HistoryFinder: Constructing Method-Level Source Code Histories

Felix Grund
Department of Computer Science
University of British Columbia
Vancouver, Canada
fgrund@cs.ubc.ca

Shaiful Chowdhury
Department of Computer Science
University of British Columbia
Vancouver, Canada
shaifulc@cs.ubc.ca

Nick C. Bradley
Department of Computer Science
University of British Columbia
Vancouver, Canada
ncbrad@cs.ubc.ca

Braxton Hall
Department of Computer Science
University of British Columbia
Vancouver, Canada
braxtonh@cs.ubc.ca

Reid Holmes
Department of Computer Science
University of British Columbia
Vancouver, Canada
rtholmes@cs.ubc.ca

***Abstract*—Source code histories are a common way for developers and researchers to reason about how software evolves. Through a survey with 42 professional software developers, we learned that developers face significant mismatches between the output provided by developers' existing approaches for examining source code histories and what they need to successfully complete their historical analysis tasks. To address these shortcomings, we propose HistoryFinder, a tool for uncovering method histories that quickly produces complete and accurate change histories for ~90% methods outperforming leading tools from both research (e.g, FinerGit) and practice (e.g., IntelliJ / `git log`). HistoryFinder helps developers to navigate the entire history of source code methods so they can better understand how the method evolved. A field study on industrial code bases with 16 industrial developers confirmed our empirical findings of HistoryFinder's correctness, low runtime overheads, and additionally showed that the approach can be useful for a wide range of industrial development tasks.**

I. Introduction

Historical data embedded within version control systems contains a wealth of information that is useful to both developers and researchers. Source code histories are used by developers to understand how a particular unit of source code evolved [1], to provide context for code reviews [2], to share information among collocated teams [3], and for identifying experts [4]. Researchers use source code histories to understand developers' work habits [5], and to predict the likelihood and location of source code changes and defects [6], [7]. Version control systems (VCS) store a project's source code history by tracking developers' line-level changes to files. Unfortunately, these systems do not provide a complete understanding of the source code's evolution [8] primarily due to the frequent moving and renaming of files across the file system [9] and groups of lines being moved between files. Typically both researchers and developers are interested in accessing only a subset of a project's history, which is not well supported by VCS [10], [11], [12]. To address these information needs, tool support that is robust to the common development transformations and is able to generate accurate source code histories is needed.

Studies have focused on improving the accuracy and usability of source code history construction (e.g., [13], [14], [9], [15]); this is often referred as "history slicing" [11], [10]. These studies mainly differ in the granularity of the generated history. For example, in functional level granularity, one can extract all the relevant commits that are connected to a specific software feature [11]. Similarly, the history can be generated only for a given file of interest [9]. There are also scenarios when the research and the developer community desire lower level source code history [12], [15], [16]. Consequently, several studies aimed for line-level history [17], [18]; unfortunately, these suffered from high false positive and false negative rates due to many code lines can be similar just by chance [9], [19]. These observations support the need for method-level source code histories to balance between being too coarse (e.g., file-level) and too fine (e.g., line-level) granularity. Unfortunately, only a few approaches specialize in building method-level histories [15], [14], [20].

Previous history construction approaches, including the recent FinerGit [20], require preprocessing the entire project history before making any queries. This up-front cost hinders a tool's usability [21]. Historical tracing tools that are commonly used in practice do not require pre-processing; these include IntelliJ's git history feature and `git log -L`, which generate code history *on demand*. Unfortunately, these tools are not resilient to common code transformations present during software development and produce inaccurate method history (Section V). We surveyed 30 industrial engineers and 12 academic developers to gain further insight into method-level source code histories to learn what questions they are trying to answer with these data and what shortcomings they experience with existing approaches. Ultimately, these participants indicated that they wanted up-to-date results without lengthly preprocessing. They also reported that inaccuracies introduced by

source code transformations inhibit existing tools causing the tools to frequently return incomplete histories.

To address the shortcomings, we propose *HistoryFinder*,[1] a tool for surfacing complete histories of source code methods. HistoryFinder builds method histories *on demand*, as desired by a developer or researcher, thus requires no pre-processing or whole-program analyses. The similarity algorithm used by the approach surfaces all changes to a source code method along with a categorization of how the method was changed. The HistoryFinder similarity algorithm is robust in the face of common filesystem and source code transformations that occur during software development.

We evaluated HistoryFinder's accuracy and runtime performance using a manually constructed oracle from 20 popular open-source project repositories and compared its accuracy to both the state-of-the-art (FinerGit) and sate-of-the-practice tools (IntelliJ and `git log -L`). We also conducted an industrial field study to verify that HistoryFinder also generates accurate histories for industrial systems. In both cases, HistoryFinder correctly determined the complete history of ~90% of the evaluated methods with a median runtime of ~2 seconds. The primary contributions of this paper include:

- A survey with 42 professional developers demonstrating a lack of tool support for the most frequently performed historical understanding tasks.
- The open source implementation of HistoryFinder, a novel approach for extracting method-level source code histories which can be used interactively through a developer-facing *web service* or a research-oriented *command line* client.[2]
- A quantitative analysis of HistoryFinder's accuracy and runtime performance using 20 popular open source projects. We also verified HistoryFinder's accuracy and runtime with 16 industrial developers.
- A manually derived history oracle for 200 methods (required ~100 hours of manual work), to facilitate future research on source code history construction algorithms.

II. Background & Related Work

Source code histories have long been recognized as a key information source for program understanding and for capturing change rationale (e.g., [22], [23], [24], [25], [26], [27], [28]). Several approaches have been proposed to help developers and researchers better leverage source code histories. We examined these approaches according to three requirements important to both industrial developers and researchers: speed, granularity, and robustness; Table I provides an overview of many of these.

Analysis burden. Many approaches require a complete project to be analyzed before any queries can be issued. These *offline* analyses can usually be queried efficiently once a history is created, but can require hours of preprocessing before they can return results. While it is possible to compute results incrementally, many tools do not support this; these tools are best geared towards mining-style analyses rather than answering developer queries. For example, Historage [14], and FinerGit [20] (an improvement over Historage) preprocess a repository to place each method in its own file; they then use Git's history mechanism to track changes on each individual method's corresponding file [14]. Sunghun *et al.* [15] proposed a function matching algorithm for the `C` language. The algorithm considers metrics including the number of incoming calls (fan-in), which require preprocessing the complete repository. This is also true for Beagle [29], APFEL [30], and C-Rex[13]. Unfortunately, preprocessing the entire repository for each change can cause high feedback-latency, discouraging developers in adopting a particular tool [21]. Recently, Li *et al.* proposed CSLICER [11] for extracting source code history; this approach requires existing test sets for conducting dynamic analysis. Tools commonly used in practice like `git-log -L` and IntelliJ do not require any up-front analysis, making them more practical for answering developer queries on-demand without any prior configuration or analysis.

