

Behind the Intent of Extract Method Refactoring: A Systematic Literature Review

Eman Abdullah AlOmar , *Member, IEEE*, Mohamed Wiem Mkaouer , *Member, IEEE*,
and Ali Ouni , *Member, IEEE*

Abstract—**Background:** Code refactoring is widely recognized as an essential software engineering practice to improve the understandability and maintainability of the source code. The *Extract Method* refactoring is considered as “Swiss army knife” of refactorings, as developers often apply it to improve their code quality, e.g., decompose long code fragments, reduce code complexity, eliminate duplicated code, etc. In recent years, several studies attempted to recommend *Extract Method* refactorings allowing the collection, analysis, and revelation of actionable data-driven insights about refactoring practices within software projects. **Aim:** In this paper, we aim at reviewing the current body of knowledge on existing *Extract Method* refactoring research and explore their limitations and potential improvement opportunities for future research efforts. That is, *Extract Method* is considered one of the most widely-used refactorings, but difficult to apply in practice as it involves low-level code changes such as statements, variables, parameters, return types, etc. Hence, researchers and practitioners begin to be aware of the state-of-the-art and identify new research opportunities in this context. **Method:** We review the body of knowledge related to *Extract Method* refactoring in the form of a systematic literature review (SLR). After compiling an initial pool of 1,367 papers, we conducted a systematic selection and our final pool included 83 primary studies. We define three sets of research questions and systematically develop and refine a classification schema based on several criteria including their methodology, applicability, and degree of automation. **Results:** The results construct a catalog of 83 *Extract Method* approaches indicating that several techniques have been proposed in the literature. Our results show that: (i) 38.6% of *Extract Method* refactoring studies primarily focus on addressing code clones; (ii) Several of the *Extract Method* tools incorporate the developer’s involvement in the decision-making process when applying the method extraction, and (iii) the existing benchmarks are heterogeneous and do not contain the same type of information, making standardizing them for the purpose of benchmarking difficult. **Conclusions:** Our study serves as an “index” to the body of knowledge in this area for researchers and practitioners in determining the *Extract Method* refactoring approach that is most appropriate

for their needs. Our findings also empower the community with information to guide the future development of refactoring tools.

Index Terms—Extract method, refactoring, quality, systematic literature review.

I. INTRODUCTION

REFACTORING is the art of restructuring code to improve it without changing its external behavior [2]. One of the basic building blocks of refactoring is *Extract Method*, i.e., the process of moving a fragment of code from an existing method into a new method with a name that explains its behavior. Method extraction is one of the main refactorings that were defined when this area was established [3], as it is a common response to the need of keeping methods concise and modular, and reducing the spread of shared responsibilities. Furthermore, *Extract Method* serves as a bridge to facilitate more complex refactorings [4]. *Extract Method* is widely employed by developers across various systems¹. It represents approximately 49.6% of the total refactorings recommended, as shown by JDeodorant [5], one of popular tools that support *Extract Method* refactoring. Moreover, open-source developers [6], [7], [8], [9], [10], [11], [12], [13] and industry professionals [14] consider it a critical refactoring operation. The popularity of this refactoring is inherited from its multifaceted utility that can be used for a myriad of reasons, such as removal of duplicate code [15], [16], [17], [18], extraction of reusable methods [6], [19], [20], wrapping older method signatures [6], decomposition of long or complex structures [21], [22], [23], [24], [25], [26], [27], and support of code testability [28], [29]. This wide variety of usage scenarios shows why method extraction is considered the *Swiss Army knife* of refactoring operations [30]. One of the typical rationales behind method extraction is the removal of duplicate code instances, which we can extract from a real-world case. In this case, the committer has documented the cleaning up of duplicate code. A closer inspection of the code changes, illustrated in Fig. 1, reveals the elimination of code duplication in four methods (i.e., `getDummy(dataType byte)`, `getNext(obj Object, dataType byte)`, `genericGetNext(obj Object, dataType byte)`, and `accumChild(child List, o Object, dataType byte)`, where four duplicates are extracted into one separate method (i.e., `genericGetNext(Object obj,`

Manuscript received 8 May 2023; revised 15 December 2023; accepted 15 December 2023. Date of publication 4 January 2024; date of current version 19 April 2024. This work was supported in part by the National Science Foundation under Grant CNS-2213765. Recommended for acceptance by F. Ferrucci. (Corresponding author: Eman Abdullah AlOmar.)

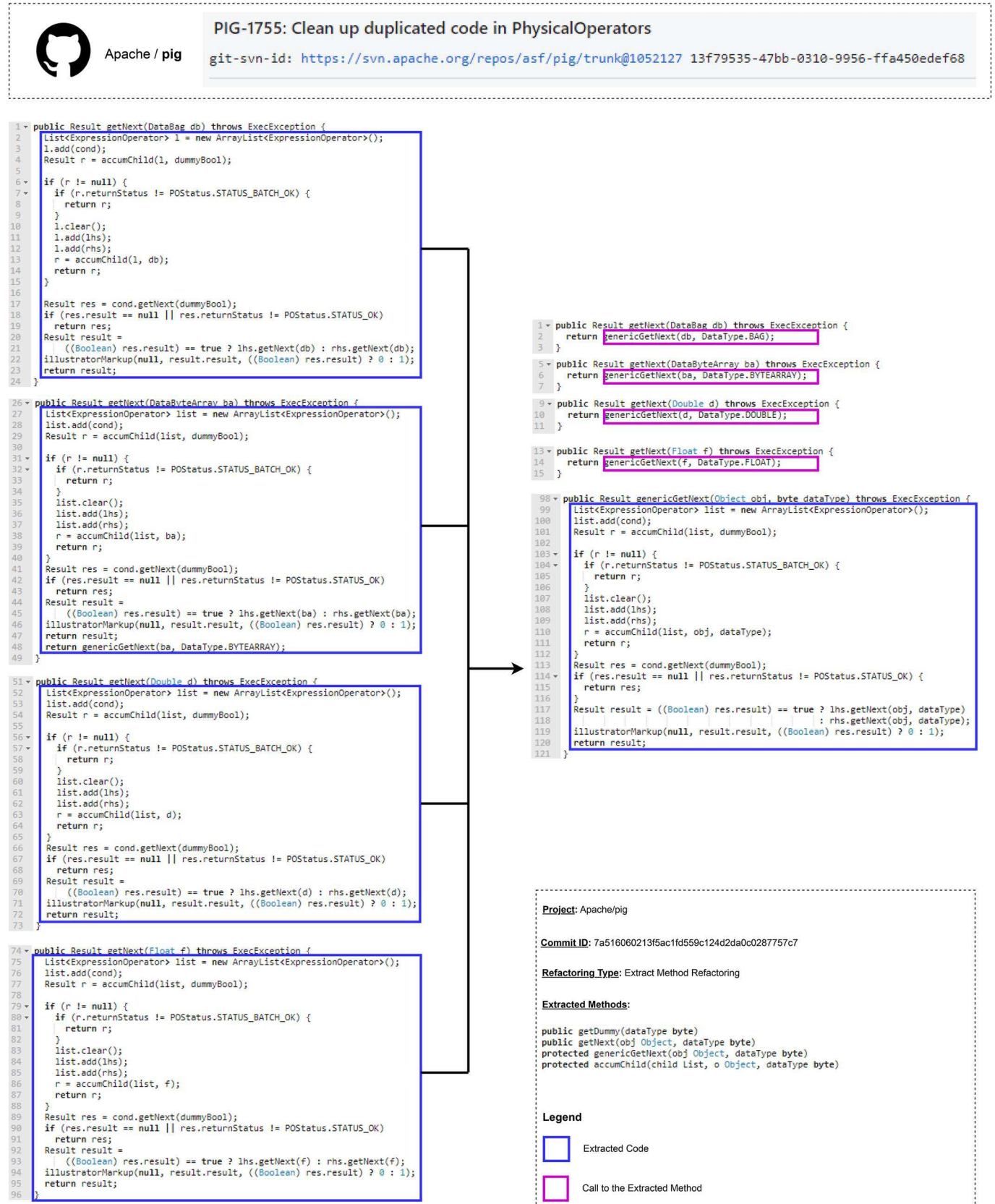
Eman Abdullah AlOmar is with the School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030 USA (e-mail: ealomar@stevens.edu).

Mohamed Wiem Mkaouer is with the College of Innovation and Technology, University of Michigan-Flint, Flint, MI 48502 USA (e-mail: mmkaouer@umich.edu).

Ali Ouni is with the Department of Software Engineering and IT, ETS Montreal, University of Quebec, Montreal, QC H3C 3P8, Canada (e-mail: ali.ouni@etsmtl.ca).

Digital Object Identifier 10.1109/TSE.2023.3345800

¹Based on JDeodorant tool usage statistics: “<https://users.ensc.concordia.ca/nikolaos/>”

Fig. 1. Sample example of *Extract Method* refactoring [1].

byte dataType) and then replaced with calls to the newly extracted method.

Given its popularity and the diversity of its usage scenarios, modern Integrated Development Environments (IDEs), such as IntelliJ IDEA, PyCharm, Eclipse, and Visual Studio offer the *Extract Method* refactoring as a built-in feature, to support the correctness of code transformation and its behavior preservation. However, the built-in feature only supports the *automation* of the refactoring and not the *recommendation* of opportunities to apply it. Therefore, various research projects focused on recommending method extraction, by identifying refactoring opportunities, such as making code more reusable [6], [19], [20], removing duplicate code [15], [16], [17], [18], improving testability through smaller test methods [28], [29], and segregating multiple functionalities [21], [22], [23], [24], [25], [26], [27]. Some of these studies have also implemented their solutions in tools and plugins.

Despite the existence of built-in IDE features, and tools, several surveys report a general reluctance of developers to adopt them [6], [31], [32], [33], [34]. In fact, surveys show that developers tend to manually extract methods despite the associated effort and error-proneness [32]. Existing research assumes that practitioners have a clear and common understanding of the intent behind method extraction, since it focuses on improving the accuracy of identifying refactoring opportunities. Yet, a recent investigation of Stack Overflow posts, related to *Extract Method*, outlines how developers are asking how to perform refactoring, whether there is tool support, and how to avoid any side effects [30]. Bridging the gap between the state-of-the-art and the state-of-the-practice starts with understanding the *intent* that drives primary studies (PSs) to identify refactoring opportunities and the extent to which they support its execution. In fact, cataloging these studies can facilitate their adoption by developers. Therefore, this paper systematically maps existing research in the recommendation of *Extract Method* refactoring from six main dimensions:

- **Intent:** refers to the motivation behind the need for a method to be extracted, *e.g.*, duplicate code removal.
- **Code Analysis:** refers to the type of source code analysis, *e.g.*, lexical and semantic code analysis.
- **Code Representation:** refers to the underlying code representation being used during the extraction, *e.g.*, source code and AST.
- **Detection:** refers to the automation degree to which a refactoring opportunity is detected, *e.g.*, manual and fully-automated.
- **Execution:** refers to the automation degree to which a refactoring opportunity is executed, *e.g.*, manual and fully-automated.
- **Validation Method:** refers to the approaches that have been suggested for evaluation method extraction, *e.g.*, case study, and experiment.

Another interesting investigation relates to the existing toolset implemented by researchers. We further classify them based on various characteristics, including their target language, availability, types of validation, etc.

Since little is known about the existing literature on *Extract Method* refactoring, this SLR serves as a comprehensive review of the body of knowledge on this topic to analyze existing techniques, and their associated programming languages. The analysis of such a wide variety of methods leads to the development of categorization and reveals areas of potential improvements. Therefore, when defining our research questions, we follow established guidelines in systematic literature review studies [35], [36], [37]. The motivation behind each question is as follows.

- **RQ₁: What approaches were considered by the PSs to recommend *Extract Method* refactoring?** We pose this RQ to study current approaches for *Extract Method*, and to get an overview of the existing approaches and their characteristics. Accordingly, for each surveyed study, we collect information about six main dimensions, together with any associated tools.
- **RQ₂: What are the main characteristics of *Extract Method* recommendation tools?** This RQ dives deeper into the characteristics of the tools. It outlines how they were implemented, maintained, and validated.
- **RQ₃: What are the datasets, and benchmarks used for evaluating and validating *Extract Method* recommendation approaches?** This RQ investigates the datasets, and benchmarks, which refers to systems and system artifacts, that are chosen and used for evaluating and validating the extraction of methods, and its results.

The main contributions of this paper are summarized as follows:

- We conduct the first SLR to review *Extract Method* refactoring, and classifying its corresponding studies from various dimensions.
- We explore the existing toolset and benchmarks generated by these studies. We provide a one-stop-shop website that links to all the tools and datasets that we were able to recover from the studies².
- We provide practical implications of our findings for researchers, developers, tool builders, and educators.

The remainder of this paper is organized as follows: Section II reviews existing studies related to systematic reviews of refactoring. Section III outlines our empirical setup in terms of search strategy, study selection, and data extraction. Section IV discusses our findings, while the research implications are discussed in Section V. Section VI captures threats to the validity of our work before concluding with Section VII.

II. RELATED WORK

Zhang et al. [38] conducted a systematic literature review (SLR) on 39 studies on bad code smells. They discussed these studies based on various aspects including the goals of the studies, the type of code smells, the approaches to detect code smells, and finally, their refactoring opportunities. Their main finding shows that *Duplicated Code* and *Long Method* are

²<https://refactorings.github.io/em-slr/>

among the most studied code smells. Furthermore, they found that nearly 49% of the primary studies aim to improve tools to detect code smells, while only 15% focus on enhancing the current knowledge of refactoring code smells. Later, Abebe and Yoo [39] conducted another systematic review of 58 studies to reveal software refactoring trends, opportunities, and challenges. Their classification helped guide researchers to address the crucial issues in software refactoring. The authors pointed out that one of the gaps in refactoring research is the lack of a refactoring tool that provides custom refactoring for all specific user needs. After that, AlDallal [40] conducted an SLR of 47 PSs published on identifying refactoring opportunities in object-oriented code. AlDallal's review classified PSs based on the considered refactoring scenarios, the approaches to determine refactoring candidates, and the datasets used in the existing empirical studies. In their study, *Extract Method* refactoring is used in refactoring identification approaches, *i.e.*, quality metrics-oriented, precondition-oriented, clustering-oriented, graph-oriented, and code-slicing-oriented approaches. In the following SLR work by AlDallal and Abdin [42], they discussed 76 PSs and classified them based on refactoring quality attributes of object-oriented code. Their finding shows that the authors of the PSs studied the impact of the *Extract Method* refactoring on quality much more frequently, and was considered by 11.8% or more of the PSs. Thereafter, Singh and Kaur [41] performed an SLR as an extension of AlDallal's SLR [40] where they analyzed 238 research items in code smell detection and its refactoring opportunities to address some research questions left open in AlDallal's SLR. Their finding reveals that *Extract Method* refactoring was used in metric-based detection techniques. Baqais and Alshayeb [44] conducted a systematic literature review on automated software refactoring. In their review, they analyzed 41 studies that propose or develop different automatic refactoring approaches, finding that *Extract Method* used in precondition-based approaches.

Other studies focus on search-based refactoring where search techniques are used to identify refactoring recommendations. Mariani and Vergilio [43] systematically reviewed 71 studies and classified them based on the main elements of search-based refactoring, including artifacts used, encoding and algorithms used, search technique, metrics addressed, available tools, and conducted evaluation. Mariani and Vergilio classified the selected PSs into five general categories related to behavior preservation methods. These categories involved (1) Opdyke's function [48], (2) Cinnéide's function [49], (3) domain-specific, (4) no evidence of behavior preservation, and (5) do not mention the method. One of their main takeaways is the need for search-based approaches to explore the need to achieve fully automated approaches for refactoring. Lacerda et al. [45] performed a tertiary systematic literature review of 40 secondary studies to identify the main observations and challenges on code smell and refactoring. Their finding shows that code smells and refactoring strongly correlate with quality attributes. They concluded that few refactoring tools exist, and some are obsolete. There is an opportunity to propose and improve *Extract Method* refactoring tools, especially tools to predict and evaluate the effects of refactoring. Abid et al. [46] analyzed the results of

TABLE I
REFACTORING-RELATED SLRs IN RELATED WORK

Study	Year	Focus	No of PSs
Zhang et al. [38]	2011	Bad smells and refactoring	39
Abebe and Yoo [39]	2014	Refactoring trends and challenges	58
AlDallal [40]	2015	Refactoring identification	47
Singh and Kaur [41]	2017	Refactoring identification	238
AlDallal and Abdin [42]	2017	Impact of refactoring on quality	76
Mariani and Vergilio [43]	2017	Search-based refactoring	71
Baqais and Alshayeb [44]	2020	Automatic refactoring	41
Lacerda et al. [45]	2020	Code smells and refactoring	40
Abid et al. [46]	2020	Refactoring research efforts	3183
AlOmar et al. [47]	2021	Refactoring behavior preservation	28
This work		<i>Extract Method</i> refactoring	83

3,183 primary studies on refactoring covering the last three decades to offer a comprehensive literature review of existing refactoring research studies. The authors derived a taxonomy focused on five key aspects of refactoring including refactoring lifecycle, artifacts affected by refactoring, refactoring objectives, refactoring techniques, and refactoring evaluation. They highlight the need to validate refactoring techniques and tools using industrial systems to bridge the gap between academic research and industry's research needs.

AlOmar et al. [47] conducted a systematic literature mapping to identify behavior preservation approaches in software refactoring. Their key finding reveals the variety of formalisms and techniques such as developing automatic refactoring safety tools and performing a manual source code analysis. However, researchers are biased toward using precondition-based and testing-based approaches although there are other techniques (*e.g.*, graph-based) that have some potential and perhaps they are effective for specific problems that have not yet been well explored. Further, the authors found that *Extract Method* refactoring is one of the most widely used refactoring operations in PSs to demonstrate behavior preservation.

Table I summarizes existing SLRs on software refactoring. Overall, we observe that all the above-mentioned studies focus on either (1) detecting refactoring opportunities through the optimization of structural metrics or the identification of design and code defects, (2) automating the generation and recommendation of the most optimal set of refactorings to improve the system's design while minimizing the refactoring effort, so that developers still can recognize their own design, or (3) demonstrating comprehensive literature review of existing refactoring research studies and the concept of behavior preservation. Our work differs from these studies, as our SLR focuses primarily on collecting and summarizing specifically *Extract Method* refactoring techniques, the "Swiss army knife of refactorings" [6], [7] with an in-depth analysis. To the best of our knowledge, no previous work has conducted a comprehensive SLR pertaining to *Extract Method* techniques in software refactoring.

