoros Amanatidis, and Lefteris Angelis. Machine learning for technical debt identification. *IEEE Transactions on Software Engineering*, 48(12):4892–4906, 2021.

[23] Emre Doğan and Eray Tüzün. Towards a taxonomy of code review smells. *Information and Software Technology*, 142:106737, 2022.

[24] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[25] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. Individual, Social and Personnel Factors Influencing Modern Code Review Process. In *Proceedings of the IEEE Conference on Open Systems (ICOS)*, pages 40–45, 2019.

[26] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. Modern Code Reviews—Survey of Literature and Practice. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–61, 2023.

[27] Patrick Rempel and Parick Mader. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Transactions on Software Engineering*, 43(8):777–797, 2017.

[28] John Terzakis. The impact of requirements on software quality across three product generations. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE)*, pages 284–289, 2013.

[29] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012.

[30] Mohamad Kassab, Joanna F DeFranco, and Phillip A Laplante. Software testing: The state of the practice. *IEEE Software*, 34(5):46–52, 2017.

[31] Zhang Zhonglin and Mei Lingxia. An improved method of acquiring basis path for software testing. In *Proceedings of the 5th International Conference on Computer Science & Education*, pages 1891–1894, 2010.

[32] Harmen Sthamer, Joachim Wegener, and Andre Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review*, 2002.

[33] Arif Ali Khan and Jacky Keung. Systematic review of success factors and barriers for software process improvement in global software development. *IET Software*, 10(5):125–135, 2016. doi: 10.1049/iet-sen.2015.0038.

[34] Enrico Fregnan, Fernando Petrulio, Linda Di Geronimo, and Alberto Bacchelli. What happens in my code reviews? An investigation on automatically classifying review changes. *Empirical Software Engineering*, 27:89, jul 2022.

[35] Asif Kamal Turzo, Fahim Faysal, Ovi Poddar, Jaydeb Sarker, Anindya Iqbal, and Amiangshu Bosu. Towards Automated Classification of Code Review Feedback to Support Analytics. In *Proceedings of the 17th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2023.

[36] Miroslaw Ochodek, Krzysztof Durczak, Jerzy Nawrocki, and Miroslaw Staron. Mining Task-Specific Lines of Code Counters. *IEEE Access*, 11: 100218–100233, 2023.

[37] Erik Arisholm and Dag IK Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):521–534, 2004.

[38] Barbara A Kitchenham, Lesley M Pickard, and Susan J Linkman. An evaluation of some design metrics. *Software Engineering Journal*, 5(1): 50–58, 1990.

[39] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. A survey on software coupling relations and tools. *Information and Software Technology*, 107:159–178, 2019.

[40] Israel Herraiz and Ahmed E Hassan. Beyond lines of code: Do we need more complexity metrics. *Making software: what really works, and why we believe it*, pages 125–141, 2010.

[41] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. *ACM SIGMETRICS Performance Evaluation Review*, 10(1):81–88, 1981.

[42] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. Correlations of software code metrics: an empirical study. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pages 255–266, 2017.

[43] Barbara Kitchenham, Lech Madeyski, and David Budgen. Segress: Software engineering guidelines for reporting secondary studies. *IEEE Transactions on Software Engineering*, 49(3):1273–1298, 2022.

[44] Nauman Bin Ali and Muhammad Usman. A critical appraisal tool for systematic literature reviews in software engineering. *Inf. Softw. Technol.*, 112:48–50, 2019.

[45] Nauman Bin Ali and Muhammad Usman. Reliability of search in systematic reviews: Towards a quality assessment framework for the automated-search strategy. *Inf. Softw. Technol.*, 99:133–147, 2018.

[46] Jürgen Börstler, Nauman Bin Ali, and Kai Petersen. Double-counting in software engineering tertiary studies - an overlooked threat to validity. *Inf. Softw. Technol.*, 158:107174, 2023.

