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 source code. The *Extract Method* refactoring is considered as the “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 to collect, analyze and reveal 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 as well as their 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 future refactoring tool development.

Index Terms—extract method, refactoring, quality

1 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 systems[1]. 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]–[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]–[18], extraction of reusable methods [6], [19], [20], wrapping older method signatures [6], decomposition of long or complex structures [21]–[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 Figure 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, byte dataType)` and then replaced with calls to the newly extracted method.

- *EA. AlOmar is with the School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ, 07030 USA. E-mail: ealomar@stevens.edu*
- *MW. Mkaouer is with the College of Innovation and Technology, University of Michigan-Flint, Flint, MI 48502 USA. E-mail: mmkaouer@umich.edu*
- *A. Ouni is with the Department of Software Engineering and IT, ETS Montreal, University of Quebec, H3C 3P8 Montreal, QC, Canada. E-mail: ali.ouni@etsmtl.ca*

Manuscript received May 7, 2023.

1. Based on `JDeodorant` tool usage statistics: ”https://users.encs.concordia.ca/ nikolaos/”

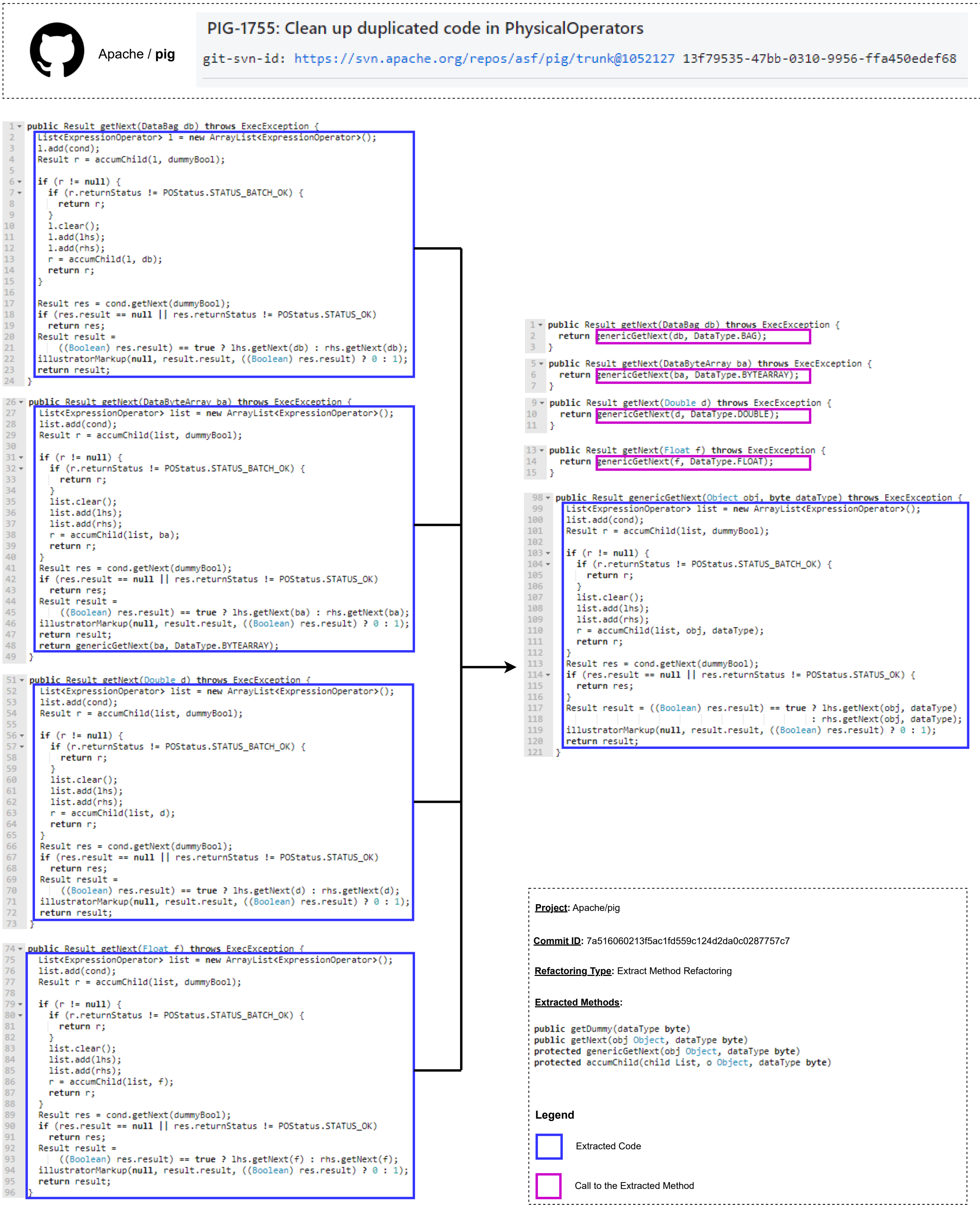

Fig. 1: Sample example of *Extract Method* refactoring [1].

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]–[18], improving testability through smaller test methods [28], [29], and segregating multiple functionalities [21]–[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]–[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]–[37]. The motivation behind each question is as follows.

- **RQ$_1$:** ***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.
- **RQ$_2$:** ***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.
- **RQ$_3$:** ***What are the datasets, and benchmarks used for evaluating and validating Extract Method recommendation tools?*** 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 studies[2].
- We provide practical implications of our findings for researchers, developers, tool builders, and educators.

The remainder of this paper is organized as follows: Section 2 reviews existing studies related to systematic reviews of refactoring. Section 3 outlines our empirical setup in terms of search strategy, study selection, and data extraction. Section 4 discusses our findings, while the research implications are discussed in Section 5. Section 6 captures threats to the validity of our work, before concluding with Section 7.

2 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 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

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

TABLE 1: Refactoring-related SLRs in related work.

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

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 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, 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 1 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 since 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.