III. STUDY DESIGN

This SLR aims to explore the landscape of approaches and tools that recommend the *Extract Method* refactoring. Based on established guidelines [35], [36], [50], [51], [52], we performed

the SLR in three main phases: planning, reviewing, and reporting the review. Creating a protocol is a major step when conducting an SLR [35]. The planning phase involves identifying the need for a review and the development of a review protocol (described in Section III-A). The review phase encompasses the selection of primary studies, the assessment of the study, data extraction, and data synthesis (described in Sections III-B and III-D). Finally, the reporting phase emphasizes recording the review, which involves observing documents, and presenting the obtained results (described in Section IV).

A. Survey Planning

The planning phase highlights the research motivation that leads to the development of research questions.

1) Identifying the Need for a Systematic Literature Review:

The absence of comprehensive and current secondary research that delves into the *Extract Method* underscores the need for a comprehensive Systematic Literature Review (SLR). While there have been SLRs in the field of refactoring, their focus remains confined to the automation of refactoring, the impact of refactoring on quality, detection of code smells and trends, challenges, and application of refactoring, which none specializes in *Extract Method*. Thus, the core motivation behind carrying out this SLR is to:

- Collect the body of knowledge of *Extract Method* refactoring approaches in the research literature.
- Combine and analyze the reported findings regarding *Extract Method* approaches.
- Identify open issues in existing research.

2) *Specifying the Research Questions:* During the process of conducting an SLR, it is of paramount importance to pinpoint pertinent research questions that have the potential to provide clear answers. We identified three such research questions:

- **RQ₁:** *What approaches were considered by the PSs to recommend Extract Method refactoring?*
- **RQ₂:** *What are the main characteristics of Extract Method recommendation tools?*
- **RQ₃:** *What are the datasets, and benchmarks used for evaluating and validating Extract Method recommendation approaches?*

B. Primary Studies Selection

In alignment with the research questions, we extracted the initial terms that encapsulated the research topic. Referring to previous reviews of the literature within the field, we developed search keywords incorporating synonyms and related terms.

1) *Search Strategy:* Similar to Fernandes et al. [53], we performed an automatic search in seven electronic data sources to find relevant studies, including ScienceDirect³, Scopus⁴, Springer Link⁵, Web of Science⁶, ACM Digital Library⁷, IEEE

```
((extract method OR extract-method OR method extract*
OR method-extract* OR extract function OR extract-
function OR function extract* OR function-extract* OR split
method OR split-method OR method split* OR method-
split* OR split function OR split-function OR function
split* OR function-split* OR separat* method OR separat*-
method OR method separat* OR method-separat* OR sepa-
rat* function OR separate-function OR function separat* OR
function-separat*) AND (long method OR long function OR
large method OR large function OR duplicat* code OR code
duplicat* OR code clone OR code bad smell OR code smell
OR bad smell OR antipattern OR anti-pattern OR design
defect OR design flaw) AND (refactor*) AND (approach OR
tool OR technique))
```

Text Box1: Search string.

Xplore⁸, and Wiley⁹. TextBox I shows our search string in these search engines.

The strategy to construct our search keywords is as follows:

- Derive the main terms from research questions and terms considered in the relevant papers.
- Include alternative spellings for major terms.
- Combine possible synonyms and spellings of the main terms using Boolean OR operators and then combine the main terms using the Boolean AND operators.

These search keywords are applied to titles, abstracts, and keywords. To verify the validity of the search string, we manually double-checked a few articles from each of the seven digital libraries, similar to Garousi and Mäntylä [54]. Also, during the review of this manuscript, reviewers pointed out a set of keywords whose incorporation helped with revealing more studies that were finally included. To get a high-level picture of the covered topics, we generated a word cloud of paper titles, as depicted in Fig. 3.

2) *Study Selection:* To collect the PSs, we adapted the search process of AIdallal and Abdin [42] and conducted a five-phased process. Literature publications were eliminated based on the defined inclusion and exclusion criteria to filter our irrelevant articles.

Inclusion criteria (IC):

The selected studies must satisfy all the following inclusion criteria:

- The article must be published in peer-reviewed venues before August 26, 2023.
- The article must report an approach to recommend *Extract Method* refactoring.

Exclusion criteria (EC):

Papers are excluded if satisfying any of the exclusion criteria, as follows:

- The study is a position paper, abstract, blog, editorial, keynote, tutorial, book, patent, or panel discussion.
- The study is not written in English.

³<https://www.sciencedirect.com/>

⁴<https://www.scopus.com>

⁵<https://link.springer.com/>

⁶<https://webofknowledge.com/>

⁷<https://dl.acm.org/>

⁸<https://ieeexplore.ieee.org/>

⁹<https://onlinelibrary.wiley.com/>

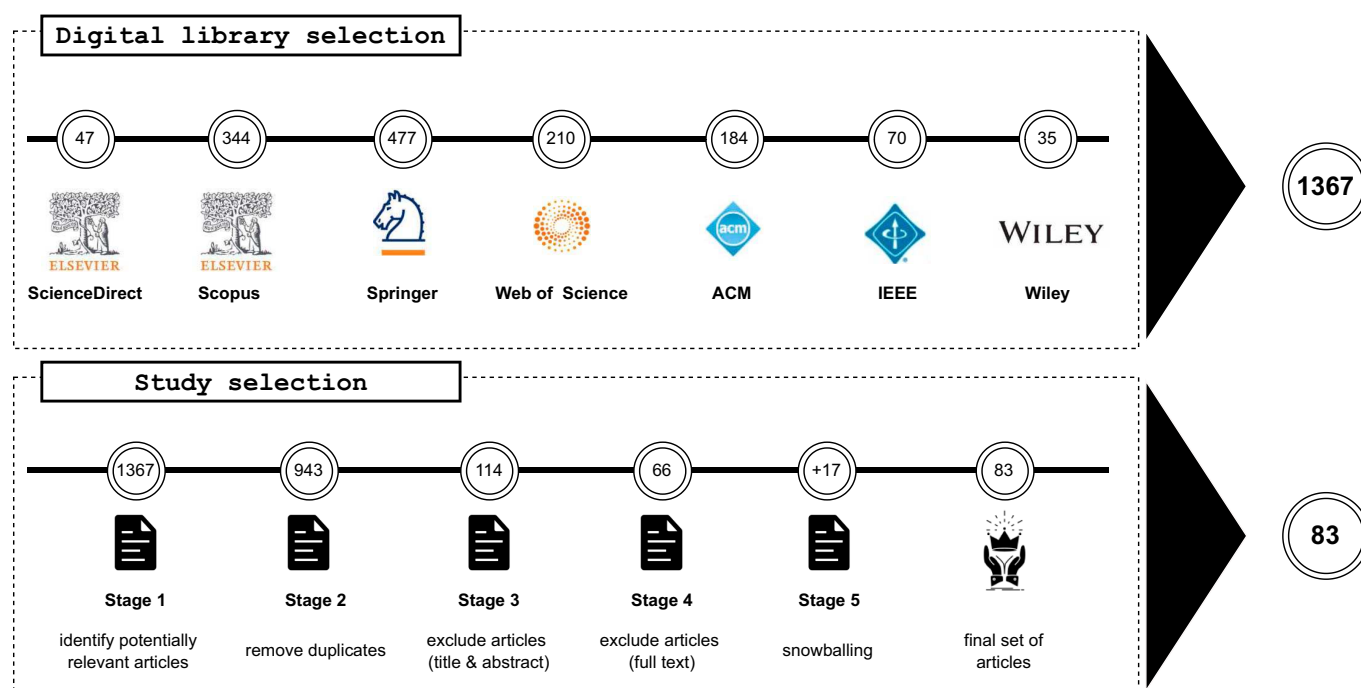


Fig. 2. Literature search process.

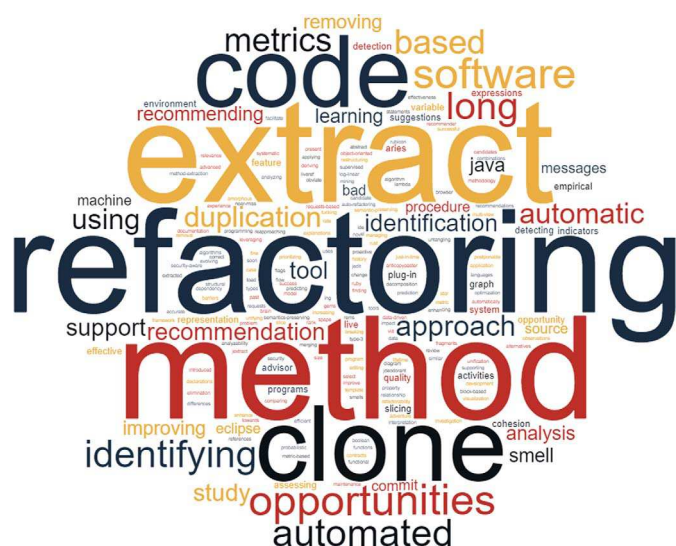


Fig. 3. Word cloud of paper titles of primary studies.

Regarding the second inclusion criteria, we only considered PSs that reported an approach to recommend *Extract Method* refactoring. We excluded any other articles that provided a broad explanation of the concept of *Extract Method* refactoring.

Stage 1: Identification of potentially relevant articles. In this first stage of the selection process, shown in Fig. 2, we searched seven digital libraries for potentially related articles. Our criteria included applying our predefined search string to the title, abstract, and keyword fields. The results of this search were not limited to specific venues. Searching through the seven digital libraries resulted in a total of 1,367 publications in the

literature. We performed the initial screening of the articles to reduce the possibility of including irrelevant articles.

Stage 2: Removal of duplicates. By merging the results obtained from the search platforms, we remove duplicate publications, books, and reports, which resulted in a total of 943 literature publications.

Stage 3: Exclusion of articles based on title and abstract. It is important to consider the abstracts at this stage because the titles of some articles could be misleading. Inclusion and exclusion rules were applied at this stage to all retrieved studies. This elimination process reduced our set of results to 114 publications in the literature. When a determination cannot be reached solely based on the title and abstracts, the studies are promoted to the next stage.

Stage 4: Exclusion of articles based on full text. To obtain the relevant PSs, the identified papers in Stage 3 were reviewed. Literature reviews were eliminated based on defined exclusion and inclusion rules. This process resulted in a total of 66 literature publications that were included in this study.

Stage 5: Snowballing. To maximize the search coverage of all relevant papers, we also performed the snowballing technique [36] on 66 papers already in the pool. Using snowballing, we extracted 1,958 references from the reference section of the studies, and extracted studies citing the 66 selected studies. We combined the results and filtered out duplicate records, along with books, and non-peer reviewed studies. Then, we compare this set with 943 primary studies obtained from Stage 2 to further refine the studies. This step resulted in the addition of 17 additional papers, where some of them did not explicitly mention the recommendation of *Extract Method* in their titles and abstracts. The updated pool size increased to 83.

C. Study Quality Assessment

To assess the quality of PSs, we followed the guidelines proposed in [35], [55], [56]. We chose three quality assessment questions that could be applicable to all PSs, and each PS is evaluated against three questions within three dimensions of study quality (*i.e.*, objective, method, and coverage of the studies). The corresponding questions are as follows: Q1) *Does the study's primary objective explicitly focus on the Extract Method refactoring?*; Q2) *Does the study include structured and preferably automatic or semi-automatic Extract Method approaches?*; and Q3) *Does the study sufficiently describe the Extract Method technique, algorithm, and evaluation?*. These questions are implicitly used in the above refinement stages. If a PS passes these quality criteria, we believe that a PS has valuable information for SLR. The answer to each of these questions is either “Yes”, “Partially”, or “No” with numerical values of 1, 0.5, or 0, respectively. If the questions did not apply to the context of a PS, they were not evaluated. The overall quality of each PS is calculated by summing up the scores of the applicable questions. In general, all the published articles in the accepted literature scored well on the quality assessment questions.

D. Data Extraction, Categorization, and Analysis

To determine the attribute(s) of the classification dimension [57], [58], we screened the full texts of the PSs and identified the attribute(s) of that dimension. We used attribute(s) generalization and refinement to derive the final map, similar to [54]. Specifically, we analyzed the PSs to create a comprehensive high-level list of themes, extracted from a thematic analysis, based on guidelines provided by Cruzes et al. [59]. Thematic analysis is among the most used methods in Software Engineering literature [6], [60], [61], for identifying and recording patterns (or “themes”) within a collection of descriptive labels, which we call “codes”. For each PS, we proceeded with the analysis using the following steps: *i)* Initial reading of the PSs; *ii)* Generating initial codes (*i.e.*, labels) for each PS; *iii)* Translating codes into themes, sub-themes, and higher-order themes; *iv)* Reviewing the themes to find opportunities for merging; *v)* Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence.

Inspired by previous studies [62], [63], we initiated our study by adopting existing taxonomies to categorize PSs. To carry out the manual coding of PSs, we used a spreadsheet application equipped with tagging capabilities. This spreadsheet provided the annotators with the following information: (1) the paper title and study link, (2) why *Extract Method* is performed (*i.e.*, intent), (3) the type of source code analysis (*i.e.*, code analysis), (4) the underlying code representation used during the extraction (*i.e.*, representation), (5) the automation degree of detecting the refactoring opportunity, (6) the automation degree of executing the recommended refactoring, and (7) the type of experiments carried out to validate the method. When creating our customized classification dimensions, annotators could select from preexisting tags in a drop-down menu or create a new one if none of the existing tags fits the specific

case (*i.e.*, each annotator had the flexibility to assign one or more tagging items).

The above-mentioned steps were performed independently by two authors. One author performed the labeling of PSs independently of the other author, who was responsible for reviewing the currently drafted themes. At the end of each iteration, the authors met and refined the themes to reach a consensus. It is important to note that the approach is not a single-step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped altogether. As the two authors progressed in translating the themes, there was some reorganization, refinement, and reclassification of the data into different or new codes. For example, we aggregated, into “Intent”, the preliminary categories “duplicated code”, “code clone”, “long method”, and “separation of concerns”. We used the thematic analysis technique to address RQ₁ and RQ₂.

E. Final Primary Studies Selection

The research method discussed in Section III resulted in 83 relevant PSs. The main venues for these relevant PSs are presented in Table II. The PSs were published in 55 different sources, including journals, conferences, and workshops. The list specifically includes 12 journals, 37 conferences, and 8 workshops. The first relevant article was published in a journal in 1998, whereas the most recent one was published in 2023. The number of literary papers published in journals, conferences, and workshops combined, is presented in Fig. 4. This figure illustrates a trend that began in 2017, resulting in a higher number of studies conducted between 2017 and 2023 compared to the total of studies published before 2017. This growing interest in this refactoring incites further research to improve its adoption in practice.

IV. RESULTS

This section reports and discusses the results of our study.

A. What Approaches Were Considered by the PSs to Recommend Extract Method Refactoring?

A detailed overview of the *Extract Method* refactoring approaches reported by the 83 PSs is shown in Table III. Upon analyzing the PSs, we extract comprehensive high-level categories grouping the techniques used to implement the *Extract Method* refactoring. These PSs are based on three main categories: (1) *Code Clone*, *Long Method*, and *Separation of Concerns* (SoC). Fig. 6 shows the percentages of *Extract Method* studies clustered by the detected intent. The *Code Clone* category had the highest number of PSs, with a ratio of 38.6%. The *Separation of Concerns* (SoC) category accounted for 34.9%, with *Long Method* representing 26.5%. Notably, these categories show minimal variation within the range of 26.5% to 38.6%. It should be noted that most of the *Extract Method* refactoring tools (49%) are primarily designed for the purpose of removing code clones. In the rest of this section, we provide a more in-depth analysis of each of these categories along with the corresponding PSs.

TABLE II
PUBLICATION VENUES

Publication Venue	PSs
Symposium on Software Reusability	[64]
International Conference on Software Engineering	[8], [65], [66], [67], [68], [69]
Conference on Software Maintenance and Reengineering	[70]
Journal of Systems and Software	[71], [72], [73]
Asia-Pacific Software Engineering Conference	[21], [74]
Workshop on Refactoring Tools	[75], [76], [77], [78], [79]
International Conference on Program Comprehension	[80], [81], [82]
Agile Processes in Software Engineering and Extreme Programming	[83]
Transactions on Software Engineering	[10], [84], [85], [86]
International Conference on Software Quality	[87], [88]
International Symposium on Software Reliability Engineering	[89]
International Conference on Software Maintenance and Evolution	[90], [91]
International Workshop on Refactoring	[15]
IEEE Access	[92]
Symposium on the Foundations of Software Engineering	[14]
Innovations in Software Engineering Conference	[23]
International Conference on Automated Software Engineering	[93], [94]
Information and Software Technology	[95], [96], [97]
Science of Computer Programming	[16]
Conference on Software: Theory and Practice	[98]
International Conference on the Art, Science, and Engineering of Programming	[99]
Computer Software and Applications Conference	[100]
International Journal of Software Engineering and Knowledge Engineering	[101]
International Conference on Software Engineering and Knowledge Engineering	[102]
Automated Software Engineering Journal	[103]
Machine Learning with Applications	[104]
Empirical Software Engineering	[105]
International Requirements Engineering Conference	[106]
Algorithms	[107]
International Conference on Software Analysis, Evolution and Reengineering	[108], [109], [110]
International Federation for Information Processing	[111]
Conference on Object-oriented programming systems and applications	[11], [112], [113]
IEICE Transactions on Information and Systems	[114]
International Conference on Computer and Communications	[115]
IASTED Conf. on Software Engineering and Applications	[116]
ACM SIGSOFT Software Engineering Notes	[117]
OOPSLA workshop on Eclipse technology eXchange	[118]
International Conference on Product Focused Software Process Improvement	[119]
Journal of Software Maintenance and Evolution: Research and Practice	[18]
International Conference on Soft Computing Techniques and Engineering Application	[120]
International Conference on Electrical Engineering/Electronics, Computer Telecommunications and Information Technology	[121]
International conference on Aspect-oriented software development	[122]
Conference on software engineering and advanced applications	[9]
Annual Computer Software and Applications Conference	[123]
International Conference on Predictive Models and Data Analytics in Software Engineering	[124]
Transactions on Software Engineering and Methodology	[125]
International Conference on Software Maintenance	[126]
Conference on Software Maintenance, Reengineering, and Reverse Engineering	[127]
Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing	[128]
International Workshop on Software Clones	[129], [130]
Workshop on Software Evolution through Transformations	[131]
Symposium on Principles of Programming Languages	[132]
ACM SIGPLAN workshop on Partial evaluation and program manipulation	[133], [134]
Working Conference on Reverse Engineering	[135]
Seminar on Advanced Techniques Tools for Software Evolution	[136]

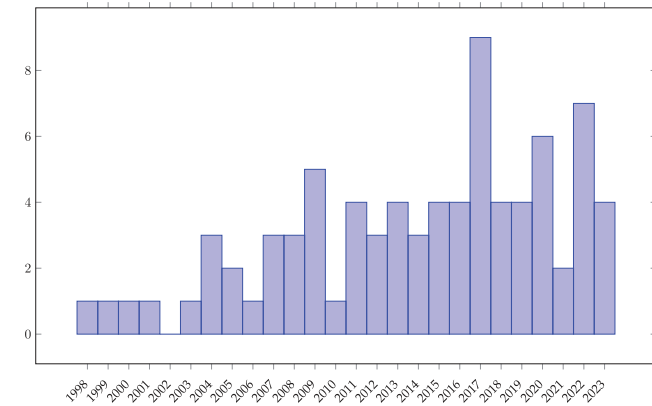


Fig. 4. Distribution of primary studies by year.