[47] Muhammad Usman, Nauman Bin Ali, and Claes Wohlin. A quality assessment instrument for systematic literature reviews in software engineering. *W Informatica Softw. Eng. J.*, 17(1):230105, 2023.

[48] Nauman bin Ali and Binish Tanveer. A comparison of citation sources for reference and citation-based search in systematic literature reviews. *E-Informatica Software Engineering Journal*, 2022.

[49] Huynh Khanh Vi Tran, Jürgen Börstler, Nauman bin Ali, and Michael Unterkalmsteiner. How good are my search strings? reflections on using an existing review as a quasi-gold standard. *E-Informatica Software Engineering Journal*, 16(1), 2022.

[50] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[51] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[52] B W Boehm, J R Brown, and M Lipow. Quantative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, 1976.

[53] David Budgen, Pearl Brereton, Nikki Williams, and Sarah Drummond. What support do systematic reviews provide for evidence-informed teaching about software engineering practice? *E-informatica Software Engineering Journal.*, 14(1):7–60, 2020.

[54] Colin Robson. *Real world research: A resource for social scientists and practitioner-researchers.* Wiley-Blackwell, NJ, USA, 2002.

[55] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach.* CRC press, FL, USA, 2019.

[56] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: a tertiary systematic review of challenges and observations. *Journal of Systems and Software*, page 110610, 2020.

[57] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[58] Hamdi A Al-Jamimi and Moataz Ahmed. Prediction of software maintainability using fuzzy logic. In *Proceedings on International Conference on Computer Science and Automation Engineering*, pages 702–705, 2012.

[59] Saleh Almugrin, Waleed Albattah, Omar Alaql, Musaad Alzahrani, and Austin Melton. Instability and abstractness metrics based on responsibility. In *Proceedings of the 38th Annual Computer Software and Applications Conference*, pages 364–373, 2014.

[60] Giulio Concas, Michele Marchesi, Alessandro Murgia, Sandro Pinna, and Roberto Tonelli. Assessing traditional and new metrics for object-oriented systems. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pages 24–31, 2010.

[61] Maria Teresa Baldassarre, Danilo Caivano, Simone Romano, and Giuseppe Scanniello. Software Models for Source Code Maintainability: A Systematic Literature Review. In *Proceedings of the 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 252–259, 2019.

[62] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. *Evidence-based software engineering and systematic reviews*, volume 4. CRC press, 2015.

[63] B. Kitchenham and S.L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE Software*, 13(1):12–21, 1996. doi: 10.1109/52.476281.

[64] Jose P. Miguel, David Mauricio, and Glen Rodriguez. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*, 5(6):31–53, 2014. doi: 10.5121/ijsea.2014.5603.

[65] Jim A McCall, Paul K Richards, and Gene F Walters. Factors in software quality, volumes i, ii, and iii. *US Rome Air Development Center Reports, US Department of Commerce, USA*, 1977.

[66] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[67] Ronald Jabangwe, Jürgen Börstler, Darja Šmite, and Claes Wohlin. Empirical Evidence on the Link between Object-Oriented Measures and External Quality Attributes: A Systematic Literature Review. *Empirical Software Engineering*, 20(3):640–693, 2015.

[68] Lionel C Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 56:97–166, 2002.

[69] ISO. IEC 9126-1: Software engineering-product quality-part 1: Quality model. *Geneva, Switzerland: International Organization for Standardization*, 21, 2001.

[70] Organización Internacional de Normalización. ISO-IEC 25010: 2011 systems and software engineering-systems and software quality requirements and evaluation (square)-system and software quality models. *Geneva, Switzerland: International Organization for Standardization*, 2011.

[71] CISQ Specifications for Automated Quality Characteristic Measures. CISQ Technical Work Groups for Reliability, Performance Efficiency, Security, Maintainability. Standard CISQ-TR 2012-01, OMG, 2012.