3 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]–[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 3.1). The review phase encompasses the selection of primary studies, the

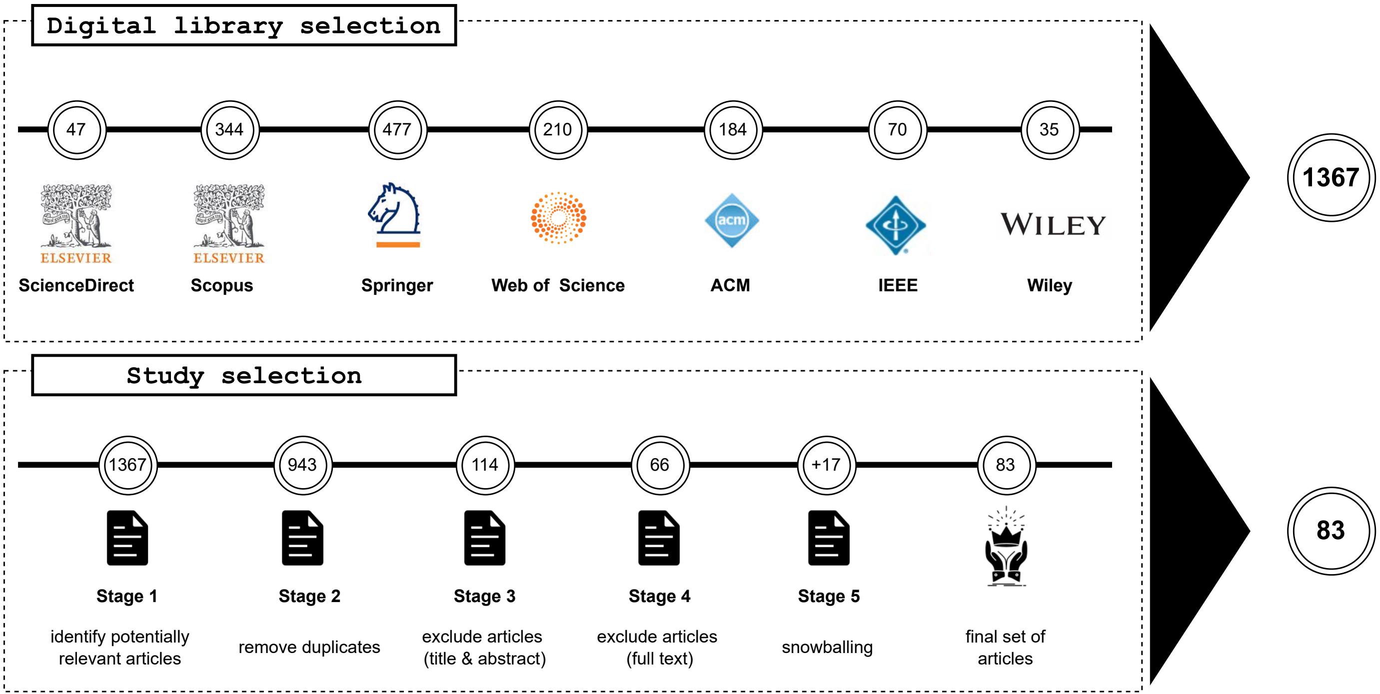

Fig. 2: Literature search process.

assessment of the study, data extraction, and data synthesis (described in Sections 3.2 and 3.4). Finally, the reporting phase emphasizes recording the review, which involves observing documents, and presenting the obtained results (described in Section 4).

3.1 Survey Planning

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

3.1.1 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.

3.1.2 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:

- **RQ$_1$:** ***What approaches were considered by the PSs to recommend Extract Method refactoring?***
- **RQ$_2$:** ***What are the main characteristics of Extract Method recommendation tools?***
- **RQ$_3$:** ***What are the datasets, and benchmarks used for evaluating and validating Extract Method recommendation tools?***

3.2 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.

3.2.1 Search strategy

Similar to Fernandes *et al.* [53], we performed an automatic search in seven electronic data sources to find relevant studies, including ScienceDirect[3], Scopus[4], Springer Link[5], Web of Science[6], ACM Digital Library[7], IEEE Xplore[8], and Wiley[9]. TextBox 1 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

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/

((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 separat* 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))

TextBox 1: Search string.

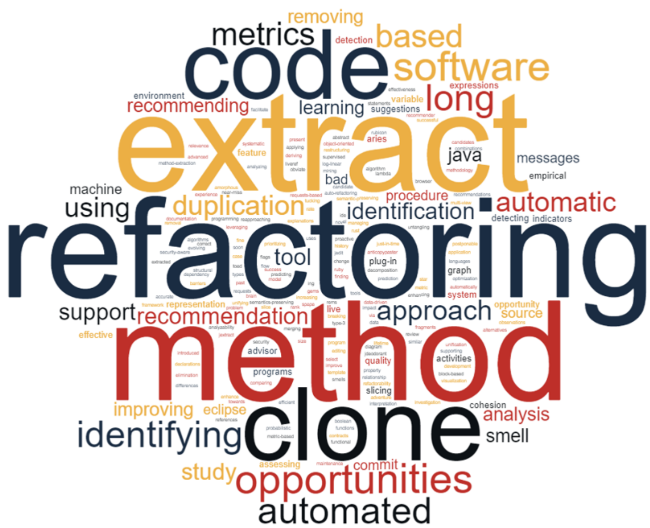

Fig. 3: Word cloud of paper titles of primary studies.

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 their 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 Figure 3.

3.2.2 Study selection

To collect the PSs, we adapted the search process of AlDallal and Abdin [42] [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.

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 Figure 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 the pool size increased to 83.

3.3 Study Quality Assessment

To assess the quality of PSs, we followed the guidelines proposed in [35], [55], [56]. We chose 3 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: 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.

3.4 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 from the other author 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 RQ_1 and RQ_2. We read the selected PSs to answer the research questions after extracting the classification dimensions. We then extracted the standard information from each article, similar to [57], [58], and included additional attributes relevant to our study in the data extraction form.

3.5 Final Primary Studies Selection

The research method discussed in Section 3 resulted in 83 relevant PSs. The main venues for these relevant PSs are presented in Table 2. 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 Figure 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 rising interest in this refactoring incites further research to improve its adoption in practice.

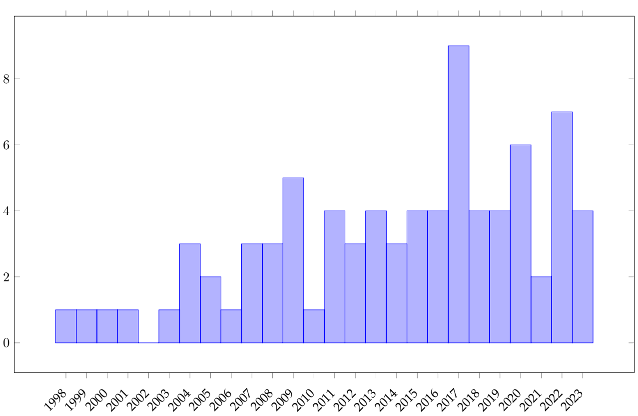

Fig. 4: Distribution of primary studies by year.

4 Results

This section reports and discusses the results of our study.

4.1 *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 3. Upon analyzing the PSs, we extract comprehensive high-level

TABLE 2: Publication venues.

