CodeShovel: constructing robust source code history

by

Felix Grund

B.Sc., Bamberg University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

June 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

CodeShovel: constructing robust source code history

submitted by **Felix Grund** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Reid Holmes, Computer Science
Supervisor

Ivan Beschastnikh, Computer Science
Supervisory Committee Member

Abstract

Source code histories are valuable resources for developers, and development tools, to reason about the evolution of their software systems. Through a survey with 42 professional software developers, we gained insight in how they use the history of their projects and what challenges they face while doing so. We discovered significant mismatches between the output provided by developers' existing approaches and what they need to successfully complete their tasks. To address these shortcomings, we created CodeShovel, a tool for navigating method histories that is able to quickly produce complete method histories in 90% of the cases. CodeShovel enables developers to navigate the entire history of source code methods quickly and reliably, regardless of the transformations and refactorings the method has undergone over its lifetime, helping developers build a robust understanding of its evolution. A field study with 16 industrial developers confirmed our empirical findings of CodeShovel's correctness and efficiency and additionally showed that our approach can be useful for a wide range of industrial development tasks.

Lay Summary

At the core of all software is its source code that is saved in (normally multiple) files on the file system. To navigate changes to source code, most software developers use version control, a range of tools that capture all changes made to the source code as dedicated versions. This work aims to provide better views and means of navigation for such version histories. In particular, these histories are normally viewed and navigated on the file level, which makes it very hard to view histories of only certain components of the source code. We introduce a new tool that enables tracing source code history of logical components of the code, rather than the text saved in a file.

Preface

All of the work presented henceforth was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey campus. All projects and associated methods were approved by the University of British Columbia's Research Ethics Board [certificate #H18-00510].

This material is the result of ongoing research at the Software Practices Laboratory. I was the lead investigator, responsible for all major areas of concept formation, data collection and analysis, as well as manuscript composition. Reid Holmes was the supervisory author on this project and was involved throughout the project in concept formation and manuscript composition. The material has not been published prior to this thesis.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I thank everyone at the Software Practices Lab of the Computer Science Department of the University of British Columbia for magnificient 2.5 years in academia. My master program at UBC has been an extraordinary experience. Moreover, I thank everyone at Scandio GmbH in Munich, my first employer, for their great support before and throughout my master program. And last but most importantly, I thank my family who have always had open ears when I was faced with complicated life situations.

Chapter 1

Introduction

Source code repositories contain a wealth of information that is valuable and relevant to support developers in their work. For instance, this information can answer questions commonly asked about source code (e.g., [15, 18, 21, 29]), help predict the location and likelihood of source code defects [22], or provide more context for code reviews [17]. We performed a survey with 30 industrial developers and 12 academic developers and found that they frequently have questions that they try to answer using source code histories, but that existing tools do not support them in accessing the historical data they need. Specifically, existing tools return results at the wrong granularity (they are file-based rather than method-based) and results are often incomplete (common file transformations inhibit retrieving complete historical data). Consequently, developers have to perform multiple queries and manually filter changes to identify the changes that are relevant to their task.

To address the shortcomings identified by our survey participants, and to help them find the information they need from source code histories, we built *CodeShovel*, a tool for surfacing complete histories of source code methods. CodeShovel works by parsing the files changed in a commit and using the resulting Abstract Syntax Tree (AST) to reason about the nature of the change. Currently, a robust language parser for Java is available, and we have started to work on JavaScript and Ruby parsers. The algorithm is easily extensible for other languages. A similarity algorithm that considers five key features compares methods in the ASTs to determine whether a method of interest was modified, and if so how. Ultimately, CodeShovel

is able to quickly and accurately return complete and precise method-level source code histories. In our previously described survey, we identified that a majority of industrial developers would find a tool working on the method- and class-levels most useful. We therefore implemented CodeShovel to work on methods with the perspective that an extension for classes would be feasible by aggregating method histories.

We evaluated CodeShovel's correctness and performance using 20 popular open source repositories with a total of 176,125 methods. We manually created oracle method histories for 10 methods in each of the 20 repositories (totaling 200 methods) and then proceeded in two phases: first, we trained our algorithm on the first 100 oracle method histories and created unit tests to prevent regression; second, we evaluated the correctness of CodeShovel on the next 100 oracle method histories after an implementation freeze. We found that CodeShovel is able to correctly determine the complete history of over 90% of methods. By running CodeShovel on all the 176,125 methods in the 20 repositories, we found that CodeShovel produces method histories with a median execution time of less than 2 seconds.

To ensure the correctness of CodeShovel translates to closed-source, industrial code bases, we additionally ran CodeShovel on 45 participant-selected methods in a field study with 16 industrial developers. The results were nearly identical to our empirical study in terms of accuracy (the developers assessed 91% of method histories to be correct and complete) and performance (median < 2 seconds execution time). Furthermore, our field study revealed that CodeShovel would be useful in the domains of provenance, traceability, onboarding, code understanding, and automation.

The primary contributions of this paper are:

- A survey with 42 professional developers to understand how they use source code history; this demonstrated a lack of tool support for the most frequently-performed historical understanding tasks.
- The implementation of CodeShovel, a novel approach for extracting method-level source code histories.[1]

[1] CodeShovel is available at: https://github.com/ataraxie/codeshovel

- A quantitative analysis of CodeShovel's accuracy and performance.
- A field study with 16 industrial developers verifying CodeShovel's correctness and performance, and an indication of use cases where CodeShovel would be most helpful.

The remainder of the paper is structured as follows: first, we describe a complete scenario in Chapter 2 where we illustrate the problems developers are commonly facing with line-based historical analysis tools. Chapter 3 then presents related work where we place CodeShovel in the context of similar tools for source code history. Chapter 4 describes the design and results from our survey with 42 professional developers. Here, we aim to answer the following research questions:

RQ1 Do developers use source code history when they are working with code? If so, what are they trying to learn when they examine source code history?

RQ2 In terms of their mental models and information needs, what level of temporal and structural granularity are most appropriate when using source code history?

RQ3 How do developers identify the history for specific code units and how well does existing tooling support these queries?

RQ4 How effectively can a language-aware code history viewer support program comprehension?

Our approach with CodeShovel is described in Chapter 5 and its implementation in Chapter 6.

We then evaluate our tool empirically in Chapter 7 based on the following research questions:

RQ5 How frequently are methods changed?

RQ6 Are the results returned by CodeShovel correct?

RQ7 Is CodeShovel fast enough for being used as an online tool that does not require preprocessing?

Chapter 8 describes our field study in which we further analyze **RQ6** and **RQ7** and one more research question:

RQ8 In which scenarios are method-level histories useful to industrial developers and why?

Discussion with threats to validity and future work follow in Chapter 9 before we conclude in Chapter 10.

Chapter 2

Motivation and Background

This section motivates the importance of source code history tooling during development tasks. We first illustrate a general high-level scenario in Section 2.1 and provide background information about existing tooling in Section 2.2. We then expand our initial scenario to be more specific using an example method in an open-source repository, demonstrating how existing tools fail to return many important historical references in practice (Section 2.3).

2.1 General Scenario

Consider the following scenario: a developer is about to review a pull request that changed a method. She has not seen this particular segment of code in a while and lacks understanding of what the method is actually doing. Since in-place documentation and comments are insufficient for a clear understanding of the code [27], and the author of the pull request is unavailable, she decides to investigate in the version control history associated with the method [17]. She believes how the method was previously evolved, and who authored those changes, will help contextualize the current pull request in terms of the problems prior developers encountered working on this code.[1].

She first uses her version control tool to view the history of the file containing the method that was changed. Unfortunately, the file history contains a large num-

[1]The approach of using version history for such program understanding tasks being very common among software developers will become more visible in our survey in Section 4.

ber of changes, among which only a few modified the method of interest to her pull request. These changes cannot be easily filtered for only the changes relevant to her pull request so she decides to use her version control tool to select a line-range based history (a functionality built into most version control systems). Because the file has undergone multiple extensive refactorings over its lifespan, this history reports many changes that are not related to the method she is interested in. This history also terminates at a fairly recent commit when the method was moved from another file. Consequently, even with a tool seemingly built for this task she was not able to obtain a complete view of the past changes that affected her methods of interest without extensive manual traversal of the version control history.

2.2 File- and line-based source code history

The scenario uses tools provided by modern version control systems to search and filter source code history. The most prominent version control system, *Git*, surfaces this functionality through the `git-log` command which includes several different options. Typically, a file path is provided for the history specific to a file. In addition, a line range can be provided if the investigator is only interested in a specific part of a file (`git log -L BEGIN,END:PATH`). While there are other advanced options such as the `-S` option which accepts a search string, most `git-log` invocations are at the file- (no parameter) and line-range (`-L`) granularity.

IDE-based version control tools that provide more abstract views are usually based on the same underlying data as `git-log` and its arguments (e.g., within the Eclipse or IntelliJ IDEs). While these tools suggest a language-aware aspect to their functionality, they remain essentially text-based. For example, the *Show history for method* feature in IntelliJ has no notion of the method as a code unit despite the term *method* in the command name within the IDE. Rather, IntelliJ extracts the line range of the method of interest and shows the history for this line range. This lack of language awareness causes these tools to produce a high proportion of false positives and miss many relevant changes (false negatives) for files that have been involved in non-trivial evolution. Additionally, these tools often report a unit as being new, where in fact it was just moved from a different file (e.g. during refactoring), even though in reality that unit may have a rich evolutionary history

prior to the refactoring.

2.3 Specific Scenario

We illustrate the challenges involved with investigating source code history of a code unit using the *Checkstyle*[2] project, a popular syntax validation tool for Java.

Suppose the developer is evaluating a pull request that changes the `CommonUtils::createPattern`[3] method which can be found on lines 93–112 in `CommonUtils.java`. She now wants to learn more about this method's history so she can better understand the context of the current pull request. She uses her version control tool to show the history of `CommonUtils.java`, but unfortunately the file history shows 47 changes to this file in three years and given that `createPattern` comprises only 3% of the file (20 LOC out of 664 LOC its current revision), it is unlikely that `createPattern` is germane to most of these changes.

Retrieving history with `git-log` She decides to examine the history for `createPattern` using its line range and issues the command `git log -L 93,112:PATH`.[4] This identifies two commits: `ce21086` which changed the body of the method and `2864c10`, which is a false positive because the change just moved the method within the file without modifying it. Unfortunately, several other changes are missing as well; in total, of the 17 years of history for this method, `git-log` is only able to return the commits that happened in the most recent 18 months. Through an exhaustive manual analysis we identified several changes to this method throughout its lifetime; these are shown in Table 2.1. Notably, the method was moved between files twice (`8d6fa33` and `1c15b6a`) and the file the method was in was renamed (or was moved between directories) and was moved between files three times (`f1efb27`, `ed595de`, and `cdf3e56`). While these changes could have helped the developer build the understanding they wanted for their task, it is nearly impossible to find them in a timely fashion with the current tools.

[2] We forked this repository to keep it stable at a specific commit for illustration purposes in this paper: https://github.com/ataraxie/checkstyle/.

[3] https://github.com/ataraxie/checkstyle/blob/746a9d69125211ff44af1cb37732e919368ba620/src/main/java/com/puppycrawl/tools/checkstyle/utils/CommonUtils.java

[4] `Full command: git log -L 93,112:src/main/java/com/puppycrawl/tools-/checkstyle/utils/CommonUtils.java`.

Retrieving history with Code Shovel CodeShovel is able to build a complete history for `CommonUtils::createPattern` on the fly with no previous analysis of the version control repository. Every row in Table 2.1 is returned by the tool, along with the annotations shown in the right-most column in the table.[5] These annotations exist to help developers quickly identify the changes they are interested in (for instance if they wanted to trace how the method moved between files they could just look at the first row when the method was introduced and any subsequent method move entries).

[5]CodeShovel actually reveals more contextual information than displayed here. These information are described in more detail in Chapter 5.

Table 2.1: Actual history of method `CommonUtils::createPattern`. Rows with ⋆ were returned by `git-log`; other rows are false negative results that `git-log` failed to identify. One false positive result returned by `git-log` (2864c10) is not included in the list. CodeShovel correctly identified all results for this method with no false positives or false negatives.

SHA	Date	Nature of change
⋆ ce21086	2017-02-24	Method body change
f65b17c	2015-11-22	Method body change
f2c6263	2015-10-29	Param change + body change
cdf3e56	2015-08-27	File move/rename
ed595de	2015-08-26	File move/rename
081c654	2015-08-01	Exception change
97f0829	2015-03-27	Method body change
ebd4afd	2015-03-24	Method body change
1c15b6a	2015-03-13	Method move to other file
b94bac0	2015-01-11	Param change + body change
f1efb27	2014-02-19	File move/rename
35d1673	2006-07-07	Method body change
e27489c	2005-05-11	Body + return type change + rename
b0db9be	2002-12-08	Method body change
419d924	2002-12-06	Exception change
7b849d5	2002-05-14	Method body change
8d6fa33	2002-01-14	Method move to other file
f0f7f3e	2001-06-28	Method body change
0fd6959	2001-06-22	Method introduced

Chapter 3

Related Work

Source code history has long been recognized in the software evolution community as a key contributor to program understanding and capturing rationale (e.g., [3, 4, 9, 14, 19, 25, 26]).

While all version control systems provide features for tracking the histories of individual lines, line ranges, and files that they are versioning, a number of tools have been built to better help engineers understand the histories of their systems. In this work, our goal is to develop an approach to help developers find comprehensive histories for their code as easily as posible. While there are many dimensions one could analyze, we have chosen to focus on three in particular in this work as we believe they are instrumental in supporting this goal. An overview of these dimensions is given in Table 3.1.

3.1 Analysis burden

Many approaches are up-front analyses that require a complete project to be analyzed before any queries can be issued. These *offline* analyses can usually be queried efficiently once a history has been created, but the up-front cost can require hours of pre-processing before these queries can be handled. These approaches typically require histories to be recomputed after code changes are made as the tools are more geared towards mining-style analyses than answering developer queries. For example, Historage pre-processes a repository, placing each method

Approach	Online	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
CodeShovel	YES	Code	M-Rename M-Signature	F-Rename M-Move

Table 3.1: Selection of tools for examining source code histories. These vary in whether they are online (computing histories on demand or requiring 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 string matching, and *M-Move* denotes move method, pull-up method, and push-down method).

in its own file and then using Git's history mechanism to track changes on each individual method's corresponding file [12]. The up-front pre-processing time allows for essentially instantaneous serving of histories for a given method. In contrast, `git-log` does not require any up-front analysis: when a query is made for a line or range within a file, `git-log` walks back through the file's history to find changes to that line or line range.

3.2 Granularity

Different tools provide answers for historical queries about different granularities of code units. By default, version control systems operate on lines within files. 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", which is supported by Beagle [30]. Tools which support queries on code elements, rather than lines, also support various levels of queries, for instance to classes (e.g., Beagle [30]),

methods (e.g., method_log[1]) or program blocks (e.g., APFEL [33]). Granularities can also vary in terms of time: while most tools in Table 3.1 try to find complete histories, pry-git[2] and Beagle only analyze changes between two specific versions of a program or file. While code-specific support adds additional complexity to these tools, they allow developers to ask queries about how specific units of code have changed over time.

3.3 Transformations

Systems are constantly being evolved, which can cause considerable challenges for history tracking tools. These changes can range from simple single-line code edits to complex refactorings that can involve renaming methods and moving them to new files. Refactorings have been described as the “bread and butter" of software restructuring [1] and the occurrence of refactoring commits is remarkably large. For example, it was shown that 80% of changes to APIs are refactorings [5] and that 19% of method introductions in the PostgreSQL source code were a result of refactorings [9].

Most approaches are able to use textual similarity to 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 (e.g., Historage [12] and C-Rex [11] are able to identify method rename refactorings). Most tools are unable to track changes through inter-file transformations, except for Historage and IntelliJ, which are robust in the face of file rename events, but cannot track other inter-file transformations (like an extract-method refactoring). Unfortunately, such refactorings are also prevalent in practice (e.g. [4, 25]).

3.4 CodeShovel

The approach described in this paper is aimed at quickly providing comprehensive code histories for developers so they can understand how their system has evolved. To do this, we have developed an online approach (to remove the need for time

[1] https://github.com/freerange/method_log, accessed Jan 29 2019
[2] https://github.com/pry/pry-git, accessed Jan 29 2019

consuming pre-processing and to allow for the results to always be up-to-date) that has been tailored for identifying histories for methods (as most program changes happen within these structures) while striving to be as robust as possible to the kinds of transformations that are common in practice (refactorings often cause code histories to be incomplete).

Chapter 4

Industrial Survey

To understand industrial developers' perspectives on code histories, we surveyed them about how they use source code histories, the scenarios in which histories are useful, how easily they can be leveraged with existing tools, and whether histories could be augmented or recast to provide more value. We structured the survey around the following survey research questions:

RQ1 Do developers use source code history when they are working with code? If so, what are they trying to learn when they examine source code history?

RQ2 In terms of their mental models and information needs, what level of temporal and structural granularity are most appropriate when using source code history?

RQ3 How do developers identify the history for specific code units and how well does existing tooling support these queries?

RQ4 How effectively can a language-aware code history viewer support program comprehension?

4.1 Survey Design

Our survey was administered online and consisted of 18 questions arranged in four parts; these parts included likert-scale questions, free-response questions, and two

scenario-based code questions. We also asked participants about their professional background, current job position, development experience, and the version control tools they use. Each survey took approximately 20 minutes to complete.

Part 1 This section was designed to gain insight into RQ1 and RQ2. We asked for a recollection of the participant's last activity with source code history (RQ1). To investigate the structural granularity (first half of RQ2) we let participants rate how interested they are in history at different levels of granularity (*class/module*, *file*, *method/function*, *block*, and *field/variable*). In terms of the temporal granularity (second half of RQ2) we asked for a description on how far in the past they normally examine history and how they decide what time span to look for.

Part 2 This section introduced a brief scenario similar to the one described in Section 2.1: a developer is faced with a pull request and wants to understand better what the code that was changed is actually doing. In this section we wanted to gain more information about whether developers use source code history for program understanding tasks and what information they are searching for (RQ1). We first asked participants how familiar this scenario appears to them and then for a description of how they would approach this problem. We then isolated the imaginary change to a change to a *single method* and asked participants how they would identify changes in history that changed this method as an example of a code unit. We asked how well the participant's described strategy would cope with complex structural changes to this method (e.g. renaming, moving to a different file). Concluding Part 2, we asked how they traced changes to methods with current tooling support and what makes this hard or easy. Responses would provide us a clear picture on whether developers struggle with the tools they use in practice (RQ3).

Part 3 Examined another concrete scenario. We created a sample pull request[1] in our forked repository of the *Checkstyle* project described in Chapter 2 that changed a specific method. We provided a link to the associated file history[2] and asked

[1] https://github.com/ataraxie/checkstyle/pull/1

[2] https://github.com/ataraxie/checkstyle/commits/746a9d/src/main/java/com/puppycrawl/tools/checkstyle/utils/CommonUtils.java

participants to describe how they would identify commits relevant to this method of interest. While we expected that the challenge of this task would become visible from these descriptions we also asked directly how well existing tools support this task (RQ3). We also asked participants about the utility of a code unit history for this scenario to investigate RQ4.

Part 4 While the first three parts of the survey asked about questions about code histories using the participant's own development tools, the fourth part introduced a tool mockup for supporting language-aware historical analysis. We mocked an output for the scenario in Part 3, showing the respective commits and diffs that changed the method alongside detailed descriptions of the changes in natural language. To inform RQ4 and the feasibility of our approach further, we now asked for a rating on (a) how helpful this output would be for a better understanding of the method, (b) how hard it would be to retrieve this type of information with their current tooling and (c) how valuable a tool would be that could generate this output. Finally, we gave participants the option to express what other information could have been valuable that was not in our mocked result view.

4.2 Survey Participants

We recruited 30 professional developers from industry (71%) and 12 from academia (29%) for a total of 42 participants. Participants were selected and contacted individually from the authors' professional networks; 87 individuals were solicited giving a final response rate of 48%. The majority of job titles (64%) 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 (50% > 10 years) and 80% had used source code history for more than 4 years (21% > 10 years). For the professional developers, 63% had been employed in industry for more than 4 years (23% > 10 years).