[72] Alois Mayr, Reinhold Plösch, Michael Kläs, Constanza Lampasona, Fraunhofer Iese, and Matthias Saft. A Comprehensive Code-based Quality Model for Embedded Systems. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering*, page 10, 2012.

[73] Karine Mordal-Manet, Francoise Balmas, Simon Denier, Stephane Ducasse, Harald Wertz, Jannik Laval, Fabrice Bellingard, and Philippe Vaillergues. The squale model &#x2014; A practice-based industrial quality model. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 531–534, 2009.

[74] Robert B Grady and Deborah L Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., NJ; USA, 1987.

[75] M. Soto and M. Ciolkowski. The qualoss open source assessment model measuring the performance of open source communities. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 498–501, 2009.

[76] Alexander Romanovsky and Tullio Vardanega. *Reliable Software Technologies–Ada-Europe 2011: 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*, volume 6652. Springer Science & Business Media, 2011.

[77] Hassan Rashidi and Mohammad Sadeghzadeh Hemayati. Software Quality Models: A Comprehensive Review and Analysis. *Journal of Electrical and*

Computer Engineering Innovations, 6(1), 2018. doi: 10.22061/jecei.2019.1076.

[78] Nauman Bin Ali and Kai Petersen. Evaluating strategies for study selection in systematic literature studies. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–4, 2014.

[79] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, 2008.

[80] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[81] Khaled El Emam. Benchmarking kappa: Interrater agreement in software process assessments. *Empirical Software Engineering*, 4(2):113–133, 1999.

[82] Mark Turner. Digital libraries and search engines for software engineering research: An overview. *Keele University, UK*, 2010.

[83] Lianipng Chen, Muhammad Ali Babar, and He Zhang. Towards an evidence-based understanding of electronic data sources. In *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–4, 2010.

[84] K.S. Sreeji and C. Lakshmi. A systematic literature review: Recent trends and open issues in software refactoring. *International Journal of Applied Engineering Research*, 10(18):39696–39707, 2015.

[85] Meng Yan, Xin Xia, Xiaohong Zhang, Ling Xu, and Dan Yang. A systematic mapping study of quality assessment models for software products. In *Proceedings of the 2017 International Conference on Software Analysis, Testing and Evolution*, pages 63–71, 2017.

[86] L.A. Martins, Jr. Afonso, P., A.P. Freire, and H. Costa. Evolution of quality assessment in spl: A systematic mapping. *IET Software*, 14(6): 572–581, 2020. doi: 10.1049/iet-sen.2020.0037.

[87] R. Malhotra and K. Lata. A systematic literature review on empirical studies towards prediction of software maintainability. *Soft Computing*, 24(21):16655–16677, 2020. doi: 10.1007/s00500-020-05005-4.

[88] M. Vogel, P. Knapik, M. Cohrs, B. Szyperrek, W. Pueschel, H. Etzel, D. Fiebig, A. Rausch, and M. Kuhrmann. Metrics in automotive software development: A systematic literature review. *Journal of Software: Evolution and Process*, 33(2), 2021. doi: 10.1002/smr.2296.

[89] S.N. Saharudin, K.T. Wei, and K.S. Na. Machine learning techniques for software bug prediction: A systematic review. *Journal of Computer Science*, 16(11):1558–1569, 2020. doi: 10.3844/JCSSP.2020.1558.1569.

[90] S. Kaur and P. Singh. How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157, 2019. doi: 10.1016/j.jss.2019.110394.

[91] S. El-Sharkawy, N. Yamagishi-Eichler, and K. Schmid. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 106: 1–30, 2019. doi: 10.1016/j.infsof.2018.08.015.

[92] M. Yan, X. Xia, X. Zhang, L. Xu, D. Yang, and S. Li. Software quality assessment model: a systematic mapping study. *Science China Information Sciences*, 62(9), 2019. doi: 10.1007/s11432-018-9608-3.

[93] S. Hosseini, B. Turhan, and D. Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, 2019. doi: 10.1109/TSE.2017.2770124.