Publication Venue	PSs
Symposium on Software Reusability	[64]
International Conference on Software Engineering	[8], [65]–[69]
Conference on Software Maintenance and Reengineering	[70]
Journal of Systems and Software	[71]–[73]
Asia-Pacific Software Engineering Conference	[21], [74]
Workshop on Refactoring Tools	[75]–[79]
International Conference on Program Comprehension	[80]–[82]
Agile Processes in Software Engineering and Extreme Programming	[83]
Transactions on Software Engineering	[10], [84]–[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]–[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]–[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]

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). Figure 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.

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]–[69], [72], [82], [86], [90], [92], [96], [109], [110], [116], [117], [119], [120], [123], [126]–[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 by specifying the clones with specific properties to 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 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 developed initially 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 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

TABLE 3: Related work in recommending the *Extract Method* refactoring opportunities.

Study	Year	Intent	Code Analysis	Code Representation	Detection	Execution	Validation Method
Lakhotia & 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 & 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 & Horwitz [82]	2003	Code Clone	Semantic	Graphs	Manual	Fully automated	Proof of Concept
Ettinger & 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
Juillerat & Hirsbrunner [131]	2006	Code Clone	Syntactic	AST	Fully automated	Fully automated	Proof of Concept
Juillerat & Hirsbrunner [78]	2007	Separation of Concerns	Syntactic	AST	Manual	Fully automated	Proof of Concept
Vittek *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 & 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 & Chatzigeorgiou [70]	2009	Long Method	Textual	Source Code	Fully automated	Suggest Alternatives	Experiment
Tsantalis & 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 & Thompson [134]	2009	Code Clone	Hybrids	AST & Tokens	Manual	Suggest Alternatives	Case Study
Brown & Thompson [133]	2010	Code Clone	Hybrids	AST & 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 & Gray [96]	2012	Code Clone	Syntactic	AST	Fully automated	Choose Candidates	Experiment
Kaya & 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 & Graphs	Manual	Fully automated	Experiment
Bian *et al.* [120]	2014	Code Clone	Syntactic	Metrics	Fully automated	Manual	Experiment
Krishnan & Tsantalis [126]	2013	Code Clone	Textual	Source Code	Fully automated	User Input	Experiment
Krishnan & Tsantalis [127]	2014	Code Clone	Hybrids	AST & Graphs	Fully automated	User Input	Experiment
Tsantalis *et al.* [86]	2015	Code Clone	Hybrids	AST & Source Code & Tokens	Fully automated	User Input	Experiment
Mazinanian *et al.* [67]	2016	Code Clone	Hybrids	AST & Source Code & Tokens	Fully automated	User Input	Experiment
Tsantalis *et al.* [68]	2017	Code Clone	Hybrids	AST & Source Code & 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 & 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 & Metrics	Fully automated	Fully automated	Case Study
Charalampidou *et al.* [9]	2018	Long Method	Syntactic	Metrics	Fully automated	Fully automated	Case Study
Haas & Hummel [87]	2016	Long Method	Hybrids	Source Code & Graphs	Manual	Suggest Alternatives	Experiment
Haas & Hummel [88]	2017	Long Method	Hybrids	Source Code & 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 & Fawcett [101]	2017	Long Method	Semantic	Graphs	Fully automated	Fully automated	Experiment
Maruyama & 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 & Tyszberowicz [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 & 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 & Tokens	Fully automated	Choose Candidates	Experiment
Shin [128]	2019	Code Clone	Syntactic	AST	Fully automated	Fully automated	Case Study
Barrs & Oprescu [136]	2019	Code Clone	Hybrids	AST & 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 & 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 & Graphs & 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 & 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 & Metrics	Fully automated	Fully automated	Experiment
Tiwari & 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

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 refactroings 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 which 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. **`AntiCopyPaster`** [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 source code quality will not decrease throughout a project's 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

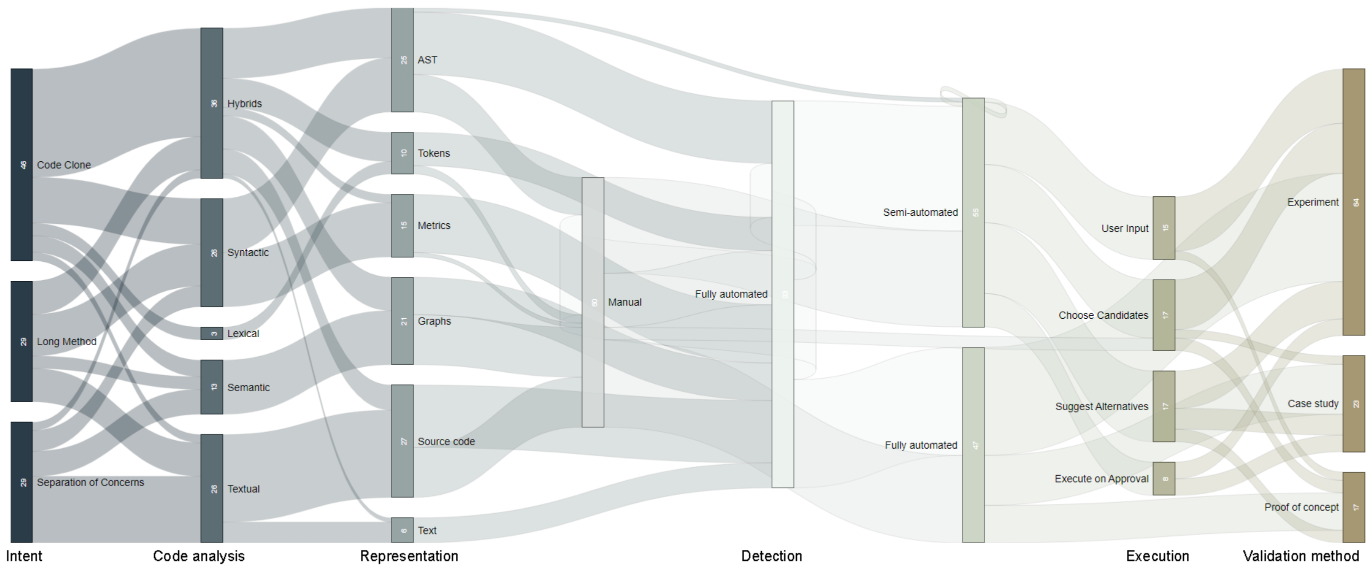

Fig. 5: The relationship among the intent, code analysis, representation, detection, execution, and validation method of the *Extract Method* refactoring.