Table 4.1 shows our participants' most frequently used source code history tools. Other tools mentioned were specific IDE add-ons, GitKraken, and GitLense. From a historical analysis standpoint, all of these tools are built upon the default data provided by the version control log utilities. None of the related tools men-

tioned in Section 3 were mentioned by our participants, which we take as an indication that the domain is currently research-driven.

Tool	Kind	% Respondents
Git (`git-log`)	Command Line	95%
GitHub	Web Service	80%
BitBucket	Web Service	76%
IDE/Editor	Application	71%
SVN (`svn log`)	Command Line	29%
TortoiseGit/TortoiseSVN	Application	17%
SourceTree	Application	17%
Mercurial (`hg log`)	Command line	14%
Other	—	29%

Table 4.1: Most commonly used history navigation tools.

4.3 Survey Results

We summarize the results of the survey organized by our survey research question.

4.3.1 Developers' usage of source code history

RQ1: Do developers use source code history when they are working with code? If so, what are they trying to learn when they examine source code history?

The great majority of our participants frequently use source code history: 76% of participants had used source code history within two days prior to performing the survey (90% < 1 week, 96% < one month). In terms of their most recent history-exploration activity, we first observe many common version control tasks, e.g "check what I modified". We also see many activities associated with accountability; participants checked "who had been contributing", "who [they] could contact for dev support" and "who is associated with [a specific] change".

A large fraction of activities were also related to program understanding. Participants wanted to "understand how the solution to a certain problem was implemented" and "how and why [a property] was changed". Some participants were

interested in the "evolving of software architecture" and in "what steps a certain component took to get to the shape/position it was in". The term *understand* appeared 8 times when they described their most recent activity.

Developers search and navigate extensively through source code history to "find reasons for [...] changes". They "browsed for a change made by a specific commit in the history", "[looked] for a specific change that might have introduced an issue" and "[looked] for an outdated implementation of a functionality". They tried to "[figure] out [a] history of changes" and "[compared] files over multiple commits". The terms *looked*, *browsed*, *searched* and *navigated* appeared 36 times in the descriptions of most recent activities.

When asked how familiar participants were with the scenario in Part 2 (evaluating a pull request), 88% replied with *very familiar* or *familiar*. When asked for a description of their strategy for addressing this scenario, most participants replied that their first step would be to approach the pull request author because "they have to be accountable". Other frequent strategies were to refer to issue trackers because they would tell "what were the design decisions that lead to the change". Others reported that examining version history was their first step: "my first step would be to look at the code history" or "fire up the history view [...] and hope for the best."

4.3.2 Source code granularity

RQ2: In terms of their mental models and information needs, what level of temporal and structural granularity are most appropriate when using source code history?

Figure 4.1 shows how interested our participants were in gathering information on source code history at different levels of *structural granularity*. Participants were at least somewhat interested in all levels but were most interested in *Method/Function* and *Class/Module* with both having roughly 85% in the categories 'Very interested' and 'Interested'. Overall, this shows an interest in examining changes at levels other than just the file or textual range (block) level as is supported by most tools.

To explore the *temporal granularity* participants considered in their exploration, we asked them how far in the past they usually examine and how they

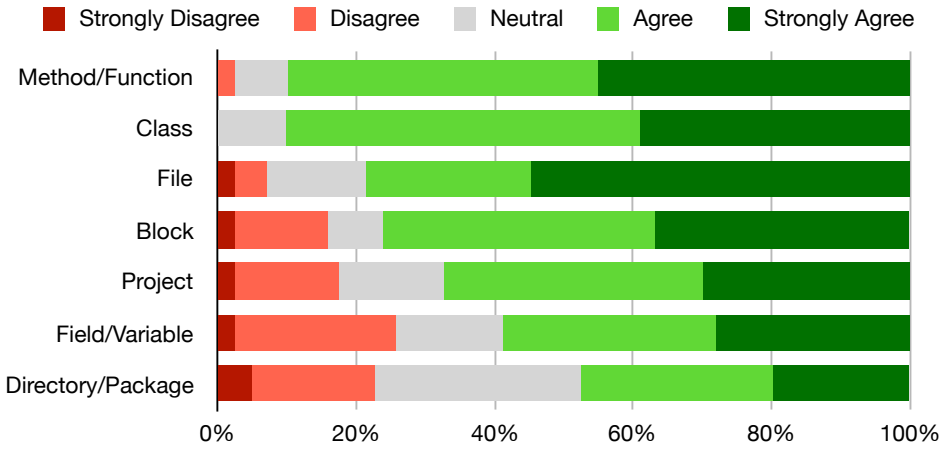


Figure 4.1: Structural granularities most desired by study participants when navigating source code histories.

determine how far they want to go. The responses were mostly consistent: "this largely depends on the goal" and that the time range can vary because the fragment of interest is based on changes and commits rather than time. Both ends of the spectrum are well represented though; some participants "don't usually go [back] more than a few weeks" or "a couple of days" while for other tasks they "look at the commit history years ago". Interestingly, the cases where they examine further in the past seem to be related to program understanding tasks most often, for example when "looking for the reason for a code change", "if some functionality seems odd or obsolete while reviewing source code" or "to understand why and how specific parts became the way they are today".

4.3.3 History and source code units

RQ3: How do developers identify the history for specific code units and how well does existing tooling support these queries?

In the context of the pull request, we asked our participants how they would *generally* identify the history of a method and how challenging this task is. They highlighted *file history* as the most important strategy. Some participants described

how they would limit the history to isolate changes to a method, for example by using a combination of `git log` and `grep`. Many also mentioned recursive or iterative calls to `git blame` and more high-level sources of information like commit messages and issue trackers.

Many participants immediately described this task as "not trivial", "not easy" and "not the funnest job". Rating the difficulty of the task in a later question, 48% chose *hard* or *very hard* while 31% chose *not very hard*, or *not hard at all*.

Figure 4.2 shows participants' ratings of how well their strategies cope with more complex structural changes. Their strategies handled *method renames* and *signature changes* well, but failed to handle *split into multiple methods* and *move to different file*. This was strongly confirmed by the responses to our question about what would make this hard or easy where the main notion was that it depends on the complexity of changes: "It's either easy (the method hasn't undergone complex changes) or challenging (the method has undergone complex changes). When it is challenging, it is VERY difficult." Many participants mentioned specifically that "not knowing the semantics" and the fact that "[there] is no direct or explicit abstraction to method/class levels" makes these historical exploration tasks challenging.

When faced with our real-world example from the *Checkstyle* repository and its history, our participants assessed the task of identifying commits that changed the method as even more challenging than in the previous questions. While some said they could not think of a strategy for this scenario ("I have no idea how I would do this", "Not at all", "I sincerely have no idea"), others gave more advanced strategies ("write a script" or a "divide and conquer mechanism"). Most participants fall back to the history of the file containing the method and "dig through the history to find the method in each commit" and "open each one of them and manually inspect". Some participants immediately pointed out the limits of this strategy: "[if] the method comes from another class [...], that would be more difficult to trace" and that "it's annoying", "there's too much garbage noise" and that "revisions are too many to go through". Some participants wondered whether "there is any tool that allows [them] to track the function across commits".

When asked how well existing tools support identifying these changes, only one participant said *very well*, 13% said *well* and 26% said *neutral*. The majority

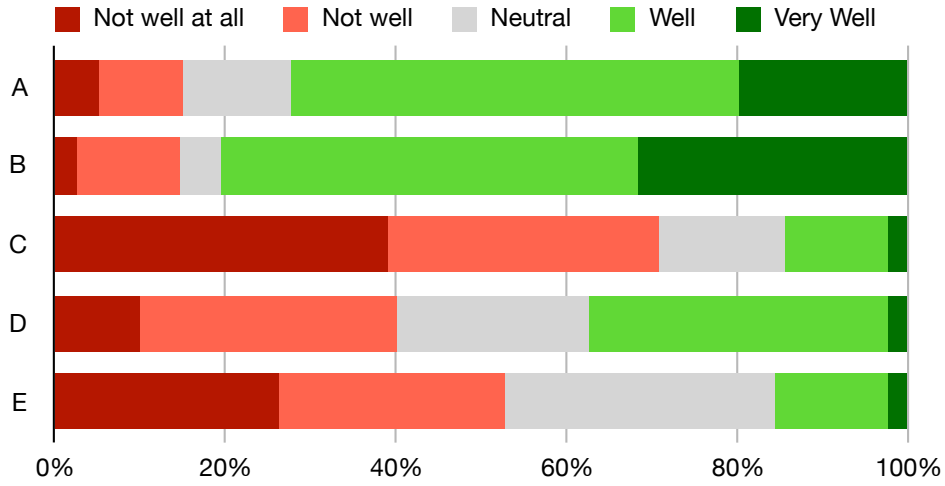


Figure 4.2: How well participants' strategies cope with common code changes. Change A: rename method; Change B: change method signature; Change C: move method to another file; Change D: split method into multiple methods; Change E: combination of previous changes.

of participants rated with either *not very well* or *not well at all* (59%). When asked how hard it would be to find the first commit that really introduced the method (rather than being just a refactoring commit), 76% replied with *hard* or *very hard*.

4.3.4 History tools and program comprehension

RQ4: How effectively can a language-aware code history viewer support program comprehension?

Some participants mentioned the lack of semantic awareness for existing history tools makes it hard to trace the changes of methods ("not knowing the semantics"). Given our tool mockup for a fictitious language-aware history tool, participants rated this tool as *very valuable* or *valuable* (94%) and 79% consider retrieving this information manually *hard* or *very hard*. When asked how helpful they would consider this information for a better understanding of the method being faced with the pull request, 70% replied with *very helpful* or *helpful*.

The positive attitudes towards comprehensive method histories also arose in the

final comments: “having the information [code histories] would be a big improvement in the daily business” and it would be great “to generate a method-, class-, file-history/protocol in a human-readable way” and to have “version control that was method-aware, or aware of classes/modules”. As indicated before in RQ2, our *method* example code unit was confirmed as “definitely the best use case” by some. Some participants commented that sometimes histories are not used “because the tools that [they] use don’t support this feature”.

Chapter 5

Approach

Figure 5.1 provides a high-level illustration of the CodeShovel approach. We partition our description as follows: Section 5.1 describes CodeShovel as a blackbox receiving one method as input and responding with the method history as output (the *Start* and *Return* circles in the figure). We then describe our approach for *finding matching methods* (the left box in Figure 5.1 in Section 5.2 and the underlying *similarity algorithm* in Section 5.3. Our approach for *identifying changes* (the right box in the Figure 5.1) is described in Section 5.4.

5.1 CodeShovel Inputs and Outputs

CodeShovel can be seen as a black box taking a number of arguments describing a method in a repository as input and producing the method's history as output. From a developer's perspective, they simply need to be able to identify the method they want a history for. Programmaticaly, these inputs are:

- *Repository*: Path to a Git version control repository on the local file system.
- *StartCommit*: SHA of the commit to start from and move backwards through history.
- *FilePath*: Path of the file containing the method relative to the root folder of the repository.

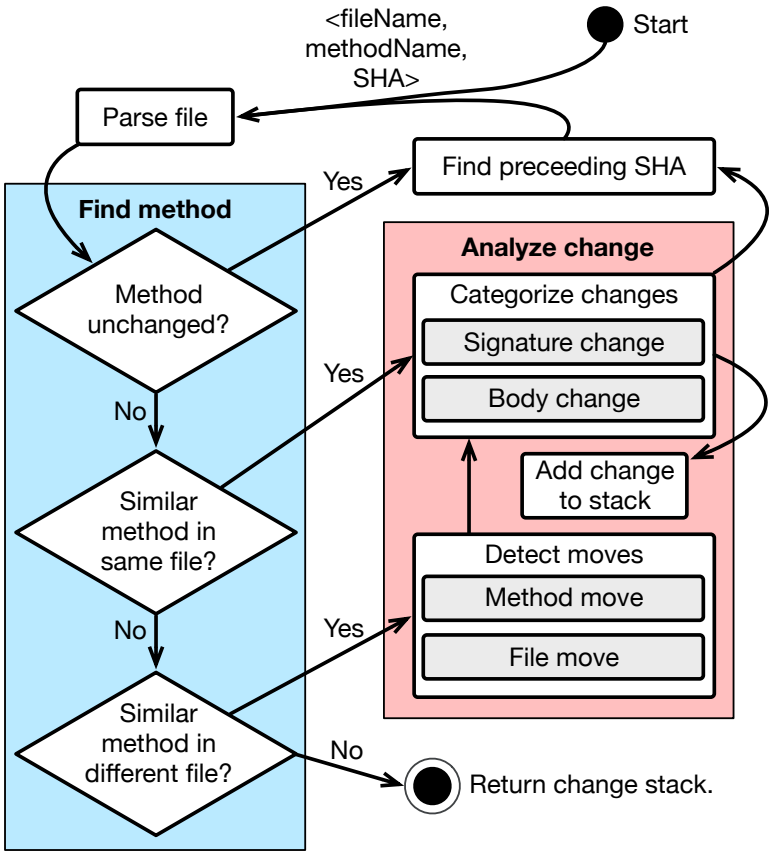


Figure 5.1: High-level approach: each query starts with a method name and SHA. CodeShovel iterates backwards through history until it finds the introducing commit for that method.

- *MethodName*: Name of the method for which the history should be produced.
- *StartLine*: Line number for the start of the method. This is used to differentiate between multiple methods with the same name.

The output of a CodeShovel execution is a JSON file with a list of commits that changed the selected method. Each commit has a change type (see Section 5.4), and includes both basic information like the *commit message, date and author* and more complex information like the *time* and number of *commits* between the

current method change and the previous one. For more complex change types, type-specific information is provided (e.g., for a method move, the old file path and the new file path).

5.2 Method Finding

After the CodeShovel query has been launched, a *method finding* process begins. The file specified by the input *FilePath* is found in the given *Repository* with the specified *StartCommit*. The programming language of the target file is identified from the file extension in the *FilePath*. A language-specific AST parser is instantiated and invoked with the extracted file content, producing an AST for the file. The method node is then identified in the AST using the input *MethodName* and *StartLine*. Using the version control system's built-in historical records, we identify all of the changes to the *FilePath* and process each change in turn from most recent to the oldest to search for the method of interest.

Since source code transformations can cause methods to move through the system, four outcomes are considered in sequence:

1. *Method unchanged*: in this case, the change to the file did not modify the method; CodeShovel can discard this change (with respect to the selected method) and proceed to the next.

2. *Similar method in same file*: in this case, the method was modified, but could be found within the file; this requires *change categorization*.

3. *Similar method in different file*: when methods are removed from a file, we search all other files modified in the same commit to determine where the method came from. This requires *move detection* first and *change categorization* thereafter.

4. *No similar method is found in the same file or a different file*: this indicates that the method was *introduced* in this commit and the CodeShovel execution can return the accumulated list of changes for the method.

We have designed this process with a recursive pattern so that if a similar method is found in a different file (case 3), a new CodeShovel sub-execution is

started for the matched method in the other file. The termination condition for this recursion is an actual *method introduction* being identified in any sub-execution. This processes is robust to an arbitrary number of method move transfomations.

5.3 Similarity Algorithm Design

At the core of the CodeShovel method finding procedure is a similarity algorithm for matching methods across file versions. In many source code history tools (e.g., *git-log*, *Historage* [12]), similarity is computed by matching a method's line range. CodeShovel instead performs the matching using method nodes obtained from ASTs. This matching procedure is based on techniques from clone detection [6, 13, 16, 20, 24, 28]. In search of an accurate and efficient way to match methods, we opted for an approach that combines string matching with the comparison of a set of metrics. For each method matching procedure, CodeShovel computes similarity of the following metrics:

- *Body similarity*: String similarity of the method body.
- *Name similarity*: String similarity of the method name.
- *Parameter similarity*: String similarity of parameter names and equality of parameter types (if language-supported).
- *Scope similarity*: Name equality of the scope of the method, e.g. the class or parent function (only in-file matching).
- *Line similiarity*: Distance between the start line of the two methods (only used for in-file matching).

It has been shown that string similarity is an efficient strategy for clone detection but lacks accuracy in many cases [6, 13, 24]. One approach for mitigating the problem of accuracy without significantly sacrificing efficiency is to combine the similarities of a set of method features and then use these similarities as metrics. Rather than measuring string similarity, we measure *code similarity* with the above-listed metrics [16, 20, 28]. We found that CodeShovel could process

method matchings accurately and efficiently[1] and did not rely on more complex strategies like AST matching techniques as in [2, 7, 8, 10, 23] that have relatively high performance overheads. For more detail into how our similarity algorithm was implemented, please see Section 6.2.

5.4 Change Analysis

Once a similar method has been found by our method matching procedure (left box in Figure 5.1), the method is given to our *change analysis* procedure (right box in Figure 5.1). If the method was found in the same file, changes are categorized immediately. If the method was found in a different file, move refactoring operations are detected first. This can be either a *method move, pull-up method,* or *push-down method* refactoring or a *file move* (or rename) operation. Each commit in the CodeShovel output is associated with one specific change type. If there are multiple change types for one commit, they are processed as a special change type containing multiple single change types. Our categorization of changes is a simplified version of the change taxonomy described by Fluri et. al. [8].

Figure 5.2 shows the hierarchical structure of our change types. At the *top level* is the abstract type *Change* from which the following types inherit:

- *NoChange* is a special type that is used as a flag for commits that did not change a method being analyzed.
- *MultiChange* is another special type that indicates that a commit contained multiple changes and maintains a list of those changes.
- *CompareFunctionChange* is another abstract type from which all actual changes between two methods inherit.
- *Introduced* indicates that a method was introduced in this commit for the first time.

[1] We are currently using the *Jaro-Winkler distance* algorithm [32] for string similarity ratings. In a later stage of the project we have learned from different sources [8, 13] that *n-gram string matching* [31] is the better similarity algorithm for source code. We expect that changing to this technique will further improve CodeShovel's accuracy.

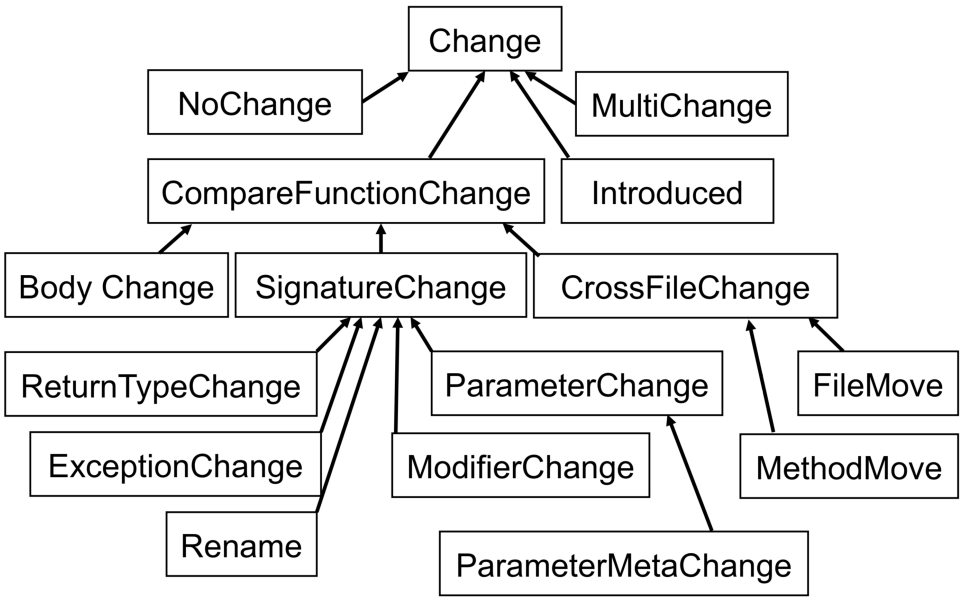


Figure 5.2: Hierarchy of change types in CodeShovel.