[94] S. Elmidaoui, L. Cheikhi, A. Idri, and A. Abran. Empirical studies on software product maintainability prediction: A systematic mapping and review. *E-Informatica Software Engineering Journal*, 13(1):141–202, 2019. doi: 10.5277/e-Inf190105.

[95] A. Kaur. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. *Archives of Computational Methods in Engineering*, 2019. doi: 10.1007/s11831-019-09348-6.

[96] R. Malhotra and M. Khanna. Software change prediction: A systematic review and future guidelines. *E-Informatica Software Engineering Journal*, 13(1):227–259, 2019. doi: 10.5277/e-Inf190107.

[97] P. Morrison, D. Moye, R. Pandita, and L. Williams. Mapping the field of software life cycle security metrics. *Information and Software Technology*, 102:146–159, 2018. doi: 10.1016/j.infsof.2018.05.011.

[98] S. Tiwari and S.S. Rathore. Coupling and cohesion metrics for object-oriented software: A systematic mapping study. *ACM International Conference Proceeding Series*, 2018. doi: 10.1145/3172871.3172878.

[99] E.M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou. A mapping study on design-time quality attributes and metrics. *Journal of Systems and Software*, 127:52–77, 2017. doi: 10.1016/j.jss.2017.01.026.

[100] R. Malhotra and A. Chug. Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, 26(8):1221–1253, 2016. doi: 10.1142/S0218194016500431.

[101] R. Malhotra and A.J. Bansal. Software change prediction: A literature review. *International Journal of Computer Applications in Technology*, 54 (4):240–256, 2016. doi: 10.1504/IJCAT.2016.080487.

[102] M. Riaz, T. Breaux, and L. Williams. How have we evaluated software pattern application? a systematic mapping study of research design practices. *Information and Software Technology*, 65:14–38, 2015. doi: 10.1016/j.infsof.2015.04.002.

[103] J. Murillo-Morera, C. Quesada-López, and M. Jenkins. Software fault prediction: A systematic mapping study. *Proceedings of XVIII Ibero-American Conference on Software Engineering*, pages 446–459, 2015.

[104] A. Bandi, B.J. Williams, and E.B. Allen. Empirical evidence of code decay: A systematic mapping study. In *Proceedings of the Working Conference on Reverse Engineering, WCRE*, pages 341–350, 2013.

[105] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013. doi: 10.1016/j.infsof.2013.02.009.

[106] B. Isong and E. Obeten. A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction. *International Journal of Software Engineering and Knowledge Engineering*, 23 (10):1513–1540, 2013. doi: 10.1142/S0218194013500484.

[107] A. Tahir and S.G. MacDonell. A systematic mapping study on dynamic metrics and software quality. *IEEE International Conference on Software Maintenance, ICSM*, pages 326–335, 2012. doi: 10.1109/ICSM.2012.6405289.

[108] J. Saraiva, E. Barreiros, A. Almeida, F. Lima, A. Alencar, G. Lima, S. Soares, and F. Castor. Aspect-oriented software maintenance metrics: A systematic mapping study. *IET Seminar Digest*, 2012(1):253–262, 2012. doi: 10.1049/ic.2012.0033.

[109] Y.A. Khan, M.O. Elish, and M. El-Attar. A systematic review on the impact of ck metrics on the functional correctness of object-oriented classes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7336 LNCS (PART 4):258–273, 2012. doi: 10.1007/978-3-642-31128-4_19.

[110] S. Montagud, S. Abrahão, and E. Insfran. A systematic review of quality attributes and measures for software product lines. *Software Quality Journal*, 20(3-4):425–486, 2012. doi: 10.1007/s11219-011-9146-7.

[111] Rachel Burrows, Alessandro Garcia, and François Taïani. Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies. *Evaluation of Novel Approaches to Software Engineering*, pages 277–290, 2009.