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 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 statement number. Initially, `LMR` is applied to a Java application core package, showing that *Long Method* bad smell can be eliminated in 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 evinced high precision regarding 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]–[78], [80], [81], [84], [85], [89], [91], [98], [100], [103]–[108], [111], [111]–[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 a plug-in for Eclipse 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 enables 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.

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. **`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 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. **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*. Figure 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 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.

Figure 5 provides detailed mappings between our six dimensions. We can observe that *Code Clone* is the most pop-

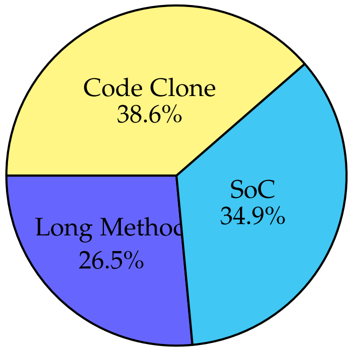

Fig. 6: Percentage of *Extract Method* studies, clustered by intent.

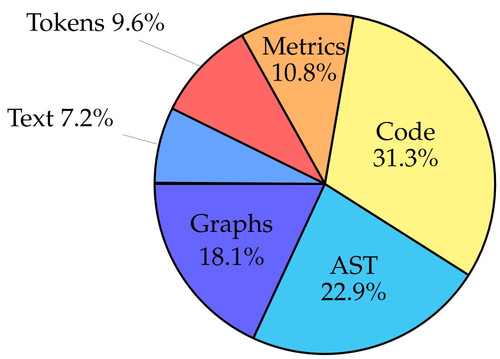

Fig. 7: Percentage of *Extract Method* studies, clustered by code representation types.

ular 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).*

4.2 *What are the main characteristics of Extract Method recommendation tools?*

To help select an appropriate *Extract Method* refactoring tool, we report in Table 4 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 4 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 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. Figure 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.

TABLE 4: 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

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 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.

Figure 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 5 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.*,

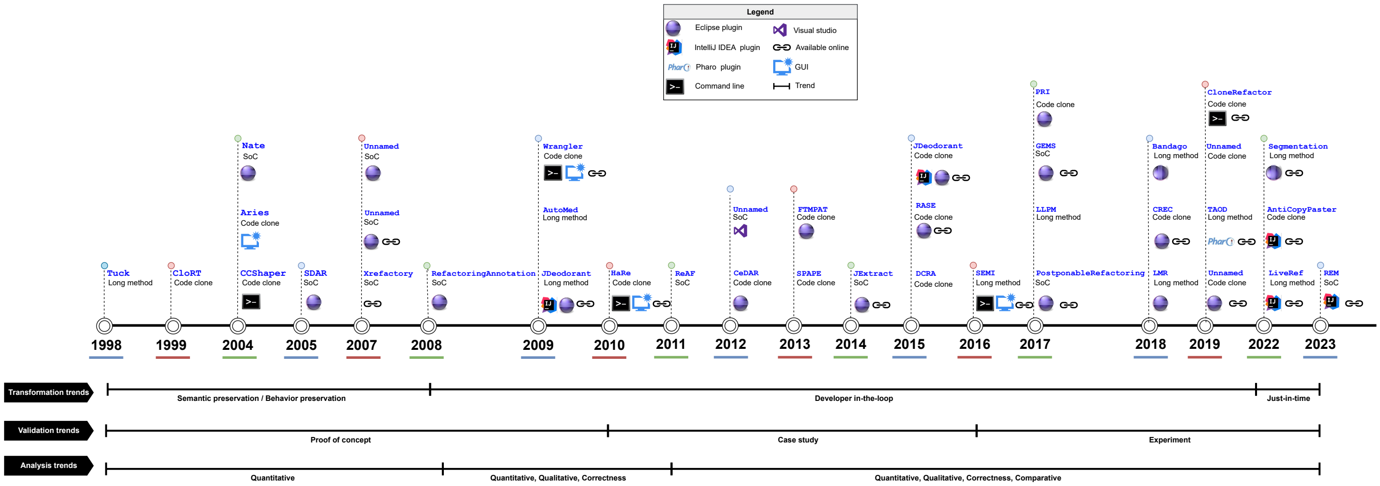

Fig. 8: Timeline of developing *Extract Method* refactoring tools.

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.*

4.3 *What are the datasets, and benchmarks used for evaluating and validating Extract Method recommendation tools?*

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 6, 7, and 8.

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 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 clone[10].

Validation Methods. Various structured evaluation approaches have been suggested, such as proof of concepts, case 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]–[16], [23], [65], [67]–[75], [80], [81], [83]–[94], [96]–[101], [101], [103]–[108], [115], [121], [126], [127], [134], [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]–[125], [128]–[130], [133], [137]. *Proof of concept* method was also adopted by 19.27% [64], [76], [78], [82], [95], [109]–[113], [118], [122], [131], [132], [135]. It is evident that experiment-based validation is becoming more popular. This is likely due to recent

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

TABLE 5: 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 & 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	Precision: 13.8% - 22.4%
			w/ JExtract	Recall: 57.1% - 92.8%
				F-measure: 22.23% - 36.09%
GEMS [89]	5 projects	w/ 4 authors	w/ JDeodorant	Precision: 13.3% - 25.3%
			w/ JExtract	Recall: 31.9% - 49.2%
			w/ SEMI	F-measure: 18.8% - 32.7%
PostponableRefactoring [66]	Unknown	No	No	Unknown
LLPM [115]	5 projects	No	w/ JDeodorant	Precision: 18.5% - 30.3%
			w/ JExtract	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	Precision: 22.81% - 38.75%
			w/ SEMI	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	Unknown
			w/ Visual Studio Rust Analyzer	

'$*$' indicates the tool is not peer-reviewed

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]–[10], [14], [15], [18], [21], [23], [64], [66]–[75], [77]–[81], [83]–[94], [96]–[100], [103]–[110], [114]–[118], [121]–[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 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 6, 7, and 8 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