Below the abstract type *CompareFunctionChange* are the following change types:

- *BodyChange* indicates that the body of the method was changed. Note that we do not distinguish changes within the body of methods further in order to keep our change hierarchy simple.
- *SignatureChange* is an abstract type indicating that some parts of the method signature have changed.
- *CrossFileChange* is an abstract type indicating a refactoring commit that either moved the method between files or changed the file name or path that contains the method.

The type *SignatureChange* is extended by:

- *ExceptionChange*: exception clause has changed (exceptions were added, removed or edited).
- *ReturnTypeChange*: the method's return type has changed.

- *ParameterChange*: the method's parameters have changed (either in type or name).
- *Rename*: the method has been renamed.
- *ModifierChange*: the method modifier has changed (e.g., from public to private).

The type *ParameterChange* has another subtype named *ParameterMetaChange* that indicates that some parameter meta-information were changed (e.g. `final` keyword added to parameter in Java).

While the previously described changes are related to *change categorization* (upper right box in Figure 5.1), the type *CrossFileChange* represents changes that span multiple files and is therefore related to *move detection* (lower right box in Figure 5.1). Its subtypes are:

- *FileMove* indicates a refactoring operation in either (1) the file name or (2) the file path.
- *MethodMove* indicates a refactoring operation where a method was moved from one file to another.

Note that the significance of the *MethodMove* change type and, in particular, its combination with the *Rename* change type has been previously identified [25]. Consequently, we consider a proper identification of this change type to be an important goal for building comprehensive method histories.

Chapter 6

Implementation

In this section, we describe how we implemented CodeShovel. We first outline the languages and platforms we used, before we illustrate how we implemented the similarity algorithm whose design we described in Section 5.3. We then provide detail on how CodeShovel traverses a method's history and detail the programming API for implementing CodeShovel language adapters.

6.1 Languages and Platforms

CodeShovel is implemented in Java. While CodeShovel is language-aware, the core approach is not language-dependent: given the required AST parser for a language (although written in Java), the implementation is extensible for other languages: all core components with language-specific functionality were realized with abstract classes and interfaces and concrete language-specific implementations. In particular, for adding other *language adapters* the only requirement is an implementation of two CodeShovel interfaces `Parser` and `Method` (see Section 6.4). To perform Git operations and to traverse commits in repositories, CodeShovel leverages the *JGit*[1] library. `JavaParser`[2] is the core of CodeShovel's Java-specific language-adapter while `Nashorn`[3] is used by the JavaScript-specific language-adapter. We have also started to write a Ruby-specific adapter using

[1] https://github.com/eclipse/jgit
[2] https://github.com/javaparser/javaparser
[3] https://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html

jRuby[4] and have found adding support for languages to be as straight-forward as we had hoped with this adapter-architecture.

6.2 Similarity Algorithm Implementation

CodeShovel's similarity algorithm takes a target method and a list of candidate methods as input and produces the method from the candidate list with the highest overall similarity as output if and only if the candidate's overall similarity is above a certain threshold. The algorithm is outlined in Listing 6.1.

Listing 6.1: CodeShovel similarity algorithm

```
IF there is a candidate with the exact same signature
    IF this is an in-file matching
        return candidate
    ELSE IF body similarity is above 0.8
        return candidate

FOREACH candidate c
    IF c has body similarity of 1
        return c
    IF c scope simiarity 1 and body similarity > 0.9
        IF this is a cross-file matching
            return c
        ELSE IF the line number distance < 10
            return c
    Compute overall similarity for candidate and save

IF there is a candidate with the same name
    IF this is a cross-file matching
        return c IF its overall similarity is > 0.8
    ELSE
        return c IF its overall similarity is > 0.5

WITH candidate c having the highest overall similarity
    IF c has body similarity > 0.82
        IF  both bodies of c and the target method have < 4
            lines and < 60 characters
```

[4] https://github.com/jruby/jruby

```
26              return c if it has overall similarity > 0.95
27          ELSE
28              return c if it has overall similarity > 0.82
29
30 RETURN NULL
```

This algorithm has a few key properties:

- The algorithm uses different metrics depending on whether this is an in-file matching or a cross-file matching. For example, if we find a candidate method with the exact same signature in the same file, we can simply return it because it *must be* the same method. On the other hand, if we search for a matching method across other files, a full match of the signature may not be enough (there might be many methods with the same signature in different files, e.g. if a class implements an interface). In this case, we add a threshold to the body similarity.

- The algorithm uses threshold numbers for different metrics that seem arbitrary at first. We derived these numbers from a test suite with a manually prepared oracle for 100 methods from 10 different open source repositories (see Chapter 7). The numbers presented in the algorithm in Listing 6.1 are the result of finding the correct originating commit for all 100 methods. We show in Chapter 7 that these values continue to work for another set of 100 methods from 10 new open-source projects.

- While looping over all candidates, we compute an overall similarity of the candidate if it has not been identified as a match previously with more simple metrics. This overall similarity is currently computed by weighing the different metrics based on the in-file/cross-file switch. These weights are the result of the training phase with the 100 sample methods (see Chapter 7).

- We treat 'short' methods (less than 4 lines and 60 characters) differently in that they need to have a higher similarity to be matched. Not doing so can lead to many incorrect matches for methods with small numbers of statements.

Our matching algorithm need not always return a match (i.e., see `RETURN NULL` in Listing 6.1). If no match is returned, the commit being considered is a *preliminary Introduced* (in the in-file case) or a *final introduced* (in the cross-file case). The latter case is the termination condition for the CodeShovel execution.

6.3 History Traversal

A *CodeShovel execution* uses the inputs described in Section 5.1 and proceeds follows: The file specified by the input *FilePath* is found in the given (local) *Repository* with the specified *StartCommit*. The programming language of the target file is identified from the file extension in the *FilePath*. A language-specific AST parser is instantiated and invoked with the extracted file content, producing an AST for the file content. The method node is then identified in the AST using the input *MethodName* and *StartLine*.[5]

With the identified start method node at hand, the file-level history of the file containing the target method is now created using an abstraction of `git-log`, starting with the parent commit (i.e. the previous commit) of the input *StartCommit*. Proceeding from new to old (i.e. child to parent) for each commit in the file history, the file content at this point in time is parsed with the same language-specific AST parser. For each iteration, a reference to the previously matched method (i.e. in the child commit) is maintained (in the first iteration this is the matched method in the *StartCommit*) and the previous version of the child commit is searched in this parent AST using the *in-file similarity detection procedure* (see Section 6.2).

For each such mapping of the method between commits, there are two cases: (1) a method sufficing the similarity threshold is found or (2) no method sufficing the similarity threshold is found. In the case of (1) a *within-file interpreter* is used to extract the change between the two revisions of the method and the loop can continue. In the case of (2) the result for this commit is a *preliminary Introduced* change type which indicates that no matching method could be found in this commit.

This is where a *cross-file interpreter* comes into play and searches through

[5]Note that the *StartLine* is necessary for languages that allow method overloading. An alternative solution would have been to specify the target method by adding its parameters to the method name. We opted for the former due to its simplicity.

all methods that were removed in this commit.[6] Each of the removed methods is compared with the target method using a *cross-file similarity detection procedure* (see Section 6.2). Again, there are now the two cases described previously: (1) *a method sufficing the similarity threshold is found* or (2) *no method sufficing the similarity threshold is found*. In the case of (1), the commit that was previously interpreted as a *preliminary Introduced* is changed to a new change type of either *FileRename* or *MethodMove* (Or a *MultiChange* that combines one of the two with other simple changes. This could be a method move in combination with a method rename, for example.) After the change type was corrected, a *new CodeShovel execution* is started for the identified method in a different file. In the case of (2) the *preliminary Introduced* change is changed to an actual *Introduced*: we are now certain that this was in fact the commit that introduced the target method. Note that a CodeShovel execution emits a recursive pattern with multiple nested *sub-executions*. The termination condition for this recursive pattern is an actual *Introduced* change being identified.

6.4 Language Adapter API

CodeShovel is extensible for arbitrary programming languages. To implement a language adapter, the developer must implement two Java interfaces (the language adapter itself must be implemented in Java):

- `Parser`: defines how methods can be found and used in a given source file of the language.
- `Method`: defines the characteristics of a method in the given language.

The following Listings 6.2 and 6.3 illustrate the `Parser` and `Method` interfaces. Note that the terms *Method* and *Function* are used interchangeably in these interfaces.

Listing 6.2: CodeShovel `Parser` interface

```
1 Method findFunctionByNameAndLine(String name, int line)
```

[6] Note that this strategy does not deal with scenarios where methods were removed in one commit and then added in a later non-subsequent commit.

```
2 List<Method> findMethodsByLineRange(int beginLine, int endLine)
3 List<Method> getAllMethods()
4 Map<String, Method> getAllMethodsCount()
5 Method findFunctionByOtherFunction(Method otherMethod)
6 boolean functionNamesConsideredEqual(String aName, String bName)
7 Method getMostSimilarFunction(List<Method> candidates, Method
      compareMethod, boolean crossFile)
8 double getScopeSimilarity(Method function, Method
      compareFunction)
9 List<SignatureChange> getMajorChanges(Commit commit, Method
      compareMethod)
10 List<Change> getMinorChanges(Commit commit, Method
      compareFunction)
11 String getAcceptedFileExtension()
```

Listing 6.3: CodeShovel `Method` interface

```
1 String getBody();
2 String getName();
3 List<Parameter> getParameters();
4 ReturnStmt getReturnStmt();
5 Modifiers getModifiers();
6 Exceptions getExceptions();
7 int getNameLineNumber();
8 int getEndLineNumber();
9 String getCommitName();
10 String getCommitNameShort();
11 Commit getCommit();
12 String getId();
13 String getMethodPath();
14 String getSourceFileContent();
15 String getSourceFilePath();
16 String getSourceFragment();
17 String getParentName();
```

The language-specific `Parser` implementation is given the file content in its constructor because the main task is the creation of an AST from this content. The `Method` implementation is given the method node from the language-specific AST implementation as input. As an example, our interface `Parser` defines a method signature `findMethodByNameAndLine(name, line)` which is implemented in

our class `JavaParser` that knows how to find a `Method` instance within a Java file, given its name and start line. Supporting a new language with this interface is relatively straightforward. In addition to our interfaces, we provide two abstract classes `AbstractParser` and `AbstractMethod` that provide generic implementations of most methods in the interfaces that we expect to work for most languages. This reduces the implementation of a *language adapter* for CodeShovel to a manageable set of methods. Both our Java- and (beta) JavaScript-language adapters are implemented in this fashion and have approx. 250 LoC (this is the only language-specific code in the CodeShovel source).

Chapter 7

Empirical Evaluation

In this section we describe how we evaluated the correctness and performance of CodeShovel using 200 methods taken from 20 popular open source Java repositories. Our research questions for this evaluation are as follows:

RQ5 How frequently are methods changed?

RQ6 Are the results returned by CodeShovel correct?

RQ7 Is CodeShovel fast enough for being used as an online tool that does not require preprocessing?

7.1 Subjects

We chose 20 popular open source Java projects of varying size for our evaluation. We chose active repositories having 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 7.1 lists our subjects and their core statistics. (The star annotations in the table are connected to the two analyses phases that will be described in Section 7.2.)

For the empirical study we selected only Java projects since we consider our CodeShovel Java adapter robust enough for this type of extensive evaluation. We are keen to work on our other parsers and reproduce this evaluation once we attribute them a comparable level of robustness.

Table 7.1: Java repositories used for our empirical analysis and their statistics. Repositories annotated with a ⋆ were used in the training analysis (total of 65171 methods) while the rest were used in the validation analysis (total of 110954 methods).

Repository	# commits	# methods	# stars
⋆ checkstyle	8010	3084	3848
commons-io	2123	996	488
⋆ commons-lang	5230	2197	1389
elasticsearch	40353	18261	33640
⋆ flink	14416	17009	4166
hadoop	19805	32888	7801
⋆ hibernate-orm	9100	23159	3318
hibernate-search	6172	5069	283
intellij-community	226106	5946	6335
⋆ javaparser	4781	3613	1883
jetty	15991	11522	2139
⋆ jgit	6065	8277	604
⋆ junit4	2228	1107	6992
⋆ junit5	4695	2078	2323
lucene-solr	30500	29888	1840
mockito	4811	1366	7358
⋆ okhttp	3262	1433	28107
pmd	13360	2567	1738
spring-boot	17818	2451	27527
⋆ spring-framework	17041	3214	22769
TOTAL	451,867	176,125	164,548

7.2 Methodology

We divided our evaluation into two analysis phases with 10 repositories each: a *training analysis* and a *validation analysis*.

In the **training analysis**, we selected 10 repositories at random, and from each of these, we then randomly selected 10 method histories with at least three commits (100 methods in total). We then constructed an oracle for each of these 100

methods to find all of the changes made to these methods. To do this we looked at all CodeShovel results and any additional manual exploration that was needed until we were convinced we found the original commit that introduced the method to the repository. For all CodeShovel results we manually evaluated whether a returned result was a true positive or false positive; for false positives we analyzed the root cause of the problem and fixed the tool. For CodeShovel true positives (including those that were deemed true positives after improving the tool), we created a unit test validating the input data for the given method and the expected result (the unit tests also prevented subsequent regression issues when changing our algorithms). We iteratively performed this process until CodeShovel found all commits in our manually-derived oracle for the 100 methods we were analyzing.

The **validation phase** was performed similarly to the training analysis, except that it was performed after the CodeShovel algorithm could no longer be updated. Again, we randomly selected 10 methods from each of the remaining repositories having at least three commits and manually constructed an oracle containing their complete history. This was done by multiple authors (and one non-author) using a combination of `git-log` and manual inspections; the oracles were independently validated by experienced developers for completeness and correctness. The CodeShovel results for these methods were compared to this oracle and from these we can assess the correctness of our tool (since we know both what all of the right results should be and can also check that the tool did not return any results it should not have).

7.3 Empirical Study Results

We summarize the results of the empirical study by research question.

7.3.1 Frequency of code changes

RQ5: How frequently are methods changed?

Figure 7.1 shows the number of modifications CodeShovel found across all 176,125 methods in 20 repositories. We can see that one-third of methods are introduced and never modified. Exploring these, we found many of these methods

were not part of the project itself but were dependencies imported into the project as source and not subsequently modified. The next most common reason for these unchanged methods is that they are getters or setters which, due to their limited functionality, had little code that needed future modification. Finally, there are methods that are simply never modified, moved, or renamed. These methods are not interesting as they contain no history beyond their introducing commit and so we exclude them from our analysis. They also represent methods for which developers would have no trouble collecting historical information with existing tools (as there is none).

The remaining methods had multiple changes and are precisely the ones whose histories would be most useful to developers as they are more likely to investigate the histories of methods that are actually changing (e.g., high churn methods) than those that do not change. They are also the ones whose introducing commit is more challenging to find since there are more changes to step through. Thus, we subsequently focus our evaluation on methods that had at least three changes.

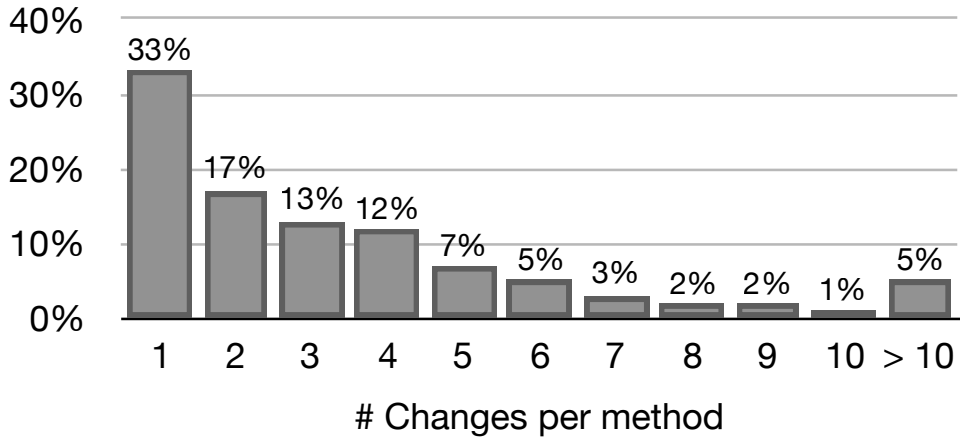


Figure 7.1: The proportion of methods having each number of changes as identified by CodeShovel. This shows that while 33% of methods are only changed once, 50% of methods are changed three times or more.

Answer RQ5: Based on 176,125 methods in our analysis, about one third of methods are introduced and never modified. Another third are changed one to three times after being introduced. Most of the remaining third of methods are modified less then ten times with only 5% being modified more than 10 times.

7.3.2 CodeShovel correctness

RQ6: Are the results returned by CodeShovel correct?

To examine the *completeness* and *correctness* of the histories found by CodeShovel, we compared the histories produced by CodeShovel with the histories in the oracle for each of the 100 methods in our validation phase. The history of 7 methods could not be determined by CodeShovel because these files were not parsable. We found that CodeShovel correctly identified the histories for 85 of the 93 remaining methods (91%).

CodeShovel was not able to find the correct introducing commit for 8 methods. In these cases, the diff contained several changes for the method making it somewhat ambiguous as to whether the method should be considered a new method or an extensively transformed version of an existing method. For example, Figure 7.2 shows a diff from one of the 8 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 CodeShovel to report that the `bindMatchers` method was introduced in this commit instead of reporting that it was a transformed version of `create`. In our training phase, we established the threshold values for CodeShovel's similarity algorithm to penalize simultaneous changes. We found that allowing the similarity algorithm to consider too many simultaneous changes significantly increased the false-positive rate. Further tuning of the threshold values, along with additional metrics would adjust how CodeShovel interprets these changes.

```
- 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;
  }
```

Figure 7.2: A diff showing a complex method transformation. The `create` method was renamed to `bindMatchers`, the parameters were changed, an exception signature was added, and the creation `Invocation` was removed from the body.

Answer RQ6: Based on the 100 methods analyzed in our validation phase, CodeShovel is able to correctly determine the complete history of 91% of methods.

7.3.3 CodeShovel performance

RQ7: Is CodeShovel fast enough for being used as an online tool that does not require preprocessing?

To evaluate the performance of CodeShovel's online similarity algorithm, we recorded the execution time for each of the methods across the 10 repositories used in the validation phased (total of 110,954 methods). We collected the runtimes on a development computer (12-core processor running at 3.30GHz with 32GB memory). Figure 7.3 shows the median execution time for all methods in each repository (point) as well as the overall median execution time for all methods.

Overall, the performance is sufficient for interactive use with a median runtime under 2 seconds and with 90% of the methods returning in less than 10 seconds. The `intellij-community` repository is the outlier with a median execution time of about 7 seconds, which was due to a combination of large source files (which take longer to parse) and a high frequency of change within these files (which increases the number of times the parser is invoked).

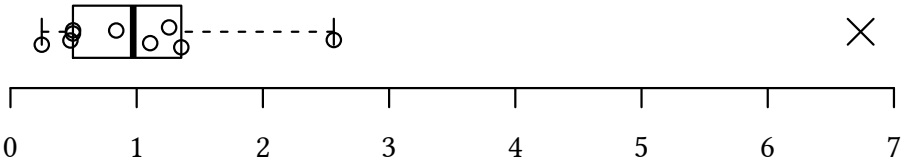


Figure 7.3: The median time it took CodeShovel to process all methods in each repository (point) listed in Table 7.1. The overall median runtime is under 2 seconds; the outlier is the `intellij-community` repository which has many large and frequently changing source files.

Answer RQ7: Based on the 110,954 methods analyzed in the validation phase, CodeShovel produces correct and complete method histories with a median under 2 seconds.

7.4 Change Tagging

We examined the proportion of the method changes that were tagged with each change type. Table 7.2 shows the proportion of methods that received each tag. The sum of these proportions is > 100% because methods can receive multiple tags (e.g., they can have their method body changed and get a new parameter in the same change).

While body changes do comprise the majority of changes, we believe this also makes it easier for the developer to filter these out if they are more interested in refactoring and restructuring tasks. Similarly, if developers want to only focus on those changes, they can elide many results that would otherwise not be of interest to them.