Category #1: Code Clone. This category refers to studies that are designed to recommend *Extract Method* refactoring opportunities to eliminate *Code Clone* design defects. Refactoring *Code Clone* consists of taking a code fragment and moving it to create a new method while replacing all instances of that fragment with a call to this newly created method. It is worth noting that some PSs [15], [18], [67], [68], [69], [72], [82], [86], [90], [92], [96], [109], [110], [116], [117], [119], [120], [123], [126], [127], [128], [129], [130], [131], [132], [133], [134], [135], [136], [137] utilized the concept of *Code Clone*

to consider some or all types of clones (*i.e.*, Type 1, Type 2, Type 3, Type 4), and others [83], [93], [97] utilized *Duplicate Code* by considering Type 1 clone.

Komondoor and Horwitz [132] proposed an algorithm to select statements that are worth extracting while ensuring semantics preservation. The authors identify conditions based on control and data dependencies, and the algorithm suggests moving the selected statements when the conditions hold. **ClORT** [135] is developed to take into account the shared elements of cloned methods while utilizing the strategy design pattern to differentiate them. A dynamic pattern matching algorithm is used to identify the semantic distinctions between clones and their translation in terms of programming language entities. Komondoor and Horwitz [82] propose a semantic preserving algorithm for extracting difficult sets of statements, including the detection of duplicated fragments and extracting them into procedures, to make them extractable, achieving ideal results in more than 70% of the difficult cases. **Aries** [18], [116], [117] is an *Extract Method* refactoring tool based on code clone analysis on top of their previous tool **CCShaper** [119], enabling users to select which clones to remove by characterizing code clones. Juillerat and Hirsbrunner [131] propose an algorithm for *Extract Method* refactoring to remove code clone. The algorithm first constructs the abstract syntax tree of Java code, then generates a list of tokens for clone identification, and finally identify clone that obeys certain constraints for *Extract Method* refactoring.

Wrangler [134] is a hybrid approach based on tokens and AST to detect code clones in Erlang/OTP programs automatically. The proposed clone detection approach is capable of reporting code fragments that are syntactically identical and support clone removal using function extraction. **HaRe** [133] is designed for Haskell to detect and eliminate code duplication for function extraction. Choi et al. [130] extract code clones for refactoring by combining clone metrics. Their observation is that the combinations of these metrics can identify refactorable clone classes with higher precision. **CeDAR** [96] is an Eclipse plug-in that sends the results of clone detection data to Eclipse, and the IDE receives the information and determines which clones can be refactored. This tool reportedly detects considerably more clone groups compared to open-source artifacts. **FTMPAT** [129] introduces a method that relies on slice-based cohesion metrics to merge software clones. The method starts by taking two similar methods as input and first detect syntactic differences between them using AST differencing. Subsequently, it identifies pairs of code fragments within these methods, to serve as suitable candidates for *Extract Method*. Then, the identified candidates are then evaluated and prioritized using slice-based cohesion metrics. **SPAPE** [72], [120] is a near-miss clone extraction method applied to ten large-scale open-source software and reportedly can extract more clones than this software. SPAPE was initially developed in C programming language to refactor near-miss clones automatically. The tool utilizes a symbolic program execution to transform data and identify duplicated code to ensure cohesiveness for programmers.

Krishnan et al. [126], [127] propose an algorithm for refactoring of software clones with two objectives: maximize the

TABLE III
RELATED WORK IN RECOMMENDING THE *EXTRACT METHOD* REFACTORING OPPORTUNITIES

Study	Year	Intent	Code Analysis	Code Representation	Detection	Execution	Validation Method
Lakhotia and Deprez [95]	1998	Long Method	Semantic	Graphs	Manual	Suggest Alternatives	Proof of Concept
Balazinska et al. [135]	1999	Code Clone	Syntactic	AST	Fully automated	Fully automated	Proof of Concept
Komondoor and Horwitz [132]	2000	Code Clone	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Maruyama [64]	2001	Separation of Concerns	Semantic	Graphs	Manual	Choose Candidates	Proof of Concept
Komondoor and Horwitz [82]	2003	Code Clone	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Ettinger and Verbaere [122]	2004	Separation of Concerns	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Higo et al. [119]	2004	Code Clone	Lexical	Tokens	Fully automated	Choose Candidates	Case Study
Higo et al. [116]	2004	Code Clone	Semantic	Graphs	Fully automated	Fully automated	Case Study
Higo et al. [117]	2005	Code Clone	Lexical	Tokens	Fully automated	Execute on Approval	Case Study
Higo et al. [18]	2008	Code Clone	Textual	Source Code	Fully automated	Execute on Approval	Case Study
O'Connor et al. [118]	2005	Separation of Concerns	Syntactic	AST	Semi-automated	Suggest Alternatives	Proof of Concept
Juillierat and Hirsbrunner [131]	2006	Code Clone	Syntactic	AST	Fully automated	Fully automated	Proof of Concept
Juillierat and Hirsbrunner [78]	2007	Separation of Concerns	Syntactic	AST	Manual	Fully automated	Proof of Concept
Vitek et al. [111]	2007	Separation of Concerns	Syntactic	AST	Manual	User Input	Proof of Concept
Corbat et al. [112]	2007	Separation of Concerns	Syntactic	AST	Manual	Choose Candidates	Proof of Concept
Murphy-Hill and Black [8]	2008	Separation of Concerns	Textual	Source Code	Manual	Choose Candidates	Experiment
Abadi et al. [79]	2008	Separation of Concerns	Textual	Source Code	Manual	Fully automated	Case Study
Abadi et al. [77]	2009	Separation of Concerns	Textual	Source Code	Manual	Fully automated	Case Study
Tsantalis and Chatzigeorgiou [70]	2009	Long Method	Textual	Source Code	Fully automated	Suggest Alternatives	Experiment
Tsantalis and Chatzigeorgiou [71]	2011	Long Method	Textual	Source Code	Fully automated	Suggest Alternatives	Experiment
Yang et al. [21]	2009	Long Method	Textual	Source Code	Manual	Suggest Alternatives	Case Study
Li and Thompson [134]	2009	Code Clone	Hybrids	AST and Tokens	Manual	Suggest Alternatives	Case Study
Brown and Thompson [133]	2010	Code Clone	Hybrids	AST and Tokens	Manual	Suggest Alternatives	Case Study
Kanemitsu et al. [75]	2011	Separation of Concerns	Semantic	Graphs	Manual	Suggest Alternatives	Experiment
Meananeatra et al. [121]	2011	Long Method	Syntactic	Metrics	Manual	Suggest Alternatives	Proof of Concept
Choi et al. [130]	2011	Code Clone	Lexical	Tokens	Fully automated	Manual	Case Study
Sharma [76]	2012	Separation of Concerns	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Cousot et al. [113]	2012	Separation of Concerns	Textual	Source Code	Manual	Fully automated	Proof of Concept
Tairas and Gray [96]	2012	Code Clone	Syntactic	AST	Fully automated	Choose Candidates	Experiment
Kaya and Fawcett [102]	2013	Long Method	Textual	Source Code	Fully automated	Manual	Experiment
Goto et al. [129]	2013	Code Clone	Syntactic	AST	Manual	Fully automated	Case Study
Bian et al. [72]	2013	Code Clone	Hybrids	AST and Graphs	Manual	Fully automated	Experiment
Bian et al. [120]	2014	Code Clone	Syntactic	Metrics	Fully automated	Manual	Experiment
Krishnan and Tsantalis [126]	2013	Code Clone	Textual	Source Code	Fully automated	User Input	Experiment
Krishnan and Tsantalis [127]	2014	Code Clone	Hybrids	AST and Graphs	Fully automated	User Input	Experiment
Tsantalis et al. [86]	2015	Code Clone	Hybrids	AST and Source Code and Tokens	Fully automated	User Input	Experiment
Mazinanian et al. [67]	2016	Code Clone	Hybrids	AST and Source Code and Tokens	Fully automated	User Input	Experiment
Tsantalis et al. [68]	2017	Code Clone	Hybrids	AST and Source Code and Tokens	Fully automated	User Input	Experiment
Silva et al. [80]	2014	Separation of Concerns	Textual	Source Code	Fully automated	Suggest Alternatives	Experiment
Silva et al. [98]	2015	Separation of Concerns	Textual	Source Code	Fully automated	Suggest Alternatives	Experiment
Fontana et al. [83]	2015	Code Clone	Hybrids	AST and Source Code	Fully automated	Suggest Alternatives	Experiment
Meng et al. [69]	2015	Code Clone	Syntactic	AST	Fully automated	Fully automated	Experiment
Charalampidou et al. [124]	2015	Long Method	Syntactic	Metrics	Fully automated	Fully automated	Case Study
Charalampidou et al. [10]	2016	Long Method	Syntactic	AST and Metrics	Fully automated	Fully automated	Case Study
Charalampidou et al. [9]	2018	Long Method	Syntactic	Metrics	Fully automated	Fully automated	Case Study
Haas and Hummel [87]	2016	Long Method	Hybrids	Source Code and Graphs	Manual	Suggest Alternatives	Experiment
Haas and Hummel [88]	2017	Long Method	Hybrids	Source Code and Graphs	Manual	Choose Candidates	Experiment
Xu et al. [89]	2017	Separation of Concerns	Textual	Source Code	Fully automated	Choose Candidates	Experiment
Imazato et al. [100]	2017	Separation of Concerns	Textual	Source Code	Fully automated	Manual	Experiment
Kaya and Fawcett [101]	2017	Long Method	Semantic	Graphs	Fully automated	Fully automated	Experiment
Maruyama and Hayashi [66]	2017	Separation of Concerns	Textual	Source Code	Manual	Choose Candidates	Proof of Concept
Xu et al. [115]	2017	Long Method	Syntactic	Metrics	Fully automated	Manual	Experiment
Chen et al. [123]	2017	Code Clone	Syntactic	AST	Manual	Fully automated	Case Study
Ettinger and Tyszbrowicz [110]	2016	Code Clone	Textual	Source Code	Manual	Fully automated	Proof of Concept
Ettinger et al. [109]	2017	Code Clone	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Meananeatra et al. [114]	2018	Long Method	Hybrids	AST and Graphs	Manual	Execute on Approval	Case Study
Choi et al. [74]	2018	Long Method	Syntactic	Metrics	Fully automated	Manual	Experiment
Yue et al. [90]	2018	Code Clone	Syntactic	AST	Fully automated	Manual	Experiment
Vidal et al. [125]	2018	Long Method	Textual	Source Code	Fully automated	Choose Candidates	Case Study
Yoshida et al. [15]	2019	Code Clone	Hybrids	AST and Tokens	Fully automated	Choose Candidates	Experiment
Shin [128]	2019	Code Clone	Syntactic	AST	Fully automated	Fully automated	Case Study
Barrs and Oprescu [136]	2019	Code Clone	Hybrids	AST and Graphs	Fully automated	Manual	Experiment
Antezana [65]	2019	Long Method	Textual	Source Code	Manual	Choose Candidates	Experiment
Alcocer et al. [16]	2020	Long Method	Textual	Source Code	Manual	Choose Candidates	Experiment
Nyamawe et al. [106]	2019	Separation of Concerns	Textual	Text	Fully automated	Manual	Experiment
Nyamawe et al. [105]	2020	Separation of Concerns	Textual	Text	Fully automated	Manual	Experiment
Krasniqi and Cleland-Huang [108]	2020	Separation of Concerns	Textual	Text	Fully automated	Manual	Experiment
Abid et al. [85]	2020	Separation of Concerns	Textual	Source Code	Manual	User Input	Experiment
Sheneamer [92]	2020	Code Clone	Hybrids	AST and Graphs and Tokens	Fully automated	Manual	Experiment
Aniche et al. [84]	2020	Separation of Concerns	Syntactic	Metrics	Fully automated	Manual	Experiment
Van der Leij et al. [14]	2021	Separation of Concerns	Syntactic	Metrics	Fully automated	Manual	Experiment
Sagar et al. [107]	2021	Separation of Concerns	Hybrids	Text and Metrics	Fully automated	Manual	Experiment
AlOmar et al. [103]	2022	Separation of Concerns	Textual	Text	Fully automated	Manual	Experiment
Nyamawe [104]	2022	Separation of Concerns	Textual	Text	Fully automated	Manual	Experiment
Shahidi et al. [73]	2022	Long Method	Hybrids	Graphs and Metrics	Fully automated	Fully automated	Experiment
Tiwari and Joshi [23]	2022	Long Method	Semantic	Graphs	Fully automated	Manual	Experiment
Fernandes et al. [94]	2022	Long Method	Syntactic	Metrics	Fully automated	Execute on Approval	Experiment
Fernandes et al. [99]	2022	Long Method	Syntactic	Metrics	Fully automated	Execute on Approval	Experiment
AlOmar et al. [93]	2022	Code Clone	Syntactic	Metrics	Fully automated	Execute on Approval	Experiment
AlOmar et al. [97]	2023	Code Clone	Syntactic	Metrics	Fully automated	Execute on Approval	Experiment
Cui et al. [81]	2023	Separation of Concerns	Semantic	Graphs	Fully automated	Manual	Experiment
Thy et al. [11]	2023	Separation of Concerns	Textual	Source Code	Fully automated	Fully automated	Case Study
Palit et al. [91]	2023	Separation of Concerns	Semantic	Graphs	Fully automated	Manual	Experiment

number of mapped statements and, at the same time, minimize the number of differences between the mapped statements. The authors compared the proposed technique with CeDAR and concluded that their approach can find a significantly larger number of refactorable clones. In other studies [67], [68], [86], JDeodorant has been extended to identify *Extract Method* opportunities for *Code Clone* extraction. The tool automatically assesses whether a pair of clones can be safely refactored while preserving the behavior. The authors were able to increase the percentage of refactorable clones to 36% on the same clone dataset used by Tairas and Gray [96]. Duplicated Code Refactoring Advisor (DCRA) [83] is released to select and suggest the best refactorings of duplicated code, aiming to reduce the human involvement during *Duplicated Code* refactoring procedures. The tool used NiCad [138] for clone detection, which adds information characterizing every clone, e.g., the clone's location in the class hierarchy, its size, and type. Next, through the refactoring advisor, the tool suggests the refactorings to remove the clones and provide a ranking of their quality. RASE [69] is a clone removal tool that can apply combinations of six refactorings. *Extract Method* is one of these refactorings used to extract common code guided by systematic edits. PRI [123] employs refactoring pattern templates and traces cloned code fragments across revisions. PRI takes as input the results from a clone detector, and then automatically identifies refactored regions through refactoring pattern rules in the subsequent revisions, and summarizes refactoring changes across revisions. Ettinger et al. [109], [110] contribute to the automation of type-3 clone elimination by preparation of non-contiguous code for extraction in a new method. CREC [90] is a learning-based approach that proposes specific clones through feature extraction. The tool initially refactors R-clones (historically refactored) and NR-clones (typically not refactored). This process is carried out using 34 features that analyze the characteristics of each clone to classify them. The implementation of CREC is done in three stages: preparation of the clone data, training, and testing, which allows it to provide the programmer with an accurate refactoring recommendation.

Yoshida et al. [15] released an *Extract Method* refactoring tool to be used as a proactive clone recommendation system. The process is meant to be implemented as an Eclipse plug-in to keep track of changes in the code. This tool suggests changes in real-time versus at the end of the project. This routine makes the code fresh in the programmer's mind, allowing for more efficient progress. This is accomplished by actively tracking the user's work in Eclipse and suggesting edits. Shin [128] proposes a refactoring method for finding duplicate code used in branch statements and refactoring them by extracting common parts. The results of case studies with unskilled developers yielded an average of 10% reduction in source code. CloneRefactor [136] detects code clones that are suitable for refactoring, based on their context and scope. Their results indicate that about 40% of code duplication can be refactored by method extraction, while other clones require other refactoring techniques. Sheneamer [92] automatically extracts features from detected code clones and trains models to inform programmers of the type to refactor. Their approach categorizes refactored clones as

distinct classes and develops a model to recognize the various types of refactored clones and those that are anonymous. **Anti-CopyPaster** [93], [97] is an IntelliJ IDEA plugin, implemented to detect and refactor duplicate code interactively as soon as a duplicate is created. The plugin only recommends the extraction of a duplicate only when it is *worth it*, i.e., the plugin treats whether a given duplicate code shall be extracted as a binary classification problem. This classification is performed using a CNN, trained using a dataset of 9,471 extract method refactorings of duplicate code collected from 13 open-source projects.

Category #2: Long Method. This category refers by studies that are designed to identify *Extract Method* refactoring opportunities to eliminate *Long Method* design defects. *Long Method* is a long and complex method that hinders the readability, reusability, and maintainability of the code. As a solution, refactoring *Long Method* was proposed by extracting independent and cohesive fragments from long methods as new, short, and reusable methods [9], [10], [16], [21], [23], [65], [70], [71], [73], [74], [87], [88], [94], [95], [99], [101], [102], [114], [115], [121], [124], [125], [139].

Lakhotia and Deprez [95] proposed a transformation tuck that restructures code and reorganizes unclear large fragments into small cohesive functions. **Tuck** [95] deconstructs large functions into small functions by restructuring programs. Wedge, split, and fold are the three parts that makeup tuck. Then, statements of meaningful functions in a wedge are split and folded into a new function. JDeodorant [70], [71] encompassed identifying specific *Extract Method* refactoring opportunities. This tool automatically identifies *Extract Method* opportunities for *Long Method* to suggest code improvement instead of requiring a set of statements from the programmer. Yang et al. identified fragments to be extracted from long methods. Their approach is implemented as a prototype called **AutoMed** [21]. The evaluation results suggested that the approach may reduce the refactoring cost by 40%. Meananeatra et al. [121] proposed an approach to select refactorings dependent on data flow and control flow graphs of software metrics. The method procedure includes calculating metrics, filter refactorings, computing maintainability for candidate refactorings, then outlining *Extract Method* refactorings with the highest maintainability. The approach has been reported to accurately resolve *Long Method* issues by suggesting refactoring techniques for the *Extract Method*, replacing temp with the query, and decomposing condition. Kaya and Fawcett [102] automate selecting program refactoring fragments to resolve defects with the *Long Method*. The paper goes over the identification process of code fragments based on a placement tree. This procedure outlines each node in the tree with variable reference counts to implement an effective process. Charalampidou et al. [9], [124] conduct a case study to evaluate several cohesion, coupling, and size metrics to serve as indicators of the existence of *Long Method*, and integrate these metrics into a multiple logistic regression model, enabling the prediction of whether a method should be refactored or extracted. The tool **SEMI** [10] ranks refactoring opportunities based on their extraction ability. This paper outlines *Long Method*, to be implemented within a method to identify refactoring opportunities. The SEMI approach determines which parts

of code are cohesive between statements. This can minimize the size of each method and create clear resulting methods that are increasingly single-responsibility principle compliant. This tool was validated with industrial and comparative case studies.