Granularity. The granularity at which a history can be generated can be a key factor for the utility of a given tool [31]. Method-level granularity is widely accepted in different areas such as bug localization [32], [33], [34], and software energy estimation [35], [36]. Our survey with professional developers reveals that method-level granularity is also desired for source code history generation. The importance of extracting previous method level changes for predicting future change patterns has been mentioned in the research community [16]. Different approaches provide histories at different granularities. CSLICER [11] extracts a minimal changeset that completely isolates a feature. Such changeset may contain information from multiple files. By default, version control systems operate on lines within files, but provide incomplete history because of file movements and renaming. Daniela *et al.* [9] address this problem using an incremental origin analysis approach. By focusing on the text itself, these approaches are language-agnostic but are unable to answer interesting queries like "find all changes to this class". Tools which support queries on code elements, rather than lines, support various levels of queries, for instance to classes (e.g., Beagle [29]), methods (e.g., method_log[3]) or blocks (e.g., APFEL [30]).

Granularities also vary in terms of time: while most tools in Table I try to find complete histories, pry-git[4], and Beagle only analyze changes between two specific versions of a program or file and do not try to uncover the complete history. This is also true for recent approaches like ClDiff [37], and GumTree [38].

Transformations. "The one constant in software is *change*". This makes histories important, but many changes can be challenging to track. Changes can range from simple single-line code edits to complex refactorings that involve renaming methods and moving them to new files. Refactoring is described as the "bread and butter" of software restructuring [39] and refactorings happen remarkably frequently during development. For example, 80% of the changes to APIs are

[1]NOTE: HistoryFinder is a code name for double-blind review.

[2]An anonymized version of the online service with no logging can be found at https://tinyurl.com/tutkm4o (VM may take a moment to start).

[3]https://github.com/freerange/method_log

[4]https://github.com/pry/pry-git

TABLE I

SELECTION OF TOOLS FOR EXAMINING SOURCE CODE HISTORIES. THESE VARY IN WHETHER THEY ARE *on demand* OR REQUIRE PRE-PROCESSING A WHOLE PROJECT, THE GRANULARITY THAT CAN BE ANALYZED (*code* MEANS A SUBSET OF CLASS, METHOD, OR STATEMENT), AND THEIR TOLERANCE TO COMMON SOURCE CODE TRANSFORMATIONS (*M-* REFERS TO METHOD, *F-* REFERS TO FILES, ≈ REFERS TO PARTIAL OR WEAK SUPPORT, AND *M-Move* DENOTES PULL-UP METHOD, AND PUSH-DOWN METHOD).

Approach	On-Demand?	Granularity	Code Transformations	
			Intra-File	Inter-File
APFEL	NO	Code	✗	✗
Beagle	NO	Code	≈	≈
Historage	NO	Code	M-Rename	F-Rename
C-Rex	NO	Code	M-Rename	✗
pry-git	YES	Code	≈	✗
method_log	YES	Code	≈	✗
git-log -L	YES	Text	≈	✗
IntelliJ	YES	Text	≈	F-Rename
FinerGit	NO	Code	M-Rename M-Signature	F-Rename M-Move
HistoryFinder	YES	Code	M-Rename M-Signature	F-Rename M-Move

refactorings [40] and 19% of the method introductions in the PostgreSQL source code were caused by refactorings [22]. Approachs like method_log (designed for Ruby methods) can detect transformations within a file (intra-file changes), as long as enough textual similarity is maintained through the transformation. Some code-based analyses are able to further categorize the changes: Historage [14] and C-Rex [13] can identify method rename refactorings. While dedicated refactoring detection tools exist (e.g., [41], [42], [43], [44]), most history tracking tools, cannot track inter-file transformations, except for Historage/FinnerGit and IntelliJ, which are robust in file rename events, but cannot track other inter-file transformations (e.g., extract-method refactoring). Unfortunately, such refactorings are prevalent in practice [23], [27].

III. INDUSTRIAL SURVEY

Many of the existing approaches we identified in the literature (Section II) were geared at the research community and did not fully consider the needs of industry developers. Prior work by Codoban *et al.* [12] found that developer-facing history tools (e.g., `git log`) are not ideal: developers complained about information overload and wanted more structured and selective information. To verify these findings and to gain additional insight into how and why developers use software histories, we conducted a survey with 42 participants. We examined the following two research questions:

RQ1 Do developers use source code histories, and if so, at what granularity?

RQ2 What mechanisms do developers use for generating histories and what shortcomings do they have?

Survey Design. The survey was administered online and consisted of 18 Likert-scale and free-response questions along with two code-oriented scenarios. Each survey took ∼20 minutes to complete. The complete survey and anonymized responses are available.[5]

[5]URL removed for double blind submission

Survey Participants. We recruited 30 professional developers from industry and 12 from academia (total 42 participants). Participants were contacted via email from the authors' professional networks; 87 individuals were solicited giving a final response rate of 48%. The majority (64%) of job titles were *software developer/engineer* or similar; all academic participants were upper-level graduate students or faculty. Across all participants, 90% had more than 4 years of programming experience and 80% had used source code history for four years or more. For the 30 professional developers, 63% had been employed in industry for four years or more.

A. RQ1: Do developers use histories?

Survey questions: (1) How recently did you last use source code history of any kind? (2) What were you looking for? (3) In terms of source code granularity, how interested are you in gathering information on source code history at the following levels? (4) When you use code history, how far in the past do you usually examine?

The majority of the survey participants frequently use source code history: 76% had used code history within two days prior to performing the survey (90% within a week). Participants use source code history for a variety of activities including version control (e.g., to "*check what I modified*"), to check change accountability (e.g., to determine "*who had been contributing*", "*who [they] could contact for dev support*", and "*who is associated with [a specific] change*"), and for program understanding (e.g., to "*understand how the solution to a certain problem was implemented*" and "*how and why [a property] was changed*").

90% of participants responded positively to using *Method-/Function* granularities, 79% responded positively about *File* granularity while 76% responded positively about using histories at *Block* granularity. This suggests developers are interested in examining histories at source code granularities other than just the file or textual range (block) level as is supported by most tools.

In terms of duration, while a few participants only used recent commits, most (67%) expressed that they would go back as far as necessary (even years) to find the changes they were looking for. For example, one participant mentioned that "*sometimes I need to trace back the lifespan of a class until it was created (which might get tricky if it was renamed).*" According to another participant, "*It'll be great to have the complete history available all the time.*"

B. RQ2: How do developers generate histories?

We asked participants to review a pull request from the *Checkstyle* project that involved reasoning about a method in a file that had changed 47 times over three years.

Survey questions: (1) Is this pull request scenario familiar to you? (2) How would you identify the commits in which the method of interest has changed? (3) How well do existing tools support identifying these changes? (4) How hard would it be to find the first commit for the given method? (5) How useful would it be to have support for a more semantic history in this scenario?