7.5 Comparison with `git-log`

Due to the popularity of the `git-log` command and its integration in many advanced history viewers, we compared CodeShovel with the basic command-line

Table 7.2: Proportion of methods tagged with each change type. These add up to > 100% because a change can be tagged with more than one kind of change. Changes, in the bottom half of the table modify the method signature.

Change type	% of changes
Modify method body	59.8
Initial method introduction	20.1
Rename file or change file path	14.3
Extract method refactoring	1.8
Parameter change (type, name, or order)	11.4
Modifier change	3.4
Method rename refactoring	2.4
Return type modification	0.9
Exception signature changes	0.6

version of `git-log`[1] for the 20 repositories used in our evaluation.

Figure 7.4 shows the proportion of history missing by `git-log` with a point for each analyzed project. For example, for a method that had 1,000 days of history with CodeShovel, `git-log` was only able to find 510 days worth of history on average.

Analyzing the *correctness of git-log*, we identified *three main categories* of scenarios in which `git-log` failed to produce correct results:

1. A file is renamed or its path changes (due to moving of the file or directory/-package renaming). While `git-log` has a `-follow` flag which enables the tracking of changes to a file through file renames, this option is not available in combination with the `-L` flag for line ranges. So if a developer is only interested in the changes to a specific segment of code, there is no option to see changes to this segment across file renames or path changes. This would mean that `git-log` will simply discontinue a method's history for these changes.

2. A method is moved to another file. This is a change not detected by the

[1] i.e. `git log -L METHOD_BEGIN_LINE,METHOD_END_LINE:FILEPATH`.

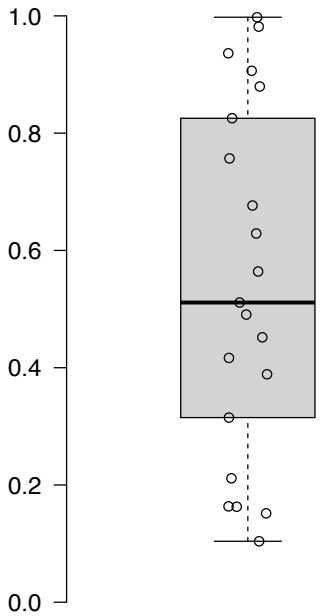


Figure 7.4: Fraction of CodeShovel data returned by `git-log` as a fraction of total duration of results. For example, for a method that had 1,000 days of history with CodeShovel, `git-log` was only able to find 510 worth of history (on average).

`git-log` command in general.

3. A method is moved within the file. In this case the baseline will continue the history of the line range from which the method was moved, although this line range now continues a new fragment of source code.

It should be noted here that, rather than `git-log`, the `git-blame` command enables to follow the moving of methods in-file and cross-file to some degree with the `-M` and `-C` flags that work in combination with the `-L` flag. However, the purpose of `git-blame` is to annotate each line in the code segment with the commit and author that changed the line most recently. There is no history functionality associated with this command, although there are approaches available that create such a history using a recursive call chain of `git-blame` (e.g. *SmartGit*[2]).

[2]https://www.syntevo.com/smartgit/, accessed 2019-06-03

Analyzing the *correctness of CodeShovel*, we found almost all of the incorrect results being related to our *method similarity algorithm* (see 5.3). While in some cases there is room for discussion on whether one method A in commit X really continues the history of method B in commit Y, other cases are obvious mistakes that we consider false positives from our tool. While we managed to eliminate all false positives from the 100 method histories in our training phase by improving the tool, we still saw a number of erroneous changes in the histories analyzed in the validation phase (see Section 7.3.2).

Chapter 8

Industrial Field Study

Our empirical analysis in Chapter 7 showed that CodeShovel provides complete and correct results with acceptable performance for our set of sample methods from open-source projects. To ensure the correctness (**RQ6**) and performance (**RQ7**) of CodeShovel translates to closed-source, industrial code bases, we ran the tool on participant-selected methods from their own industrial projects and had them independently verify that the generated histories were correct. As a follow-up to the industrial survey (Section 4), we also elicited participant insight into how they would apply CodeShovel in their industrial setting. For this purpose, we aim to additionally answer one research question:

RQ8 In which scenarios are method-level histories useful to industrial developers and why?

8.1 Study Participants

We conducted our field study with 16 developers at a medium-size (approximately 60 employee) software company in Munich, Germany.[1] Since the current version of CodeShovel is most stable for Java methods, developers were required to have some industry Java background, and to be able to provide a set of Java methods with whose histories they were familiar. Participants had a median of 10 years of

[1] 4 of the 16 particpants had also participated in the survey described in Section 4.

programming experience, 3.5 years working as professional software developers, and 8.5 years experience with version control. Each participant was given a 10 Euro coffee gift cards as compensation for their time.

8.2 Study Design

We designed our field study as on-site interview sessions lasting approximately 45 minutes per participant. Participants were asked to chose 2-4 Java methods from their own repositories that they were familiar with and that had been revised multiple times. We queried CodeShovel using each of these methods on the participant's computer and recorded the runtime. We then had the industrial developers evaluate the correctness of the results.

After we sent each participant the CodeShovel executable and had the parcitipants run it on each of their methods. For each method, the participant then sent the JSON result file to the moderator. For each commit in the result file, the process was then:

1. The participant opened the commit in the history viewer of their choice on their machine.
2. We summarized the information in the CodeShovel output (and especially the change types described in Section 5.4).
3. The participant looked in their history viewer and verified the information.

After this process, we asked participants three questions:

Q1 Was the method history correct? In particular, was the method really introduced in the first commit?

Q2 For what scenarios do you think CodeShovel would be useful?

Q3 Is the information produced by CodeShovel helpful?

Finally, we asked participants about their professional experiences with source code histories before concluding the interview.

8.3 Study Results

We summarize the results of the field study by the associated research questions RQ6, RQ7, and RQ8.

8.3.1 CodeShovel correctness

RQ6: Are the results returned by CodeShovel correct?

We produced 45 method histories in our field study. CodeShovel found the correct introducing commit for 41 methods (91%).[2] For the four methods for which CodeShovel was not able to find the introducing commit, two had commits containing multiple changes causing the overall similarity to be below the matching threshold, one contained a non-parsable file, and for one we could not reproduce the problem. Otherwise, participants confirmed that CodeShovel performed well and found the relevant commits without including any unrelated commits.

Answer RQ6: Based on the 45 participant-selected methods analyzed in our field study, CodeShovel is able to correctly determine the complete history of 91% of methods. This conforms to the results of our empirical results in Section 7.3.2.

8.3.2 CodeShovel performance

RQ7: Is CodeShovel fast enough for being used as an online tool that does not require preprocessing?

Figure 8.1 shows the runtimes of CodeShovel on 42 of the 45 methods (we forgot to record the runtimes for three methods). The outlying method that took 8 seconds to execute was due to the method being changed 44 times requiring a large file be parsed as many times. Overall, the median runtime was less than 2 seconds showing that CodeShovel's online algorithm is fast enough on an industrial codebase for interactive use.

[2]The alert reader may notice that this number exactly coincides with the results of our empirical evaluation (see Section 7.3.2).

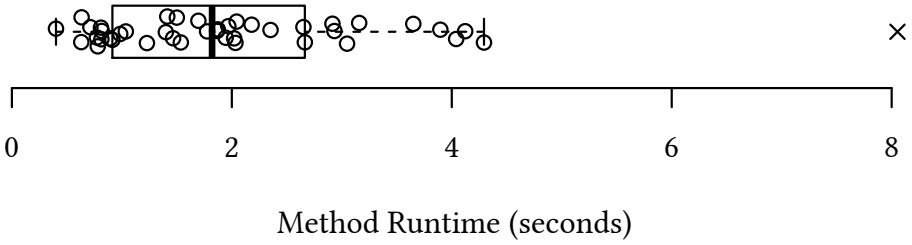


Figure 8.1: CodeShovel runtimes on 42 methods from an industrial codebase. The slow performance of the outlying method was due to the containing file needing to be parsed a large number of times.

Answer RQ7: Based on the 45 participant-selected methods analysed in our field study, CodeShovel produces correct and complete method histories with a median under 2 seconds. This conforms to the results of our empirical results in Section 7.3.3.

8.3.3 Scenarios for method-level histories

RQ8: In which scenarios are method-level histories useful to industrial developers and why?

We were also interested for which scenarios professional developers would use CodeShovel. To help facilitate the discussion, we asked the participants after they had seen the history generated by CodeShovel for their methods. Participants described several scenarios in which CodeShovel would be useful, and to understand these scenarios more generally, two authors independently open coded the transcribed responses.

The history generated by CodeShovel allows developers to determine a method's ***provenance*** because they "*can see easily who introduced a method*" *(P3)* and 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)*, and so developers can "*focus on moves and other refactoring operations that would not be tracable 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)*. They also thought it would be useful for "***code understanding** for code you're not used to*" *(P10)* so that "*one can learn more about the codebase in an easy way*" *(P7)*. And, because developers "*already do what this tool is doing, we just do it manually*" *(P8)*, CodeShovel ***automates history-related tasks***.

Participants also offered some suggestions as to how CodeShovel could be improved: it "*definitely **needs a UI***" *(P2)* that should be integrated into the IDE so it doesn't "*take me out of my workflow*" *(P16)*, and that tracking "***method-level annotations** would be valuable*" *(P1)*.

Answer (RQ8) Participants provided several scenarios in which they thought CodeShovel would be useful. From these, we identified provenance, traceability, onboarding, code understanding, and automation as the most important aspects. Participants also noted the need for a UI and support for method-level annotations before they would consider adopting CodeShovel.

Overall, participants rated the method histories (including change tags) as very helpful (7/16, 43%), somewhat helpful (6/16, 37%), or neither helpful nor unhelpful (3/16, 19%). No participants rated the information as unhelpful. Two of the participants who rated CodeShovel as neither helpful nor unhelpful were either not active developers or had not yet worked on a "big" project.

Chapter 9

Discussion

In this section we discuss our threats to validity and future work.

9.1 Threats to Validity

In the following, we internal and external threats to validity.

9.1.1 Internal Validity

The primary threat to the internal validity of our empirical study is the construction of our oracle. While we had at least one external developer verify the method histories we manually generated, it is possible that some histories in the oracle were incorrect or incomplete. This is especially true for the introducing commit which was found on a best-effort basis since it is not feasible to manually examine all commits in a repository.

Another threat is due 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.

Moreover, both the survey and the field study there is a certain degree of moderator bias, since participants' were selected from the authors' personal networks. We claim though, that it is very unlikely that participants—mostly experienced

software developers—responded in favor of the author just to influence the results positively. This is especially true for free text questions, where content would have to be "made up" by participants.

9.1.2 External Validity

The primary threat to external validity in our empirical study is our sample size. While we evaluated CodeShovel on both open-source and closed-source industrial codebases, we manually evaluated the correctness of the approach on a limited number of methods. In both the evaluation phase and the field study, this was done due to time constraints, and we acknowledge that our findings may not generalize.

In our survey and field study, the participants we recruited through may not be representative of all developers in practice. We sought to reduce this threat by having a fairly large number of participants (42 in the survey and 16 in the field study).

Furthermore, we acknowledge that our experiments were run only on Java projects. Although we claim that CodeShovel is easily extensible to other languages, we have only started to implement other language adapters (currently JavaScript and Ruby). So far though, we have only evaluated our tool on the Java programming language, which may not generalize to many other languages.

9.2 Future Work

There are a variety of ways for improving CodeShovel in the future. These ideas stem from participants' ideas in our survey and field study, as well as from our own evaluation.

9.2.1 Interaction with CodeShovel

Presently, developers must interact with CodeShovel via a command-line interface. We see a proper user interface as a logical next step for practical usage, a sentiment echoed by many of the developers who took part in our field study (Chapter 8). Some further suggested that CodeShovel should be integrated into the IDE to avoid context switches, or to be shown as a timeline with clickable commits. An option to reorder the commits from old to new was also suggested.

We are currently working on running CodeShovel as a web service with UI: users point the service to their Git repository and are then guided to choose their method of interest. The history of the given method is then shown in the UI (after a processing time that was evaluated in previous sections) and the user can navigate through the changes. We expect that the completion of this service, and the associated transformation from command-line tool to UI-based web service will be a significant step towards practical usage of CodeShovel.

9.2.2 Integration of CodeShovel in other software

Since CodeShovel is an open source Maven project, the CodeShovel Java API can already be used in other Java projects. For example, a Java-based Git hosting service could add a method history feature to its UI which will then trigger CodeShovel procedures on the backend. This process will also be enhanced by said web service that is currently in development: rather than the CodeShovel Java API, any platform can then interact with the REST endpoints of our service using HTTP. This enables integration of CodeShovel in projects not written in Java (but even Java projects may choose to use the REST API for architectural reasons).

9.2.3 More code granularities and change details

Developers who took part in our survey (Section 4) said that being able to navigate history at both the method-level and the class-level would be most useful. Currently, CodeShovel only supports method-level navigation but it would be relatively straight forward to track non-method blocks (e.g., fields and constructors) and include them with the aggregate of all method histories in a particular class, to provide support for class-level navigation.

We would also like to inspect the body of methods more in order to provide more details on changes that involve changing of the body of methods (by far the most present change type in Section 7.4). So far, we have not interpreted changes within method bodies for performance reasons (after all it is what makes related tools using ASTs perform considerably slower). More investigation and runtime analysis may direct us to a strategy to do such interpretations with less performance costs. We would also like to provide more detailed descriptions of changes to

method metadata (e.g., method annotations in Java). One participant also suggested call-site analysis for method changes; i.e. CodeShovel could also analyze where methods are called throughout the code base and how this changes over time.

9.2.4 Source code statistics

Several participants described a desire to see more statistics for method histories; for example, statistics could indicate that a method needs refactoring or one could infer from method histories what features were robust and which ones needed continuous repair and improvement. This would move us to a meta-analysis level of method history data that is very interesting.

9.2.5 CodeShovel robustness

Finally, CodeShovel currently fails to produce method histories as soon as it encounters a file revision that cannot be parsed While we would like to deal with this problem more gracefully, we can also see from our evaluation that this does not occur often in practice. Still, it would be a great improvement to skip over such "unparseable" commits and continue history traversal.

Although we consider CodeShovel's measured correctness of 91% a good result, we would certainly like to evaluate our tool with more methods and languages: more language adapter need to be implemented and more methods need to be evaluated. With the concept of the training phase and the associated unit tests (see Chapter 7) we are certain that we can make CodeShovel produce the correct history of almost any method in the world.

Chapter 10

Conclusion

Source code histories are valuable sources of information about how a system has evolved. In this paper, we described a formative survey with 30 industrial and 12 academic developers to learn how they use source code history and what challenges they face when doing so. Through this survey, we learned that existing tools do not effectively surface the results developers need to answer their development questions. To address this, we built CodeShovel, a tool that uses a combination of AST parsing and metrics-based string similarity to analyze method-level source code changes. With an empirical analysis across 10 open-source projects, we demonstrated that CodeShovel can return complete method histories in more than 90% of cases with a median execution time of less than 2 seconds. A field study with 16 industrial developers confirmed these results translate to industrial code bases and from these developers we determined that CodeShovel would be useful for a wide range of industrial development tasks. We have a wide range of ideas to improve CodeShovel in the future. Most importantly, would like to extend our tool to support more programming languages and provide an appropriate user interface for practical usage. Further evaluation, both in the lab and in the field, will make our findings more generalizable and our tool even more accurate. In the end, we believe the results that CodeShovel returns quickly and reliably will be helpful for future developers as they investigate the history of their source code.

Bibliography

[1] . *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2. → page 12

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998. doi:10.1109/ICSM.1998.738528. → page 27

[3] A. W. Bradley and G. C. Murphy. Supporting software history exploration. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 193–202, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi:10.1145/1985441.1985469. URL http://doi.acm.org/10.1145/1985441.1985469. → page 10

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. doi:10.1145/353171.353183. URL http://doi.acm.org/10.1145/353171.353183. → pages 10, 12

[5] D. Dig and R. Johnson. The role of refactorings in api evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi:10.1109/ICSM.2005.90. URL http://dx.doi.org/10.1109/ICSM.2005.90. → page 12

[6] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching: Research articles. *J. Softw. Maint. Evol.*, 18

(1):37–58, Jan. 2006. ISSN 1532-060X. doi:10.1002/smr.v18:1. URL http://dx.doi.org/10.1002/smr.v18:1. → page 26

[7] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi:10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982. → page 27

[8] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007. ISSN 0098-5589. doi:10.1109/TSE.2007.70731. URL http://dx.doi.org/10.1109/TSE.2007.70731. → page 27

[9] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. volume 31, pages 166–181, 02 2005. doi:10.1109/TSE.2005.28. URL doi.ieeecomputersociety.org/10.1109/TSE.2005.28. → pages 10, 12

[10] M. Hashimoto and A. Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288, Oct 2008. doi:10.1109/WCRE.2008.44. → page 27

[11] A. E. Hassan and R. C. Holt. C-rex : An evolutionary code extractor for c. 2004. → page 12

[12] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 96–100, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0848-9. doi:10.1145/2024445.2024463. URL http://doi.acm.org/10.1145/2024445.2024463. → pages 11, 12, 26

[13] Johnson. Substring matching for clone detection and change tracking. In *Proceedings 1994 International Conference on Software Maintenance*, pages 120–126, Sep. 1994. doi:10.1109/ICSM.1994.336783. → pages 26, 27

[14] H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *2008 IEEE International Conference on Software Maintenance*, pages 157–166, Sep. 2008. doi:10.1109/ICSM.2008.4658064. → page 10

[15] 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)*, pages 344–353, 2007. ISBN 0-7695-2828-7. doi:10.1109/ICSE.2007.45. → page 1

[16] Kodhai and Perumal. Clone detection using textual and metric analysis to figure out all types of. 2011. → page 26

[17] 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)*, pages 1028–1038, 2016. doi:10.1145/2884781.2884840. URL http://doi.acm.org/10.1145/2884781.2884840. → pages 1, 5

[18] 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)*, pages 492–501, 2006. → page 1

[19] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi:10.1145/1134285.1134355. URL http://doi.acm.org/10.1145/1134285.1134355. → page 10

[20] Mayrand, Leblanc, and Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253, Nov 1996. doi:10.1109/ICSM.1996.565012. → page 26

[21] A. Mockus and J. D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 503–512, 2002. → page 1

[22] 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)*, pages 284–292, 2005. doi:10.1145/1062455.1062514. URL http://doi.acm.org/10.1145/1062455.1062514. → page 1

[23] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011. ISSN 2150-8097.

doi:10.14778/2095686.2095692. URL http://dx.doi.org/10.14778/2095686.2095692. → page 27

[24] F. V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 336–339, Sep. 2004. doi:10.1109/ASE.2004.1342759. → page 26

[25] F. V. Rysselberghe, M. Rieger, and S. Demeyer. Detecting move operations in versioning information. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 8 pp.–278, March 2006. doi:10.1109/CSMR.2006.23. → pages 10, 12, 29

[26] F. Servant and J. A. Jones. History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi:10.1145/2393596.2393646. URL http://doi.acm.org/10.1145/2393596.2393646. → page 10

[27] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 139–145, 1998. ISBN 0-8186-8779-7. doi:10.1109/ICSM.1998.738502. → page 5

[28] M. Sudhamani and L. Rangarajan. Structural similarity detection using structure of control statements. *Procedia Computer Science*, 46:892 – 899, 2015. ISSN 1877-0509. doi:https://doi.org/10.1016/j.procs.2015.02.159. URL http://www.sciencedirect.com/science/article/pii/S1877050915002239. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India. → page 26

[29] 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)*, pages 563–572, 2004. ISBN 0-7695-2163-0. → page 1

[30] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Proceedings 10th International Workshop on Program Comprehension*, pages 127–136, June 2002. doi:10.1109/WPC.2002.1021334. → page 11

[31] 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*, 10:253–260, 07 1974. doi:10.1016/0020-0271(74)90020-5. → page 27

[32] W. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. *Proceedings of the Section on Survey Research Methods*, 01 1990. → page 27

[33] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 16–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-621-1. doi:10.1145/1188835.1188839. URL http://doi.acm.org/10.1145/1188835.1188839. → page 12

Appendix A

Survey

A.1 Survey Form

The following pages show the form that was presented to the participants of the survey described in Section 4.