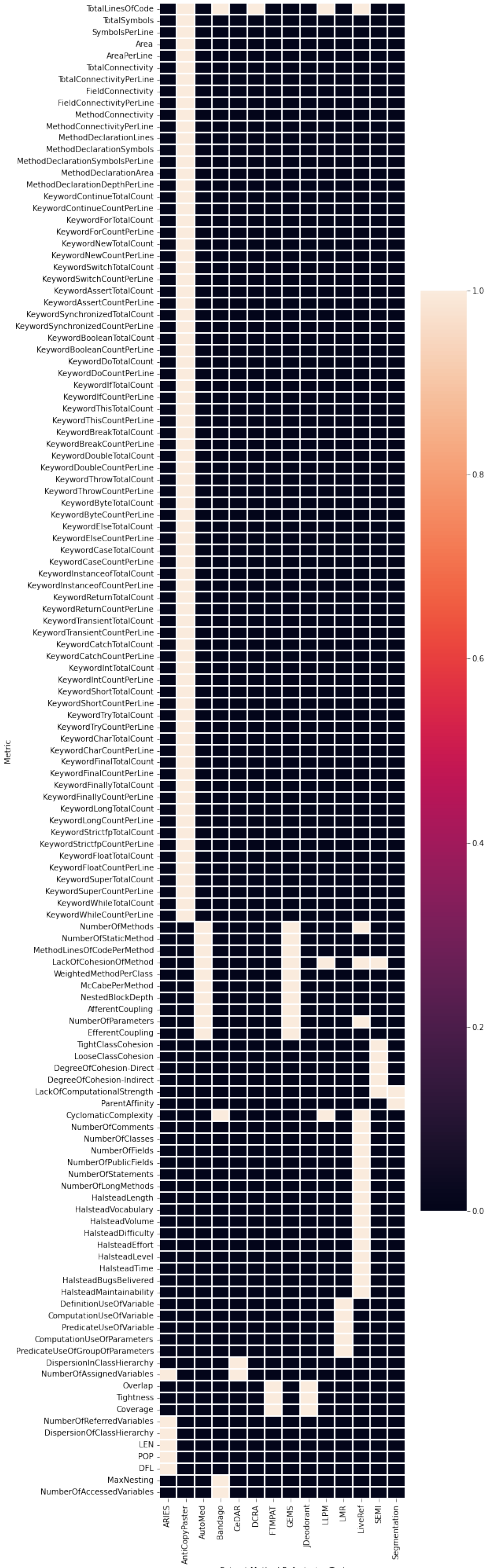

Fig. 9: Software metrics considered in the *Extract Method* refactoring tool.

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 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.*

5 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 RQ_1 and RQ_2, 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 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 4 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.

TABLE 6: 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	houtReader 1.8.0	LOC: 20,000 / 269 classes	Unknown	Caee Study
Meananeatra *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		
					Cayanne 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 [16], [65]	Long Method	Pharo	N/A	9	GitMultipileMatrix	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			

Lack of clarity of how the approaches leverage metrics and decide the associated threshold to make the decision. From Figure 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. Additionally, 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 RQ_2, there are an existence of multiple *Extract Method* refactoring tools; however, RQ_1 and RQ_2 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]–[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

TABLE 7: 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	LOC: 30,872	Unknown	Experiment
					Mnesia	LOC: 28,152		
					Yaws	LOC: 29,603		
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	KLOC: 67	Unknown	Experiment
					Columba 1.4	KLOC: 75		
					EMF 2.4.1	KLOC: 118		
					Hibernate 3.3.2	KLOC: 209		
					Jakarta-JMeter 2.3.2	KLOC: 54		
					JEdit 4.2	KLOC: 51		
					JFreeChart 1.0.10	KLOC: 76		
					JRuby 1.4.0	KLOC: 101		
					Squirrel-SQL 3.0.3	KLOC: 141		
FTMPAT [129]	Code Clone	Java	3	1	Ant 1.7.0	Unknown	Unknown	Caee Study
SPAPE [72]	Code Clone	Java	Unknown	10	Linux 2.6.6/kernel	LOC: 30,629	Unknown	Experiment
		Procedural			Unix/make 3.82	LOC: 33,864		
					httpd 2.2.2/server	LOC: 36,926		
					devecot 2.0.8/src/auth	LOC: 18,243		
					gstreamer 0.10.31/gst	LOC: 66,637		
					gtk 2.91.5/gdk/x11	LOC: 30,118		
					iptables 1.4.10/extensions	LOC: 19,668		
					nginx-0.8.15/src/core	LOC: 17,126		
					proftpd 1.3.3c/src	LOC: 34,404		
					PostgreSQL 9.0.2/src/backend/access	LOC: 65,046		
Bian *et al.* [120]	Code Clone	Java	Unknown	5	Linux 2.6.6/arch	Unknown	Unknown	Experiment
					Linux 2.6.6/net	Unknown		
					Linux 2.6.6/sound/drivers	Unknown		
					Unix/make 3.82	Unknown		
					http2.2.2/server	Unknown		
JDeodorant [67], [68], [86], [126], [127]	Code Clone	Java	N/A	9	Ant 1.7.0 / Ant 1.9	KLOC: 67	Unknown	Experiment
					Columba 1.4	KLOC: 75		
					EMF 2.4.1	KLOC: 118		
					JMeter 2.3.2 / JMeter 2.9	KLOC: 54		
					JEdit 4.2	KLOC: 51		
					JFreeChart 1.0.10 / JFreeChart 1.0.14	KLOC: 76		
					JRuby 1.4.0 / JRuby 1.7.3	KLOC: 101		
					Hibernate 3.3.2	KLOC: 209		
					SQuirreL SQL 3.0.3	KLOC: 141		
DCRA [83]	Code Clone	Java	1	50	Qualitas Corpus [168] (v. 20120401)	Unknnown	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	8,723 commits	[153]	Experiment
					Eclipse.jdt.core	22,358 commits		
					Elastic Search	14,766 commits		
					JFreeChart	3,603 commits		
					JRuby	24,434 commits		
					Lucene	22,061 commits		
PRI [123]	Code Clone	Java	N/A	6	AlgoUML	LOC: 127,145 / 1,559 files	Unknown	Caee Study
					Tomcat	LOC: 215,584 / 1,537 files		
					Log4j	LOC: 59,499 / 817 files		
					Eclipse AspectJ	LOC: 326,563 / 4,758 files		
					JEdit	LOC: 107,368 / 561 files		
					JRuby	LOC: 186,514 / 1,256 files		
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	KLOC: 260 / 990 classes	Unknown	Experiment
					JUnit	KLOC: 43 / 449 classes		
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
Sheneamer [92]	Code Clone	Java	N/A	6	Previous work [90]	[153]	Dataset of [153]	Experiment
				6	netbeans	200 paired clones	Unknown	
					eclipse-jdtcore	400 paired clones		
					EITC	426 paired clones		
					J2sdk1.4.0-javax	482 paired clones		
					eclipse-ant	522 paired clones		
					cocoon	655 paired clones		
AntiCopyPaster [93], [97]	Code Clone	Java	78	13	arthas	73,884 total commits	[174]	Experiment
					easyexcel			
					camel-quarkus			
					commons-lang			
					flink			
					iceberg			
					jena			
					pulsar			
					storm			
					apollo			
					JavaGuide			