85% of the participants are either *Very familiar* or *familiar* with the pull request scenario where they need to inspect code history. Developers generally extract code history with their preferred tools (either using the tooling within the IDE or in the shell). Participants mentioned several tools that they used to do this, with `git log` and IntelliJ's history feature among the most popular. 56% of the participants, however, responded that the existing tools do not support these tasks well while 27% responded neutrally. Overall, 79% participants stated that it is *Hard* or *Very hard* to find the commit that actually introduced a method (i.e., to extract the method's complete history). In particular, 67% participants believe their approaches are suited *Not very well* or *Not well at all* to deal with complex structural changes such as method move. The majority of the participants (91%) stated that it would be *Very helpful* or *Helpful* to have a tool that is robust to structural changes and can generate complete and accurate method level history. These results align well with the prior study by Codoban *et al.* [12] that showed that developers need enhanced support for eliciting source code histories.

Summary: Developers frequently use source code histories. Existing tools are inadequate for extracting history at the most desired levels of granularity.

IV. HistoryFinder: Surfacing Method Histories

Motivated by the drawbacks of previous approaches (Section II) and feedback from the developer survey (Section III), we now describe HistoryFinder, a tool for quickly constructing accurate source code histories. HistoryFinder has been explicitly designed to robustly identify and track changes in the face of common code transformations. It generates histories at the granularity of individual methods; class-level histories can be constructed by aggregating all method-level histories in a class. To ensure HistoryFinder results are always up-to-date, and to minimize unnecessary overhead, histories are computed *on demand* with no pre-processing. To allow developers to explore the full history of a method, HistoryFinder searches backwards through time to identify all relevant commits until it finds the method's introducing commit. HistoryFinder can be used as a command line tool and as a web service.[6] Its source code, including scripts that can install, build, and run HistoryFinder with a single command, is available.[7]

Figure 1 provides a high-level illustration of HistoryFinder's heuristic approach. To build a history, HistoryFinder starts with the most recent commit for a method and iteratively steps back through past commits in the version control repository to find other commits that also modified the method. This process continues until the introducing commit is found. This entails two main tasks: First, all of the commits that modified the method need to be found among the commits in the repository (Figure 1, left rectangle). Second, the changes to the method need to be analyzed to determine how the method was changed; this information can help developers find specific changes of interest (Figure 1, right rectangle).

Inputs. The inputs HistoryFinder requires are readily available to the developer: a repository identifier (e.g., a `git clone` URL), the path of the file containing the method, the method name, and the line number[8] of the method declaration. The starting commit SHA for building the history can be provided, but `HEAD` is used by default.

Outputs. To provide presentation flexibility, HistoryFinder emits a JSON object containing a list of commits that modified the specified method and relevant metadata. The web service and command line clients render the JSON output object to increase usability.

A. Method Matching

At the core of HistoryFinder's method finding procedure is a similarity algorithm for matching methods across file versions. Our selection of the matching algorithm is driven by two factors: First, the algorithm must be *on demand*; we can not use complex method features (e.g., whole program call graph) that require processing a complete repository, making the algorithm non-performant. Second, developers want the full method history. This is also true for the MSR community who are interested in source code origin analysis [22], [9]. As we show later, some methods have years long history and may have been modified ~50 times. To locate a method even in one single commit, we sometimes need to parse many other files (because the method moved) and compare with all the methods in those files, which negatively impacts the runtime. These observations discouraged us from using complex strategies like AST matching techniques [45], [46], [47], [48], [38].

Our matching algorithm relies on techniques from clone detection (e.g., [49], [50], [51], [52], [53], [54]). Textual similarity is an efficient strategy for clone detection but lacks accuracy in many cases [49], [51], [52]. One approach for mitigating this problem without significantly sacrificing runtime efficiency is to compare different source code metrics [50], [55]. Therefore, HistoryFinder measures similarity between two methods by comparing their body similarity and signature similarity; it also considers the name of the type containing the method and its line number when needed. When calculating text-based similarities (e.g., body and signature), HistoryFinder uses the *Jaro-Winkler distance* algorithm [56]. As we show later, this simple algorithm achieves high accuracy in both open and closed-source projects with efficient runtime performance.

When invoked for the first time, HistoryFinder locates the specified method by cloning the repository, checking out the appropriate SHA, and reading the method text from the provided file path and line number. HistoryFinder also uses a language-specific parser to generate an AST of method signatures in the files it analyzes; while for most commits this is just one file, when a method has been moved, it can include parsing all files modified in a given commit. The AST enables quickly and systematically identifying all methods,

[6] Anonymous service (no logs): https://tinyurl.com/tutkm4o

[7] URL removed for double blind submission.

[8] The line number is used to differentiate overloaded methods.

their signatures, and bodies for the matching algorithm. For a given commit, HistoryFinder stores the current file path (*path*), line number (*num*), method signature (*sig*), and method body (*body*) for the specified method. To build the history for the method, HistoryFinder then considers the preceding commit that modified the *path* containing the method. HistoryFinder includes all the branches that contributed to the method's current state.

Since HistoryFinder works *backwards* in time from the most recent commit to the oldest commit, it is generally trying to determine where the method *came from* while it searches for the commit that introduced the method into the repository. Using a greedy approach it tries to identify the method using a four phase heuristic; the description of these phases is presented below, and the pseudo-code for the heuristic can be found in Figure 2. HistoryFinder uses several thresholds when comparing program elements; these thresholds were derived using the data-driven approach described in Section V.

Phase 1: Method unchanged. Modifications to the file containing the method do not necessarily imply that the specified method was changed. To check for this, HistoryFinder first looks for a method with textually-identical *sig* and *body* within the *path*. If there exists such a method, this means that relative to the preceding commit, this commit changed some other part of the file but not the method itself. Therefore, this change is not added to the method's history. HistoryFinder then iterates, using the version control system to find the preceding commit that modified *path* and executes again from the beginning. For efficiency, HistoryFinder's algorithm *only* considers those changes that modify the file containing the method.

Phase 2: Method modified within current file. If an identical method is not found in *path* in Phase 1, HistoryFinder then considers all other methods within the file to check for instances where the method was modified. To do this it looks at all of the method bodies within *path*. If a method is found with at least an 80% similar *body*, the inputs are updated for subsequent searches (e.g., to reflect any changes to the *sig*, *body*, or *num*) and the commit is added to the method's history for the next iteration.