Hass and Hummel [87], [88] introduce refactoring and orders, each with a scoring function developed to reduce complexity and improve the way users read the code. This open-source software filters out invalid *Extract Method* refactorings and then ranks to obtain different suggestions with the previously mentioned scoring function. Kaya and Fawcett [101] strive to implement *Extract Method* refactoring and urge developers to utilize understandable implementation and modular structures so that the source code quality will not decrease throughout the project development. The goal is to refactor without requiring the user to select a code section. The approach searches for opportunities to refactor by declaring variables and regions of code that are fully extractable. The user can visualize the available refactoring options and choose which to apply without relying on a foreign code base. **LLPM** [115] combines method-level software metrics applying a log-linear probabilistic model for accustomed refactorings. This application was tested with refactorings of real-world *Extract Method* applications allowing the researchers to obtain parameter sets that capture the reason behind such refactorings. This analysis was completed by identifying the code to refactor and prioritizing various method groups to refactor. The proposed model optimizes parameters that maximize the probability of the collected dataset to refactor *Long Method* bad smells accurately. **LMR** [114] is an *Extract Method* refactoring approach that utilizes program analysis and code metrics by implementing refactoring enabling conditions. This approach uses two guidelines for practical refactoring sets: code analyzability level and the statement number. Initially, LMR is applied to a Java application core package, showing that *Long Method* bad smell can be eliminated in the code without removing behavior or making it more challenging to analyze. Choi et al. [74] investigates change metrics and *Extract Method* throughout two studies. The relationship results deduce a clear relationship between change metrics and *Extract Method*. Product and change metrics must be available to accurately recommend refactorings for *Extract Method*. The main contributions highlight metric change differences between extracted and not-extracted entities. Vidal et al. [125] proposed **Bandago**, that is implemented on top of JSPiRiT, an Eclipse plugin for identifying and prioritizing code smells in Java. Bandago performs a heuristic search using a simulated annealing algorithm [139] that repeatedly applies the *Extract Method* refactoring. Their findings reveal that the tool can automatically fix more than 60% of *Brain Methods*, and when comparing the performance of Bandago with JDeodorant, the authors found that other types of code smells are also fixed after applying the *Extract Method* refactoring suggestions.

TOAD [16], [65] searches specific portions of the source code that include the developer's original code selection and meet ideal conditions for the *Extract Method*. The approach operates during the workflow of refactorings and chooses fragments of code with correct syntax and outlined necessities.

The tool explicitly recommends auto-refactoring alternatives when the user selects a piece of code and requests refactoring options. Overall, TOAD reduced failed attempts significantly at a lower cognitive cost for *Extract Method* refactoring. Shahidi et al. [73] automatically identified and refactored the *Long Method* code smells in Java code using advanced graph analysis techniques. Their proposed approach was evaluated in five different Java projects. The findings reveal the applicability of the proposed method in establishing the single responsibility principle with a 21% improvement. In another study, Tiwari and Joshi introduced **Segmentation** [23] that identifies *Extract Method* opportunities concentrating on achieving higher performance with fewer suggestions. Compared with other tools, Segmentation outperformed F-measure approaches and suggested that it showed high precision concerning small methods and *Long Method* in opportunities with the *Extract Method*. Empirical validations were applied to six open-source code applications to assess beneficial suggestions. Segmentation improves comparable recall and precision while identifying extract method refactorings. **LiveRef** [94], [99] is a tool implemented for live refactoring Java code. It works to resolve problems with long feedback loops that allow code to be maintainable and readable. The environment provides efficient refactoring suggestions by diminishing the time needed to apply, recommend, and identify the refactoring loop. The plugin for Java IntelliJ IDEA implemented a live refactoring environment that automatically applies *Extract Method*. The tool results in improvements in the quality of the code along with faster programming solutions.

Category #3: Separation of Concerns. The Separation of Concerns (SoC) category refers to studies segregating methods into multiple sub-methods based on their behavior so the code becomes less complex and effectively reused [140]. One of the main limitations of these studies [8], [11], [14], [64], [66], [75], [76], [77], [78], [80], [81], [84], [85], [89], [91], [98], [100], [103], [104], [105], [106], [107], [108], [111], [112], [113], [118], [122] is the absence of any context related to the application of refactorings, *i.e.*, it is not clear how developers would identify the need to apply these refactoring, *e.g.*, improving design metrics or removing design defects. Maruyama [64] solves the burden of manual refactoring by implementing automatic support when initiated by the programmer. It can be used by (1) selecting a fragment of code, (2) choosing a method, and (3) naming it. A new method is created from the parts of code from an existing method through block-based slicing. This mechanism is based on data-flow and control-flow analysis, so the user will not have to test the refactored fragment. **Nate** [122] performs the *Extract Method* refactoring by extracting the slice into a new method, replacing it with a method call. For each extracted statement, the tool determines whether to remove it from the original method or to keep it there because it is still relevant. **SDAR** [118] is an Eclipse plug-in that detects and applies local and global refactoring through star diagrams. The tool offers *Extract Method* refactoring options that improve code and aid development opportunities and allows the refactoring option for every node in the diagram that passes the JDT *Extract Method* conditions. Juillerat and Hirsbrunner

[78] construct an algorithm to recognize the arguments and outcomes of an extraction method. The implementation is an Eclipse plugin and uses the Java Development Tools library provided by Eclipse.

Xrefactory [111] allows the application of *Extract Method* refactoring using a back-mapping preprocessor to perform at the level of compilers in addition to other refactorings such as renaming, adding, and moving method parameters. Although this tool only involves limited refactoring, the quality of the analysis indicates the quality of the whole refactoring tool. Corbat et al. [112] developed a plug-in for the Eclipse Ruby development tools IDE since automated refactorings are not included in Ruby. Dynamic typing of Ruby makes implementing refactorings very difficult since it can be impossible for an IDE to determine an object type; therefore, *Extract Method* refactoring was applied loosely adapted from JDT. The tool **RefactoringAnnotation** [8] for *Extract Method* refactoring allows the user to find solutions to coding errors. The annotations depend on what code section the programmer suggests and applies relevant refactoring recommendations. This is done automatically by implementing an arrow to be drawn on parameters and return values. The study concluded that speed, accuracy, and user satisfaction increase with the application of new tools. Usability recommendations are implemented, and the goal is to cultivate a new generation of tools that are user-friendly for programmers. Abadi et al. [79] re-approach the refactoring Rubicon by providing more general support for method extraction. The authors performed a case study to convert a Java servlet to use the model-view-controller pattern. Abadi et al. [77] introduces the foundation of fine slicing, a method that computes program slices. These slices can be transformed with the data removal and control dependencies as their surrounding code is extractable/executable. Cousot et al. [113] highlight the problem of automatically inferring contracts such as validity, safety, completeness, and generality with method extraction. The proposed solution was to create two fast and capable tools that interact in an environment while maintaining precision. The practical solution is comprised of forward/backward methods that are iterative. Silva et al. [80] used a similarity-based approach to recommend automated *Extract Method* refactoring opportunities that hide structural dependencies rarely used by the remaining statements in the original method. Their evaluation on a sample of 81 *Extract Method* opportunities achieved precision and recall rates close to 50% when detecting refactoring instances. In another study, Silva et al. [98] extended their work by designing an Eclipse plugin called **JExtract** that automatically identified, ranked, and applied refactorings upon request. The tool begins by generating all possibilities of *Extract Method* for each method and then ranks these methods between dependencies in the code.

ReAF [75] is a prototype tool that handles all Java language grammar. Initially, the user inputs source files to form a software system that the tool will visualize and build a procedural PDG for every method in the input. The tool can only handle Java source code but can be developed to handle other languages. Sharma [76] propose *Extract Method* candidates based on the data and the structure dependency graph. Their suggestions

were obtained by eliminating the longest dependency edge in the graph. **GEMS** [89] is an *Extract Method* refactoring recommender that extracts structural and functional features related to complexity, cohesion, and coupling. It then uses this information to identify code fragments from a given source method that can be extracted. This method was tested comparatively with JDeodorant [70], [71], JExtract [80], [98] and SEMI [10] to highlight the superiority of this tool. The Eclipse plug-in was created to support software reliability with method extraction. GEMS validates potential code for a method and assigns a “goodness” score to it and recommends refactoring with *Extract Method*. Imazato et al. [100] propose a technique to find refactoring opportunities in the code using machine learning. The history of software development was analyzed as the basis of this tool to automatically suggest *Extract Method* refactoring in the latest source code. This technique utilizes machine learning to identify potential refactoring opportunities. It consists of two phases: learning and predicting. The learning phase involves analyzing the characteristics of past cases and criteria, while the predicting phase involves detecting the location of possible refactorings. This design has the advantage of reducing the risk of overlooking refactorings. **PostponableRefactoring** [66] tool checks the code’s conditions and reports each defined error. These normal, fatal, and recoverable errors alert users when to apply the refactoring. Each error is refactorable since code may be rewritten altogether, but knowing which segments need work proves useful to programmers, especially throughout large projects. Nyamawe et al. [105], [106] recommended *Extract Method* refactorings based on the history of previously requested features, applied refactoring, and information about code smells. This learning-based approach is evaluated using a set of open-source projects with an F-measure of 70% to recommend refactorings. Krasniqi and Cleland-Huang [108] develop a model first to detect refactoring commit messages from non-refactoring commits, then differentiate between 12 refactoring types. Their findings showed that SVM has an F-measure of 15% when predicting *Extract Method* refactorings. Abid et al. [85] highlights security throughout refactoring while attempting to improve various quality attributes. The proposed idea emphasizes security metrics and balancing code qualities through multi-objective refactoring. Compared with other approaches, this tool performs above existing approaches to improve the security of systems at a low cost while not sacrificing the quality of code. The paper determined that developers must prioritize security and other important qualities when establishing refactoring systems. Aniche et al. [84] use a machine learning approach to predict refactorings using code, process, and ownership metrics. The resulting models predict 20 different refactorings at the class, method, and variable levels. Their model achieved an accuracy of 84% when predicting *Extract Method* refactoring using Random Forest and Neural Network. Another experiment that predicts refactorings was conducted using quality metrics.

Van der Leij et al. [14] explore the recommendation of the *Extract Method* refactoring at ING. They observed that machine learning models could recommend *Extract Method* refactorings with high accuracy, and the user study reveals that ING experts

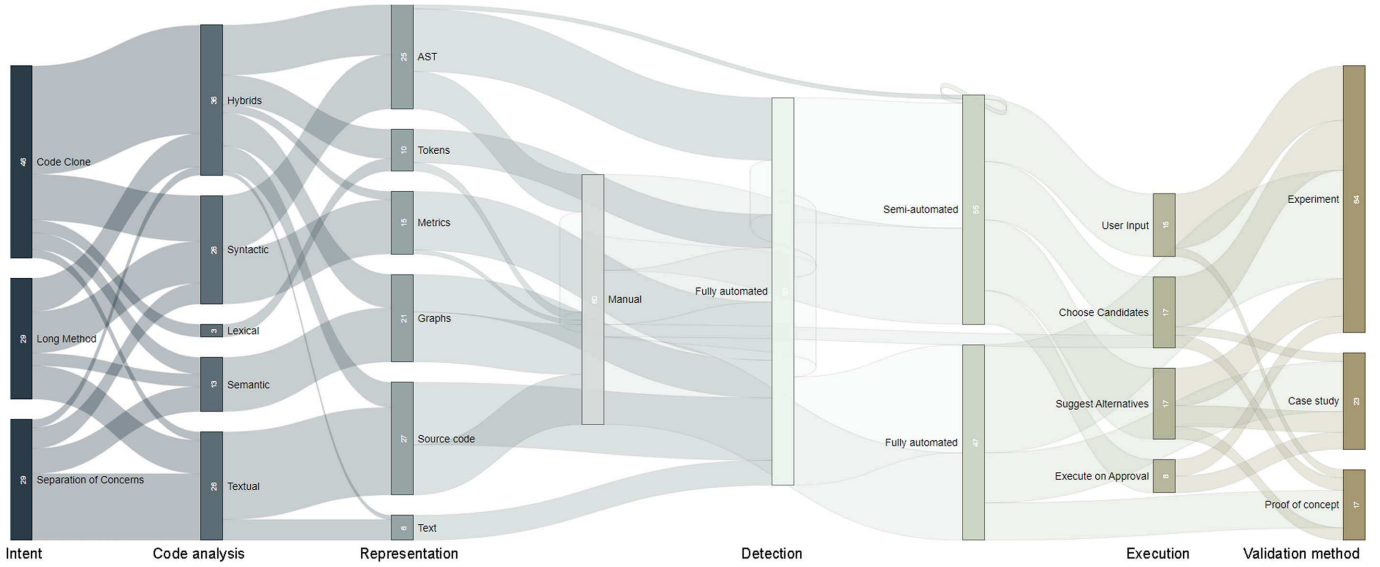


Fig. 5. The relationship among the intent, code analysis, representation, detection, execution, and validation method of the *Extract Method* refactoring.

tend to agree with most of the model's recommendations. Sagar et al. [107] compare commit messages and source code metrics to predict *Extract Method* refactoring. Their main findings show that the Random Forest trained with commit messages or code metrics resulted in the best average accuracy of around 60%. AlOmar et al. [103] formulate the prediction of refactorings as a multiclass classification problem, *i.e.*, classifying refactoring commits into six method-level refactoring operations, applying nine supervised machine learning algorithms. The prediction results for *Extract Method* ranged from 63% to 93% in terms of F-measure. To predict *Extract Method* refactorings, Nyamawe [104] employs a binary classifier and recommends required refactorings with a multi-label classifier. This is done with the help of traditional refactoring detectors and commits message analysis to detect applied refactorings through machine learning. **REMS** [81] recommend *Extract Method* refactoring opportunities via mining multi-view representations from code property graph. The results show that their approach outperforms four state-of-the-art refactoring tools, including GEMS [89], JExtract [80], [98], SEMI [10], and JDeodorant [70], [71] in effectiveness and usefulness. **REM** [11] proposed an automated *Extract Method* built on top of the IntelliJ IDEA plugin for Rust. Results reveal that REM can extract a larger class of feature-rich code fragments into semantically correct functions, can reproduce method extractions performed manually by human developers, and is efficient enough to be used in interactive development. Palit et al. [91] employ a self-supervised autoencoder to acquire a representation of source code generated by a pre-trained large language model for *Extract Method* refactoring. Their experiments show that their approach outperforms the state-of-the-art by 30% in terms of the F1 score.

Next, we elaborate on the code analysis and code representation techniques as they were mentioned in their primary studies.

Code Analysis. The nature of a code can be represented by the design properties of its specification. These properties can

be decomposed into: (1) *Textual*: no transformation or normalization is done to the source code, and generally the raw source code or textual information is used directly in the detection process; (2) *Structural*: changes the source code into a series of lexical “tokens” using a compiler-style lexical analysis; (3) *Syntactic*: employs a parser to transform source programs into parse trees or abstract syntax trees (ASTs). These can then be examined using either tree matching or structural metrics to detect code smells; (4) *Semantic*: captures the control and data flow of the program. It utilizes static program analysis to give more exact data than syntactic similarity. It generates a Program Dependence Graph (PDG), encompassing Control Flow Graphs (CFG) and Call Graphs (CG); and (5) *Hybrids*: refers to techniques that use a combination of characteristics of other approaches.

Code Representation. It spotlights the internal representation of the artifacts to be refactored. We extract comprehensive categories grouping the representation types used to implement the *Extract Method* refactoring. These PSs are based on six main categories: (1) *Source Code*, (2) *Abstract Syntax Tree (AST)*, (3) *Graphs*, (4) *Metrics*, (5) *Tokens*, and (6) *Text*. Fig. 7 illustrates the percentages of types of internal representation that the PSs used to make a decision on the extraction of the method. As can be seen, 31.3% of the PSs use *Source Code* to recommend *Extract Method* refactoring. Furthermore, 22.9% of the approaches support the execution of the *Extract Method* refactoring using *AST*. The categories *Graphs*, *Metrics*, *Tokens*, and *Text* had the least number of PSs, with a ratio of 18.1%, 10.8%, 9.6%, and 7.2%, respectively.

We notice how the 3 *Intent* clusters have used all categories of *Code Analysis*, along with its associated types of *Code Representation*. The *Code Clone* cluster, despite being the largest in terms of studies, has the least number of papers that require developers to manually input the code to be refactored. This demonstrates how the existence of code clone detection tools has been supporting the refactoring studies since their early

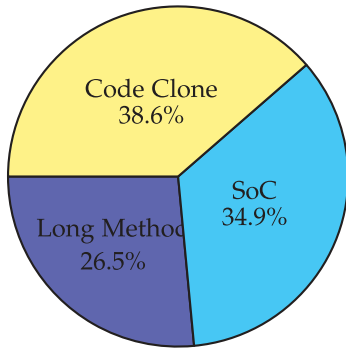


Fig. 6. Percentage of *Extract Method* studies, clustered by intent.

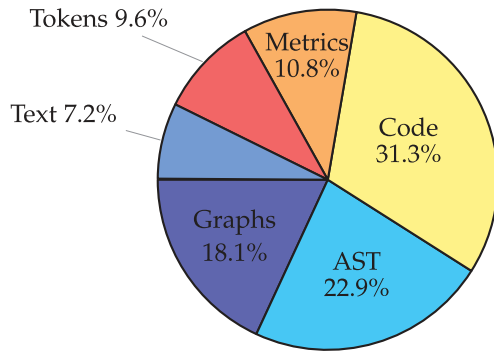


Fig. 7. Percentage of *Extract Method* studies, clustered by code representation types.

days. With the advancement in IDE support, studies shifted to automating the identification of refactoring opportunities, primarily by matching code smell patterns, then by mining patterns previously executed similar refactorings.

As for automating the recommendation, 53% of the studies opted to include the developer in the loop. Incorporation can be in the form of asking for information to complete the transformation, such as requesting the name of the extracted method [141], [142]. 61% of the studies provide multiple candidate solutions, either for the developer to choose from (e.g., [88], [96]), or to also suggest other similar alternatives (e.g., [70], [133]).