[112] B. Kitchenham. What's up with software metrics? - a preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51, 2010. doi: 10.1016/j.jss.2009.06.041.

[113] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[114] Ramon Abilio, Pedro Teles, Heitor Costa, and Eduardo Figueiredo. A Systematic Review of Contemporary Metrics for Software Maintainability. In *Proceedings of Sixth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 130–139, 2012.

[115] Juliana Saraiva, Sergio Soares, and Fernando Castor. Towards a catalog of Object-Oriented Software Maintainability metrics. In *Proceedings of*

the 4th International Workshop on Emerging Trends in Software Metrics (WETSoM), pages 84–87, 2013.

[116] Ruchika Malhotra and Anuradha Chug. Software Maintainability: Systematic Literature Review and Current Trends. *International Journal of Software Engineering and Knowledge Engineering*, 26(08):1221–1253, 2016. doi: 10.1142/S0218194016500431.

[117] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017. doi: 10.1016/j.jss.2017.03.044.

[118] Rashina Hoda, Norsaremah Salleh, John Grundy, and Hui Mien Tee. Systematic literature reviews in agile software development: A tertiary study. *Information and Software Technology*, 85:60–70, 2017.

[119] Jose L Barros-Justo, Fabiane BV Benitti, and Santiago Matalonga. Trends in software reuse research: A tertiary study. *Computer Standards & Interfaces*, 66:103352, 2019.

[120] Karina Curcio, Rodolfo Santana, Sheila Reinehr, and Andreia Malucelli. Usability in agile software development: A tertiary study. *Computer Standards & Interfaces*, 64:61–77, 2019.

[121] IEEE. 1074-2006–ieee standard for developing a software project life cycle process, 2006.

[122] Samuel Daniel Conte, Hubert E Dunsmore, and Vincent Y Shen. *Software engineering metrics and models.* Benjamin-Cummings Publishing Co., Inc., CA, USA, 1986.

[123] Andrew R Gray and Stephen G MacDonell. A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology*, 39(6):425–437, 1997.

[124] Esmeralda Yamileth Hernandez-Gonzalez, Angel Juan Sanchez-Garcia, Maria Karen Cortes-Verdin, and Juan Carlos Perez-Arriaga. Quality metrics in software design: A systematic review. In *Proceedings of 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 80–86, 2019.

[125] Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, and Alexander Chatzigeorgiou. Guidelines for Managing Threats to Validity of Secondary Studies in Software Engineering. In Michael Felderer and Guilherme Horta Travassos, editors, *Contemporary Empirical Methods in Software Engineering*, pages 415–441. Springer International Publishing, 2020.

[126] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings of the conference on Software Maintenance*, pages 337–338, 1992.

[127] Lionel C Briand, Sandro Morasca, and Victor R Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[128] R Harrison, S Counsell, and R Nithi. An overview of object-oriented design metrics. In *Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, pages 230–235, 1997.

[129] Wei Li. Another metric suite for object-oriented programming. *Journal of Systems and Software*, 44(2):155–162, 1998.

[130] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th International Conference on Software Quality*, volume 186, pages 1–8, 1994.

[131] Mark Lorenz and Jeff Kidd. *Object-oriented Software Metrics: A Practical Guide.* Prentice-Hall, Inc., NJ, USA, 1994.

[132] Mahmoud O Elish, Ali H Al-Yafei, and Muhammed Al-Mulhem. Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of eclipse. *Advances in Engineering Software*, 42(10):852–859, 2011.

[133] Tobias Mayer and Tracy Hall. Measuring oo systems: a critical analysis of the mood metrics. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 108–117, 1999.

[134] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[135] Magne Jørgensen. A critique of how we measure and interpret the accuracy of software development effort estimation. In *Proceedings of the First International Workshop on Software Productivity Analysis and Cost Estimation*, pages 1–6, 2007.