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 extaction 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 RQ_2 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

TABLE 8: 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
Juillerat & 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	Unknown	Unknown	Experiment
					GanttProject			
					JasperReports			
					Java 1.4.2 libraries			
Abadi *et al.* [79]	Separation of Concerns	Java	N/A	Unknown	Unknown	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	Unknown	[148]	Experiment
					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			
GEMS [89]	Separation of Concerns	Java	48	5	Wikidev	56 methods	Unknown	Experiment
					SelfPlanner	25 methods		
					MyWebMarket	23 methods		
					JUnit	12 methods		
					JHotDraw	14 methods		
Imazato *et al.* [100]	Separation of Concerns	Java		5	Ant	LOC: 260,624 / 1,532 methods	Unknown	Experiment
					ArgoUML	LOC: 370,750 / 1,470 methods		
					JEdit	LOC: 187,166 / 1,066 methods		
					jFreeChart	LOC: 327,865 / 180 methods		
					Mylyn	LOC: 166,149 / 980 methods		
PostponableRefactoring [66]	Separation of Concerns	Java	N/A	Unknown	Unknown	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	KLOC: 170/ 2,382 commits	[176]	Experiment
					Drools	KLOC: 371 / 840 commits		
					Groovy	KLOC: 141 / 4,892 commits		
					Infinispan	KLOC: 299 / 2,349 commits		
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	LOC: 20,157	[160]	Caee Study
					gitoxide	LOC: 20,211		
					kickof	LOC: 1,502		
					sniffnet	LOC: 7,304		
					beerus	LOC: 302		
Palit *et al.* [91]	Separation of Concerns	Java	61	410	Previous work [84]	55,268 commits	[183]	Experiment

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]–[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]–[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.

6 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]–[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 3. 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. Additionally, 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.

7 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.

This research is partially by the National Science Foundation under Grant No. CNS-2213765.

References

[1] https://github.com/apache/pig/commit/7a516060213f5ac1fd559c124d2da0c0287757c7.

[2] M. Fowler, *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Transactions on Software Engineering and Methodology (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 *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 922–925, 2015.

[5] "Jdeodorant, https://github.com/tsantalis/jdeodorant," 2011.

[6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 858–870, ACM, 2016.

[7] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity.," in *CASCON*, pp. 132–146, 2013.

[8] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for Extract Method," in *Proceedings of the 30th international conference on Software engineering*, pp. 421–430, 2008.

[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 *2018 44th Euromicro Conference on software engineering and advanced applications (SEAA)*, pp. 234–238, IEEE, 2018.

[10] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying Extract Method refactoring opportunities based on functional relevance," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954–974, 2016.

[11] S. THY, A. COSTEA, K. GOPINATHAN, and I. SERGEY, "Adventure of a lifetime: Extract method refactoring for rust," *a) A*, vol. 15, p. 16.

[12] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[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 *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 51–58, IEEE, 2019.

[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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1337–1347, 2021.

[15] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive clone recommendation system for Extract Method refactoring," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 67–70, IEEE, 2019.

[16] J. P. S. Alcocer, A. S. Antezana, G. Santos, and A. Bergel, "Improving the success rate of applying the Extract Method refactoring," *Science of Computer Programming*, vol. 195, p. 102475, 2020.

[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 *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pp. 73–82, 2010.

[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," *Journal of Software Maintenance and Evolution: Research and Practice*, 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," *Innovations in Systems and Software Engineering*, pp. 1–31, 2022.

[20] E. A. AlOmar, P. T. Rodriguez, J. Bowman, T. Wang, B. Adepoju, K. Lopez, C. Newman, A. Ouni, and M. W. Mkaouer, "How do developers refactor code to improve code reusability?," in *Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19*, pp. 261–276, Springer, 2020.

[21] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," in *2009 16th Asia-Pacific Software Engineering Conference*, pp. 43–49, IEEE, 2009.

[22] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of systems and software*, vol. 128, pp. 236–251, 2017.

[23] O. Tiwari and R. Joshi, "Identifying Extract Method Rrefactorings," in *15th Innovations in Software Engineering Conference*, pp. 1–11, 2022.

[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 Software Engineering*, 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 *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 101–110, IEEE, 2014.

[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 Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[27] W. Oizumi, A. C. Bibiano, D. Cedrim, A. Oliveira, L. Sousa, A. Garcia, and D. Oliveira, "Recommending composite refactorings for smell removal: Heuristics and evaluation," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pp. 72–81, 2020.

[28] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 437–443, IEEE, 2011.

[29] M. Harman, "Refactoring as testability transformation," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 414–421, IEEE, 2011.

[30] A. Hora and R. Robbes, "Characteristics of method extractions in java: A large scale empirical study," *Empirical Software Engineering*, vol. 25, pp. 1798–1833, 2020.

[31] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 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 *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1303–1313, 2021.

[33] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini, "Industry experiences with large-scale refactoring," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1544–1554, 2022.

[34] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, "Refactoring inspection support for manual refactoring edits," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 365–383, 2017.

[35] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[36] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp. 1–10, 2014.

[37] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, pp. 1–10, 2008.

[38] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maint. Evol.*, 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, 01 2014.

[40] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, 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 Engineering Journal*, 2017.

[42] J. A. Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[43] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14 – 34, 2017.

[44] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, 2020.

[45] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[46] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig, "30 years of software refactoring research: a systematic literature review," *arXiv preprint arXiv:2007.02194*, 2020.

[47] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Information and Software Technology*, vol. 140, p. 106675, 2021.

[48] W. F. Opdyke, *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[49] M. O. Cinnéide, *Automated application of design patterns: a refactoring approach*. Trinity College Dublin, 2001.

[50] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26, 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," *Journal of systems and software*, 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," *Information and software technology*, 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 *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1–12, 2016.

[54] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Information and Software Technology*, vol. 80, pp. 195 – 216, 2016.

[55] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and software technology*, vol. 131, p. 106449, 2021.

[56] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and software technology*, vol. 50, no. 9-10, pp. 833–859, 2008.

[57] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Information and Software Technology*, 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," *Journal of Systems and Software*, vol. 171, p. 110827, 2021.

[59] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*, pp. 275–284, IEEE, 2011.

[60] E. A. AlOmar, M. Chouchen, M. W. Mkaouer, and A. Ouni, "Code review practices for refactoring changes: an empirical study on

OpenStack," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 689–701, 2022.

[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 *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 348–357, IEEE, 2021.

[62] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, 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," *Informe Técnico*, vol. 1, p. 2011, 2011.

[64] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, pp. 31–40, 2001.