For the *Validation*, 16% of mostly earlier studies handcrafted their own synthetic examples to assess the correctness of their solutions. The need for a more developer-centric assessment triggered validation to perform case studies. Evaluating the recommendation performance with developers provides a more grounded basis for judgement, at the expense of relatively specific setting that does not necessarily generalize. The rise of information retrieval in general, along with refactoring mining in particular, allowed studies to benefit from mined refactorings to assess accuracy and conduct comparative analysis.

Fig. 5 provides detailed mappings between our six dimensions. We can observe that *Code Clone* is the most popular intent-driving method extraction with a ratio of 38.6%, followed

up by *Separation of Concerns*, taking 34.9%, and finally *Long Method* represented by 26.5%. Interestingly, this is not matched in terms of the toolset, as the highest ratio of tools goes to *Code Clone* with 49%, then *Long Method* and *Separation of Concerns* with 26.5% and 24.5%, respectively. Such observation has caught our attention particularly as *Separation of Concerns* is the only category that relies on all existing detection techniques and has its own unique one, i.e., Evolutionary-based, and yet, there is a lack of concretizing this amount of research into practical tools. As for code representation, it is unsurprising that *Code* is the most popular representation to identify need-to-refactor code fragments. This is being inherited from how research couples refactoring to a natural response to code smells, e.g., *Long Method*. So, metric-based detection rules are the most popular for detecting code smells [143], and so they become a go-to in the context of *Extract Method*. Finally, existing studies offer a wide variety of static and dynamic techniques to execute the refactoring. They mainly rely on variants techniques of code slicing and graph analysis.

Summary. 38.6% of *Extract Method* refactoring studies are primarily addressing code clones. These studies commonly employ textual and structural code analysis as their internal representation to decide on method extraction. This representation is typically based on source code or Abstract Syntax Trees (AST).

B. What Are the Main Characteristics of *Extract Method Recommendation Tools*?

To help select an appropriate *Extract Method* refactoring tool, we report in Table IV the following main characteristics that can be considered to make an informed decision about tools usage:

- *Language*: Indicates the programming language the tool supports.
- *Number of Metric*: Indicates the number of software metrics used by the tool.
- *Interface*: Indicates what IDE/user interface the tool supports.
- *Usage Guide?*: Indicates the availability of instructions on how to use the tool.
- *Tool Link*: Points to the online source code repository.
- *Last Update*: Indicates whether the tool has been consistently updated/maintained since its development.

Among the 83 primary studies, we identified 37 *Extract Method* refactoring tools. Table IV provides the results for each of the 37 tools. We report any of these characteristics as ‘Unknown’ in the table if we cannot locate the needed information and ‘N/A’ if the information is not applicable to the study. It is evident from the table that the majority of *Extract Method* tools are intended to recommend refactoring exclusively for Java-based systems. As for metrics, most studies only mention quality attributes without the names of the metrics. Next, in terms of how developers interact with these tools, we found that most of

TABLE IV
CHARACTERISTICS OF *EXTRACT METHOD* REFACTORING TOOLS

Tool	Language	No of Metric	Interface	Usage Guide?	Tool Link	Last Update
Tuck [95]	Unknown	Unknown	Unknown	No	Unknown	Unknown
CloRT [135]	Java	N/A	Unknown	No	Unknown	Unknown
Nate [122]	Java	Unknown	Eclipse	No	Unknown	Unknown
CCShaper [119]	Java	6	Command line	No	Unknown	Unknown
Aries [18], [116], [117]	Java	6	GUI-based	No	Unknown	Unknown
SDAR [118]	Java	N/A	Eclipse	No	Unknown	Unknown
Unnamed [78]	Java	N/A	Eclipse	No	Unknown	Unknown
Xrefactory [111]	C++	N/A	Unknown	Yes	[144]	2007
Unnamed [112]	Ruby	N/A	Eclipse	Yes	[145]	2012
RefactoringAnnotation [8]	Java	Unknown	Eclipse	No	Unknown	Unknown
JDeodorant [67], [68], [70], [71], [86], [126], [127]	Java	3	IntelliJ / Eclipse	Yes	[5]	2019
AutoMed [21]	Java	10	Unknown	No	Unknown	Unknown
Wrangler [134]	Erlang/OTP	N/A	GUI-based / Command line	Yes	[146]	2023
HaRe [133]	Haskell 98	N/A	GUI-based / Command line	Yes	[147]	2017
ReAF [75]	Java	Unknown	Unknown	No	Unknown	Unknown
Unnamed [113]	C#	Unknown	Visual Studio extension	No	Unknown	Unknown
CeDAR [96]	Java	2	Eclipse	No	Unknown	Unknown
FTMPAT [129]	Java	3	Eclipse	No	Unknown	Unknown
SPAPE [72]	Procedural / Java	Unknown	Unknown	No	Unknown	Unknown
JExtract [80], [98]	Java	Unknown	Eclipse	Yes	[148]	2016
DCRA [83]	Java	1	Unknown	No	Unknown	Unknown
RASE [69]	Java	N/A	Eclipse	Yes	[149]	2015
SEMI [10]	Java	5	GUI-based / Command line	Yes	[150]	2016
GEMS [89]	Java	48	Eclipse	Yes	[151]	2017
PostponableRefactoring [66]	Java	N/A	Eclipse	Yes	[152]	2018
LLPM [115]	Java	4	Unknown	No	Unknown	Unknown
PRI [123]	Java	N/A	Eclipse	No	Unknown	Unknown
LMR [114]	Java	5	Eclipse	No	Unknown	Unknown
CREC [90]	Java	N/A	Eclipse	Yes	[153]	2018
Bandago [125]	Java	4	Eclipse	No	Unknown	Unknown
Unnamed [15]	Java	N/A	Eclipse	No	[154]	2019
Unnamed [128]	Java	N/A	Unknown	No	Unknown	Unknown
CloneRefactor [136]	Java	N/A	Command line	No	[155]	2020
TOAD [16], [65]	Pharo	N/A	Pharo	Yes	[156]	2019
Segmentation [23]	Java	2	Eclipse	No	[157]	2022
LiveRef [94], [99]	Java	20	IntelliJ	Yes	[158]	2022
AntiCopyPaster [93], [97]	Java	78	IntelliJ	Yes	[159]	2023
REM [11]	Rust	N/A	IntelliJ	Yes	[160]	2023

the tools are in the form of IDE plugins, *i.e.*, Eclipse or IntelliJ, and user interface or command line. Regarding tool availability, we searched for a link to the tool website or binaries. In case the link is absent or no longer functional, we contacted the publication's authors. From these 37 *Extract Method* tools, we could only locate 18 tools. Fig. 8 depicts a timeline of releasing 37 *Extract Method* refactoring tools, in which 18 tools are made publicly available online by the research community. There has been a considerable increase in the number of tools in the last two decades. The earlier tools were responsive to the challenge of ensuring the correctness of the transformation and its behavior preservation, given the lack of IDE support. The evaluation of these tools was mainly handcrafted, using fewer examples as a proof of concept. When IDEs started supporting the execution of code extraction, studies shifted toward automating the identification of refactoring opportunities while including developers in the tool design and evaluation. The rise of refactoring mining tools has enabled another dimension for studies to leverage previously performed extractions as *ground truth* for predictive modeling, or for comparison baselines between existing solutions. Finally, recent techniques have taken a proactive fashion to immediately recommend refactoring, as soon as the opportunity is detected, in order to facilitate the adoption of the proposed change.

Several approaches have different automation support for detection and correction of *Extract Method* refactoring

identification. In the rest of this section, we analyze the following level of automation for the *Extract Method* refactoring tools.

Category #1: Manual approach refers to using code inspection to detect or correct code smells.

Category #2: Full automated approach refers to providing explicit full tool support to the users without human intervention.

Category #3: Semi-automated approach for the semi-automated approaches, it is broken down into four categories:

- *Suggest Alternatives*: refers to the tool that is capable of carrying out the task automatically and proposing options or alternatives to the user. Nevertheless, the user must still manually select and implement the suggestion;
- *Choose Candidates*: refers to the tool that proposes alternative tasks to be done and requires the user to confirm the selection;
- *Execute on Approval*: refers to the tool that displays the activity that is about to be carried out and requests the user's permission. The user can either accept the activity in its entirety or cancel it;
- *User Input*: refers to the tool that asks the user to select the code fragment as input to the tool.

Regarding the automaticity in the *Extract Method* refactoring, we observe that most tools perform fully automated or semi-automatic refactoring tools. For example, the tool suggests

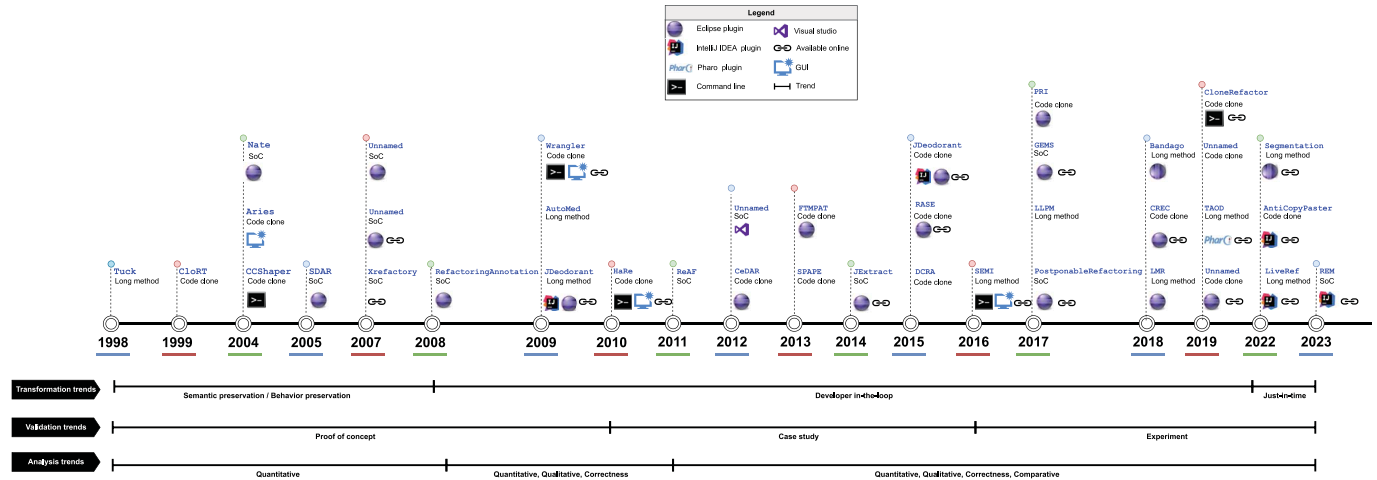


Fig. 8. Timeline of developing *Extract Method* refactoring tools.

an *Extract Method* refactoring for the code clone fragments, and the developer decides whether to apply or reject that refactoring. It is essential to highlight that automated refactoring alone cannot eliminate the need for manual verification after applying refactoring or manual refactoring in particular scenarios. That explains why many *Extract Method* refactoring tools support semi-automatic refactoring. Furthermore, we observe that some tools utilize existing code smell detectors, and others integrate the detection of code smell and the execution of refactoring in the same tool. The latter eliminates the need to set up the dependency on a separate *Long Method* splitter or *Code Clone* detector.

Fig. 9 depicts the software metrics used by the 14 *Extract Method* refactoring tools (the white color indicates that the tool computes the respective metric, while black signifies that the tool does not). It is worth noting that we only include metrics that the PSs report. Some PSs indicated the usage of metrics without specifying the metric names. As can be seen, 14 of the *Extract Method* refactoring tools, namely, Aries, AntiCopyPaster, AutoMed, Bandago, CeDAR, DCRA, FTMPAT, GEMS, JDeodorant, LLPM, LMR, LiveRef, SEMI, and Segmentation, indicated the metrics. These metrics relate to cohesion, coupling, complexity, size, keyword, and clone pairs. We found that ‘TotalLinesOfCode’, ‘CyclomaticComplexity’, ‘LackOfCohesionOfMethod’, ‘NumberOfMethods’, ‘NumberOfParameters’, and ‘NumberOfAssignedVariables’ are common metrics utilized by most of the tools. It should be noted that some of these metrics are used to assess quality improvement in refactoring research [161], [162].

Table V shows the quantitative, qualitative, comparative, and correctness data analysis of *Extract Method* refactoring tools. It is evident from the table that there is a noticeable absence of validation-related information from both quantitative and qualitative perspectives. While the quantitative analysis seems to be the default experimentation by most of the primary studies, only 34% reported the correctness of their tools through the standard performance metrics (e.g., precision, recall). On the other hand, 26% of tools were purely evaluated qualitatively. Only 15%

of the tools undergo both quantitative and qualitative analysis. Moreover, JDeodorant and JExtract are widely used by 23% of the studies for comparative analysis. To summarize, most studies rely on quantitative analysis or qualitative analysis to create oracles for their recommendation. Therefore, they need to go beyond the correctness and investigate the usefulness of their recommendations from the developer’s standpoint, which was done only for 15% of the tools. Additionally, many studies do not position their recommendations properly with respect to existing literature reviews through proper comparative analysis. Regarding correctness, most tools do not indicate details around their accuracy. From the set of 37 *Extract Method* tools, only 11 tools provide information about the tool’s accuracy.

Summary. A total of 37 *Extract Method* refactoring tools have been developed, with 49% designed for refactoring code clones and 24% intended to break down lengthy methods. Among these tools, approximately 58% are developed as plugins, 9% are command-line tools, and 9% feature graphical user interfaces (GUIs). Several of these tools incorporate the developer’s involvement in the decision-making process when applying the method extraction.

C. What are the Datasets, and Benchmarks Used for Evaluating and Validating *Extract Method* Recommendation Approaches?

We investigate the datasets, and benchmarks that are used to evaluate and validate *Extract Method* refactoring studies. We follow the same extraction procedure as described in Abgaz et al. [163]. A summary of the findings is illustrated in Tables VI, VII, and VIII.

Codebases. The evaluation of proposed *Extract Method* studies depends on the availability of datasets and benchmarking data, which is a relatively unexplored area. We identified that most of the studies used a dataset created by the paper’s authors, corresponding to 86.74%. Only 13.25% reused datasets

TABLE V
QUANTITATIVE, QUALITATIVE, AND COMPARATIVE ANALYSIS OF *EXTRACT METHOD* REFACTORING TOOLS

Tool	Quantitative	Qualitative	Comparative	Correctness
Tuck [95]	Unknown	No	No	Unknown
CloRT [135]	Unknown	Unknown	Unknown	Unknown
Nate [122]	Unknown	No	No	Unknown
CCShaper [119]	1 project	No	No	Unknown
Aries [18], [116], [117]	1 project	No	No	Unknown
SDAR [118]	Unknown	No	No	Unknown
Xrefactory [111]	Unknown	No	No	Unknown
Unnamed [112]	Unknown	No	No	Unknown
RefactoringAnnotation [8]	5 projects	w/ 16 developers	No	Unknown
JDeodorant [70], [71]	1 project	w/ 1 developer	No	Precision: 33.3% - 100% Recall: 25% - 100 % Precision (AVG): 51% Recall (AVG): 69%
JDeodorant [67], [68], [86]	9 projects	No	w/ CeDAR	Accuracy: increase to 36%
JDeodorant [126], [127]	7 projects	No	w/ CeDAR	Accuracy: increase to 83%
AutoMed [21]	1 project	No	No	Accuracy: 3.57% - 92.86%
Wrangler [134]	3 projects	No	No	Unknown
HaRe [133]	13 programs	No	No	Unknown
ReAF [75]	1 project	w/ 14 developers	w/ JDeodorant	Unknown
Unnamed [113]	Unknown	w/ 4 authors	No	Unknown
CeDAR [96]	9 projects	No	w/ Aries and Supremo*	Unknown
FTMPAT [129]	1 project	No	No	Unknown
SPAPE [72]	10 projects	No	No	Unknown
JExtract [80], [98]	12 projects	No	w/ JDeodorant	Precision: 38% - 48% Recall: 38% - 48%
DCRA [83]	50 projects	No	No	Unknown
RASE [69]	2 projects	w/ experts	w/ RASE entire methods	Accuracy: 58%
SEMI [10]	5 projects	w/ 3 developers	w/ JDeodorant w/ JExtract	Precision: 13.8% - 22.4% Recall: 57.1% - 92.8% F-measure: 22.23% - 36.09%
GEMS [89]	5 projects	w/ 4 authors	w/ JDeodorant w/ JExtract w/ SEMI	Precision: 13.3% - 25.3% Recall: 31.9% - 49.2% F-measure: 18.8% - 32.7%
PostponableRefactoring [66]	Unknown	No	No	Unknown
LLPM [115]	5 projects	No	w/ JDeodorant w/ JExtract	Precision: 18.5% - 30.3% Recall: 52.6% - 62.1% F-measure: 27.4% - 40.7%
PRI [123]	6 projects	No	No	Accuracy: 94.1%
LMR [114]	1 project	No	No	Unknown
CREC [90]	6 projects	No	No	F-measure: 76% - 83%
Bandago [125]	10 projects	w/ 35 developers	w/ JDeodorant	Unknown
Unnamed [128]	Unknown	w/ 6 teams	No	Unknown
Unnamed [15]	2 projects	w/ 8 developers	No	Unknown
CloneRefactor [136]	1,343 projects	No	No	Unknown
TOAD [16], [65]	9 projects	w/ 10 developers	No	Unknown
Segmentation [23]	6 projects	No	w/ JExtract w/ SEMI	Precision: 22.81% - 38.75% Recall: 24.58% - 41.75% F-measure: 23.66% - 40.19%
LiveRef [94], [99]	3 projects	w/ 42 developers	No	Unknown
AntiCopyPaster [93], [97]	13 projects	w/ 72 developers	No	Precision: 82% Recall: 82% F-measure: 82% PR-AUC: 86%
REM [11]	5 projects	No	w/ IntelliJ's Rust w/ Visual Studio Rust Analyzer	Unknown

* indicates the tool is not peer-reviewed

from previous studies. The selection of applications for experimentation is based on the availability of the source code, and the *Extract Method* tools. Due to the absence of agreed-upon evaluation benchmarks, studies have generally used custom evaluations. Generally, PSs have mostly employed relatively small- or medium-scale open-source applications, typically containing less than 225,000 lines of code. Examples of open-source systems utilized by some PSs with the intent of *Long*

Method and *Separation of Concerns* include JHotDraw and JUnit. Ant and JFreeChart are becoming popular Java systems for *Extract Method* evaluation when extracting code clone¹⁰.

Validation Methods. Various structured evaluation approaches have been suggested, such as proof of concepts, case

¹⁰Due to space constraints, we report project names if the number of projects considered is less than or equal to 15.