[136] Barbara A Kitchenham, Lesley M Pickard, Stephen G. MacDonell, and Martin J. Shepperd. What accuracy statistics really measure [software estimation]. *IEE Proceedings-Software*, 148(3):81–85, 2001.

[137] Tron Foss, Erik Stensrud, Barbara Kitchenham, and Ingunn Myrtveit. A simulation study of the model evaluation criterion mmre. *IEEE transactions on Software Engineering*, 29(11):985–995, 2003.

[138] Fatima Nur Colakoglu, Ali Yazici, and Alok Mishra. Software product quality metrics: A systematic mapping study. *IEEE Access*, 9:44647–44670, 2021. doi: 10.1109/ACCESS.2021.3054730.

[139] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 85–94, 2005.

[140] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Using in-process testing metrics to estimate post-release field quality. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE)*, pages 209–214, 2007.

[141] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[142] Puja Saxena and Monika Saini. Empirical studies to predict fault proneness: A review. *International Journal of Computer Applications*, 22(8): 41–45, 2011.

[143] Organización Internacional de Normalización. ISO-IEC 25023: 2016-systems and software engineering-systems and software quality requirements and evaluation (square)-measurement of system and software product quality. *Geneva, Switzerland: International Organization for Standardization*, 2016.

[144] John Spray, Roopak Sinha, Arnab Sen, and Xingbin Cheng. Building maintainable software using abstraction layering. *IEEE Transactions on Software Engineering*, 48(11):4397–4410, 2021.

[145] Ingunn Myrtveit, Erik Stensrud, and Martin Shepperd. Reliability and validity in comparative studies of software prediction models. *IEEE Transactions on Software Engineering*, 31(5):380–391, 2005.

[146] Ingunn Myrtveit and Erik Stensrud. Validity and reliability of evaluation procedures in comparative studies of effort prediction models. *Empirical Software Engineering*, 17:23–33, 2012.

[147] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003.

[148] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, pages 712–721, 2013.

[149] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working Conference on Mining Software Repositories*, pages 192–201, 2014.

[150] Jipeng Qiang, Zhenyu Qian, Yun Li, Yunhao Yuan, and Xindong Wu. Short text topic modeling techniques, applications, and performance: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3): 1427–1445, 2020.

[151] Xueqi Cheng, Xiaohui Yan, Yanyan Lan, and Jiafeng Guo. BTM: Topic modeling over short texts. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2928–2941, 2014.

[152] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. Automatic classification of review comments in pull-based development model. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, pages 572–577, 2017.

[153] Miroslaw Ochodek, Miroslaw Staron, Wilhelm Meding, and Ola Söder. Automated code review comment classification to improve modern code reviews. In *Proceedings of the 14th International Conference on Software Quality*, pages 23–40, 2022.

[154] Camila Costa Silva, Matthias Galster, and Fabian Gilson. Topic modeling in software engineering research. *Empirical Software Engineering*, 26(6): 1–62, 2021.

[155] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[156] Yeasir Arafat and Syeda Sumbul Hossain Shamma. Categorizing review comments by mining software repositories. *International Conference on Advances in Computing and Data Sciences*, page 12, 2020.

[157] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 35th International Conference on Software Engineering*, pages 522–531, 2013.

[158] Steve McConnell. *Code complete.* Pearson Education, London, UK, 2004.

[159] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 202–211, 2014.

[160] Robert Chatley and Lawrence Jones. Diggit: Automated code review via software repository mining. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 567–571, 2018.

[161] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality, 2015.

[162] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 43(1): 56–75, 2017.

[163] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 163–174, 2021.

[164] Maarten Grootendorst. BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794*, 2020.

[165] Roman Egger and Joanne Yu. A topic modeling comparison between LDA, NMF, top2vec, and BERTopic to demystify twitter posts. *Frontiers in Sociology*, 7:886498, 2022.

[166] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. CommentFinder: A simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 507–519, 2022.

[167] Nicole Davila and Ingrid Nunes. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177: 110951, 2021.