Phase 3: Method moved through file rename or move. If no match is found in the first two phases which only examine *path*, HistoryFinder widens its search to consider all other files that were modified in the commit. The files modified in this commit are important, because HistoryFinder knows that by not matching previously, either the method was moved from another file (e.g., because the file changed paths or the file was renamed) or the method was introduced. To check for this, HistoryFinder examines the signature ASTs for all other files modified in the commit. In this phase, HistoryFinder accounts for path rename refactorings (e.g., the filename is the same but the overall path has changed). To do this it searches all files for a method that have the same *sig*, a body that is at least 90% similar, is within 10 lines of `num` (e.g., has the same position in the file), and has the same parent type name as the previous method (e.g., if the method was contained in type `Bar`, HistoryFinder confirms that the new type is also called

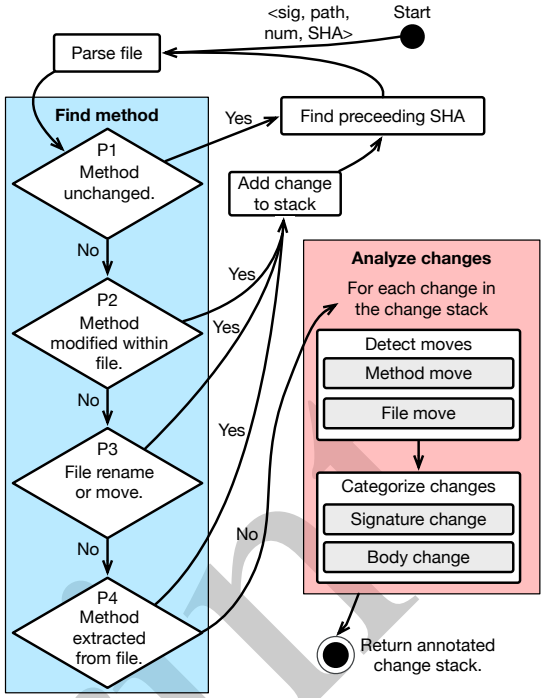

Fig. 1. High-level approach: each query starts with a method name and SHA. HistoryFinder iterates backwards through history until it finds the introducing commit for that method.

`Bar`). If such a method is found, the inputs are updated and the change is added to the method's history for the next iteration.

Phase 4: Method extracted from different file. Finally, HistoryFinder considers the most challenging form of transformation: method extractions. In an ideal situation, an extract method refactoring will just move a method from one file to another. In reality, the methods are often changed along the way (e.g., their signatures are modified and their body may be changed). HistoryFinder ranks all methods within all files modified by a change by their body similarity. The most-similar method is matched if either a method is 95% similar and is < 4 lines of code (LOC), or is 82% similar and is > 4 LOC. This LOC-based discrimination is needed to decrease the chances of erroneously matching short methods. If a match is made, the inputs are updated for subsequent searches and the commit is added to the method's history for the next iteration.

Preparing method history. If no candidate is matched in the final phase, the last change added to the method's history is considered the method's introducing commit. At the end of this process, the history contains only a list of the changes to the method, each consisting of $\langle$ *path, sig, SHA, num* $\rangle$. To increase the utility of this history, we further analyze each change to analyze *how* the method changed before returning the history to the developer.

B. Change Analysis

Once the list of changes for a method have been identified, the change analysis phase examines each change to determine how the method was modified. Each commit in the HistoryFinder output is associated with one or more specific change

```
// Inputs:
//  sig:   method signature
//  body:  method body text
//  path:  path to file method is in
//  num:   file line number for method signature
//  files: list of all files changed in the commit

// Phase 1
// Find unchanged method within same file
FOREACH meth in files[path]
  IF sim(meth['sig'], sig) == 1.0 &&
     sim(meth['body'], body) == 1.0
      return NO_CHANGE

// Phase 2
// Find modified method within same file
FOREACH meth in files[path]
  sim(meth['body'], body) >= 0.8
    return meth // method found in file

// Phase 3
// Find method within renamed or moved file
FOREACH file in files
  FOREACH meth in file
    IF sim(meth['sig'], sig) == 1.0 &&
       sim(meth['body'], body) >= 0.9 &&
       ABS(meth['num'] - num) <= 10
         return meth // method found in moved file

// Phase 4
// Find method modified from different file
methods = all methods in all files
// Sort methods by decreasing body similarity
methods = sort(methods, sim(entry['body'], body))

// Find highest matching method
FOREACH meth in methods
  IF meth['loc'] > 4 &&
    sim(meth['body'], body) >= 0.82
      return meth
  IF meth['loc'] <= 4 &&
    sim(meth['body'], body) >= 0.95
      return meth

// No match, last commit was introducing commit
return null;
```

Fig. 2. HistoryFinder method matching algorithm: `meth` refers to method, `sim` refers to the previously described similarity matching approach. Thresholds are explained in Section V.

kinds. This categorization of changes, presented in Figure 3, is a simplified version of the change taxonomy described by Fluri et. al. [46]. At the top-most level, there are four primary change kinds. The goal of HistoryFinder's analysis is to find the complete change history back to the method's introductory commit; this is captured by the *Introduced* change kind. Most changes in practice are *MethodChange* which captures the primary modifications that methods undergo. The *MultiChange* kind is used to maintain an unordered list of other change kind instances so the developer can examine these compound changes. The *NoChange* is a special kind that indicates methods that did not change in an identifiable way (e.g., when a commit modified some other part of the method's containing file).

Method changes can occur in several different ways. The most common of these by far is the *BodyChange* which

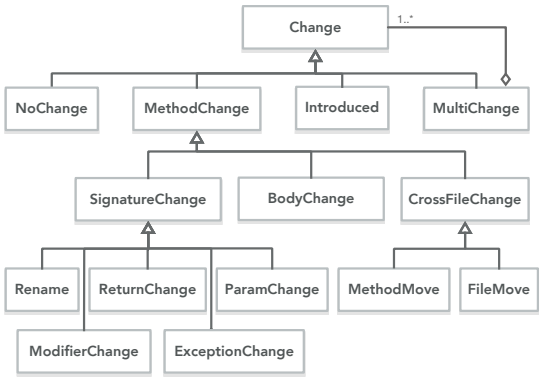

Fig. 3. Hierarchy of change kinds in HistoryFinder.

occurs whenever the text of the method body changes. The *SignatureChange* kind occurs when the method is renamed (*Rename*), or its parameters (*ParamChange*), return type (*ReturnChange*), modifiers (*ModifierChange*), or thrown exceptions (*ExceptionChange*) are altered.

Finally, some method changes arise from `CrossFileChange`; *FileMove* occurs due to common filesystem transformations like file rename or path changes. More complex changes that move methods between files (*MethodMove*) include extract method, push-down, and pull-up refactorings. The significance of the *MethodMove* change kind, especially in combination with the *Rename* change kind, has been previously identified [23]. Consequently, we consider a proper identification of this change kind to be an important goal for building comprehensive method histories.