TABLE VI
BENCHMARKS AND DATASETS USED IN *EXTRACT METHOD* REFACTORING STUDIES FOR *LONG METHOD* DECOMPOSITION

Study	Intent	Language	No of Metric	No of Project	Project	Other Properties	Dataset Link	Validation Method
Tuck [95]	Long Method	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Proof of Concept
JDeodorant [70], [71]	Long Method	Java	3	1	Violet 0.16	LOC: 4,100/ 61 classes/ 144 methods	Unknown	Experiment
AutoMed [21]	Long Method	Java	10	1	hourReader 1.8.0	LOC: 20,000 / 269 classes	Unknown	Caee Study
Meaneatna et al. [121]	Long Method	Java	3	Unknown	Unknown	Unknown	Unknown	Experiment
Kaya & Fawcett [102]	Long Method	C++	N/A	Unknown	Unknown	Unknown	Unknown	Experiment
Charalampidou et al. [124]	Long Method	Java	5	1	jFlex	Unknown	Unknown	Caee Study
Charalampidou et al. [9]	Long Method	Java	8	1	jFlex	Unknown	Unknown	Caee Study
SEMI [10]	Long Method	Java	5	5	Wikidev	Unknown	[165]	Caee Study
					MyPlanner			
					MyWebMarket			
					JUnit			
					JHotDraw			
Haas & Hummel [87]	Long Method	Java	2	3	Agilefant	LOC: 36,116/ 2,841 methods	Unknown	Experiment
					JabRef	LOC: 128,145 / 5,665 methods		
					JChart2D	LOC: 50,728 / 1,849 methods		
Haas & Hummel [88]	Long Method	Java	9	13	Unknown	Unknown	Unknown	Experiment
Kaya & Fawcett [101]	Long Method	C++	N/A	Unknown	Unknown	Unknown	Unknown	Experiment
LLPM [115]	Separation of Concerns	Java	4	5	Wikidev	130 total methods	Unknown	Experiment
					SelfPlanner			
					MyWebMarket			
					JUnit			
					JHotDraw			
LMR [114]	Long Method	Java	5	1	JFreeChart 1.0.17	LOC: 5,665 / 20 classes / 552 methods	Unknown	Caee Study
Choi et al. [74]	Long Method	Java	6	1	JEdit	LOC: 97,116 - 313,706	Unknown	Experiment
Bandago [125]	Long Method	Java	4	10	Columba 1.4	LOC: 26,600/ 436 classes	[166]	Caee Study
					JGraphT 0.9.0	LOC: 14,180 / 218 classes		
					SportTracker 5.7	LOC: 5,200 / 40 classes		
					Cayenne 4.0	LOC: 45,000 / 533 classes		
					CheckStyle 6.4.1	LOC: 60,000 / 399 classes		
					Jena 2.12.1	LOC: 54,410 / 697 classes		
					JGroups 3.4.8	LOC: 76,570 / 644 classes		
					Quartz 2.1.7	LOC: 26,810 / 176 classes		
					Roller 5.1.2	LOC: 47,460 / 452 classes		
					Squirrel 3.6.0	LOC: 79,070 / 879 classes		
TOAD [65], [16]	Long Method	Pharo	N/A	9	GitMultipleMatrix	Unknown	[167]	Experiment
					TestDeviator			
					DrTest			
					Regis			
					SmallSuiteGenerator			
					Roassal			
					Live Robot Programming			
					KerasBridge			
					GToolkit Documenter			
Shahidi et al. [73]	Long Method	Java	Unknown	5	JEdit 4.5.1	LOC: 107,212 / 1,141 classes / 6,663 methods	Unknown	Experiment
					FreeMind 0.9.0	LOC: 40,933 / 696 classes / 4,583 methods		
					ArgoUML 0.34	LOC: 249,538 / 2,539 classes / 17,485 methods		
					JFreeChart 1.0.14	LOC: 222,814 / 8,630 classes / 619 methods		
					jVLT 1.3.2	LOC: 29,161 / 420 classes / 2,036 methods		
Segmentation [23]	Long Method	Java	2	6	JUnit	Unknown	[157]	Experiment
					JHotDraw			
					MyWebMarket			
					EventBus			
					Mockito			
					XData			
LiveRef [94], [99]	Long Method	Java	20	3	Space Invaders	Unknown	[158]	Experiment
					JHotDraw			
					Movie rental system			

studies, and experiments. Proof of concept involves demonstrating how the identification process works with the help of examples. Case studies examine the migration process in depth by looking at relevant cases, using one or multiple projects as a target. Experiments involve selecting the chosen codebases and then experimentally evaluating them using metrics such as coupling, cohesion, complexity, and code size, or comparing them with other tools. It should be noted that validation methods are reported as they were mentioned in their primary studies.

Previous studies have classified validation methods into *proof of concepts*, *case studies*, and *experiments* [163], [164]. In our study, *experiment-based* validation is the most widely used method, with 59.03% of the studies that use it [8], [10], [14], [15], [16], [23], [65], [67], [68], [69], [70], [71], [72], [73], [74], [75], [80], [81], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94], [96], [97], [98], [99], [100], [101], [103], [104], [105], [106], [107], [108], [115], [121], [126], [127], [134]. Some of these studies even combined a survey or user study with their experiment (*e.g.*, [93], [94], [97], [99], [125]). The *case study* is the second most dominant method,

with 21.68% of the papers applying it to evaluate their methods [9], [11], [18], [21], [77], [79], [114], [116], [117], [123], [124], [125], [128], [129], [130], [133], [137]. *Proof of concept* method was also adopted by 19.27% [64], [76], [78], [82], [95], [109], [110], [111], [112], [113], [118], [122], [131], [132], [135]. It is evident that experiment-based validation is becoming more popular. This is likely due to recent advances in metrics and benchmarks that make it easier to compare different *Extract Method* techniques.

Programming Languages. The majority of studies (81.92%) centralize on Java-based applications [8], [9], [10], [14], [15], [18], [21], [23], [64], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [77], [78], [79], [80], [81], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94], [96], [97], [98], [99], [100], [103], [104], [105], [106], [107], [108], [109], [110], [114], [115], [116], [117], [118], [121], [122], [123], [124], [125], [126], [127], [128], [129], [130], [131], [135], while C++ [72], [111], [120], Ruby [112], C# [113], Pharo [16], [65], Haskell [137], Erlang/OTP [134] and Rust [11], Java and Procedural in combination [72], accounts for 18.07%. It is evident that *Extract Method* studies tend to

TABLE VII
BENCHMARKS AND DATASETS USED IN *EXTRACT METHOD* REFACTORING STUDIES FOR *CODE CLONE* EXTRACTION

Study	Intent	Language	No of Metric	No of Project	Project	Other Properties	Dataset Link	Validation Method
CloRT [135]	Code Clone	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Komondoor & Horwitz [132]	Code Clone	Procedural	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Komondoor & Horwitz [82]	Code Clone	Procedural	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
CCShaper [119]	Code Clone	Java	6	1	Ant 1.6.0	LOC: 180,000 / 627 files	Unknown	Caee Study
Aries [18], [116], [117]	Code Clone	Java	6	1	Ant 1.6.0	LOC: 180,000 / 627 files	Unknown	Caee Study
Juillerat & Hirsbrunner [131]	Code Clone	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Wrangler [134]	Code Clone	Erlang/OTP	N/A	3	Wrangler Mnesia Yaws	LOC: 30,872 LOC: 28,152 LOC: 29,603	Unknown	Experiment
HaRe [133]	Code Clone	Haskell 98	N/A	13	Previous work [137]	Unknown	Unknown	Caee Study
Choi et al. [130]	Code Clone	Java	3	1	Unknown	KLOC: 110 / 296 files	Unknown	Caee Study
CeDAR [96]	Code Clone	Java	2	9	Ant 1.7.0 Columba 1.4 EMF 2.4.1 Hibernate 3.3.2 Jakarta-JMeter 2.3.2 JEdit 4.2 JFreeChart 1.0.10 JRuby 1.4.0 Squirrel-SQL 3.0.3	KLOC: 67 KLOC: 75 KLOC: 118 KLOC: 209 KLOC: 54 KLOC: 51 KLOC: 76 KLOC: 101 KLOC: 141	Unknown	Experiment
FTMPAT [129]	Code Clone	Java	3	1	Ant 1.7.0	Unknown	Unknown	Caee Study
SPAPE [72]	Code Clone	Procedural	Unknown	10	Linux 2.6.6/kernel Unix/make 3.82 htpd 2.2.2/server devcot 2.0.8/src/auth gstreamer 0.10.31/gst gtk 2.91.5/gdk/x11 iptables 1.4.10/extensions nginx-0.8.15/src/core proftpd 1.3.3c/src PostgreSQL 9.0.2/src/backend/access	LOC: 30,629 LOC: 33,864 LOC: 36,926 LOC: 18,243 LOC: 66,637 LOC: 30,118 LOC: 19,668 LOC: 17,126 LOC: 34,404 LOC: 65,046	Unknown	Experiment
Bian et al. [120]	Code Clone	Java	Unknown	5	Linux 2.6.6/arch Linux 2.6.6/net Linux 2.6.6/sound/drivers Unix/make 3.82 http2.2.2/server	Unknown Unknown Unknown Unknown Unknown	Unknown	Experiment
JDeodorant [67], [68], [86], [126], [127]	Code Clone	Java	N/A	9	Ant 1.7.0 / Ant 1.9 Columba 1.4 EMF 2.4.1 JMeter 2.3.2 / JMeter 2.9 JEdit 4.2 JFreeChart 1.0.10 / JFreeChart 1.0.14 JRuby 1.4.0 / JRuby 1.7.3 Hibernate 3.3.2 Squirrel-SQL 3.0.3	KLOC: 67 KLOC: 75 KLOC: 118 KLOC: 54 KLOC: 51 KLOC: 76 KLOC: 101 KLOC: 209 KLOC: 141	Unknown	Experiment
DCRA [83]	Code Clone	Java	1	50	Qualitas Corpus [168] (v. 20120401)	Unknown	Unknown	Experiment
RASE [69]	Code Clone	Java	N/A	2	Previous works [169], [170]	Unknown	[171]	Experiment
CREC [90]	Code Clone	Java	N/A	6	Axis2 Eclipse-jdt.core Elastic Search JFreeChart JRuby Lucene	8,723 commits 22,358 commits 14,766 commits 3,603 commits 24,434 commits 22,061 commits	[153]	Experiment
PRI [123]	Code Clone	Java	N/A	6	AlgoUML Tomcat Log4j Eclipse AspectJ JEdit JRuby	LOC: 127,145 / 1,559 files LOC: 215,584 / 1,537 files LOC: 59,499 / 817 files LOC: 326,563 / 4,758 files LOC: 107,368 / 561 files LOC: 186,514 / 1,256 files	Unknown	Caee Study
Ettinger et al. [109], [110]	Code Clone	Java	N/A	Unknown	Previous work [172]	59 clone pairs	Unknown	Proof of Concept
Unnamed [15]	Code Clone	Java	N/A	2	JFreeChart JUnit	KLOC: 260 / 990 classes KLOC: 43 / 449 classes	Unknown	Experiment
Unnamed [128]	Code Clone	Java	N/A	Unknown	Unknown	Unknown	Unknown	Caee Study
CloneRefactor [136]	Code Clone	Java	N/A	1,343	Previous work [173]	LOC (AVG): 980	Unknown	Experiment
Shenamer [92]	Code Clone	Java	N/A	6	Previous work [90] netbeans eclipse-jdtcore EITC J2sdk1.4.0-javaw eclipse-ant cococon	[153] 200 paired clones 400 paired clones 426 paired clones 482 paired clones 522 paired clones 655 paired clones	Dataset of [153] Unknown	Experiment
AntiCopyPaster [93], [97]	Code Clone	Java	78	13	arhas easyexcel camel-quarkus commons-lang flink iceberg jena pulsar storm apollo JavaGuide	73,884 total commits	[174]	Experiment

incorporate Java codebases. This could be because many tools *Extract Method* are designed for Java.

Dataset Availability. Dataset availability is one of the essential factors that allow the reproducibility and extension of studies. We collect all artifacts associated with the PSs, which encompasses studies providing raw datasets that require processing by researchers, as well as those that offer solely user survey responses from developers. It is observed from Tables VI, VII, and VIII that 78.31% of *Extract Method* datasets are not

publicly available. This observation highlights the need for public datasets to enable replication and extension of studies and mitigate benchmark bias when comparing the proposed approach with existing studies.

We conjecture that the ground truth used to compare with existing studies might be biased. Also, the comparison against the state-of-the-art may not be appropriate unless these tools are called in the same context or intent as in the original paper. For instance, JDeodorant applies the *Extract Method* refactoring

TABLE VIII
BENCHMARKS AND DATASETS USED IN *EXTRACT METHOD* REFACTORING STUDIES FOR *SEPARATION OF CONCERNS*

Study	Intent	Language	No of Metric	No of Project	Project	Other Properties	Dataset Link	Validation Method
Maruyama [64]	Separation of Concerns	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Nate [122]	Separation of Concerns	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
SDAR [118]	Separation of Concerns	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Juillerrat & Hirsbrunner [78]	Code Clone	Java	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Xrefactory [111]	Separation of Concerns	C++	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
Unnamed [112]	Separation of Concerns	Ruby	N/A	Unknown	Unknown	Unknown	Unknown	Proof of Concept
RefactoringAnnotation [8]	Separation of Concerns	Java	Unknown	5	Azureus GanttProject JasperReports	Unknown	Unknown	Experiment
Abadi et al. [79]	Separation of Concerns	Java	N/A	Unknown	Java 1.4.2 libraries	Unknown	Unknown	Caee Study
Abadi et al. [77]	Separation of Concerns	Java	N/A	Unknown	Unknown	Unknown	Unknown	Caee Study
ReAF [75]	Separation of Concerns	Java	Unknown	1	Ant 1.8.1	Unknown	Unknown	Experiment
Sharma [76]	Separation of Concerns	C/C++	N/A	1	CppCheck	Unknown	Unknown	Proof of Concept
Unnamed [113]	Separation of Concerns	C#	Unknown	Unknown	Unknown	Unknown	Unknown	Proof of Concept
JExtract [80, 98]	Separation of Concerns	Java	Unknown	12	MyWebMarket JUnit 3.8 / 4.10 JHotDraw 5.2 Ant 1.8.2 ArgoUML 0.34 Checkstyle 5.6 FindBugs 1.3.9 FreeMind 0.9.0 JFreeChart 1.0.13 Quartz 1.8.3 Squirrel SQL 3.1.2 Tomcat 7.0.2	Unknown	[148]	Experiment
GEMS [89]	Separation of Concerns	Java	48	5	Wikidev SelfPlanner MyWebMarket JUnit JHotDraw	56 methods 25 methods 23 methods 12 methods 14 methods	Unknown	Experiment
Imazato et al. [100]	Separation of Concerns	Java		5	Ant ArgoUML JEdit JFreeChart	LOC: 260,624 / 1,532 methods LOC: 370,750 / 1,470 methods LOC: 187,166 / 1,066 methods LOC: 327,865 / 180 methods LOC: 166,149 / 980 methods	Unknown	Experiment
PostponableRefactoring [66]	Separation of Concerns	Java	N/A	Unknown	Mylyn	Unknown	Unknown	Proof of Concept
Nyamawe et al. [105], [106]	Separation of Concerns	Java	N/A	55	[175]	Unknown	[175]	Experiment
Krasniqi & Cleland-Huang [108]	Separation of Concerns	Java	N/A	4	Derby Drools Groovy Infinispan	KLOC: 170/ 2,382 commits KLOC: 371 / 840 commits KLOC: 141 / 4,892 commits KLOC: 299 / 2,349 commits	[176]	Experiment
Abid et al. [85]	Separation of Concerns	Java	8	30	[177]	Unknown	[177]	Experiment
Aniche et al. [84]	Separation of Concerns	Java	61	11,149	[178]	8.8 million commits	[178]	Experiment
Van der Leij et al. [14]	Separation of Concerns	Java	7	11,149	Previous work [84]	8.8 million commits	Dataset of [84]	Experiment
Sagar et al. [107]	Separation of Concerns	Java	60	800	Previous work [179]	748,001 commits	Dataset of [103]	Experiment
AlOmar et al. [103]	Separation of Concerns	Java	N/A	800	Previous work [179]	748,001 commits	[180]	Experiment
Nyamawe [104]	Separation of Concerns	Java	N/A	65	Previous works [105], [108], [181]	7,520 commits	Datasets of [105], [108], [181]	Experiment
Cui et al. [81]	Separation of Concerns	Java	N/A	Unknown	Previous works [6], [89]	Unknown	[182]	Experiment
REM [11]	Separation of Concerns	Rust	N/A	5	petgraph gitoxide kickoff sniffnet beerus	LOC: 20,157 LOC: 20,211 LOC: 1,502 LOC: 7,304 LOC: 302	[160]	Caee Study
Palit et al. [91]	Separation of Concerns	Java	61	410	Previous work [84]	55,268 commits	[183]	Experiment

to deal with long methods. If this tool is being tested against an *Extract Method* performed to remove duplicates, it is expected not to recommend any code changes. Therefore, performing experimentation with techniques that address different intents may not be adequate. In a similar context, building a universal model that extracts methods based on the history of code changes without understanding the intent must be human-verified to see whether it is useful.

Summary. Out of the 83 primary studies analyzed, almost 78% of the datasets are not publicly available. There is a lack of sharing datasets, which is detrimental to reproducing research. Primary studies have mostly employed small or medium-scale open-source applications, often developed using Java, typically containing less than 225,000 lines of code. These datasets are heterogeneous and do not contain the same type of information, making their standardization, for the purpose of benchmarking, difficult.