[65] A. S. Antezana, "Toad: A tool for recommending auto-refactoring alternatives," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 174–176, IEEE, 2019.

[66] K. Maruyama and S. Hayashi, "A tool supporting postponable refactoring," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 133–135, IEEE, 2017.

[67] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: clone refactoring," in *Proceedings of the 38th international conference on software engineering companion*, pp. 613–616, 2016.

[68] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 60–70, IEEE, 2017.

[69] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 392–402, IEEE, 2015.

[70] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method refactoring opportunities," in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 119–128, IEEE, 2009.

[71] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, 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," *Journal of Systems and Software*, 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," *Journal of Systems and Software*, vol. 187, p. 111221, 2022.

[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 *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 653–657, IEEE, 2018.

[75] T. Kanemitsu, Y. Higo, and S. Kusumoto, "A visualization method of program dependency graph for identifying Extract Method opportunity," in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 8–14, 2011.

[76] T. Sharma, "Identifying extract-method refactoring candidates automatically," in *Proceedings of the Fifth Workshop on Refactoring Tools*, pp. 50–53, 2012.

[77] A. Abadi, R. Ettinger, and Y. Feldman, "Fine slicing for advanced method extraction," in *3rd workshop on refactoring tools*, vol. 21, 2009.

[78] N. Juillerat and B. Hirsbrunner, "Improving method extraction: A novel approach to data flow analysis using boolean flags and expressions.," in *WRT*, pp. 48–49, 2007.

[79] A. Abadi, R. Ettinger, and Y. A. Feldman, "Re-approaching the refactoring rubicon," in *Proceedings of the 2nd Workshop on Refactoring Tools*, pp. 1–4, 2008.

[80] D. Silva, R. Terra, and M. T. Valente, "Recommending automated Extract Method refactorings," in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 146–156, 2014.

[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 *Proceedings of the 31st International Conference on Program Comprehension*, 2023.

[82] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *11th IEEE International Workshop on Program Comprehension, 2003.*, pp. 33–42, IEEE, 2003.

[83] F. Arcelli Fontana, M. Zanoni, and F. Zanoni, "A duplicated code refactoring advisor," in *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings 16*, pp. 3–14, Springer, 2015.

[84] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Transactions on Software Engineering*, 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 Transactions on Software Engineering*, vol. 48, no. 3, pp. 864–878, 2020.

[86] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[87] R. Haas and B. Hummel, "Deriving Extract Method refactoring suggestions for long methods," in *International Conference on Software Quality*, pp. 144–155, Springer, 2016.

[88] R. Haas and B. Hummel, "Learning to rank extract method refactoring suggestions for long methods," in *Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies: 9th International Conference, SWQD 2017, Vienna, Austria, January 17-20, 2017, Proceedings 9*, pp. 45–56, Springer, 2017.

[89] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "GEMS: An Extract Method refactoring recommender," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 24–34, IEEE, 2017.

[90] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 115–126, IEEE, 2018.

[91] I. Palit, G. Shetty, H. Arif, and T. Sharma, "Automatic refactoring candidate identification leveraging effective code representation," 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, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Anticopypaster: Extracting code duplicates as soon as they are introduced in the ide," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–4, 2022.

[94] S. Fernandes, A. Aguiar, and A. Restivo, "Liveref: a tool for live refactoring java code," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–4, 2022.

[95] A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions," *Information and Software Technology*, vol. 40, no. 11-12, pp. 677–689, 1998.

[96] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Information and Software Technology*, vol. 54, no. 12, pp. 1297–1307, 2012.

[97] E. A. AlOmar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Just-in-time code duplicates extraction," *Information and Software Technology*, p. 107169, 2023.

[98] D. Silva, R. Terra, and M. T. Valente, "JExtract: An eclipse plug-in for recommending automated Extract Method refactorings," *In Brazilian Conference on Software: Theory and Practice*, 2015.

[99] S. Fernandes, A. Aguiar, and A. Restivo, "A live environment to improve the refactoring experience," in *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, pp. 30–37, 2022.

[100] A. Imazato, Y. Higo, K. Hotta, and S. Kusumoto, "Finding extract method refactoring opportunities by analyzing development history," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 190–195, IEEE, 2017.

[101] M. Kaya and J. W. Fawcett, "Identification of extract method refactoring opportunities through analysis of variable declarations and uses," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 01, pp. 49–69, 2017.

[102] M. Kaya and J. W. Fawcett, "Identifying extract method opportunities based on variable references (s).," in *SEKE*, pp. 153–158, 2013.

[103] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, and Z. Yu, "On the documentation of refactoring types," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–40, 2022.

[104] A. S. Nyamawe, "Mining commit messages to enhance software refactorings recommendation: A machine learning approach," *Machine Learning with Applications*, vol. 9, p. 100316, 2022.

[105] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Software Engineering*, vol. 25, pp. 4315–4347, 2020.

[106] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Automated recommendation of software refactorings based on feature requests," in *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pp. 187–198, IEEE, 2019.

[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, p. 289, 2021.

[108] R. Krasniqi and J. Cleland-Huang, "Enhancing source code refactoring detection with explanations from commit messages," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 512–516, IEEE, 2020.

[109] R. Ettinger, S. Tyszberowicz, and S. Menaia, "Efficient method extraction for automatic elimination of type-3 clones," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 327–337, IEEE, 2017.

[110] R. Ettinger and S. Tyszberowicz, "Duplication for the removal of duplication," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pp. 53–59, IEEE, 2016.

[111] M. Vittek, P. Borovansky, and P.-E. Moreau, "A c++ refactoring browser and method extraction," in *Software Engineering Techniques: Design for Quality*, pp. 325–336, Springer, 2007.

[112] T. Corbat, L. Felber, M. Stocker, and P. Sommerlad, "Ruby refactoring plug-in for eclipse," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 779–780, 2007.

[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 *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 213–232, 2012.

[114] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Refactoring opportunity identification methodology for removing long method smells and improving code analyzability," *IEICE TRANSACTIONS on Information and Systems*, 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 *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 2503–2507, IEEE, 2017.

[116] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words, "Aries: Refactoring support environment based on code clone analysis.," in *IASTED Conf. on Software Engineering and Applications*, pp. 222–229, 2004.

[117] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Aries: refactoring support tool for code clone," *ACM SIGSOFT Software Engineering 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 *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pp. 16–20, 2005.

[119] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis," in *Product Focused Software Process Improvement: 5th International Conference, PROFES 2004, Kansai Science City, Japan, April 5-8, 2004. Proceedings 5*, pp. 220–233, Springer, 2004.