C. Implementation

We implemented HistoryFinder in Java. While HistoryFinder is language-aware, the core approach is language-independent, given the required AST parsers. All core components with language-specific functionality use abstract classes and interfaces with concrete language-specific implementations. To add support for a new language, two HistoryFinder interfaces, `Parser` and `Method`, need to be implemented. For example, `Parser` defines a method signature `findMethodByNameAndLine(name, line)` which finds a `Method` instance within a file given its name and start line, and is relatively easy to implement. Our Java implementation of these two interfaces has ~250 LOC, and is the only language-specific code required to support a new language. To perform Git operations and to traverse commits in repositories, HistoryFinder leverages the *JGit* library and `JavaParser` to generate ASTs. We have also started to implement support for Python (using `ANTLR`), JavaScript (using `Nashorn`), and Ruby (using `jRuby`), but have not extensively evaluated HistoryFinder on these languages.

Summary: HistoryFinder leverages different source code metrics (e.g., body similarity, signature similarity, and line similarity) to decide if two given methods are similar. If these similarities exceed our data-informed thresholds, then two methods are considered the same. This process continues until the first (introduction) commit is found for a given method. Each change commit is also associated with a change kind (e.g., *BodyChange*), which makes HistoryFinder's output helpful for program comprehension.

V. Empirical Evaluation

From our literature review and the developer survey, we identified two important requirements for method history extraction tools: first, they need to be able to extract complete histories and second, they need to run on demand without requiring pre-processing. These requirements necessitate that HistoryFinder be robust to transformations and quickly perform online analyses of the version control repository. To assess how effectively HistoryFinder meets these requirements, we examine the following research questions:

RQ3 How accurate and robust is HistoryFinder for producing complete and correct method histories?

RQ4 What is HistoryFinder's runtime performance, and is it acceptable for on-demand use?

A. Methodology

This section describes how we constructed an oracle of method histories and how we tuned HistoryFinder's matching algorithm to evaluate the correctness of the histories HistoryFinder produced. We chose to develop an oracle to allow us to compute both recall (the proportion of complete histories an approach can detect) as well as the precision (the proportion of histories that do not contain incorrect results); we believe recall to be most important metric for history tracking as this measures how likely a developer will be able to find the complete history of a method of interest.

Subjects. For our evaluation, we chose 20 popular open source Java projects, each with at least 2,000 commits, 900 methods, and 250 stars on GitHub. These projects span a range of domains, and we consider them a representative set of mid- to large-scale open source Java projects. Table II lists the projects and their statistics.

Oracle construction. To verify the correctness of the histories produced by HistoryFinder, we manually constructed an oracle of method histories. To do this we randomly selected 10 methods having at least 3 commits from each project in Table II, for a total of 200 methods. This was laborious manual work, and required ~30 minutes per method. Method histories were extracted by multiple authors and one non-author using a combination of tools (e.g., `git-log`) and manual inspection. It was then independently validated by two experienced developers for completeness and correctness. Disagreements were resolved by consulting another individual.

Training phase. As described in Section IV, HistoryFinder uses a heuristic approach to match methods across changes.

TABLE II
Java repositories used for our empirical evaluation.

	Repository	# commits	# methods	# stars
Training	checkstyle	8,010	3,084	3,848
	commons-lang	5,230	2,197	1,389
	flink	14,416	17,009	4,166
	hibernate-orm	9,100	23,159	3,318
	javaparser	4,781	3,613	1,883
	jgit	6,065	8,277	604
	junit4	2,228	1,107	6,992
	junit5	4,695	2,078	2,323
	okhttp	3,262	1,433	28,107
	spring-framework	17,041	3,214	22,769
Validation	commons-io	2,123	996	488
	elasticsearch	40,353	18,261	33,640
	hadoop	19,805	32,888	7,801
	hibernate-search	6,172	5,069	283
	intellij-community	226,106	36,387	6,335
	jetty	15,991	11,522	2,139
	lucene-solr	30,500	29,888	1,840
	mockito	4,811	1,366	7,358
	pmd	13,360	2,567	1,738
	spring-boot	17,818	2,451	27,527
	TOTAL	451,867	206,566	164,548

We initially set HistoryFinder's thresholds using our intuition. For example, we set a high body similarity threshold to reduce false positives. We then used 100 methods from the oracle as our training set. We modified the algorithm (e.g., adding a special condition for short methods) and updated the threshold values until we achieved 100% training accuracy. In order to alleviate regression issues while we tuned the thresholds, we created a separate test method for each training method which compares the expected method history with the HistoryFinder's generated history.

Validation phase. During the validation phase, HistoryFinder's thresholds were fixed at the values previously shown in Figure 2 and could no longer be changed. When computing accuracy, we compare the histories generated by HistoryFinder using these threshold values with the remaining 100 validation methods that were not used for training.

B. Results

This section describes HistoryFinder's accuracy and runtime performance in accordance with RQ3 and RQ4.

1) RQ3: HistoryFinder's recall and precision

To evaluate HistoryFinder, we examined precision and recall, recall by different change types, and recall compared to

```
- public InvocationMatcher create(Object proxy, Method method) {
-     Invocation invocation = new Invocation(proxy, method);
-
+ public InvocationMatcher bindMatchers(Invocation invocation)
          throws InvalidUseOfMatchersException {
      InvocationMatcher im = new InvocationMatcher(invocation);
      return im;
  }
```

Fig. 4. A diff showing a complex method transformation that HistoryFinder failed to recognize.

IntelliJ and `git log -L`, and FinerGit.

Completeness and correctness. We compared the histories produced by HistoryFinder with the histories of the 100 validation methods in our validation phase. HistoryFinder correctly identified the exact histories for 90 of the 100 methods; that is, the tool had 90% recall for this oracle. Encouragingly, HistoryFinder could still be useful for the 10 methods for which it failed to uncover the complete history. For one method, HistoryFinder found the complete history including the introducing commit, but unfortunately continued tracking another prior method due to its similarity. This is the only case from one hundred queries where a returned history was not part of the queried method's history (e.g., a false positive was returned); that is, the tool had 99% precision for this oracle. HistoryFinder was only able to return partial histories for the other 9 missed methods. For example, for one of the validation methods, HistoryFinder successfully extracted the first 15 commits out of the 16 that actually occurred (15/16). For the other 8 methods that did not return the correct complete history, the fraction of histories returned were 6/8, 3/6, 15/17, 7/18, 19/26, 5/9, 6/9, and 1/4.

When does HistoryFinder fail? Figure 4 shows a `diff` from one of the ten incomplete methods. From a developer's perspective, the method was modified to take an instance of `Invocation` instead of creating it in the method body, which can be seen by the changes to the parameters, the exception signature, and the removal of the single line in the body. The method was also renamed in the same commit. Collectively, these changes caused HistoryFinder to fail to report that the second method represented a transformation of the first. It remains an interesting challenge to us to solve these kind of scenarios without significantly affecting HistoryFinder's runtime performance.