[168] Sanuri Gunawardena, Ewan Tempero, and Kelly Blincoe. Concerns identified in code review: A fine-grained, faceted classification. *Information and Software Technology*, 153:107054, 2023.

[169] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. Using code reviews to automatically configure static analysis tools. *Empirical Software Engineering*, 27(1):28, 2022.

[170] Ruiyin Wen, Maxime Lamothe, and Shane McIntosh. How does code reviewing feedback evolve?: A longitudinal study at Dell EMC. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 151–160, 2022.

[171] 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.

[172] Daniel M German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. "was my contribution fairly reviewed?" A framework to study the perception of fairness in modern code reviews. In *Proceedings of the 40th International Conference on Software Engineering*, pages 523–534, 2018.

[173] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *Proceedings of the 31st*

International Conference on Software Maintenance and Evolution, pages 81–90, 2015.

[174] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, pages 261–270, 2015.

[175] Jianhua Yin and Jianyong Wang. A dirichlet multinomial mixture model-based approach for short text clustering. In *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining*, pages 233–242, 2014.

[176] Amritanshu Agrawal, Wei Fu, and Tim Menzies. What is wrong with topic modeling? And how to fix it using search-based software engineering. *Information and Software Technology*, 98:74–88, 2018.

[177] Mubin Ul Haque, Leonardo Horn Iwaya, and M. Ali Babar. Challenges in docker development: A large-scale study using stack overflow. In *Proceedings of the 14th International Symposium on Empirical Software Engineering and Measurement*, pages 1–11, 2020.

[178] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering*, 25:2694–2747, 2020.

[179] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? An analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19:619–654, 2014.

[180] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 38–41, 2018.

[181] Lauren R Biggers, Cecylia Bocovich, Riley Capshaw, Brian P Eddy, Letha H Etzkorn, and Nicholas A Kraft. Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering*, 19: 465–500, 2014.

[182] Qingye Wang, Xin Xia, David Lo, and Shanping Li. Why is my code change abandoned? *Information and Software Technology*, 110:108–120, 2019.

[183] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering*, 23(6):3346–3393, 2018.

[184] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, NY, USA, 2007.

[185] Maurice H Halstead. *Elements of Software Science.* Elsevier Science Ltd., AMS, Netherlands, 1977.

[186] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[187] Maria Caulo and Giuseppe Scanniello. A taxonomy of metrics for software fault prediction. In *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 429–436, 2020.

[188] Amit Sharma and Sanjay Kumar Dubey. Comparison of software quality metrics for object-oriented system. *International Journal of Computer Science & Management Studies (IJCSMS)*, 12:12–24, 2012.

[189] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[190] Hani Abdeen, Stephane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 103–112, 2009.

[191] Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler, and Muhammad Usman. A tertiary study on links between source code metrics and external quality attributes. *Information and Software Technology*, 165:–, 2024.

[192] Arvinder Kaur, Kamaldeep Kaur, and Kaushal Pathak. A proposed new model for maintainability index of open source software. In *Proceedings of the 3rd International Conference on Reliability*, pages 1–6, 2014.

[193] Umar Iftikhar, Nauman Bin Ali, Jürgen Börstler, and Muhammad Usman. Dataset for a catalog of source code metrics – a tertiary study. `https://doi.org/10.5281/zenodo.7219870`, 2022. Accessed: 2022-10-20.

[194] The Centre for Reviews and Dissemination (CRD) Database of Abstracts of Reviews of Effects (DARE). `https://www.crd.york.ac.uk/CRDWeb/`, 2022. Accessed: 2022-10-20.

[195] Wray L Buntine and Swapnil Mishra. Experiments with non-parametric topic models. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining*, pages 881–890, 2014.

[196] Yueshen Xu, Yuyu Yin, and Jianwei Yin. Tackling topic general words in topic modeling. *Engineering Applications of Artificial Intelligence*, 62: 124–133, 2017.