Section 1: Source Code History

In this section we aim to collect a general understanding how developers use source code history. By *source code history*, we are referring to any activity, system, or tool associated with past changes to source code; common examples are the history of a specific file, commit diffs or pull requests.

Q1.1 How recently did you last use source code history of any kind?

- O Less than 2 work days
- O Less than 1 week
- O Less than 1 month
- O Less than 1 year
- O More than 1 year
- O can't remember

Q1.2 Please describe this most recent activity. How did you use source code history? What were you looking for? Did you find it? Did the tools you used support you in this investigation or could they be improved?

Q1.3 In terms of source code granularity, how interested are you in gathering information on source code history at the following levels?

	Very interested	Interested	Neutral	Not very interested	Not interested at all	Don't know
Project	O	O	O	O	O	O
Directory/Package	O	O	O	O	O	O
File	O	O	O	O	O	O
Class/Module	O	O	O	O	O	O
Field/Variable	O	O	O	O	O	O
Method/Function	O	O	O	O	O	O
Block*	O	O	O	O	O	O

* By *block* we are refering to a group of declarations or statements that we commonly see between curly braces ({}) or keywords like *begin*/*end* in programming languages.

Q1.4 When you use code history, how far in the past do you usually examine? How do you determine how far in the past you want to go?

Section 2: pull request scenario

Section 2: Pull Request Example

For the following questions, imagine the following situation: you are reviewing a change to a code fragment in a pull request* but you are not certain about what the code is actually doing. Your goal is to better understand the code and what led to the change being made.

**Pull request* is a common term in version control for the activity of a source code contributor requesting that a project maintainer merges a change into the code base of the project. If you are unfamiliar with this terminology, you can simply assume a simple change to the source code base that you are reviewing.

Q2.1 Does this scenario sound familiar to you (i.e. have you encountered this in the past)?

Very familliar	Familiar	Neutral	Not very familiar	Not familiar at all	Don't know
○	○	○	○	○	○

Q2.2 Please describe very briefly how you would approach this problem. What kinds of questions would you like to answer? What tools or approaches would you use to answer them?

Suppose the change you are reviewing is related to a single method. You want to understand this method better and what led to it being changed.

Q2.3 Using source code history, how would you find changes to **this method only**? Please describe briefly.

Q2.4 How well would your strategy cope with more complex structural changes, e.g. method renaming, moving of a method, refactoring?

	Very well	Well	Neutral	Not very well	Not well at all	Don't know
Renaming of method	○	○	○	○	○	○
Signature changes (parameters, return type)	○	○	○	○	○	○
Move to a different file	○	○	○	○	○	○
Splitting into multiple methods	○	○	○	○	○	○
Combinations of the previous	○	○	○	○	○	○

Q2.5 Using current tooling support, how hard is it generally to trace changes to a specific method?

Very hard	Hard	Neutral	Not very hard	Not hard at all	Don't know
○	○	○	○	○	○

Q2.6 Given your answer to the previous question (Q2.5), what makes this hard or easy?

Section 3: Historical Scenario Overview

Section 3: Specific Scenario - Overview

We have chosen a specific scenario that illustrates how source code history relates to development in practice. Please read the description below and answer the questions that follow. Please allow a few minutes and click on the links provided to understand the scenario better. The choice of the Java language for the example is arbitrary and does not require Java experience.

Example Scenario

Imagine yourself in the dev team of Checkstyle, a popular syntax checker for Java. You are to review this pull request with a change to the method *CommonUtils.hasWhitespaceBefore*. In order to review this pull request, you want to get a better picture on this method and how it has changed over the past. You decide to look into the history of the file *CommonUtils.java* as seen here. You discover that this file has a history of 47 revisions in 3 years.

Q3.1 In the above version history, how would you identify the commits in which the method of interest has changed? Please describe your strategy briefly.

Q3.2 How well do existing tools support identifying these changes?

Very well	Well	Neutral	Not very well	Not well at all	Don't know
O	O	O	O	O	O

Q3.3 How useful would it be to have support for a more semantic history in this scenario (e.g. history for this method or class only)?

Very useful	Useful	Neutral	Not very useful	Not useful at all	Don't know
O	O	O	O	O	O

Q3.4 How hard would it be to find the first commit for the given method and whether the method was really created then or if it was moved there from somewhere else (e.g. through a file renaming, or through a refactoring)?

Very hard	Hard	Neutral	Not very hard	Not hard at all	Don't know
O	O	O	O	O	O

Section 4: Historical Scenario Detail

Section 4: Historical Scenario Detail

For the same real world example as above (Checkstyle pull request), we have analyzed the version control history of the method of interest. The following descriptions and diff snippets show where and how the method has been changed over the past. Please have a look at these and answer the questions that follow.

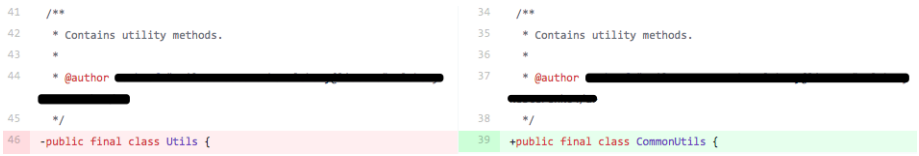
Example Scenario (details)

The history of the file CommonUtils.java shows that a refactoring commit on Aug 28 2015 (46a52f8) renamed the method *whitespaceBefore* to *hasWhitespaceBefore:*

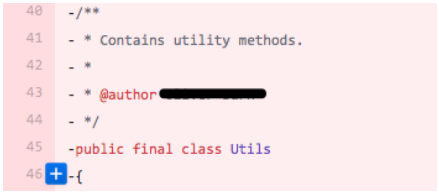
```
93  -    public static boolean whitespaceBefore(int index, String
    line) {
94          for (int i = 0; i < index; i++) {
95              if (!Character.isWhitespace(line.charAt(i))) {
96                  return false;
```

```
93  +    public static boolean hasWhitespaceBefore(int index, String
    line) {
94          for (int i = 0; i < index; i++) {
95              if (!Character.isWhitespace(line.charAt(i))) {
96                  return false;
```

This is the third-oldest commit in the file's history. The message of the oldest commit in the file's history on Aug 26 2015 (cdf3e56) is "Utils class has been splitted to CommonUtils and TokenUtils". The diff confirms that a file Util.java was split into these two separate files CommonUtils.java and TokenUtils.java and that the method *whitespaceBefore* came from this file:



Inspecting the history of Utils.java reveals 41 revisions throughout the year 2015. However, in the oldest commit from Jan 21 2015 (204c073), the method *whitespaceBefore* was not present in this file. Searching for the commit that introduced the method reveals a commit from March 15 2015 (1c15b6a) with the message “move all methods from checkstyle.api.Utils to checkstyle.Utils”. Again, this was a refactoring commit that combined two classes with the same name (Utils.java) to one file:



```
72  -    /**
73  -     * Returns whether the specified string contains only whitespace up to the
74  -     * specified index.
75  -     *
76  -     * @param index index to check up to
77  -     * @param line the line to check
78  -     * @return whether there is only whitespace
79  -     */
80  -    public static boolean whitespaceBefore(int index, String line)
81  -    {
82  -        for (int i = 0; i < index; i++) {
83  -            if (!Character.isWhitespace(line.charAt(i))) {
84  -                return false;
85  -            }
86  -        }
87  -        return true;
88  -    }
```

```
40  /**
41   * Contains utility methods.

45  public final class Utils
46  {
```

```
108 +    /**
109 +     * Returns whether the specified string contains only whitespace up to the
110 +     * specified index.
111 +     *
112 +     * @param index index to check up to
113 +     * @param line the line to check
114 +     * @return whether there is only whitespace
115 +     */
116 +    public static boolean whitespaceBefore(int index, String line)
117 +    {
118 +        for (int i = 0; i < index; i++) {
119 +            if (!Character.isWhitespace(line.charAt(i))) {
120 +                return false;
121 +            }
122 +        }
123 +        return true;
124 +    }
```

The history of the old Utils.java file from which the method came reveals 69 revisions with the first one dating back to Feb 20 2002 (e10faf3). The details of this commit show that the method *whitespaceBefore* was introduced in this commit for the first time.

```
+final class Utils
+{
+    /** stop instances being created **/
+    private Utils()
+    {
+    }
+
+    /**
+     * Returns whether the specified string contains only whitespace up to the
+     * specified index.
+     *
+     * @param aIndex index to check up to
+     * @param aLine the line to check
+     * @return whether there is only whitespace
+     */
+    static boolean whitespaceBefore(int aIndex, String aLine)
+    {
+        for (int i = 0; i < aIndex; i++) {
+            if (!Character.isWhitespace(aLine.charAt(i))) {
+                return false;
+            }
+        }
+        return true;
+    }
```

Q4.1 Consider again the described situation of being faced with a pull request for a change of a method. How helpful would you consider the information above for getting a better understanding of the method and its history?

Very helpful	Helpful	Neutral	Not very helpful	Not helpful at all	Don't know
○	○	○	○	○	○

Q4.2 How hard would you consider retrieving information on the history of a method with the above level of detail?

Very hard	Hard	Neutral	Not very hard	Not hard at all	Don't know
○	○	○	○	○	○

Q4.3 If a tool could generate information in the fashion of the above on any method or other code unit, how valuable would you consider this tool?

Very valuable	Valuable	Neutral	Not very valuable	Not valuable at all	Don't know
○	○	○	○	○	○

Q4.4 What other information that is not in the descriptions above would you consider valuable?

Section 5: Background Information

Background Information

Q5.1 How many years have you been programming?

< 1 year	1-3 years	4-10 years	> 10 years
○	○	○	○

Q5.2 How long have you been working as a professional software developer?

< 1 year	1-3 years	4-10 years	> 10 years
○	○	○	○

Q5.3 How many years have you been using source code version control?

< 1 year	1-3 years	4-10 years	> 10 years
○	○	○	○

Q5.4 What is your current job title?

Q5.5 What version control systems and tools do you use? Please select one or more options.

☐ Git	☐ Bitbucket	☐ GitKraken
☐ Mercurial	☐ SourceTree	☐ TFS (Team Foundation Server)
☐ CVS	☐ IDE/Editor	☐ Visual Studio Online
☐ SVN	☐ SmartGit/SmartSVN	☐ Other (see next question)
☐ Github	☐ TortoiseGit/TortoiseSVN	

Q5.6 If you selected IDE/Editor in the previous question, please specify what IDE/Editor (and if you know, what underlying version control system) you are using. If you selected "Other", please specify what other tools you use.

Q5.7 Do you have any final comments? Do you have any other ideas for tool support or systems to solve the general problems described in this survey and its scenarios? Is there anything else on your mind?

Q5.8 If you are interested in the results of this survey and/or you want to enrol for the $100 (CAD) Amazon gift card lottery, please provide your email address. (Your email address will not be stored with the survey data.)

A.2 Survey Results

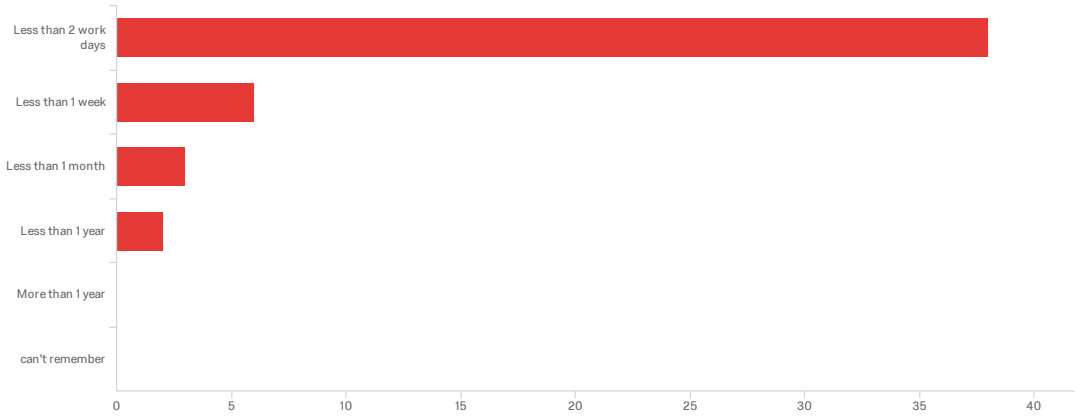
The following pages show the results of the survey described in Section 4.

Survey Results Export

CodeShovel

June 25, 2019 5:33 PM MDT

Q1.1 - Q1.1 How recently did you last use source code history of any kind?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q1.1 How recently did you last use source code history of any kind?	1.00	4.00	1.37	0.77	0.60	49

#	Field		Choice Count
1	Less than 2 work days	77.55%	38
2	Less than 1 week	12.24%	6
3	Less than 1 month	6.12%	3
4	Less than 1 year	4.08%	2
5	More than 1 year	0.00%	0
6	can't remember	0.00%	0
			49

Showing rows 1 - 7 of 7

Q1.2 - Q1.2 Please describe this most recent activity. How did you use source code history? What were you looking for? Did you find it? Did the tools you used support you in this investigation or could they be improved?

Q1.2 Please describe this most recent activity. How did you use source code...
Figure out who did certain changes
I opened a project I have not been working on for some time, so I used the commit-diff feature to see where I left the project, what changes I have made and where to keep on working.
Looked for changes. Tools that have been used, were useful.
shell, code changes, yes, yes
Used local individual file history to restore a previous iteration of my own code and used line annotations to identify when certain lines were changed last. Regarding the file history, I used only the most fitting version because there were so many revisions that finding the right one was too time consuming without knowing the approximate time where that version was actually in use.
who wrote the code initially / who made the last change and what did he change
I Iteratively fix bugs in code driving a data processing pipeline. I try out a few things, then commit them to the repo when they are tested.
Using git to manage multiple branches that depend on one another. Each branch corresponds to a separate feature, and each feature is submitted for code review independently. I was ensuring that each branch contained all the changes it needed to be atomic. This is possible in git but requires advanced knowledge of git -- I wouldn't say it's a first-class use case.
Reviewed a pull-request, merged a pull-request, changed to branches to see other people's code. I was looking to see that a bug fix was implemented. The tools I used were Git command line, to switch branches, and Bit Bucket online, where line changes were listed in red and green. The Bit Bucket tool could certainly be improved. Unlike Github, it is difficult to click into full files and the entire code history. I suppose that the Git command line could begin to use plugins, if they do not already do, to implement interesting features.
I searched in some diffs between two git commits for reasons why something in the code broke.
Tagging the repo for a release. Pushing the tag to origin. Used git CLI.
I was looking for the origin of a merge conflict. I was able to find the origin by using standard Git commands like git diff, git show and git ls-files -s.
I did some changes to the source code but needed to look at the previous state to resolve some bugs I introduced. I found the problem. I used the GitHub website to look at the history.
git log, git diff, git show to find reasons for recent changes
I had to do some version updates of dependencies that were done in another project too. So I had a look at the history to check which versions did we use in the other project. The tool (Bitbucket) I used supported me in this investigation.
checked the last commits to identify was was changed recently - and yes found it

Q1.2 Please describe this most recent activity. How did you use source code...

Figure out history of changes. Continuous Integration tool was not doing a build anymore. No change was made to source code. So I could concentrate on supicous configuration. Tool support was okay for me.

Used git to make and push some changes to the repo. Also, browsed for a change made by specific commit in the history (for 2 month old commit), i.e., looked at the "git diff" based on the SHA of the commit. Git was good enough for what I wanted and I did get the work done.

I reviewed recent changes. I used source code history to identify the changes, verify they were correct, looking for a specific change that might have introduced an issue. I confirmed that the problem was not recently introduced. The tools were quite good at helping me find those changes.

I was looking to merge branches into the main branch. (There were several disparate branches because each individual team member was working in isolation.) The worst aspect was that all of the deleted garbage files cluttered the history, so I had to filter that out. I used a lot of git diff.

Looked through a diff in a pull request

I searched for an old version of a file. I used the bitbucket for that. I found it and the bitbucket interface was great help.

I was merging few code changes that I implemented few days ago with my current working branch. I use git and I'm completely satisfied with it so far.

I was using github to go check who had been contributing to a enterprise project before and who I could contact for dev support.

I was reviewing a pull request in GitHub. It was a small PR and I had no issues with it. But reviewing PR is usually a tedious and boring task, because this may involve reviewing many dozens of files with unrelated changes (especially at the beginning of a project, you may need to refactor a few things while implementing a functionality). You could group them under commits, but when you review a PR it's easier to review the whole changes instead of commit by commit.

I used “git log” and “git show” to check what I modified. I was ensuring I didn't commit the wrong thing. I was also trying to have a overview what I have done. Just many commands involved. Better to have an easier ones.

Had to look if the stage branch contained all hotfixes from production, and how to name the next stage release. Used git on the commandline for merges, but SourceTree too look at the graph (tube map).

Looking for which commit a block of code was inserted in. I did find it eventually, but the tools aren't great, since I couldn't find a way to track lineage of a particular line of code, just the last commit which modified it (or my git/hg-fu aren't good enough!)

I was using it to identify when a colleague made a change, and also looking to see how commits were made. I was easily able to find it. I was using git-x. I also used history to identify what was changed previously and why. I found it using an annotation addon in intellij.

pull/commit/push cycle in git and the equivalent in hg and looking through history to identify what changed recently (since my previous commits) and also who is associated with some changes that I noticed.

I was checking the diffs in my branch against the committed code. A normal diff on the terminal was sufficient.

- Check when a change was made - Read code from a pervious version Of course I found what I was looking for. I can't thing of anything to improve.

I was looking for an outdated implementation of a functionality we aimed to re-introduce (in a better way)

Mainly for the basic tasks pulling, committing and pushing. I did not search for something specific. It's mostly a problem of myself that in addition to a certain improvement or bug fix, I put irrelevant enhancements into commits instead of committing the irrelevant ones separately. So I like the Bitbucket function of commenting on certain code parts of a commit and highlighting a particular spot that was responsible for a bug.

Looked for an old git commit using IntelliJ. Found it because of good commit messages. IntelliJ helped excellently.

Q1.2 Please describe this most recent activity. How did you use source code...

Checking for which ticket the changes in a file have been made. Yes I found it.

git

1. Embedded Eclipse plugin for GIT. 2. Recover changes overwritten by a team member by accident. 3. Yes. 4. Yes, Yes, visualization could be better, had to walk through all files line by line to compare changes.

Analyzed code changes over a period of time to find a bug/introduction of the bug. Tool support was sufficient (git/bitbucket/IDE)

I wanted to understand how the solution to a certain problem was implemented. I did find what I was looking for. The tool (Stash) was perfectly adequate in displaying the added and the changed lines of code.

Evolving of software architecture. Understanding what steps a certain component took to get to the shape/position it was in. I used GitHub and Tower to navigate the repository and the module's diffs. Navigating the file history and searching through the history of near-by modules by navigating the repository at different positions in time. No, the architectural evolution and reasoning of the modules under consideration was not reconstructible through observing the repository at different positions in time.

I often study the history of my source code to understand how and where other people worked since my last commit/push. I do not need any particular advanced tool to do that, since I am usually looking at small modifications that can be easily managed by the built-in tools of git/svn.