V. DISCUSSION AND OPEN ISSUES

To ensure that the *Extract Method* refactoring is properly identified/applied, we recommend retrofitting these tools with the following dimensions:

✦ **Provide context to guide developers on how to use *Extract Method* refactoring tools.** Based on the findings from RQ₁ and RQ₂, it becomes apparent that certain tools offer the context in which the *Extract Method* refactoring is being performed (e.g., JDeodorant, SEMI, AntiCopyPaster). The opportunities of applying this refactoring might be related to *Duplicate Code* removal, *Long Method* extraction, etc. However, other tools (e.g., ReAF, SDAR) lack the context in which the *Extract Method* is being performed. It is worth noting that without properly considering the context, the ground truth used to compare against existing studies might be biased. Also, the comparison against the state-of-the-art may not be appropriate unless these tools are called in the same context or intent as their original papers. For instance, JDeodorant applies the *Extract Method* refactoring to deal with long methods. If this tool is being tested against an *Extract Method* performed to remove duplicates, it is expected not to recommend any code changes. Therefore, performing experimentation against

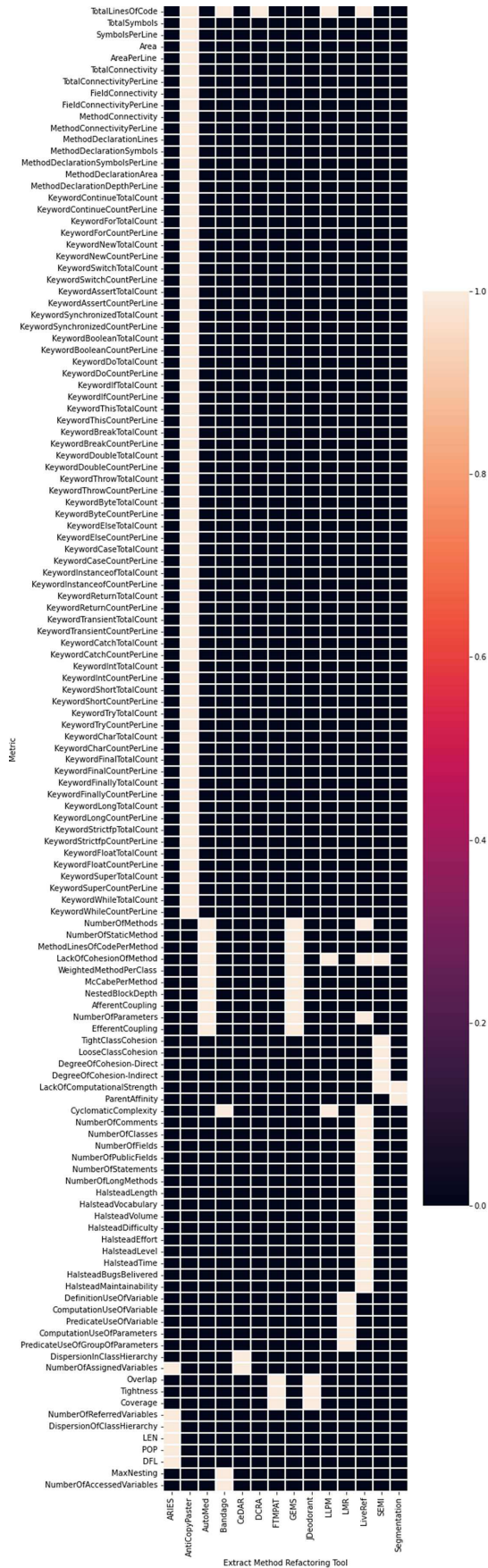


Fig. 9. Software metrics considered in the *Extract Method* refactoring tool.

techniques tackling different intents may not be adequate. In a similar context, building a universal model that extracts methods based on the history of code changes without understanding the intent must be human-verified to see whether it is useful.

✦ **Recommend appropriate naming for the method after the extraction.** Since the main purpose of the tools listed in Table IV is the recommendation of *Extract Method* refactoring, developers will ultimately need to provide a clear name for the extracted method, which is considered one of the most influential factors in the developer's decision on whether to perform *Extract Method* or not [141], [142]. The appropriate name assists in expressing its role and meaning to the extracted code. The existing approaches can complement their recommendation of the *Extract Method* with the naming recommendation of the extracted method.

✦ **Lack of clarity of how the approaches leverage metrics and decide the associated threshold to make the decision.** From Fig. 9, we observe different software quality metrics related to various quality attributes used by the tools. For instance, AntiCopyPaster has used 78 metrics related to size, complexity, coupling, and keywords to extract duplicate code. In contrast, LiveRef utilized around 20 metrics related to complexity, cohesion, and maintainability to identify the extraction targets of *Long Method* code smell. However, the implementation of these metrics may vary between these tools based on the context. In addition, there may be cases where different metric names are used to improve some quality attributes. This phenomenon might impact the interpretation of the correctness of the recommended tools.

✦ **Adapt *Extract Method* refactoring operations for multiple programming languages.** As reported in RQ₂, there are an existence of multiple *Extract Method* refactoring tools; however, RQ₁ and RQ₂ findings show that most of these tools are limited to supporting Java systems which narrow *Extract Method*-related research to Java systems. Hence, restricting research to a single language will not accurately reflect real-world scenarios [184]; there are opportunities for researchers to evolve the field further and increase the diversity of their research. The developers of non-Java systems gain no benefit without a tool to use in their development workflow. Furthermore, recent trends have shown a rise in the popularity of dynamically typed programming languages (e.g., Python), giving more urgency for the research community to construct tools that support non-traditional research languages.

✦ **Lack of benchmarks.** With the rise of refactoring mining tools [184], [185], [186], such tools were used to create datasets that already performed *Extract Method* refactorings from open-source software repositories. The collected refactorings became one of the main sources of already quantitative analysis for refactoring recommendation studies. For instance, the mined *Extract Method* refactorings were used either as an oracle to validate the correctness of recommendations [80], [98], [187], [188], or as training and testing sets for machine learning models and deep learning models [93], [97], [104]. While these tools have demonstrated high detection accuracy [189], they solely parse source code changes to identify refactoring patterns. So, there is no association between the performed refactoring and

the developer's rationale behind it. Even the reliance on the developer's documentation of the code change may not necessarily reveal the needed details behind the refactoring intent. Without such information, it becomes difficult to guess whether a mined *Extract Method* was performed to split a long method, segregate concerns from a complex method, or remove a clone. Therefore, studies using these data sets make assumptions concerning their intent, which may or may not hold. Any refactoring being performed outside of the paper's presumed context is noise that may hinder the data quality for training or validation. That is why it is essential to curate any collected refactorings by associating them with their proper context. Yet, the task of labeling refactorings' contexts may not be trivial.

✦ **Lack of clarity on potential *Extract Method* drawbacks.** All reviewed studies primarily focus on motivating the need for method extraction to improve readability, maintainability, and reusability. However, it is critical to raise the developer's awareness of the potential limitations inherited from the solutions' design or execution. One of the main design-level limitations of these approaches is the potential increase in the code's cognitive complexity. In fact, when a new method is extracted, it may introduce additional local variables and parameters. Such addition can adversarially hinder program comprehension and add a maintenance burden. Additionally, adding new method calls comes with additional overhead, such as method dispatch and return, which may reduce the program's performance, especially when the extracted code breaks tight loops [190]. Finally, depending on where the extracted method lives, it can change the scope or visibility of its variables or objects, leading to a violation of the behavior preservation property. While the benefits of the proposed refactorings may outweigh the drawbacks, studies should warn developers to avoid introducing regressions in their systems.

✦ **Integration of *Extract Method* tools into the developer workflow.** While our finding from RQ₂ shows that researchers proposed an approach to recommend *Extract Method* refactoring opportunities, not all approaches can be used in practice. Hence, the community needs to better collaborate with established tool/IDEs vendors in integrating their contributions with popular tools and IDEs to promote the usage of their artifacts. As for the existing tools, in addition to providing extensive and innovative refactoring functionality, researchers must ensure that their products exhibit an optimal user experience. Usability and trustworthiness are essential to refactoring tool adoption and are among the reasons for the limited usage [12], [191], [192], [193].

✦ ***Extract Method* refactoring support using Large Language Models (LLMs).** While *Extract Method* is considered as one of the most popular refactoring operations and represents approximately 49.6% of the total refactorings recommended [5], it is recognized as one of the most difficult and error-prone refactorings [6], [12], [32]. Even though we have shown in this systematic review multiple studies on *Extract Method* in the literature using multiple artificial intelligence (AI) techniques, its adoption is still challenging for developers [6], [12]. More recently, Large Language Models (LLMs) have made rapid advancements that have brought AI to a new level, enabling

and empowering even more diverse software engineering applications and industrial domains with intelligence [194], [195], [196], [197], [198]. Such LLMs are pre-trained on large corpora of data which enclose numerous commonsense knowledge and support Transformer architecture with millions, even billions of parameters. We believe that the *Extract Method* can benefit significantly from LLM advances. For instance, dedicated LLMs can be used to identify code fragments that need to be extracted and to recommend appropriate names for the extracted methods. LLMs can also automatically generate the documentation of *Extract Method* refactoring changes, e.g., generate the commit message or pull request description along with the intent behind the refactoring. It can also help with code review by explaining the intent of the *Extract Method* refactoring and providing a summary of the code change before and after the refactoring. We thus believe that LLMs represent a unique technique to empower *Extract Method* refactoring and open up various research venues in the field of *Extract Method* in particular and refactoring in general.

VI. THREATS TO VALIDITY

In this section, threats are discussed in the context of three types of threats of validity: internal validity, construct validity, and external validity.

Internal threats to validity: Obtaining a representative set of literature publications for this SLR can be considered a validity threat due to the search process. To minimize this threat, we followed the SLR guidelines [35], [36], [50], [51], [52]. In particular, we have carefully established search engines, search terms, and inclusion/exclusion criteria to ensure that the review of the literature is comprehensive. Additionally, we considered related search terms and the main terms of the research questions to construct the search string and select relevant articles. Furthermore, we followed a five-stage study selection process and applied each stage's inclusion and exclusion criteria described in Section III. Moreover, the analysis involved snowballing to expand the paper collection. These study design steps reduce the possibility that papers are missed. Another threat is the limitation of search terms and search engines, which might lead to incomplete literature publications. To limit this threat, we used carefully defined keywords and comprehensive academic search engines (i.e., ScienceDirect, Scopus, Springer, Web of Science, ACM, IEEE, and Wiley) that cover the main publishers' venues. We observed that when using search engines, particularly IEEE, some papers containing our keywords were not being found despite being indexed in their libraries. This issue has been reported in previous studies when using the IEEE search engine [199], [200]. However, we found these missed papers during the snowballing process. Regarding the quality of the selected PSs, only the studies that underwent peer review by leading academic publishers were included. Furthermore, selected studies that were within the search timeline were included. To our knowledge, all PSs relevant to our research goal and within the search window have been included.

Construct threats to validity: Concerning the subjectivity of the assessment of the PSs, the primary studies were reviewed

independently by two authors. The first author performed data analysis and extraction from the second author, who reviewed the currently selected PSs. At the end of each iteration, the authors met and performed any necessary refinements. In the event of disagreements, the researchers discussed these cases to reach a consensus. Furthermore, to avoid personal bias during manual analysis, two authors conducted each step in the manual analysis, and the results were always cross-validated. Moreover, some PSs do not make a clear distinction between how refactoring opportunities are detected, and how the refactoring is actually performed. Therefore, for these studies, we consider detection to refactoring opportunities to be part of the correction if the end goal of the PSs is *Extract Method* refactoring identification.

External threats to validity: The collected papers contain a significant proportion of academic works, forming an adequate basis for concluding findings that could be useful for academia. However, we cannot claim that the same *Extract Method* detection and execution is used in industry. Additionally, our findings are mainly within the field of software refactoring. We cannot generalize our results beyond this subject.

VII. CONCLUSION

In this paper, we map and review the body of knowledge on *Extract Method* refactoring opportunities. We systematically reviewed 83 papers and classified them. This research aims to aggregate, summarize and discuss the practical approaches that recommend *Extract Method* refactoring. Our main findings show that (i) 38.6% of *Extract Method* refactoring studies primarily focus on addressing code clones; (ii) Several of the *Extract Method* tools involve the developer in the decision-making process when applying the method extraction, and (iii) the existing benchmarks vary widely and lack uniform information, posing challenges in standardizing them for benchmarking purposes. This existing research empowers the community with information to guide future *Extract Method* tool development. Future work includes evaluation of each tool to determine the extent to which tools recommend *Extract Method* refactoring given the same context.

ACKNOWLEDGMENTS

The authors sincerely thank the anonymous reviewers for their invaluable feedback and constructive comments, which enhanced the quality and rigor of this work. Their thoughtful insights and suggestions have been instrumental in shaping the final version of this paper.

REFERENCES

- [1] GitHub. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/apache/pig/commit/7a516060213f5ac1fd559c124d2da0c0287757c7>
- [2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 2018.
- [3] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 2, no. 3, pp. 228–269, 1993.
- [4] A. V. Zarras, T. Vartziotis, and P. Vassiliadis, "Navigating through the archipelago of refactorings," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 922–925.
- [5] "JDeodorant." GitHub, 2011. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/tsantalis/jdeodorant>
- [6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA: ACM, 2016, pp. 858–870.
- [7] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proc. Conf. Center Adv. Stud. Collab. Res. (CASCON)*, 2013, pp. 132–146.
- [8] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 421–430.
- [9] S. Charalampidou, E.-M. Arvanitou, A. Ampatzoglou, P. Avgeriou, A. Chatzigeorgiou, and I. Stamelos, "Structural quality metrics as indicators of the long method bad smell: An empirical study," in *Proc. 44th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 234–238.
- [10] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE Trans. Softw. Eng.*, vol. 43, no. 10, pp. 954–974, Oct. 2017.
- [11] S. Thy, A. Costea, K. Gopinathan, and I. Sergey, "Adventure of a lifetime: Extract method refactoring for rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, pp. 658–685.
- [12] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan./Feb. 2012.
- [13] E. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? An exploratory study on how developers document their refactoring activities in commit messages," in *Proc. IEEE/ACM 3rd Int. Workshop Refactoring (IWor)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 51–58.
- [14] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, and M. Aniche, "Data-driven extract method recommendations: A study at ING," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 1337–1347.
- [15] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive clone recommendation system for extract method refactoring," in *Proc. IEEE/ACM 3rd Int. Workshop Refactoring (IWor)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 67–70.
- [16] J. P. S. Alcocer, A. S. Antezana, G. Santos, and A. Bergel, "Improving the success rate of applying the extract method refactoring," *Sci. Comput. Program.*, vol. 195, 2020, Art. no. 102475.
- [17] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution? An empirical study on open source software," in *Proc. Joint ERCIM Workshop Softw. Evolution (EVOL) Int. Workshop Princ. Softw. Evolution (IWPSSE)*, 2010, pp. 73–82.
- [18] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *J. Softw. Maintenance Evolution: Res. Pract.*, vol. 20, no. 6, pp. 435–461, 2008.
- [19] E. A. AlOmar, T. Wang, V. Raut, M. W. Mkaouer, C. Newman, and A. Ouni, "Refactoring for reuse: An empirical study," *Innov. Syst. Softw. Eng.*, vol. 18, pp. 1–31, Mar. 2022.
- [20] E. A. AlOmar et al., "How do developers refactor code to improve code reusability?" in *Proc. Reuse Emerg. Softw. Eng. Practices: 19th Int. Conf. Softw. Syst. Reuse ICSR 2020*, Hammamet, Tunisia. New York, NY, USA: Springer-Verlag, Dec. 2020, pp. 261–276.
- [21] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," in *Proc. 16th Asia-Pacific Softw. Eng. Conf.*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 43–49.
- [22] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software antipatterns," *J. Syst. Softw.*, vol. 128, pp. 236–251, Jun. 2017.
- [23] O. Tiwari and R. Joshi, "Identifying extract method refactorings," in *Proc. 15th Innov. Softw. Eng. Conf.*, 2022, pp. 1–11.
- [24] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.
- [25] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 101–110.

- [26] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, 2018.
- [27] W. Oizumi et al., "Recommending composite refactorings for smell removal: Heuristics and evaluation," in *Proc. XXXIV Brazilian Symp. Softw. Eng.*, 2020, pp. 72–81.
- [28] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshops*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 437–443.
- [29] M. Harman, "Refactoring as testability transformation," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshops*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 414–421.
- [30] A. Hora and R. Robbes, "Characteristics of method extractions in Java: A large scale empirical study," *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 1798–1833, May 2020.
- [31] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, Jul. 2014.
- [32] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: A large-scale survey of software refactoring," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 1303–1313.
- [33] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini, "Industry experiences with large-scale refactoring," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1544–1554.
- [34] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, "Refactoring inspection support for manual refactoring edits," *IEEE Trans. Softw. Eng.*, vol. 44, no. 4, pp. 365–383, Apr. 2018.
- [35] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Technical Report EBSE-2007-01, 2007.
- [36] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–10.
- [37] K. Petersen, R. Feldt, S. Muftaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proc. 12th Int. Conf. Eval. Assessment Softw. Eng. (EASE)*, 2008, pp. 1–10.
- [38] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: A review of current knowledge," *J. Softw. Maintenance Evolution*, vol. 23, pp. 179–202, Apr. 2011.
- [39] M. Abebe and C.-J. Yoo, "Trends, opportunities and challenges of software refactoring: A systematic literature review," vol. 8, pp. 299–318, Jan. 2014.
- [40] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 231–249, 2015.
- [41] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Eng. J.*, vol. 9, no. 4, pp. 2129–2151, 2017.
- [42] J. A. Dallal and A. Abidin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 44–69, Jan. 2018.
- [43] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Inf. Softw. Technol.*, vol. 83, pp. 14–34, Mar. 2017.
- [44] A. A. B. Baqaïs and M. Alshayeb, "Automatic software refactoring: A systematic literature review," *Softw. Qual. J.*, vol. 28, no. 2, pp. 459–502, 2020.
- [45] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhenec, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *J. Syst. Softw.*, vol. 167, 2020, Art. no. 110610.
- [46] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," 2020, *arXiv:2007.02194*.
- [47] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Inf. Softw. Technol.*, vol. 140, 2021, Art. no. 106675.
- [48] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. thesis, Univ. Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.
- [49] M. O. Cinnéide, *Automated Application of Design Patterns: A Refactoring Approach*. Dublin, Ireland: Trinity College, 2001.
- [50] B. Kitchenham, "Procedures for performing systematic reviews," Keele Univ., Keele, U.K., Tech. Rep. TR/SE-0401, 2004.
- [51] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *J. Syst. Softw.*, vol. 80, no. 4, pp. 571–583, 2007.
- [52] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering—A systematic literature review," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 7–15, 2009.
- [53] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, pp. 1–12.
- [54] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Inf. Softw. Technol.*, vol. 80, pp. 195–216, Dec. 2016.
- [55] S. Li et al., "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Inf. Softw. Technol.*, vol. 131, 2021, Art. no. 106449.
- [56] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Inf. Softw. Technol.*, vol. 50, nos. 9–10, pp. 833–859, 2008.
- [57] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2049–2075, 2013.
- [58] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *J. Syst. Softw.*, vol. 171, 2021, Art. no. 110827.
- [59] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 275–284.
- [60] E. A. AlOmar, M. Chouchen, M. W. Mkaouer, and A. Ouni, "Code review practices for refactoring changes: An empirical study on Open-Stack," in *Proc. 19th Int. Conf. Mining Softw. Repositories*, 2022, pp. 689–701.
- [61] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at Xerox," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.: Softw. Eng. Pract. (ICSE-SEIP)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 348–357.
- [62] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.
- [63] J. Pérez, C. López, N. Moha, and T. Mens, "A classification framework and survey for design smell management," 2011 [Online]. Available: <https://www.semanticscholar.org/paper/A-Classification-Framework-and-Survey-for-Design-Pérez-López/250b17c498a9262d08a6b749c6057ca1667c1aa/citing-papers>
- [64] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in *Proc. Symp. Softw. Reusability: Putting Softw. Reuse Context*, 2001, pp. 31–40.
- [65] A. S. Antezana, "TOAD: A tool for recommending auto-refactoring alternatives," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: Companion Proc. (ICSE-Companion)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 174–176.
- [66] K. Maruyama and S. Hayashi, "A tool supporting postponable refactoring," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 133–135.
- [67] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "JDeodorant: clone refactoring," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 613–616.
- [68] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 60–70.
- [69] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2015, pp. 392–402.
- [70] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *Proc. 13th Eur. Conf. Softw. Maintenance Reengineering*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 119–128.
- [71] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, 2011.