[120] Y. Bian, X. Su, and P. Ma, "Identifying accurate refactoring opportunities using metrics," in *Proceedings of International Conference on Soft Computing Techniques and Engineering Application: ICSCTEA 2013, September 25-27, 2013, Kunming, China*, pp. 141–146, Springer, 2014.

[121] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell," in *The 8th Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand-Conference 2011*, pp. 492–495, IEEE, 2011.

[122] R. Ettinger and M. Verbaere, "Untangling: a slice extraction refactoring," in *Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 93–101, 2004.

[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 *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 288–297, IEEE, 2017.

[124] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 1–10, 2015.

[125] S. Vidal, I. Berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the refactoring of brain methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 1, pp. 1–43, 2018.

[126] G. P. Krishnan and N. Tsantalis, "Refactoring clones: An optimization problem," in *2013 IEEE International Conference on Software Maintenance*, pp. 360–363, IEEE, 2013.

[127] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 104–113, IEEE, 2014.

[128] W. Shin, "A study on the method of removing code duplication using code template," *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 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 *2013 7th International Workshop on Software Clones (IWSC)*, pp. 23–29, IEEE, 2013.

[130] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones*, pp. 7–13, 2011.

[131] N. Juillerat and B. Hirsbrunner, "An algorithm for detecting and removing clones in java code," in *Proceedings of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change, SeTra*, vol. 2006, pp. 63–74, 2006.

[132] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 155–169, 2000.

[133] C. Brown and S. Thompson, "Clone detection and elimination for haskell," in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 111–120, 2010.

[134] H. Li and S. Thompson, "Clone detection and removal for erlang/otp within a refactoring environment," in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 169–178, 2009.

[135] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of java software systems based on clone analysis," in *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pp. 326–336, IEEE, 1999.

[136] S. Baars and A. Oprescu, "Towards automated refactoring of code clones in object-oriented programming languages," tech. rep., 2019.

[137] S. Thompson, *Haskell: the craft of functional programming*. 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 *2008 16th iEEE international conference on program comprehension*, pp. 172–181, IEEE, 2008.

[139] S. S. Skiena, *The algorithm design manual*, vol. 2. Springer, 1998.

[140] A. Daga, S. de Cesare, and M. Lycett, "Separation of concerns: techniques, issues and implications," *Journal of Intelligent Systems*, vol. 15, no. 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," *arXiv preprint arXiv:2108.11011*, 2021.

[142] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," *Recommendation Systems in Software Engineering*, pp. 387–419, 2014.

[143] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[144] "Xrefactory,http://www.xref-tech.com/sitemap/," 2007.

[145] "Unnamed,https://github.com/misto/Ruby-Refactoring," 2012.

[146] "Wrangler, https://github.com/RefactoringTools/wrangler," 2023.

[147] "HaRe, https://github.com/RefactoringTools/HaRe," 2017.

[148] "JExtract, http://aserg-ufmg.github.io/jextract/," 2015.

[149] "RASE, https://people.cs.vt.edu/nm8247/research.html," 2015.

[150] "SEMI, http://www.cs.rug.nl/search/uploads/Resources/," 2016.

[151] "GEMS, https://www.comp.nus.edu.sg/ specmine/gems/," 2017.

[152] "PostponableRefactoring, https://github.com/katsuhisamaruyama/PostponableRefactoring," 2017.

[153] "CREC,https://github.com/soniapku/CREC," 2018.

[154] "Unnamed,https://github.com/noyosida/ProactiveCloneRecommendation," 2019.

[155] "Clonerefactor,https://github.com/simonbaars/clonerefactor,"

[156] "TOAD, https://github.com/Aleli03/TOAD," 2020.

[157] "Segmentation, https://www.cse.iitb.ac.in/ omkarendra/," 2022.

[158] "LiveRef, https://github.com/saracouto1318/LiveRef," 2022.

[159] "AntiCopyPaster,https://github.com/JetBrains-Research/anti-copy-paster," 2023.

[160] "REM,https://zenodo.org/record/8124395," 2023.

[161] M. Mohan and D. Greer, "Multirefactor: automated refactoring to improve software quality," in *Product-Focused Software Process Improvement: 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29–December 1, 2017, Proceedings 18*, pp. 556–572, Springer, 2017.

[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 *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, IEEE, 2019.

[163] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, 2023.

[164] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: intentions, strategies, and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481–490, IEEE, 2019.

[165] "http://www.cs.rug.nl/search/uploads/Resources/TSEdataset," 2016.

[166] "https://goo.gl/SHi2UB," 2018.

[167] "https://github.com/Aleli03/LinksToMethods," 2020.

[168] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia pacific software engineering conference*, pp. 336–345, IEEE, 2010.

[169] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 502–511, IEEE, 2013.

[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, https://people.cs.vt.edu/nm8247/projects/projectGroup-Rase.xml," 2015.

[172] R. Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Software Quality Journal*, vol. 19, pp. 295–331, 2011.

[173] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th working conference on mining software repositories (MSR)*, pp. 207–216, IEEE, 2013.

[174] "AntiCopyPaster,https://zenodo.org/record/7428835," 2023.

[175] "Nyamawe, https://github.com/nyamawe/FR-Refactor," 2020.

[176] "Krasniqi, https://zenodo.org/record/3596397," 2020.

[177] "Chima, https://doi.org/10.7302/0bgn-vt27," 2022.

[178] "Aniche, https://zenodo.org/record/3547639," 2022.

[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 Systems with Applications*, vol. 167, p. 114176, 2021.

[180] *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," *Information and Software Technology*, vol. 126, p. 106332, 2020.

[182] "REMS, https://anonymous.4open.science/r/REMS-A23C/README.md," 2023.

[183] "https://github.com/SMART-Dal/extract-method-identification," 2023.

[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 Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

[185] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, IEEE, 2010.

[186] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 483–494, IEEE, 2018.

[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 *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 331–336, 2014.

[188] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.

[189] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2020.

[190] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 60–71, 2014.

[191] A. M. Eilertsen and G. C. Murphy, "The usability (or not) of refactoring tools," in *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*, pp. 237–248, IEEE, 2021.

[192] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proceedings of the 36th international conference on software engineering*, pp. 1106–1116, 2014.

[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 *2012 34th international conference on software engineering (icse)*, pp. 233–243, IEEE, 2012.

[194] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[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 *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.

[196] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 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," *arXiv preprint arXiv:2305.15005*, 2023.

[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," *arXiv preprint arXiv:2303.07839*, 2023.

[199] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 507–518, IEEE, 2017.

[200] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Journal on Computing and Cultural Heritage*, 2023.