- reviewing a pull request - with the user interface of Atlassian bitbucket - compare the changes to the previous version of the file - yes - yes

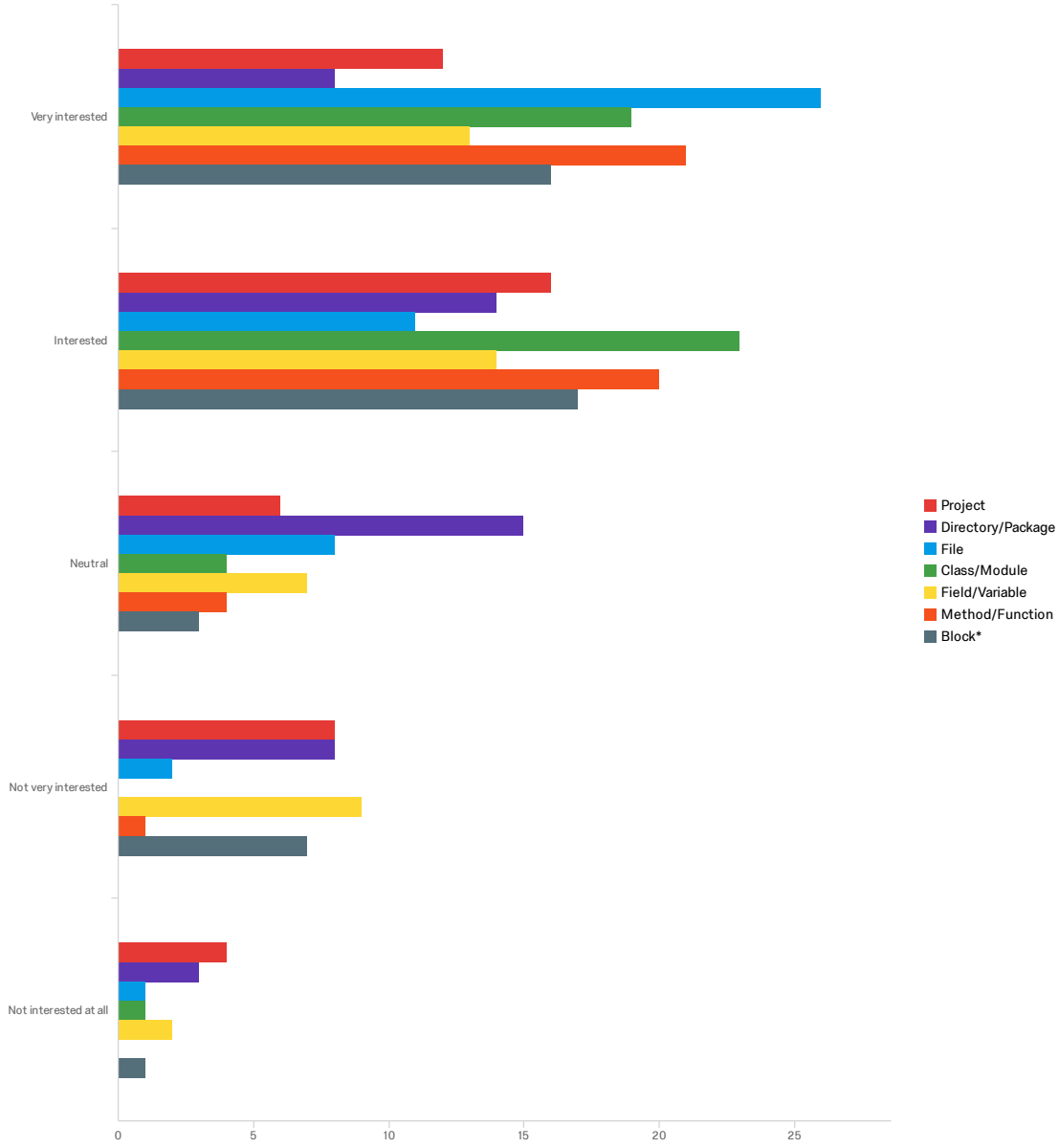
I was trying to revert a previously deleted feature, which wasn't tagged correctly. Unfortunately it took some time to find the right revision. In PHPStorm there is only a history view without further search functionality.

I search for a property in a configuration file hosted in a Git repository to understand how and why it was changed. I found the commit in which the property was changed. I used ' git blame' and 'git diff' to trace back the changes to the property.

Pull requests to review code using Bitbucket. Works fine most of the time, but sometimes changes (especially refactoring/reformating, but even adding of new methods) get displayed in a confusing way. Diffs of specific files (mostly changes of the last few hours) while developing using IDE. (Mostly to restore old states after realizing "I should have committed that".)

Contents of a file contradicted what has been documented elsewhere. Wanted to identify who changed the file at which point and whether there were any regressions (e.g. due to merges). (result: the discrepancy was intended) Used the Bitbucket Server interface to navigate first the history of a file, looking at various diffs. Color-highlighting of the changes definitely helped. Sometimes the context of what other changes where done in the commit were a bit sparse.

Q1.3 - Q1.3 In terms of source code granularity, how interested are you in gathering information on source code history at the following levels?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count

#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Project	1.00	5.00	2.48	1.28	1.64	46
2	Directory/Package	1.00	5.00	2.67	1.12	1.26	48
3	File	1.00	5.00	1.77	1.00	1.01	48
4	Class/Module	1.00	5.00	1.74	0.78	0.62	47
5	Field/Variable	1.00	5.00	2.40	1.22	1.48	45
6	Method/Function	1.00	4.00	1.67	0.72	0.52	46
7	Block*	1.00	5.00	2.09	1.12	1.26	44

#	Field	Very interested		Interested		Neutral		Not very interested		Not interested at all		Total
1	Project	26.09%	12	34.78%	16	13.04%	6	17.39%	8	8.70%	4	46
2	Directory/Package	16.67%	8	29.17%	14	31.25%	15	16.67%	8	6.25%	3	48
3	File	54.17%	26	22.92%	11	16.67%	8	4.17%	2	2.08%	1	48
4	Class/Module	40.43%	19	48.94%	23	8.51%	4	0.00%	0	2.13%	1	47
5	Field/Variable	28.89%	13	31.11%	14	15.56%	7	20.00%	9	4.44%	2	45
6	Method/Function	45.65%	21	43.48%	20	8.70%	4	2.17%	1	0.00%	0	46
7	Block*	36.36%	16	38.64%	17	6.82%	3	15.91%	7	2.27%	1	44

Showing rows 1 - 7 of 7

Q1.4 - Q1.4 When you use code history, how far in the past do you usually examine? How do you determine how far in the past you want to go?

Q1.4 When you use code history, how far in the past do you usually examine?...
if I see an interesting change i don't care how far back it was introduced...
~ 1-2 Months. Really depends on the project (how many developers are working on it, commit style: micro vs major, developer skills: is an intern involved?)
as far as necessary, it depends
Usually a version number or release date can narrow down the time frame of relevant commits to reasonable levels. Depending on the number of possible elements to analyze, I usually adapt the code level and more precisely select individual files or even just parts of a file to look into.
i am interested in the history of a code snippet (who did it / what was the code before), it doesn't matter how old it is - what matters is the number of iterations where the functionality was changed
It's generally for a few reasons. - I either want to figure out why/when a bug was introduced (by bisecting). There's no hard limit on how far back this might go in the history. I pick a point in history that I know was a good spot, and the current buggy point, and search between the two. In larger teams, it's often due to bad merges, and so it's interesting to track persons or features that introduce a regression. - Often I just want to revert a recently committed change, generally in my local history (stuff I haven't pushed). I like staging things, and using "git diff" as a tool to keep track of where I am or what I'm doing at the bleeding edge. -- especially from one day to the other. I'll often try a few things, and rebase/squash into the local commit that fixes that stuff. - I go by concept/feature for my commits. So sometimes there is no right granularity to look at patches. A refactoring operation, for instance, is conceptually simple, but ends up touching everything often. I have a vague notion of how things used to work, and so to find a commit in the past, I'll often rely on both my own commit messages, and I'll go by file. For projects that have very few (large) files, I will have to go by function, or local spots, so I use "blame" tools to figure out which commit touched each line last. If I don't remember where I changed something, but I remember an identifer, or a comment I've typed, I'll use git grep and go from there.
Depends on the pace of code development. In a project that sees several commits daily I might look back as far as 50-80 commits. Also I regularly use the functionality of git blame, which can document changes to a file that occurred arbitrarily long ago, and I find this useful when I want to understand the context in which a particular line of code was written.
Within a few months. I usually check to see if something has been modified that is relevant to the current issue that I am working on. If it is current, I want to see other related changes to keep in mind. I don't want to destroy other people's changes, so I want to understand their business logic and see why they implemented some code.
Depends: On older projects there is no exact time frame, in current projects it's often from now until the last one of my commits (i.e. all commits someone else made).
1-3 commits.
Most of the time, I only need to go back a couple of days. However, if some functionality seems odd or obsolete while reviewing source code, I need to inspect commits that are many months old.
Usually the last few changes (most of the time only the last change). To go back I use the commit history.
difficult to answer as it depends on the priority / impact of the defect / feature and how crucial the understand of it is ...
Usually I examine the last few commits (maybe 3 to 5 last commits). I have a look at the git network and read the commit messages, to find the correct commit.

Q1.4 When you use code history, how far in the past do you usually examine?...

Only couple of Commits which is mostly a couple of days. Sometimes I've been looking for a feature that was/wasn't implemented some months ago.

Last 5 - 10 commits, not depending on time.

> how far in the past do you usually examine? Usually 2-3 month, rarely anything more than a year (or all the way to the origin). > How do you determine how far in the past you want to go? I usually stop when I find what I am looking for. It usually happens to be 2-3 month old. I'll go all the way to the origin until I find what I am looking for.

Usually, I search back one or two weeks. Occasionally I will search back farther than that. How far depends upon the goal - if I am looking to find when an issue was introduced, I might search back years.

Not very far at all. I would keep it within the current sprint which is generally 2 weeks. Anything beyond that will be classified as a bug not a work in progress. This is because the branches are closed once things are merged into master and deployed as part of continuous deployments. This is very difficult to retrieve after the code is deployed. Unfortunately, we'd end up having to talk to the person who developed the feature rather than relying on code commits history. It also doesn't help that teams commit tiny changes 3-4 times a day, so it's a lot to sort through.

Depends on what I want to do. If I review a pull reuest, I only look at the latest changes. When I need to look up, how something was done in the past, I might even go back to the beginning of the project

Based on the current problem I try to solve, I go back in history until I find relevant code changes in this part of the code

I don't usually go more than a few weeks before in the history. Also it is very rare that I search commit history based on timestamp.

As long as I need

I don't have any fixed answer for this, it really depends a lot on the particular case. Sometimes I need to trace back the lifespan of a class until it was created (which might get tricky if it was renamed). Sometimes I need to check the very first commit of the project. I usually have to go to the file history and select the commit where the change I'm looking for might had happened.

Probably days at most a week if I want to ensure my work and have a overview. If using git blame, it can go very far.

Phew, as far as needed. Usually as far as the cause of some bug. Sometimes I use git bisect to find that. Sometimes I also use the history for "When did we launch v1" and that can be some years...

Usually a couple of weeks back to try and locate what caused a regression. Sometimes, have to go much further to find a changeset which is unaffected.

Time does not matter in distance I go back. The only thing that matters most is changes. I go back only 1-2 changes max.

Usually examine up to my last commit since that was the last time that I had a snapshot of the code in my head in a consistent form. If using branches then consider the entire branch, especially if am trying to figure out a merge.

I go as far as the last commit *I* made to the project, or from where I started working on the project, whichever is more recent. If this does not help in my search, I look at the commit messages to find something that seems relevant and then explore the diff.

As far as the change i'm looking for is in the past. I use tags and commit messages.

Most of the time I'm looking for relevant Jira issues in the first place, so that I know in what time frame the changes I'm searching for were made.

Until the last release tag.

Q1.4 When you use code history, how far in the past do you usually examine?...

Depends on what I'm looking for. Usually not further than 10 commits, because of different branches.

Depends, sometimes years if I'm looking for the reason for a code change. This depends on how old the project is.

till i find what i am looking for

Depends...usually I know who committed changes in the past days as I am constantly checking for updates in the SCM. When I'm uncertain about the history I have a look at the merge graph (GIT) and who of my teammembers contributed changes. Depending on the (hopefully good commit comment) I decide which commit to analyse, when searching for a specific change in history e.g. in a class file.

At least back to the oldest currently productive code

Usually only one step into the past. Rarely more. Only for very specific investigations one might have to step back a few times or look at a diff compared to a certain time point when an issue was believed to have been introduced.

Starting at a blame I get a rough overview of the age of certain parts of a module. I then try to triage down to issues I am looking into. Often line based visualisation is not really helpful for that. One has to jump back and forth in time between often unrelated changes/parts of history.

I am usually updating myself on what it has been done since I last worked on the project, so it highly depends on when I last worked on it. However, usually, I look at 3 to 6 days of work.

- usually around 1-4 weeks in the past - i move back in time by commits

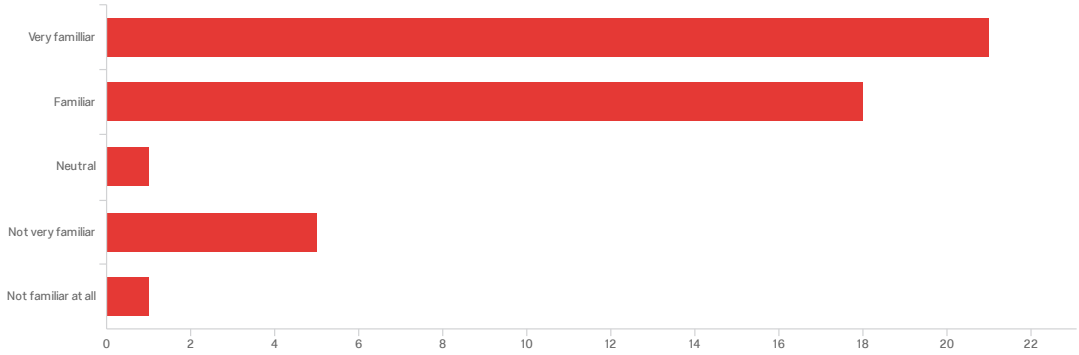
It'll be great to have the complete history available all the time.

It depends: On a project I'm actively working on I usually look at the code history for the past week. When using an open source project and depending on the change I'm interested in there are times where I have to look at the commit history years ago.

Mostly between a few hours to about two weeks. But for some cases (e.g. to understand why and how specific parts became the way they are today) it can expand to several years.

This largely depends of the goal. On larger projects with multiple developers, I most often start at the last commit I did myself (if any are available) trying to recreate the context in my mind and then go from there forward in time.

Q2.1 - Q2.1 Does this scenario sound familiar to you (i.e. have you encountered this in the past)?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q2.1 Does this scenario sound familiar to you (i.e. have you encountered this in the past)?	1.00	5.00	1.85	1.04	1.09	46

#	Field		Choice Count
1	Very familliar	45.65%	21
2	Familiar	39.13%	18
3	Neutral	2.17%	1
4	Not very familiar	10.87%	5
5	Not familiar at all	2.17%	1
			46

Showing rows 1 - 6 of 6

Q2.2 - Q2.2 Please describe very briefly how you would approach this problem. What kinds of questions would you like to answer? What tools or approaches would you use to answer them?

Q2.2 Please describe very briefly how you would approach this problem. What...
I always use the intelliJ "inspect tool" to quickly navigate through the files
First, have a look at any linked issues to find problem or feature descriptions, then consult documentation or the pull request creator to get more information. If too unsure, then at least test it locally or on an existing test system whether the intended functionality works as intended. At the end of the day, the responsibility of verifying the correct implementation is on the programmer side, not on the reviewer's.
first i ask for an informal description from the author. second i ask for a unit test / integration test. normally after understanding the test, one understands the code what was changed (ideally: test failed before the change)
The purpose of a patch is often not visible in the code alone. It It is my opinion that good comments (or PR messages) should describe the "why". The "what" and the "how" are essentially the code itself. I would want to know: - it is a fix, or a feature improvement - is it worth merging in (PRs can introduce other bugs) - is it related to something that's been repeatedly breaking. (is it a regression). looking at the code history for the affected region might reveal some important things. - is the patch tested properly. does it cover all cases. - are the limitations of the improvement/fix documented. is it a silver bullet. - is it linked to a particular issue. does the code submitted refer to that issue.
Tools I've used for PR code review, mainly Github and Bitbucket, usually don't make it easy to browse parts of the code that weren't modified in the PR. Yet, if the changes are to a part of the code with which I am unfamiliar, this is necessary to understand the full context of the changes. For this reason I will usually pull the feature branch in question to my local copy of the repository, and use my IDE to browse the changes, and the context in which they occur, at my leisure. This usually involves switching between e.g. Github, which highlights the changes and allows me to make comments on them, and the IDE, for semantic browsing of the code.
I would like better explanations of the bug/feature that is fixed or implemented. Sometimes context from a comment helps monumentally. Tying a code change to a ticket in Github, BitBucket, RT, etc would be beneificial in the code-viewer, so that context is always available and it does not have to be tracked down.
The obvious questions are why was this change made and what is it doing? I would check who made the commit and get in contact (in our case comment in Bitbucket on the specific pull request). Or look for a test for this function and check the test if this clarifies the reason and function behind the change.
Look at the whole file. Look at corresponding ticket. Ask commiter for intention of change.
To solve this problem, I would view the diffs of the relevant commits and try to run the relevant function in the original and the modified form and compare their results.
I would look at the code and follow its execution path (might need to look at other modules/classes). I would add a comment on parts that I don't know so that the person that created the pull request could answer my questions. The questions will mostly be general, like: Why does it need to be changed? What other parts of the code will be affected by the change? Are all boundary conditions satisfied? Will it break something? Tools and approaches: GitHub website for reviews and comments, IDE to look through the code, IRC to ask questions to the contributor.
pairing with the developer and find it out :-)
1. Checkout the complete code 2. Try to understand the general problem that the code should solve 3. Run & Debug the code, to see what happens at that specific part where the changes being made Bitbucket, IntelliJ IDEA, Tools that I need to run the code.

Q2.2 Please describe very briefly how you would approach this problem. What...

If I don't understand the code that I should review, I assume to contact the developer to get all needed background information. In addition I would check if there is any ticket related to the changes that provides more details.

I would analyze and use an editor to write down my comments. If code is very very complicated I will print out on paper and use a marker.

Usually there is a bug report (e.g., JIRA) for a small pull requests (PR) or a separate documentation for the bigger PR (e.g., feature). I prefer to get the higher level picture by reading docs, followed by the actual code.

I would read the code and reason about what it is doing. Generally I don't need anything more than an editor to read such code.

In a code review, I would be familiar with the code because only teammates with expert domain knowledge should review and give +1 to a PR. I would have to ping the person and ask what the change is for, but the PR will be linked to a JIRA task that I should review before bothering them. PRs are assigned within a small team and we were generally working on top of each other's code and very aware of any buggy code or upcoming enhancements.

Comment on the pull request. Either comment on the code directely or regularly on the pr itself

If I come to this kind of problem, this is mostly a sign of bad code quality. I will then ask the pull request creator what this should do and think about possible refactoring actions and comment them in the pull request

I never had to look for the past changes made to a code to understand it. But I had used git blame command to look for who was the last person changed some lines of code.

I would read through the code and try out the branch locally, in addition to running the tests and follow the flow of the code. Github and editor.

The one that triggered this PR should add a description with any information relevant to the reviewers (i.e. why, how). Besides that, the tool could help by grouping the changes by topic (not by commit, which may not contain final changes).

Read the commit messages first, then follow the thought, examine codes. Since commits are likely to be small, it should not take too long to figure out each commits, then the entire pull request.

Ask my IDE to show all callers of that function in my IDE, and then also look at the diff in this PR...

I'd really like much more context around the diffs. The other thing I'd like is subcommits to a larger commit; for example, assume a change that modifies a function and several call sites. I'd like to be able to group all the actual functional changes in a single subcommit, and then group together all the other call site changes and the like. Individual commits don't really work well because they break bisection.

I would use two tools, and issue tracker and github diff The issue tracker would tell me what were the design decisions that lead to the change. Github diff tells me the previous change with the current code.

Look at the comment in the pull request -- who made it, and what issue it is referencing. If the pull request is huge (sigh) then consider the tests first, then iterate with the person in the comments section to figure out what is what and why it is so large. Usually relying on the person who made the pull request is my go to strategy -- they have to be accountable for the pull request and in their interest to get it merged, so I expect them to volunteer answers quickly and to the point.

First, I do a diff to understand the relevant sections that I need to review. I then see if the change seems proper. To inspect a code fragment that I can't reason about, I use cscope to find how it is being used. I trace the calls to that function and try to reason about its usage. If the change does not make sense to me and seems sketchy, I make notes on the PR and ask for clarification.