- [72] Y. Bian, G. Koru, X. Su, and P. Ma, "SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones," *J. Syst. Softw.*, vol. 86, no. 8, pp. 2077–2093, 2013.
- [73] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, "An automated extract method refactoring approach to correct the long method code smell," *J. Syst. Softw.*, vol. 187, 2022, Art. no. 111221.
- [74] E. Choi, D. Tanaka, N. Yoshida, K. Fujiwara, D. Port, and H. Iida, "An investigation of the relationship between extract method and change metrics: A case study of JEdit," in *Proc. 25th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 653–657.
- [75] T. Kanemitsu, Y. Higo, and S. Kusumoto, "A visualization method of program dependency graph for identifying extract method opportunity," in *Proc. 4th Workshop Refactoring Tools*, 2011, pp. 8–14.
- [76] T. Sharma, "Identifying extract-method refactoring candidates automatically," in *Proc. 5th Workshop Refactoring Tools*, 2012, pp. 50–53.
- [77] A. Abadi, R. Ettinger, and Y. Feldman, "Fine slicing for advanced method extraction," in *Proc. 3rd Workshop Refactoring Tools*, vol. 21, pp. 1–14, 2009.
- [78] N. Juillerat and B. Hirsbrunner, "Improving method extraction: A novel approach to data flow analysis using Boolean flags and expressions," in *Proc. 1st Workshop Refactoring Tools (WRT)*, 2007, pp. 48–49.
- [79] A. Abadi, R. Ettinger, and Y. A. Feldman, "Re-approaching the refactoring rubicon," in *Proc. 2nd Workshop Refactoring Tools*, 2008, pp. 1–4.
- [80] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 146–156.
- [81] D. Cui, Q. Wang, J. Wang, Chi, J. Li, L. Wang, and Q. Li, "REMS: Recommending extract method refactoring opportunities via multi-view representation of code property graph," in *Proc. 31st Int. Conf. Program Comprehension*, 2023, pp. 191–202.
- [82] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *Proc. 11th IEEE Int. Workshop Program Comprehension*, Piscataway, NJ, USA: IEEE Press, 2003, pp. 33–42.
- [83] F. Arcelli Fontana, M. Zanon, and F. Zanon, "A duplicated code refactoring advisor," in *Proc. 16th Int. Conf. Agile Processes Softw. Eng. Extreme Program. (XP)*, Helsinki, Finland, in Proceedings 16, New York, NY, USA: Springer-Verlag, 2015, pp. 3–14.
- [84] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1432–1450, 2020.
- [85] C. Abid, M. Kessentini, V. Alizadeh, M. Dhaouadi, and R. Kazman, "How does refactoring impact security when improving quality? a security-aware refactoring approach," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 864–878, 2020.
- [86] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, 2015.
- [87] R. Haas and B. Hummel, "Deriving extract method refactoring suggestions for long methods," in *Proc. Int. Conf. Softw. Qual.*, New York, NY, USA: Springer-Verlag, 2016, pp. 144–155.
- [88] R. Haas and B. Hummel, "Learning to rank extract method refactoring suggestions for long methods," in *Proc. 9th Int. Conf. Softw. Qual. Complexity Challenges Softw. Eng. Emerg. Technol. (SWQD)*, Vienna, Austria, in Proceedings 9, New York, NY, USA: Springer-Verlag, 2017, pp. 45–56.
- [89] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "GEMS: An extract method refactoring recommender," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 24–34.
- [90] R. Yue, Z. Gao, N. Meng, X. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 115–126.
- [91] I. Palit, G. Shetty, H. Arif, and T. Sharma, "Automatic refactoring candidate identification leveraging effective code representation," in *2023 IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, IEEE, 2023, pp. 369–374, 2023.
- [92] A. M. Sheneamer, "An automatic advisor for refactoring software clones based on machine learning," *IEEE Access*, vol. 8, pp. 124978–124988, 2020.
- [93] E. A. AlOmar et al., "Anticopypaster: Extracting code duplicates as soon as they are introduced in the IDE," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–4.
- [94] S. Fernandes, A. Aguiar, and A. Restivo, "LiveRef: A tool for live refactoring Java code," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–4.
- [95] A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions," *Inf. Softw. Technol.*, vol. 40, nos. 11–12, pp. 677–689, 1998.
- [96] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, 2012.
- [97] E. A. AlOmar et al., "Just-in-time code duplicates extraction," *Inf. Softw. Technol.*, vol. 18–58, 2023, Art. no. 107169.
- [98] D. Silva, R. Terra, and M. T. Valente, "JExtract: An eclipse plugin for recommending automated extract method refactorings," in *Proc. Brazilian Conf. Softw.: Theory Pract.*, 2015, pp. 1–8.
- [99] S. Fernandes, A. Aguiar, and A. Restivo, "A live environment to improve the refactoring experience," in *Companion Proc. 6th Int. Conf. Art. Sci., Eng. Program.*, 2022, pp. 30–37.
- [100] A. Imazato, Y. Higo, K. Hotta, and S. Kusumoto, "Finding extract method refactoring opportunities by analyzing development history," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2017, pp. 190–195.
- [101] M. Kaya and J. W. Fawcett, "Identification of extract method refactoring opportunities through analysis of variable declarations and uses," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 1, pp. 49–69, 2017.
- [102] M. Kaya and J. W. Fawcett, "Identifying extract method opportunities based on variable references (s)," in *Proc. 25th Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, 2013, pp. 153–158.
- [103] E. A. AlOmaret al., "On the documentation of refactoring types," *Automated Softw. Eng.*, vol. 29, no. 1, pp. 1–40, 2022.
- [104] A. S. Nyamawe, "Mining commit messages to enhance software refactorings recommendation: A machine learning approach," *Mach. Learn. Appl.*, vol. 9, 2022, Art. no. 100316.
- [105] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Softw. Eng.*, vol. 25, pp. 4315–4347, Sep. 2020.
- [106] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Automated recommendation of software refactorings based on feature requests," in *Proc. IEEE 27th Int. Requirements Eng. Conf. (RE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 187–198.
- [107] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, "Comparing commit messages and source code metrics for the prediction refactoring activities," *Algorithms*, vol. 14, no. 10, 2021, Art. no. 289.
- [108] R. Krasniqi and J. Cleland-Huang, "Enhancing source code refactoring detection with explanations from commit messages," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 512–516.
- [109] R. Ettinger, S. Tyszbrowicz, and S. Menaia, "Efficient method extraction for automatic elimination of type-3 clones," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 327–337.
- [110] R. Ettinger and S. Tyszbrowicz, "Duplication for the removal of duplication," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evolution, Reengineering (SANER)*, vol. 3, pp. 53–59, Piscataway, NJ, USA: IEEE Press, 2016.
- [111] M. Vittek, P. Borovansky, and P.-E. Moreau, "A C++ refactoring browser and method extraction," in *Software Engineering Techniques: Design for Quality*. New York, NY, USA: Springer-Verlag, 2007, pp. 325–336.
- [112] T. Corbat, L. Felber, M. Stocker, and P. Sommerlad, "Ruby refactoring plugin for eclipse," in *Proc. Companion 22nd ACM SIGPLAN Conf. Object-Oriented Program. Syst. Appl. Companion*, 2007, pp. 779–780.
- [113] P. M. Cousot, R. Cousot, F. Logozzo, and M. Barnett, "An abstract interpretation framework for refactoring with application to extract methods with contracts," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2012, pp. 213–232.
- [114] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Refactoring opportunity identification methodology for removing long method smells and improving code analyzability," *IEICE Trans. Inf. Syst.*, vol. 101, no. 7, pp. 1766–1779, 2018.
- [115] S. Xu, C. Guo, L. Liu, and J. Xu, "A log-linear probabilistic model for prioritizing extract method refactorings," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 2503–2507.

- [116] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, and K. Words, "ARIES: Refactoring support environment based on code clone analysis," in *Proc. IASTED Conf. Softw. Eng. Appl.*, 2004, pp. 222–229.
- [117] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "ARIES: Refactoring support tool for code clone," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–4, 2005.
- [118] A. O'Connor, M. Shonle, and W. Griswold, "Star diagram with automated refactorings for eclipse," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange*, 2005, pp. 16–20.
- [119] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis," in *Proc. Product Focused Softw. Process Improvement: 5th Int. Conf. (PROFES)*, Kansai Science City, in Proceedings 5. New York, NY, USA: Springer-Verlag, 2004, pp. 220–233.
- [120] Y. Bian, X. Su, and P. Ma, "Identifying accurate refactoring opportunities using metrics," in *Proc. Int. Conf. Soft Comput. Techn. Eng. Appl.: (ICSCTEA)*, Kunming, China. New York, NY, USA: Springer-Verlag, 2014, pp. 141–146.
- [121] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell," in *8th Elect. Eng./Electronics, Comput., Telecommun. Inf. Technol. (ECTIT) Assoc. Thailand-Conf.*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 492–495.
- [122] R. Ettinger and M. Verbaere, "Untangling: A slice extraction refactoring," in *Proc. 3rd Int. Conf. Aspect-Oriented Softw. Develop.*, 2004, pp. 93–101.
- [123] Z. Chen, M. Mohanavilasam, Y.-W. Kwon, and M. Song, "Tool support for managing clone refactorings to facilitate code review in evolving software," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2017, pp. 288–297.
- [124] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proc. 11th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 2015, pp. 1–10.
- [125] S. Vidal, I. Berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the refactoring of brain methods," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 27, no. 1, pp. 1–43, 2018.
- [126] G. P. Krishnan and N. Tsantalis, "Refactoring clones: An optimization problem," in *2013 IEEE Int. Conf. Softw. Maintenance*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 360–363.
- [127] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *Proc. Softw. Evolution Week-IEEE Conf. Softw. Maintenance, Reengineering, Reverse Eng. (CSMR-WCRE)*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 104–113.
- [128] W. Shin, "A study on the method of removing code duplication using code template," *Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, pp. 27–41, 2019.
- [129] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, "How to extract differences from similar programs? A cohesion metric approach," in *Proc. 7th Int. Workshop Softw. Clones (IWSC)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 23–29.
- [130] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proc. 5th Int. Workshop Softw. Clones*, 2011, pp. 7–13.
- [131] N. Juillerat and B. Hirsbrunner, "An algorithm for detecting and removing clones in java code," in *Proc. 3rd Workshop Softw. Evolution Transformations: Embracing Change, SeTra*, vol. 2006, 2006, pp. 63–74.
- [132] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proc. 27th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2000, pp. 155–169.
- [133] C. Brown and S. Thompson, "Clone detection and elimination for Haskell," in *Proc. ACM SIGPLAN Workshop Partial Eval. Program Manipulation*, 2010, pp. 111–120.
- [134] H. Li and S. Thompson, "Clone detection and removal for Erlang/OTP within a refactoring environment," in *Proc. ACM SIGPLAN Workshop Partial Eval. Program Manipulation*, 2009, pp. 169–178.
- [135] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Proc. 6th Work. Conf. Reverse Eng. (Cat. No. PRO0303)*, Piscataway, NJ, USA: IEEE Press, 1999, pp. 326–336.
- [136] S. Baars and A. Opreescu, "Towards automated refactoring of code clones in object-oriented programming languages," in *12th Seminar Adv. Tech. Tools Softw. Evol. (SATToSE 2019)*. [Online]. Available: <http://sattose.org/2019>
- [137] S. Thompson, *Haskell: The Craft of Functional Programming*. Reading, MA, USA: Addison-Wesley, 1999.
- [138] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Piscataway, NJ, USA: IEEE Press, 2008, pp. 172–181.
- [139] S. S. Skiena, *The Algorithm Design Manual*, vol. 2. New York, NY, USA: Springer-Verlag, 1998.
- [140] A. Daga, S. de Cesare, and M. Lycett, "Separation of concerns: Techniques, issues and implications," *J. Intell. Syst.*, vol. 15, nos. 1–4, pp. 153–176, 2006.
- [141] J. Yamanaka, Y. Hayase, and T. Amagasa, "Recommending extract method refactoring based on confidence of predicted method name," 2021, *arXiv:2108.11011*.
- [142] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation System Software Engineering*, Berlin, Germany: Springer, 2014, pp. 387–419.
- [143] T. Sharma and D. Spinellis, "A survey on software smells," *J. Syst. Softw.*, vol. 138, pp. 158–173, Apr. 2018.
- [144] "Xrefactory," [xref-tech.com](http://www.xref-tech.com), 2007. Accessed: Dec. 15, 2023. [Online]. Available: <http://www.xref-tech.com/sitemap/>
- [145] "Unnamed," GitHub, 2012. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/misto/Ruby-Refactoring>
- [146] "Wrangler," GitHub, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/RefactoringTools/wrangler>
- [147] "HaRe," GitHub, 2017. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/RefactoringTools/HaRe>
- [148] "JExtract," GitHub, 2015. Accessed: Dec. 15, 2023. [Online]. Available: <http://aserg-ufmg.github.io/jextract/>
- [149] "RASE," Virginia Tech, Blacksburg, VA, USA, 2015. Accessed: Dec. 15, 2023. [Online]. Available: <https://people.cs.vt.edu/nm8247/research.html>
- [150] "SEMI," Rijksuniversiteit Groningen, Groningen, The Netherlands, 2016. Accessed: Dec. 15, 2023. [Online]. Available: <http://www.cs.rug.nl/search/uploads/Resources/>
- [151] "GEMS," NUS Computing, 2017. Accessed: Dec. 15, 2023. [Online]. Available: <https://www.comp.nus.edu.sg/specmine/gems/>
- [152] "PostponableRefactoring," GitHub, 2017. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/katsuhisamaruyama/PostponableRefactoring>
- [153] "CREC," GitHub, 2018. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/soniapku/CREC>
- [154] "Unnamed," GitHub, 2019. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/noyosida/ProactiveCloneRecommendation>
- [155] "Clonerefactor," GitHub, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/simonbaars/clonerefactor>
- [156] "TOAD," GitHub, 2020. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/Aleli03/TOAD>
- [157] "Segmentation," Indian Institute of Technology Bombay, Mumbai, India, 2022. Accessed: Dec. 15, 2023. [Online]. Available: <https://www.cse.iitb.ac.in/omkarendra>
- [158] "LiveRef," GitHub, 2022. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/saracouto1318/LiveRef>
- [159] "AntiCopyPaster," GitHub, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/JetBrains-Research/anti-copy-paster>
- [160] "REM," Zenodo.org, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://zenodo.org/record/8124395>
- [161] M. Mohan and D. Greer, "Multirefactor: Automated refactoring to improve software quality," in *Proc. 18th Int. Conf. Product-Focused Softw. Process Improvement (PROFES)*, Innsbruck, Austria, in Proceedings 18, New York, NY, USA: Springer-Verlag, 2017, pp. 556–572.
- [162] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–11.
- [163] Y. Abgaz et al., "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4213–4242, Aug. 2023.
- [164] J. Fritzsche, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution (ICSME)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 481–490.
- [165] Rijksuniversiteit Groningen, 2016. Accessed: Dec. 15, 2023. [Online]. Available: <http://www.cs.rug.nl/search/uploads/Resources/TSEdataset>

- [166] Google, 2018. Accessed: Dec. 15, 2023. [Online]. Available: <https://goo.gl/SHi2UB>
- [167] GitHub, 2020. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/Aleli03/LinksToMethods>
- [168] E. Tempero et al., "The qualitas corpus: A curated collection of Java code for empirical studies," in *Proc. Asia Pacific Softw. Eng. Conf.*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 336–345.
- [169] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.
- [170] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.
- [171] "RASE-dataset," Virginia Tech, 2015. [Online]. Available: <https://people.cs.vt.edu/nm8247/projects/projectGroup-Rase.xml>
- [172] R. Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Softw. Qual. J.*, vol. 19, pp. 295–331, 2011.
- [173] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 207–216.
- [174] "AntiCopyPaster," zenodo.org, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://zenodo.org/record/7428835>
- [175] "Nyamawe," GitHub, 2020. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/nyamawe/FR-Refactor>
- [176] "Krasniqi," zenodo.org, 2020. Accessed: Dec. 15, 2023. [Online]. Available: <https://zenodo.org/record/3596397>
- [177] "Chima," 2022, [Online]. Available: <https://doi.org/10.7302/0bgn-vt27>
- [178] "Aniche," zenodo.org, 2022. Accessed: Dec. 15, 2023. [Online]. Available: <https://zenodo.org/record/3547639>
- [179] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Syst. Appl.*, vol. 167, 2021, Art. no. 114176.
- [180] Accessed: Dec. 15, 2023. [Online]. Available: [self-affirmed-refactoring/](https://smilevo.github.io/self-affirmed-refactoring/)
- [181] S. Rebai, M. Kessentini, V. Alizadeh, O. B. Sghaier, and R. Kazman, "Recommending refactorings via commit message analysis," *Inf. Softw. Technol.*, vol. 126, 2020, Art. no. 106332.
- [182] "REMS," Anonymous GitHub, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://anonymous.4open.science/r/REMS-A23C/README.md>
- [183] GitHub, 2023. Accessed: Dec. 15, 2023. [Online]. Available: <https://github.com/SMART-Dal/extract-method-identification>
- [184] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2786–2802, Dec. 2021.
- [185] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 1–10.
- [186] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 483–494.
- [187] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 331–336.
- [188] W. Mkaouer et al., "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.
- [189] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 930–950, Mar. 2022.
- [190] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 60–71.
- [191] A. M. Eilertsen and G. C. Murphy, "The usability (or not) of refactoring tools," in *Proc. IEEE Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 237–248.
- [192] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1106–1116.
- [193] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 233–243.
- [194] A. Fan et al., "Large language models for software engineering: Survey and open problems," 2023, *arXiv:2310.03533*.
- [195] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. Chi Conf. Human Factors Comput. Syst. Extended Abstr.*, 2022, pp. 1–7.
- [196] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2023.
- [197] W. Zhang, Y. Deng, B. Liu, S. J. Pan, and L. Bing, "Sentiment analysis in the era of large language models: A reality check," 2023, *arXiv:2305.15005*.
- [198] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," 2023, *arXiv:2303.07839*.
- [199] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of Java reflection-literature review and empirical study," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 507–518.
- [200] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM J. Comput. Cult. Heritage*, vol. 55, no. 13S, pp. 1–48, 2023.