Characteristics of the validation set. Two reasonable questions are whether HistoryFinder's recall was high because the validation set contained only methods with similar size characteristics or methods that have short change histories. In the validation set, 20% of the methods have SLOC ≥20, while 27% have SLOC ≤4. In terms of changes, 60% of the methods were changed more than 5 times and 20% of the methods were changed at least 10 times. With respect to change complexity, most commits changed by fewer than 20 lines, but there are some commits that changed by almost 100 lines. Ultimately, the difference in distributions (when compared in pairs between all of the methods in the validation set and the methods HistoryFinder correctly identified from that set) are statistically insignificant (nonparametric Wilcoxon-Mann-Whitney test, P-value > 0.05). This suggests that the validation set contained a diverse set of methods, and HistoryFinder's high accuracy was not influenced by a particular group (e.g., methods with short change history).

Robustness across different transformation types. Many kinds of source code transformations happen during development ranging from simple body changes to a complex refactorings such as a pulling-up, pushing-down, or extracting methods. Our initial survey participants acknowledged the difficulty of tracking methods that have undergone complex refactorings. Existing approaches (Section II) have difficulty constructing histories for methods that have undergone these transformations making it important to evaluate HistoryFinder's robustness across these complex transformations.

For a given method, our validation set contains all the commits and change types in which the method changed. To calculate the accuracy of each change type, we counted instances where change type produced by HistoryFinder did not match the change type in the oracle. For example, the validation set contained a total of 501 commits with the type `BodyChange`. The change types of 4 of the commits produced by HistoryFinder were different resulting in a 99.2% accuracy for `BodyChange`. Table III provides the accuracies of each change type. The lowest accuracy of HistoryFinder is 91.3% caused by 2 failures for the `Rename` change type. We conclude that HistoryFinder can robustly construct method histories regardless of the types of changes a method undergoes.

TABLE III
HISTORYFINDER'S ACCURACY ACROSS DIFFERENT TYPES OF SOURCE CODE TRANSFORMATIONS. HISTORYFINDER DOES NOT EXHIBIT WEAKNESSES ON ANY PARTICULAR TYPE OF CHANGE.

Change Type	Occurrence	Accuracy (# failures)
`BodyChange`	527	~99.2% (4)
`FileRename`	167	100.0% (0)
`Introduced`	100	~98.0% (2)
`ParameterChange`	73	100.0% (0)
`MoveFromFile`	41	100.0% (0)
`Rename`	23	~91.3% (2)
`ModifierChange`	20	100.0% (0)
`ReturnTypeChange`	17	100.0% (0)
`ParameterMetaChange`	14	100.0% (0)
`ExceptionsChange`	8	100.0% (0)
`MultiChange`	99	~97.9% (2)

2) HistoryFinder accuracy relative to prior work

Among all the tools discussed in Section II, only four work for Java methods: IntelliJ's git history feature, `git log -L`, Historage, and FinerGit. FinerGit [20] is an improvement over Historage and it is the state-of-the-art tool for Java method history tracking, despite its problem with large projects that we discuss later. IntelliJ and `git log -L` were frequently mentioned in the developers' answers and discussions from our survey: IntelliJ was mentioned 26 times, and `git log` was mentioned 24 times. We thus compare HistoryFinder with IntelliJ, `git log -L`, and FinerGit. It was extremely laborious and time consuming to manually run and check all the validation methods against these tools to evaluate their accuracy; unlike HistoryFinder, there is no test suite that can automatically run and evaluate them. This validation took one of the authors ≈30 hours to complete.

HistoryFinder compared to `git log -L` / IntelliJ. There are two modes for using `git log -L`: one works with line range (`git log -L start,end:filename`), and the other directly works with a given method name (`git log -L :funcname:filename`)[9], we evaluate each of these

[9] `*.java diff=java` must be in the `.gitattributes` file.

modes. In contrast to the 90% recall of HistoryFinder, IntelliJ was able to identify the complete history for 68% of the validation methods. `git log -L` identifies the complete history for 63% of the complete histories using the `start,end:filename` mode and 41% of complete histories using the `:funcname:filename` mode.

To examine recall on particularly challenging tasks, we also investigated 30 complex methods from our validation set to compare the accuracy of these two tools with HistoryFinder. These methods have undergone different types of transformations throughout their lifetime. For selecting such complicated methods, we counted the total number of unique transformation kinds for each method. For example, the `count` is 3 for a method that had 4 `BodyChange`, 2 `Rename`, and 1 `Introduced` commits in its history. We then ordered the methods based on those counts, and selected the top 30 of them. For this set, HistoryFinder identifies the complete method history with 87% (26/30) accuracy. IntelliJ achieves 50% accuracy (15/30). `git log -L` achieves 47% (14/30) accuracy using the `start,end:filename` mode and 37% (11/30) accuracy with the `:funcname:filename` mode. Combining the *best* results from both `git log` and IntelliJ, the accuracy is 57%, which is 30% lower than HistoryFinder alone. This shows that HistoryFinder significantly outperforms the state-of-the-practice tools used by practitioners today.

HistoryFinder compared to FinerGit. When comparing HistoryFinder to FinerGit we encountered a problem as FinerGit ran out of memory (with 16 GB of RAM for the FinerGit process) or did not finish pre-processing within 60 minutes for the four largest projects in the validation data set (`intellij-community, elasticsearch, lucene-solr, hadoop`). For the 60 methods in the validation set from the smaller six projects, FinerGit identified the complete history for 39 (65%) of the methods. In contrast, HistoryFinder identified the complete history for 54 (90%) of these same 60 methods. This demonstrates that HistoryFinder has higher recall than the state-of-the art without the memory and computation downside associated with pre-processing.

3) RQ4: HistoryFinder's runtime performance

To evaluate HistoryFinder's runtime performance we recorded the wall clock time for each of the methods from the 10 validation repositories (total of 141,395 methods). We collected the runtimes on a development computer (12-core processor running at 3.30GHz with 32GB memory). HistoryFinder has a median runtime under 2 seconds; 90% of the methods returned in less than 10 seconds while the worst-case runtime was $<$ 20 seconds. We were also interested to see HistoryFinder's runtime performance across different repositories. We calculated the median (for graph readability) of all the methods' runtimes for each validation repository. Figure 5 shows the distribution of the medians. The `intellij-community` repository is the outlier with a median execution time of about 7 seconds, which is due to a combination of large source files (which take longer to parse) and a high frequency of change within these files.

We believe that HistoryFinder's latency acceptable, with a median runtime of ~2 seconds per method. A previous study by Kochhar *et al.* found that developers are very satisfied with software analysis tools having feedback-latency less than one minute [21].