[197] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. The relationship of code churn and architectural violations in the open source software jabref. In *Proceedings of the 11th European Conference on Software Architecture*, pages 152–158, 2017.

[198] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, pages 672–681, 2013.

[199] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[200] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering*, 47(5):1041–1059, 2019.

[201] Anna Huang et al. Similarity measures for text document clustering. In *Proceedings of the sixth New Zealand Computer Science Research Student Conference*, volume 4, pages 9–56, 2008.

[202] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.

[203] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[204] H Gornaa Wael and Fahmy Aly. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013.

[205] Annibale Panichella. A Systematic Comparison of search-Based approaches for LDA hyperparameter tuning. *Information and Software Technology*, 130:106411, 2021.

[206] David Mimno, Hanna Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. Optimizing semantic coherence in topic models. In *Proceedings of the International Conference on Empirical Methods in Natural Language Processing*, pages 262–272, 2011.

[207] 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 *Proceedings of the 22nd International IEEE Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, 2015.

[208] Tanay Gottigundala, Siriwan Sereesathien, and Bruno Da Silva. Qualitatively Analyzing PR Rejection Reasons from Conversations in Open-Source Projects. In *Proceedings of the IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 109–112, 2021.

[209] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart De Water. Studying pull request merges: a case study of shopify's active merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 124–133, 2018.

[210] Umar Iftikhar, Jürgen Börstler, and Nauman Bin Ali. On potential improvements in the analysis of the evolution of themes in code review comments. *49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, -:–, In Press.

[211] Abhinandan Udupa, K N Adarsh, Anvitha Aravinda, Neelam H Godihal, and N Kayarvizhy. An Exploratory Analysis of GSDMM and BERTopic on Short Text Topic Modelling. In *Proceedings of the Fourth International Conference on Cognitive Computing and Information Processing*, pages 1–9, 2022.

[212] Oliver Kopp, Carl Christian Snethlage, and Christoph Schwentker. Jabref: Bibtex-based literature management software. *TUGboat*, 44(3):441–447, 2023.

[213] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F Redmiles. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology*, 59:67–85, 2015.

[214] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[215] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[216] OpenAI. GPT-4 Technical Report, 2023. URL `http://arxiv.org/abs/2303.08774`. arXiv:2303.08774 [cs].

Context: Software quality has a multi-faceted description encompassing several quality attributes. Central to our efforts to enhance software quality is to improve the quality of the source code. Poor source code quality impacts the quality of the delivered product. Empirical studies have investigated how to improve source code quality and how to quantify the source code improvement. However, the reported evidence linking internal code structure information and quality attributes observed by users is varied and, at times, conflicting. Furthermore, there is a further need for research to improve source code quality by understanding trends in feedback from code review comments.

Objective: This thesis contributes towards improving source code quality and synthesizes metrics to measure improvement in source code quality. Hence, our objectives are 1) To synthesize evidence of links between source code metrics and external quality attributes, & identify source code metrics, and 2) To identify areas to improve source code quality by identifying recurring code quality issues using the analysis of code review comments.

Method: We conducted a tertiary study to achieve the first objective, an archival analysis and a case study to investigate the latter two objectives.

Results: To quantify source code quality improvement, we reported a comprehensive catalog of source code metrics and a small set of source code metrics consistently linked with maintainability, reliability, and security. To improve source code quality using analysis of code review comments, our explored methodology improves the state-of-the-art with interesting results.

Conclusions: The thesis provides a promising way to analyze themes in code review comments. Researchers can use the source code metrics provided to estimate these quality attributes reliably. In future work, we aim to derive a software improvement checklist based on the analysis of trends in code review comments.

Printed by Media-Tryck, Lund 2024 NORDIC SWAN ECOLABEL 3041 0903

Blekinge Institute of Technology
Licentiate Dissertation Series No. 2024:02
ISSN: 1650-2140
ISBN: 978-91-7295-474-8