Everything is in the code. Of course I know what it is doing.

Asking the Pull Request author? :D

Q2.2 Please describe very briefly how you would approach this problem. What...

Phew ... good question. I always ask the pull request creator directly. Of course you can comment with the already existing tools code parts and ask why was it done the way it was done. But typing takes time and during working hours you do not want to write big explanations. So my priority approach would be the Hipchat Call.

The PR should be linked to an issue which creates enough semantic context most of the time. If you then still don't understand the code it's either because the code is bad or you do not know the context well enough. In that case you would need to read the codebase.

I'd either talk to the developer directly or comment the file in the pull request.

read the code read comments aks committing person test the code

To understand "why" the code was changed, the easiest way is to ask the developer committed the change and ask for the reason. To answer the question "what" was changed in the code I would use a diff tool to compare and analyze the changes.

What was changed? Did the behaviour change (was a test modified to match the new behaviour)?

Investigate what the whole repository or relevant sub-systems are meant to do, then determine how this PR fits into that picture. Consult the PR description or associated design docs as to what the PR is supposed to achieve and then determine whether the code changes seem reasonable in that context.

Instead of understanding the evolution of the module/change-set I tend to ask questions or perform a pair review. Reconstructing the knowledge from plain diffs and the PR's change request is often to cumbersome. I often wish for more Architecture Decision Records I can read to understand why certain architecture decisions (APIs etc) arrived at the stage they're in.

If the change in the code is not self-explanatory, I would expect to find a clar and explanatory message in the commit.

Find the bug/feature ticket in which the problem is described and try to understand the business case. Questions are very specific to the problem. - is there a mockup (if ui related task)

In general the pull request should contain all aspects of what has changed and how the changes should behave. Also it must contain a detailed setup to build an exact test environment.

I would discuss the pull request either directly with the developer who opened the pull request or in a code review meeting in order to clarify the functionality of the code and establish a common understanding of how to use pull request and their descriptions to improve the overall code review process. Some question would be: Who did this change and when? Was it a single change or did it evolve over time? Why was the change made? Is the change related to an issue/bug or is it an improvement? In order to answer these questions my first step would be to look at the code history and then look at the tools involved in the development process (requirements documents, isse tracking system, ...).

Inspect the diff, read the comments and Javadoc, checkout the branch and browse it in the IDE. In more important/complex cases or projects I'm very new to, talk to the developer or ask questions as comments. Questions can be general like what the responsibility of each class is or target specific changes and choices made while implementing them.

Hopefully there is some textual description of the goal that has been achieved with the code changes. From that functionality I would work backwards to see what the different changes in themselves actually achieve and what their role is. Questions I would try to answer: - Why were the changes made? - Any sideeffects? - Does it harmonize with the general code-architecture of the project? Toolwise I would do most of this in the web UI of the code repository (Github, Bitbucket, ...). For larger changes I also run that version locally and step through the different functions using a debugger.

Q2.3 - Q2.3 Using source code history, how would you find changes to this method only?

Please describe briefly.

Q2.3 Using source code history, how would you find changes to this method o...
Annotate
Text search
Use multi line history for the method block to identify commits with changes in this area. If that is not enough, expand to file history or check the identified commits globally to see interconnections between files.
git blame
That's hard with line-based history search. I find it painful enough that I don't even bother. I'll narrow by file. Methods can move between files, but they generally don't span multiple files. Then, I would probably word grep for commits containing strings in that function -- if I know that an identifier changed, for instance. Short functions are easy. Long functions are harder. I will often fallback to a line-based blame output. I'll have commits that touched each line in a file (git/svn/hg blame) last, and work from there. Most often I'm interested in only the latest commit to have touched something -- if that commit was a whitespace fix (e.g. change a line ending), I'd have to recursively look back from that point. If I needed anything fancier, I would need a program to produce snapshots in time, and then track where the function is with an index (like cscope). But that's language specific. It is an interesting problem. I think searching by method has its uses, but the syntactic scope of where a change was made is not always relevant, or not always possible to track, so I feel the tools haven't pushed that feature forward too much. In Javascript for instance, you have these closures all over the place, and they are often anonymous -- so you can't fully rely on names, really to narrow down to only what you want to find. I feel that most SCMs throw the towel and give you generic "region-aware" history, rather than an informed history search based on program structure. It is interesting though, that when you are staging commits, the tools will often pull out the name of the function/class in which a hunk is to be applied, even if that line is outside the hunk. So there is value in extracting context. Gitk will give you the same thing when looking at the history (it will print the name of the object/class/function before each diff hunk). I'm not sure you can search on it though! -- or at least I don't remember doing this. You could ultimately look through the history with a visual tool, and let gitk extract the function name for you, but that's a lot of clicking.
I would use git blame to show changes to this file. This would work as long as this method hasn't been moved from one file to another. Github provides a decent UI around this, where they show the list of commits that have made changes to a file. However, this requires clicking on each commit separately, and navigating to the method in question within the list of changes brought about by that commit.
I would typically check to see if the file has been changed on a commit, and then git diff those changes to see if the method is included.
I would use the git history of the containing file in IntelliJ.
Looking at the history of the file where the method is placed in.
I would use options to limit the diff output of git log.
I would also look at the places where the method is used to see if the change breaks something. Also unit tests come in handy here.
git log -L :<funcname>:<file>
1. Checkout the branch with the changes (if there is an extra branch for that changes) 2. git diff from branch to develop or git diff between the commit with the changes and the commit before 3. Have a look at this method
I would use revision history and click through revisions.

Run 'git blame' on the file, get the 'git commit SHA' for the function, and 'git show commit_sha'. Most functions tend to be part of the single git commit. Or only 2-3 commits (modified the source code), in which case I'd see all commits one-by-one. I have not encountered the case where a function was modified by more than 2-3 commits, say 10, but if that's the case, I would probably look at the git history of the lines I am interested in. I am very unlikely to see all 10 git commits.

This can be more challenging, especially if the code has moved across files. I would need to establish the provenance of the code, then look at changes to the specific files. I would need to then read the code and understand how it changed.

This occurs in circumstances where the person responsible for this method has left (this is bad practice, eg. silo'ing). Then we would dig into the code history for this one method by looking at changes to the file itself. One method would not exist in isolation because they typically should be no longer than a screen's length. So generally the entire file and dependent files are also investigated. Things are trickier when the code doesn't link up in git, for example in microservices the API calls are difficult to connect and trace in git. For this we had a custom tool that linked calls (similar to Google's Dapper but not as nice) where we could trace the history of success/failures. From those metrics we dig into the code history to find a correlating change.

Have a look at the ticket connected to the PR. If done correctely, it should describe what should be done. However, if the ticket is done sloppy, I probably would have to contact the author or maybe the product owner

I would check the diff of the file and scroll to the method

As I said, I never had to do this. But if I had to do this, I probably would check the log of that particular file along with the changes made to it. git log -p command for example.

I use GitLens which makes it easy to find the exact commit and, hopefully, read a good summary on it.

You would need to check the "Blame" option for the file where the method is and check the commit that generated the last change. Then do the same for every deeper step you would like to go. You could guess the reason why it was changed with the commit message. In any case, it's not trivial.

use git blame and see which lines come from which commits.

Look at the file history for the last X commits.

blame->move to revision->blame recursively to rebuild lineage

I would search for changes to `this file` using gitx.

git log an grep for method if the norm is to mention this mention (unlikely); otherwise git blame (just looked up usage again in a blog since I use it so infrequently). I would figure out a person who last changed it and talk to the person if I can -- getting to the person is critical for me since I don't want to waste time making up a story about a change sequence that isn't true in reality (intent is important and is missing entirely from the source code repos).

I use git log -L to review a particular method name. Of course, this relies on the function name and path remaining constant.

Use the annotate feature in IntelliJ and klick on the commits in the method.

I'd use the bitbucket history of the file the method is in or the history of my IDE to see the changes commit by commit.

If the method is still in the same file, I would look at the diff of the file. But this approach is not really leading the way. The method could have moved from one file to another or the diff of the file is just too confusing because several places were adjusted at the same time in the course.

Don't know how, but would be great to be able to do it easily! I would probably look at commit messages and issues that I would expect to have impacts on said method.

Q2.3 Using source code history, how would you find changes to this method o...

Don't know, I'd probably check the file history.

using a diff tool

Aswing for a diff of the file the method is assumed to be included. Changes will be shown, if method is there. If method was (re)moved from file, I'll have a problem...

(graphical) diff tool

Use `git blame` to see the most recent changes to the relevant lines of code within that method and step through to the most recent change listed there.

Reading the blame of the method and jumping back and forth in history of the method. To the question blow: linting, automatic formatting and tests at different levels would ensure that the change is valid.

Unfortunately, it is not easy to directly and explicitly search changes in methods or variables or classes. I would search the diffs of the related files.

compare the file diff to the latest commit and find the changes for this function.

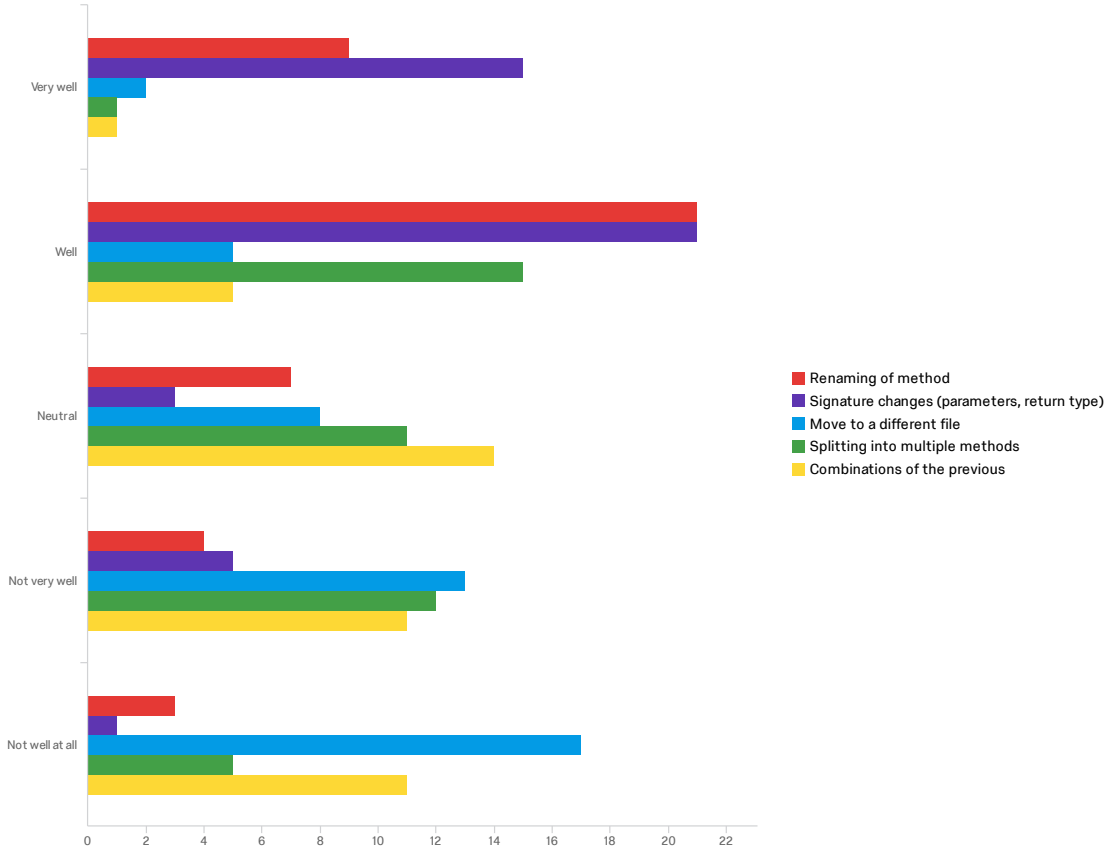
That´s a good question. Actually i would navigate to the file and open up the file git-history. Then i have to browse different versions to investigate the changes. It´s not the funniest job but it´s possible.

I would try different approaches: * First I would look at the source code history of the file that contains the method in order to understand the changes * If the first action does not result in a better understanding I would search for the method name in the descriptions of source code history

In IDE "show history" for the file, searching for the commit changing this method (by the latest change date of the specific lines), then using the commit message that hopefully includes a ticket ID.

I would look at the history for the lines of the method, for example using IntelliJ's "Show history for selection"

Q2.4 - Q2.4 How well would your strategy cope with more complex structural changes, e.g. method renaming, moving of a method, refactoring?

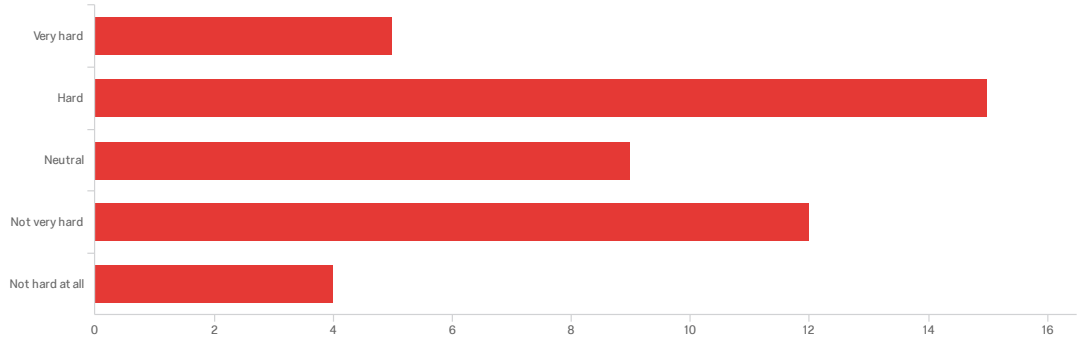


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Renaming of method	1.00	5.00	2.34	1.11	1.22	44
2	Signature changes (parameters, return type)	1.00	5.00	2.02	1.02	1.04	45
3	Move to a different file	1.00	5.00	3.84	1.17	1.38	45
4	Splitting into multiple methods	1.00	5.00	3.11	1.07	1.15	44
5	Combinations of the previous	1.00	5.00	3.62	1.07	1.14	42

#	Field	Very well	Well	Neutral	Not very well	Not well at all	Total
1	Renaming of method	20.45% 9	47.73% 21	15.91% 7	9.09% 4	6.82% 3	44
2	Signature changes (parameters, return type)	33.33% 15	46.67% 21	6.67% 3	11.11% 5	2.22% 1	45
3	Move to a different file	4.44% 2	11.11% 5	17.78% 8	28.89% 13	37.78% 17	45
4	Splitting into multiple methods	2.27% 1	34.09% 15	25.00% 11	27.27% 12	11.36% 5	44
5	Combinations of the previous	2.38% 1	11.90% 5	33.33% 14	26.19% 11	26.19% 11	42

Showing rows 1 - 5 of 5

Q2.5 - Q2.5 Using current tooling support, how hard is it generally to trace changes to a specific method?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q2.5 Using current tooling support, how hard is it generally to trace changes to a specific method?	1.00	5.00	2.89	1.18	1.39	45

#	Field		Choice Count
1	Very hard	11.11%	5
2	Hard	33.33%	15
3	Neutral	20.00%	9
4	Not very hard	26.67%	12
5	Not hard at all	8.89%	4
			45

Showing rows 1 - 6 of 6

Q2.6 - Q2.6 Given your answer to the previous question (Q2.5), what makes this hard or easy?

Q2.6 Given your answer to the previous question (Q2.5), what makes this har...
If you use a good IDE with Git Plugins it isn't too hard
Accessing history for a specific part of a file works pretty well. Problems only arise once the changes go beyond the scope of the originally identified section either through involvement of different files or major structural changes. This makes partial file history basically useless if what you are looking for is very old and there were multiple other changes to the file you are working with.
See previous answer. I can only surmise that it is not easy because correctly defining the scope of a method automatically without false positives and false negatives is a hard problem. Each language has its own grammar, there can be syntax errors which would throw off counting scopes, or even preprocessing mish-mashes of various files. Some methods/functions don't have names. The golden standard seems to be to pick the line in the source file which "looks" like some sort of header. (I've now realized at this point that we may not be on the same page as to what would be considered a method).
Most often, methods don't get moved across files, in which case it is easy. In the cases that they do, it becomes hard as the forms a discontinuity in the source code history.
There is not a specific way to track this information with a Git tool. You have to track the file and mentally investigate changes in the file.
Currently using git and this has file based granularity.
History is on file level makes it hard.
Most of the time, Git repositories are used. Git does not track changes but uses snapshots.
It might affect parts of the code that are not obvious to spot
git log -L ....
git or bitbucket show me all changes that being made. So it should be very easy to trace changes.
Refactoring is hard to track. Without refactoring it's ok.
Function can be split into multiple functions or moved to another file. These cases make it hard for 'git blame' based history browsing.
It's either easy (the method hasn't undergone complex changes) or challenging (the method has undergone complex changes). When it is challenging, it is VERY difficult.
It's very difficult in distributed systems because IntelliJ and other IDEs aren't great at remote debugging. Combine this with microservice architecture and the problem explodes. Calls occur between different repositories and the github search doesn't link these well. However, with good naming practices and logging behaviour, it can be made more intuitive.
This depends on whether the PR is connected to a ticket. If so, it also depends on how detailed the ticket is written
The diff in bitbucket and github are pretty good.

Q2.6 Given your answer to the previous question (Q2.5), what makes this har...

Certainly git is not designed to do this, but usually developers don't move around methods or functions much i guess. So file level history is good enough. At least as far as I'm concerned

GitLens shows it all

Last change is relatively easy: "Blame" and find the file within the commit. The older the change is, the harder it gets. If the method has change file or name, it becomes more complicated. In general, every additional change you have to trace back, more chances you have to get lost on the way.

git blame does not show the blames being replaced by subsequent commits. And if the method got moved, or modified too much, git blame will get lost

When the method moves elsewhere you have to change to that other place...

Most of the tooling I'm aware of works on file and line granularity, but mostly ignores function boundaries in changeset metadata

It is hard using gitx. It is more easy generally using annotation (git history in intellij). But if the method is moved then much more difficult.

source code repo tools don't have lang semantics, so somehow attempting to capture 'method' in git command line options is just asking for a bad day.

The change information at sub-file level is difficult to identify and track. Like the previous questions identified, method changes like renaming and splitting are difficult to track.

Just go through the history of that file and look at the method. Don't know why it should be hard

Well, it's not that hard since changes are clearly marked in the history of my IDE if you one commit after another.

Commits refer to file changes without any further reference. One would have to tag a piece of code, be it a method, class, or block somehow during the commit, so that any associated changes could be identified across files and independently of other changes in a commit.

You just can't do it AFAIK :D

Depends like described in Q2.4. if method is not renamed, moved to another file than it should be possible to trace changes.

Sophisticated history support in SCM/SCM integration in IDE (we use git)

It's not very hard because it's possible through decent IDE support to step through different git commits but could definitely be made easier.

Not knowing the semantics. Not being able to visualize the change of e.g. a method renaming or split in a graph.

The is no direct or explicit abstraction to method/class levels.

Hard: - most of the time a PR/Commit contains more then a single change to the method - no clear way of seeing the impact of a specific change if not explicitly grouped in a commit - developer needs discipline to commit often and in terms of logical changes Easy: - if a commit contains only the renaming or signature changing of a function it should be easy to trace back the changes

It´s just time consuming but not that hard. I´d prefer a better overview with a timeline with specific filters and stuff. Not a single side-by-side view.

The tool I use the most is Git and as far as I know you can only narrow down the search results by specifying a path. Classes, methods, properties are not supported.

Q2.6 Given your answer to the previous question (Q2.5), what makes this har...

The more stable a method is from external perspective (keeping its signature and general purpose) the easier inspecting its history becomes.

Q3.1 - Q3.1 In the above version history, how would you identify the commits in which the method of interest has changed? Please describe your strategy briefly.

Q3.1 In the above version history, how would you identify the commits in wh...