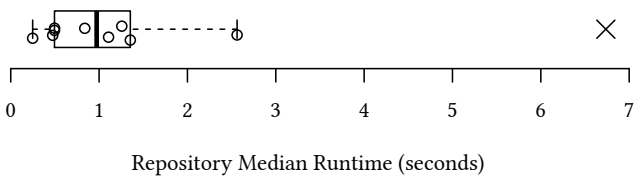

Fig. 5. The median wall clock time it took HistoryFinder to process all methods in each validation repository (circle) listed in Table II.

Summary: HistoryFinder's recall exceeds related tools and returns results quickly enough for on demand use.

VI. Industrial Field Study

To ensure HistoryFinder's accuracy and the runtime performance translates to industrial closed-source systems, we performed an industrial field study. Industrial participants independently verified the accuracy and completeness of generated histories on methods they selected from their own projects. We also captured HistoryFinder's runtimes for the participant-selected methods. As a follow-up to the industrial survey (Section III), we also asked participants how they would apply HistoryFinder in their industrial setting. For this purpose, we aim to additionally answer one research question:

RQ5 In which scenarios are method-level histories useful to industrial developers and why?

A. Study Participants

We conducted our field study with 16 industrial engineers. The majority of the participants (12/16) did not participate in the survey described in Section III. Participating developers were required to have Java background and had to be able to provide a set of Java methods whose histories they were familiar with. They had a median of 10 years of programming experience, 3.5 years working as professional software developers, and 8.5 years experience with version control. Each participant was given a coffee gift card for their time.

B. Study Design

We conducted the field study as on-site interview sessions lasting ~45 minutes per participant. Each participant was asked to choose 2-4 Java methods from their own repositories that they were familiar with and that had been revised multiple times. We executed HistoryFinder using the participant selected methods on their computer and recorded the results and runtime. Each participant then evaluated the correctness of the results for their selected methods. We asked participants three questions:

1) Was the method history correct?
2) Which scenarios might HistoryFinder be useful for?
3) Is the information produced by HistoryFinder helpful?

C. Study Results

We summarize the results of the field study by the associated research questions RQ3, RQ4, and RQ5.

1) RQ3: HistoryFinder's correctness

The industrial participants produced 45 method histories during the field study. HistoryFinder found the correct and complete histories for 41 of the 45 methods (91%). For the four methods for which HistoryFinder failed, two had commits containing multiple complicated changes causing the overall similarity to be below the matching threshold. For one method, HistoryFinder stopped while parsing a file,[10] and for one we could not reproduce the problem. Otherwise, participants confirmed that HistoryFinder performed well and identified the relevant commits without any false positives.

2) RQ4: HistoryFinder's runtime performance

The median wall clock runtime was less than 2 seconds for the methods chosen by our industrial participants on their projects. One outlier method, which had changed 44 times in an extremely large file, took 8 seconds.

Summary: HistoryFinder's runtime performance and accuracy is similar for open source (90%) and closed-source (91%) projects.

3) Scenarios for method-level histories (RQ5)

Participants described several scenarios in which HistoryFinder would be useful. To understand these scenarios, two authors independently card sorted the transcribed responses using an open coding approach [57]. HistoryFinder allows developers to determine a method's ***provenance*** because they "*can see easily who introduced a method*" *(P3)*. It can help answer "*[how] this code came to be*" *(P8)*. It can aid in ***traceability***, "*especially [...] through refactorings [since] other tools like IntelliJ and git-log don't help us here*" *(P9)*. Developers can "*focus on moves and other refactoring operations that would not be traceable with conventional Git history*" *(P5)*. Participants thought that the "*histories are very helpful for **onboarding** [since] Git blame isn't useful because formatting commits destroy everything*" *(P14)*, or "*if you're new to a codebase*" *(P10)*. Participants also thought it would be useful for **code understanding** because "*one can learn more about the codebase in an easy way*" *(P7)*, "*for code you're not used to*" *(P10)*. HistoryFinder ***automates history-related tasks*** because developers "*already do what this tool is doing, we just do it manually*" *(P8)*. Overall, 13/16 (81%) participants rated the method histories as very helpful or somewhat helpful while the remaining 3 were neutral.

VII. Discussion

Here we discuss future HistoryFinder improvements and present some research questions that should be investigated.

A. Improving HistoryFinder

HistoryFinder leverages manually constructed histories to tune its thresholds for deciding if two methods are similar. Machine learning is an alternative for such a data-driven approach. However, the difficulty in building the oracle limited the size of our training and validation sets. Without enough training samples, there is evidence that a heuristic approach is more accurate than a machine learning approach [58], [59]. Our heuristic approach explains why it considers two methods similar or different (explainable AI), which is often not easy with machine learning. Software practitioners are less confident to accept predictions based on unexplainable models [60]. Nevertheless, it would be interesting to develop a larger oracle to see how machine learning algorithms perform in source code history construction.

HistoryFinder currently uses the *Jaro-Winkler distance* algorithm for string similarity ratings [56]. For source code, *n-gram string matching* algorithm [61] can be a better alternative [49], [46]. Presently, HistoryFinder can be run from a web service and from the command line interface; integration with an IDE (e.g., IntelliJ) would be useful. With a small training and validation set, we evaluated HistoryFinder with Python source code. The initial results are promising and we continue to build out support for additional languages.

B. Impact on the MSR research community

We believe that HistoryFinder can help extend MSR research. For example, for the entire corpus of methods in Table II, while 33% of methods were never changed after they were inntroduced, 50% are changed three times or more, and 5% are changed ten times or more. Why do some methods change so frequently, and what impact do they have on software maintenance? Can we discover information from these methods for writing more stable code? Can HistoryFinder build accurate history of test methods as well, so we can study the evolution of test methods alongside source methods? HistoryFinder is currently being used for three different *software evolution* studies by two different research groups.

C. Threats to Validity

Internal Validity. The primary threat to the internal validity is related to the construction of our oracle. This threat was mitigated by two experienced developers who validated the oracle independently. Another threat is related to our sampling method: the methods selected to be used in our oracle were randomly chosen from all methods having more than three commits. This was meant to focus the evaluation on more interesting and challenging histories but we may have missed certain classes of histories by using random sampling. In both the survey and the field study there may be moderator bias, since participants were selected from the authors' personal networks. An additional *placebo* could have been used, but we were concerned this would reduce the participant pool [62].

External Validity. Although we evaluated HistoryFinder on both open-source and closed-source industrial codebases, the number of methods was small, so our findings may not generalize. In our survey and field study, our recruited participants we recruited may not be representative of all developers.

[10] This was related to the `javaparser` which we subsequently fixed.

VIII. Conclusion

In this paper, we described a formative survey with developers from both industry and academia to learn how they use source code history and what challenges they face when doing so. We learned that existing tools do not effectively surface the results developers need to answer their source code history questions. To address this, we built HistoryFinder, a tool that is robust to common source code transformations and can generate accurate method-level source code histories on demand. Empirical analysis with open-source and closed-source projects shows that HistoryFinder can return complete and accurate histories for ~90% of methods outperforming FinerGit, the current state-of-the-art, and both IntelliJ and `git log`, the current state-of-the-practice. An industrial field study further confirmed that HistoryFinder would be useful for a wide range of industrial development tasks such as traceability and program understanding.

Having access to robust source code histories is also useful for extending research in *mining software repositories* and *software evolution*, for example enabling studying the structural properties of methods that make a method more (or less) prone to future changes. It is our hope that both developers and the research community will find HistoryFinder useful for providing a richer understanding of how their systems have evolved in the face of the kinds of source code transformations that frequently occur in practice.

References

[1] K. Maruyama, E. Kitsu, T. Omori, and S. Hayashi, "Slicing and replaying code change history," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2012, p. 246249.

[2] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 1028–1038.

[3] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007, pp. 344–353.

[4] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002, pp. 503–512.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 492–501.

[6] S. D. Thomas Zimmermann, Peter Weisgerber and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004, pp. 563–572.

[7] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 284–292.

[8] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *European Conference on Object-Oriented Programming (ECOOP)*, 2012, pp. 79–103.

[9] D. Steidl, B. Hummel, and E. Juergens, "Incremental origin analysis of source code files," in *Proceedings Working Conference on Mining Software Repositories (MSR)*, 2014, p. 4251.

[10] R. Funaki, S. Hayashi, and M. Saeki, "The impact of systematic edits in history slicing," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2019, pp. 555–559.

[11] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *Transactions on Software Engineering (TSE)*, vol. 44, no. 2, pp. 182–201, 2018.

[12] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: A study on why and how developers examine it," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 1–10.

[13] A. E. Hassan and R. C. Holt, "C-rex: An evolutionary code extractor for c," 2004.

[14] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for java," in *Proceedings of the International Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL)*, 2011, pp. 96–100.

[15] S. Kim, K. Pan, and E. J. Whitehead, "When functions change their names: automatic detection of origin relationships," in *Proceedings Working Conference on Reverse Engineering (WCRE)*, 2005, pp. 143–152.

[16] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *Transactions on Software Engineering (TSE)*, vol. 30, no. 9, pp. 574–586, 2004.

[17] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in *Proceedings Workshop on Mining Software Repositories (MSR)*, 2007, pp. 14–21.

[18] A. Chen, E. Chou, J. Wong, A. Y. Yao, Qing Zhang, Shao Zhang, and A. Michail, "Cvssearch: searching through source code using cvs comments," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2001, pp. 364–373.

[19] F. Servant and J. A. Jones, "Fuzzy fine-grained code-history analysis," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2017, pp. 746–757.

[20] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking java methods with git mechanisms," *Journal of Systems and Software*, vol. 165, p. 110571, 2020.

[21] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners expectations on automated fault localization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 165–176.

[22] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *Transactions on Software Engineering (TSE)*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[23] F. V. Rysselberghe, M. Rieger, and S. Demeyer, "Detecting move operations in versioning information," in *Proceedings Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 271–278.

[24] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 492–501.

[25] H. Kagdi, M. Hammad, and J. I. Maletic, "Who can help me with this source code change?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2008, pp. 157–166.

[26] A. W. Bradley and G. C. Murphy, "Supporting software history exploration," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 193–202.

[27] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000, pp. 166–177.

[28] F. Servant and J. A. Jones, "History slicing: Assisting code-evolution tasks," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[29] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 2002, pp. 127–136.

[30] T. Zimmermann, "Fine-grained processing of cvs archives with apfel," in *Proceedings OOPSLA Workshop on Eclipse Technology eXchange (eTX)*, 2006, pp. 16–20.

[31] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[32] L. Pascarella, F. Palomba, and A. Bacchelli, "On the performance of method-level bug prediction: A negative result," *Journal of Systems and Software (JSS)*, vol. 161, Mar. 2020.

[33] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 171–180.

[34] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 200–210.

[35] R. Pereira, "Locating energy hotspots in source code," in *Proceedings of the International Conference on Software Engineering Companion (ICSE)*, 2017, pp. 88–90.

[36] M. U. Farooq, S. U. Rehman Khan, and M. O. Beg, "Melta: A method level energy estimation technique for android development," in *Proceedings of the International Conference on Innovative Computing (ICIC)*, 2019, pp. 1–10.

[37] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: Generating concise linked code differences," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2018, p. 679690.

[38] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[39] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[40] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2005, pp. 389–398.

[41] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007, pp. 427–436.

[42] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 325–334.

[43] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2017, pp. 269–279.

[44] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 483–494.

[45] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998, pp. 368–377.

[46] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Transactions on Software Engineering (TSE)*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[47] M. Hashimoto and A. Mori, "Diff/ts: A tool for fine-grained structural change analysis," in *Proceedings Working Conference on Reverse Engineering (WCRE)*, 2008, pp. 279–288.

[48] M. Pawlik and N. Augsten, "Rted: A robust algorithm for the tree edit distance," *Proceedings VLDB Endowment*, vol. 5, no. 4, pp. 334–345, Dec. 2011.

[49] Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1994, pp. 120–126.

[50] Mayrand, Leblanc, and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1996, pp. 244–253.

[51] F. V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2004, pp. 336–339.

[52] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching: Research articles," *Journal Software Evolution and Process*, vol. 18, no. 1, pp. 37–58, Jan. 2006.

[53] E. Kodhai, A. Perumal, and S. Kanmani, "Clone detection using textual and metric analysis to figure out all types of clones," *International Journal of Computer Communication and Information System*, vol. 2, no. 1, Jul. 2010.

[54] M. Sudhamani and L. Rangarajan, "Structural similarity detection using structure of control statements," *Procedia Computer Science*, vol. 46, pp. 892–899, 2015.

[55] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2008, p. 153162.

[56] W. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage," 1990.

[57] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, March 1990.

[58] A. Amini, K. Banitsas, and J. Cosmas, "A comparison between heuristic and machine learning techniques in fall detection using Kinect v2," in *Proceedings of the International Symposium on Medical Measurements and Applications (MeMeA)*, 2016, pp. 1–6.

[59] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2019, pp. 93–104.

[60] H. K. Dam, T. Tran, and A. Ghose, "Explainable software analytics," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 53–56.

[61] G. W. Adamson and J. Boreham, "The use of an association measure based on character structure to identify semantically related words and document titles," *Information Storage and Retrieval*, vol. 10, no. 7–8, pp. 253–260, Jul. 1974.

[62] S. Sahlqvist, Y. Song, F. Bull, E. Adams, J. Preston, and D. Ogilvie, "Effect of questionnaire length, personalisation and reminder type on response rate to a complex postal survey: Randomised controlled trial," *BMC Medical Research Methodology*, vol. 11, May 2011.