Use selective history for the lines in question, which reduces the number of commits to those with changes in these. Obviously, this has the downside that once the method moves around more than a few lines, the history is losing track of it because we are only referencing line numbers. Then, the commit log might give an indication of where the method came from. Continue the search there...

git blame

the comment links to an upstream issue. I would read that first. I will state upfront that I don't like the github interface. I would create a temp branch, set to the revision against which the pull request is made. Then I'd run git blame on the file. And spot the revisions affecting lines of the method of interest. Then I would inspect these commits. If the function was refactored from a different file, i'll reach a point where the function is added to CommonUtils.java. Then I'd have to git grep to find where it came from (hopefully the removal of the initial function is in the same commit as the addition). If CommonUtils.java was called something else before, I'd have to find the corresponding delete operation.

Click through each commit and search for the method in question.

I would take a look at the commit messages and any time-periods where I know people were working on related features where this method is important. I would specifically look for the development of that method, where it would have had to have been changed, by finding a relevant git commit message.

You could diff the first and the latest version and then check the lines of method for commit hashes in which they got changed, then go to these commits and check if there are earlier annotations on these lines and so on.

Use google and found command: "git log --follow -p -- file"

I would run git log -p -- src/main/java/com/puppycrawl/tools/checkstyle/utils/CommonUtils.java and search for all chunks affecting CommonUtils.hasWhitespaceBefore.

Above the code field you can select changes from all or certain commits.

If "git log -L ..." is no option here I would possible do samples of different dates looking at the original file and comparing it (as the method is quite short)

I would check the commit messages of the above version history to find the correct commits.

I would try to figure out of the comments if the method was concerned. If I wan't to be sure I would have to check every commit.

I would use 'git blame' to get the last commit/change on the lines I am interested. I would not try to find 'function level' changes from the entire commit history to CommonUtils.java. As you pointed out, 47 revisions are too many to go through.

Usually I would use binary search to walk through the change set. The descriptions could be useful in eliminating unlikely changes.

I scrolled down to the bottom of the commits and found the commons-3 commit which looked interested but was actually not directly related. I clicked on that diff and looked for the same file and saw that it was only libs and constants changed. I then scrolled through all the irrelevant minors and version bump commits up towards the top. I found the most recent related commit which shows improvements to the whitespace method. I searched for the file CommonUtils.java (using the browser search for text) because the smaller changes just to the method call were arranged at the top, making it difficult to find what I was looking for. Ultimately, I'm looking for just the CommonUtils.java file and there's too much garbage noise because it's such a commonly used file.

Q3.1 In the above version history, how would you identify the commits in wh...

Looked at the commit messages and randomly clicked through commits. Not very easy

I would click through each version and check if there is a change

git log -p And then search for the function name

I have no idea how I would do this.

In the PR, click "View" file, click "Blame", search for "hasWhitespaceBefore", check commit next to the method. For older commits, click "View blame prior to this change". Only commits older than 29th August 2015 are related to this method. But the method comes from another class (Utils.java), that would be more difficult to trace.

Open each one of them and manually inspect. ;-(

It's annoying, but looking at each commit if stuff changed in that method.

Recursively looking up blame on method lines to see when they were last modified

I would first use a better tool to look at history of the file. Like gitx, then I would just scroll through all the changes to the file if I had no idea why the change was made.

Oh boy. Well, if the git interface allowed me to expand all the diff snippets then I would expand and search for method name. As is, I would have to click through each one (doesn't seem like this method name appears in any of the comments, plus could miss some this way), so for completeness I would expand each diff snippet and search using the browser page text search).

Basically, I would dig through the history to find the method in each commit. I don't know if there is any tool that allows me to track the function across commits.

Go through every commit and check the method

I don't knooow! Click every single commit? Damn... you got me with this one!

Click on each commit and search with Ctrl+F for method name.

Look at the commit messages and diffs for each.

I'm trying to figure out by looking at the commit description if the change affects the method I'm interested in and check the diff if so.

split the time to equals parts, compare with revision 24, then revision 12 or 36

Having a look at the commit comments to get a knowledge of what is the change committed is all about. In Git there is an option to search in comments or search for logges changes in a specific file. If there is a hint in the comment, I would have a deeper look into the specifit commit. If not, one would probably have to have a deeper look into every of the 47 commits to investigate...

1. commit message 2. linked issues 3. diff tool

Look at each commit diff to determine if the method was changed.

Not at all.

Q3.1 In the above version history, how would you identify the commits in wh...

I sincerely have no idea. I usually don't find myself in such complex situations. Personally, I would need to search commit-by-commit. However, as I said, I usually don't have to do this, so I never explored more efficient solutions.

open the different commits and check manually

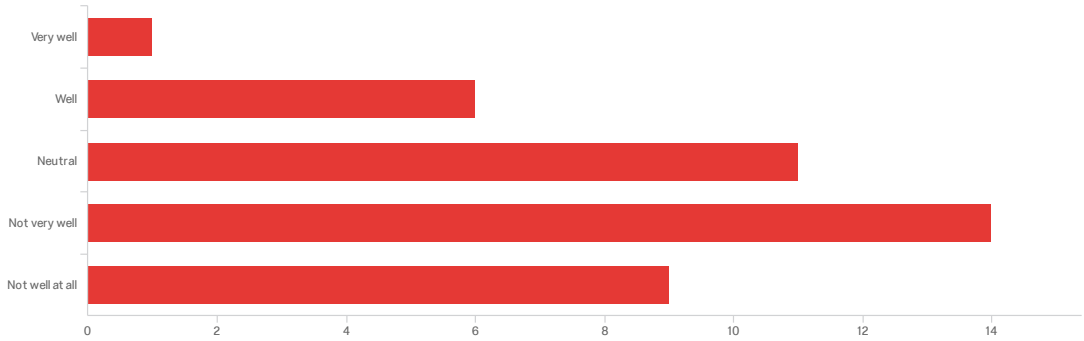
I´ll have to pick some previous versions to hopefully find commits related to this single method.

Looking at the version history I would try to identify changes related to the 'hasWhitespaceBefore' method by looking at the commit messages or issues. If this does not lead to a positive result I would probably write a simple script in order to step through the source code history and generate a diff of the affected file and only print out the relevant parts for method 'hasWhitespaceBefore'.

On web interface: Click through history? Search for "whitespace"/"blanks" in commit messages? Or using IDE, as described before.

I would first check if the function existed already in the very first commit. If not attempt some divide and conquer mechanism to see when it was introduced. From there I would go forward through all commits.

Q3.2 - Q3.2 How well do existing tools support identifying these changes?

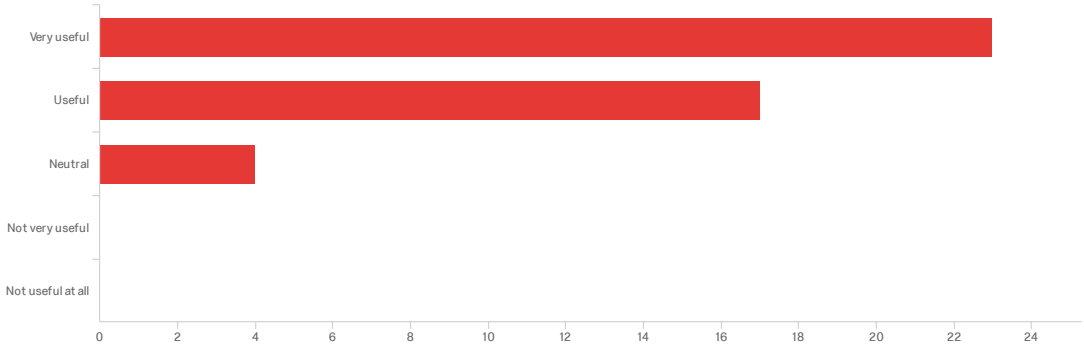


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q3.2 How well do existing tools support identifying these changes?	1.00	5.00	3.59	1.06	1.12	41

#	Field		Choice Count
1	Very well	2.44%	1
2	Well	14.63%	6
3	Neutral	26.83%	11
4	Not very well	34.15%	14
5	Not well at all	21.95%	9
			41

Showing rows 1 - 6 of 6

Q3.3 - Q3.3 How useful would it be to have support for a more semantic history in this scenario (e.g. history for this method or class only)?

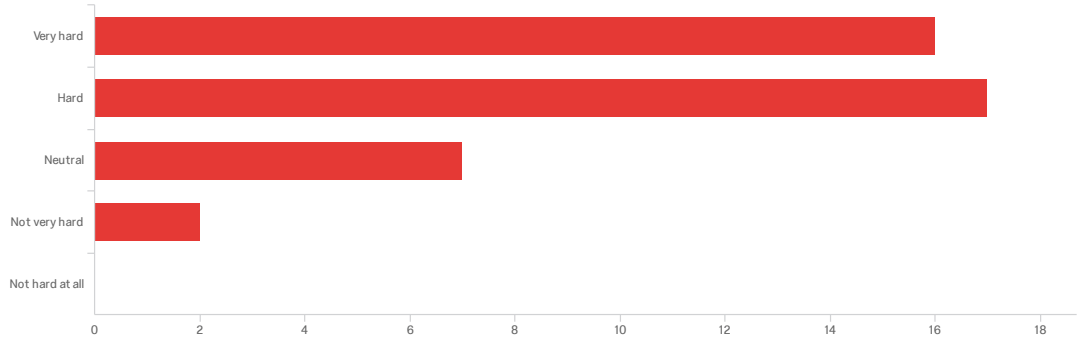


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q3.3 How useful would it be to have support for a more semantic history in this scenario (e.g. history for this method or class only)?	1.00	3.00	1.57	0.65	0.43	44

#	Field		Choice Count
1	Very useful	52.27%	23
2	Useful	38.64%	17
3	Neutral	9.09%	4
4	Not very useful	0.00%	0
5	Not useful at all	0.00%	0
			44

Showing rows 1 - 6 of 6

Q3.4 - Q3.4 How hard would it be to find the first commit for the given method and whether the method was really created then or if it was moved there from somewhere else (e.g. through a file renaming, or through a refactoring)?

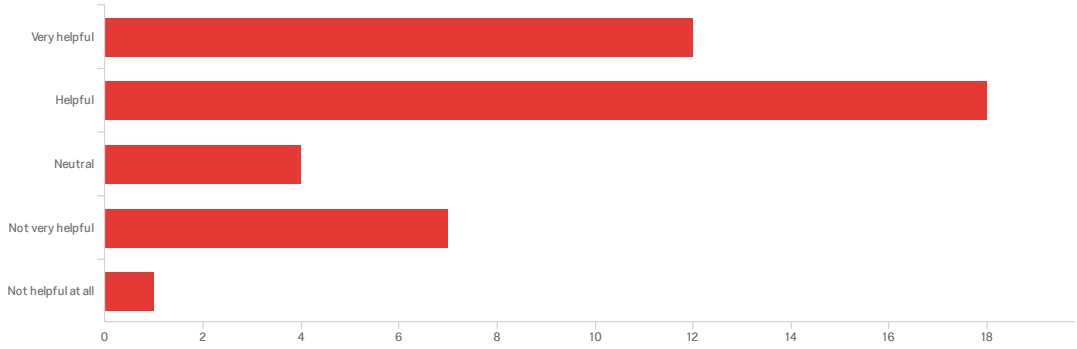


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q3.4 How hard would it be to find the first commit for the given method and whether the method was really created then or if it was moved there from somewhere else (e.g. through a file renaming, or through a refactoring)?	1.00	4.00	1.88	0.85	0.72	42

#	Field		Choice Count
1	Very hard	38.10%	16
2	Hard	40.48%	17
3	Neutral	16.67%	7
4	Not very hard	4.76%	2
5	Not hard at all	0.00%	0
			42

Showing rows 1 - 6 of 6

Q4.1 - Q4.1 Consider again the described situation of being faced with a pull request for a change of a method. How helpful would you consider the information above for getting a better understanding of the method and its history?

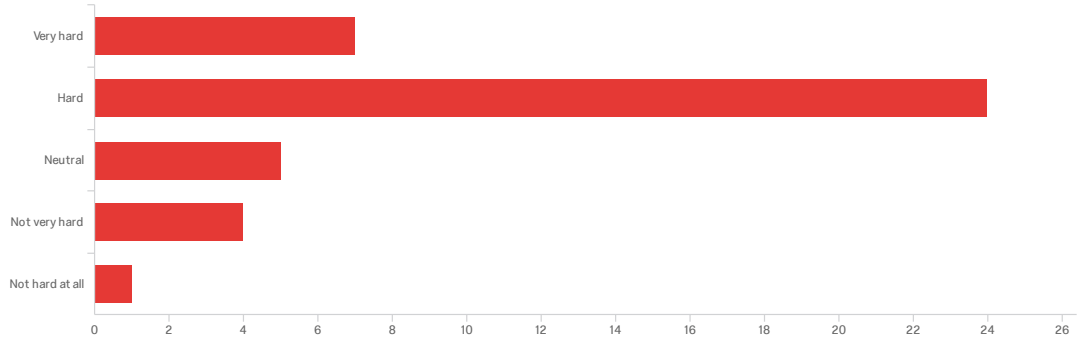


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q4.1 Consider again the described situation of being faced with a pull request for a change of a method. How helpful would you consider the information above for getting a better understanding of the method and its history?	1.00	5.00	2.21	1.10	1.22	42

#	Field		Choice Count
1	Very helpful	28.57%	12
2	Helpful	42.86%	18
3	Neutral	9.52%	4
4	Not very helpful	16.67%	7
5	Not helpful at all	2.38%	1
			42

Showing rows 1 - 6 of 6

Q4.2 - Q4.2 How hard would you consider retrieving information on the history of a method with the above level of detail?

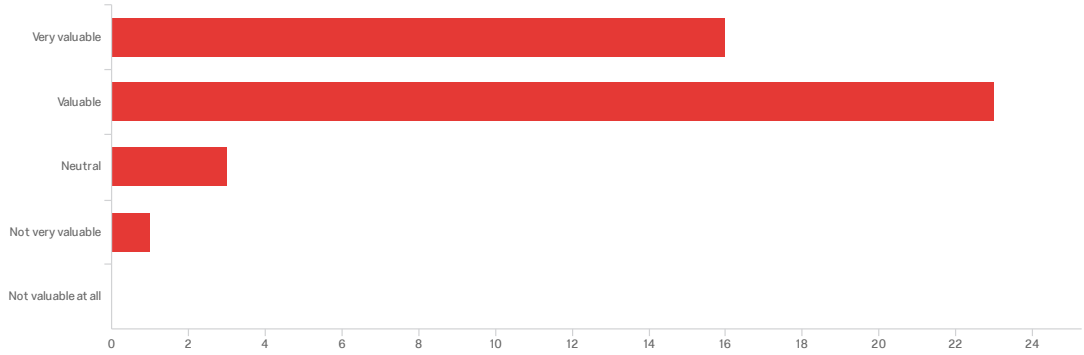


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q4.2 How hard would you consider retrieving information on the history of a method with the above level of detail?	1.00	5.00	2.22	0.92	0.85	41

#	Field		Choice Count
1	Very hard	17.07%	7
2	Hard	58.54%	24
3	Neutral	12.20%	5
4	Not very hard	9.76%	4
5	Not hard at all	2.44%	1
			41

Showing rows 1 - 6 of 6

Q4.3 - Q4.3 If a tool could generate information in the fashion of the above on any method or other code unit, how valuable would you consider this tool?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q4.3 If a tool could generate information in the fashion of the above on any method or other code unit, how valuable would you consider this tool?	1.00	4.00	1.74	0.69	0.47	43

#	Field		Choice Count
1	Very valuable	37.21%	16
2	Valuable	53.49%	23
3	Neutral	6.98%	3
4	Not very valuable	2.33%	1
5	Not valuable at all	0.00%	0
			43

Showing rows 1 - 6 of 6

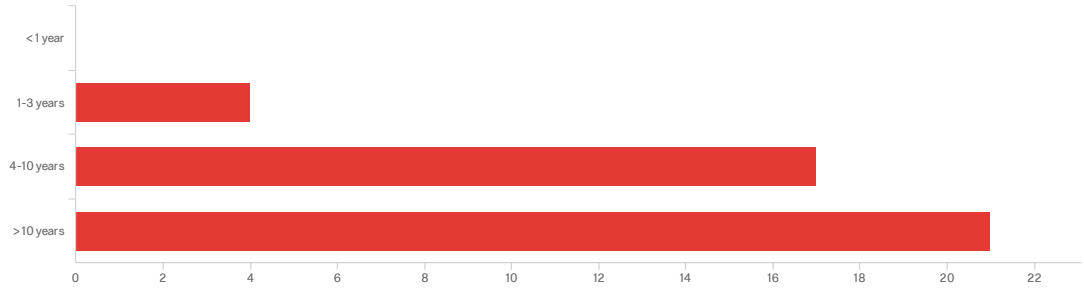
Q4.4 - Q4.4 What other information that is not in the descriptions above would you consider valuable?

Q4.4 What other information that is not in the descriptions above would you...
Information about whether content has only been moved from one class to another or gotten a real change. In the above example, I had to compare the method contents every time because it was marked as new content in the new files. For lines in that file, this may be true, but the method content itself didn't change at all.
I'm a bit torn here. On one hand I can appreciate the value that such a tool could provide. On the other hand, I don't think that in a real scenario i would want to trace such a small function (~10 LOC) back to the project's inception. It seems like more work than it's worth. If the question is "Should I merge this PR?", then I'm really trying to evaluate if the PR is a strict improvement over the previous commit. I'm not convinced going back to the roots of the project is necessary to make that decision. In other words, in what way does the function's provenance help assess whether the PR change is needed? There are other things that need to be done to evaluate the previous PR. For instance, there might be other calls to that function hasWhitespaceBefore that should also be refactored. There might be tests to worry about. I might be interested in finding commits that have modified both a test and the given function -- that might be a more compelling use case. Another thing to note is that the PR changes the semantics of what characters are allowed at the start of the line. isWhitespace() is more inclusive than just checking for ' '. Are callers of the modified functions aware that the semantics have changed?
It would be great if it was part of a git command line :) That is something that I would use to search for a method and its changes for sure.
Who made the changes and why (but this is available in the linked commits so not really missing).
Not just the information when and what changed but why the changes being made.
There is enough info here.
Short summary of all the places that are calling this method, so I could determine how important it is when scoping out changes.
I don't work on java a lot. Mostly C or C++. These languages don't have a restriction for giving same name file to the public class in the file. So I never had to face this problem.
I'd love to be able to specify some inputs and outputs to act as tests for a method. If it could show me how each version of the method would function for those; for instance, if I found a bug where it broke on a particular input, I could easily see if it were always broken or if some change added the bug.
I would like to have the issue number and design decisions that required the change.
The actual history of the method itself seems less relevant to me than what the method does right now (why/how would I need/use history; to blame people for bugs?). It seems that one useful bit of history is to find methods that were co-changed with this one. i.e., if there is another method that is a dependency or that somehow uses logic that is similar to this method, then knowing that they were changed together (or not!) would be useful. i.e., either to inform a change I would to make, or to learn about other parts of the code that are relevant to this method (e.g., if I'm learning about this method/this part of the code). Clearly knowing the people involved is key (for me), which you left out above. Knowing that Bob and Jane made all these changes would be super helpful (because I know Bob's on Slack and Jane left the company, so I know who to ask about the implementation or if I want to change the method -- code ownership comes into play).
If it is not an API, the most important information for me is that the method will still be used.
Maybe it would be interesting to see where it's (been) used (in the past).
The call reference to this method which in turn has been refracted just because of this method change

Q4.4 What other information that is not in the descriptions above would you...

Method usage
no answer
Associated test runs/suites and bug reports. Dependency updates which may be have trigger source file changes. Discussions of PR reviews going into more detail of reasonings.
- references / how often and where is the method used before and after the changes
Having the information above would be a big improvement in the daily business.
The commit message associated with every commit. Normally the commit message should include a description why the change was made and also references to other tools (e.g. issue tracking system, ...).
Who worked on the method within which time spans. Associated issue ids. Changes of signature. History of Javadoc of this specific method.
If the programming language supports strong typing like this example, I think the signature (parameters and return type) could be retrieved and would help a lot.

Q5.1 - Q5.1 How many years have you been programming?

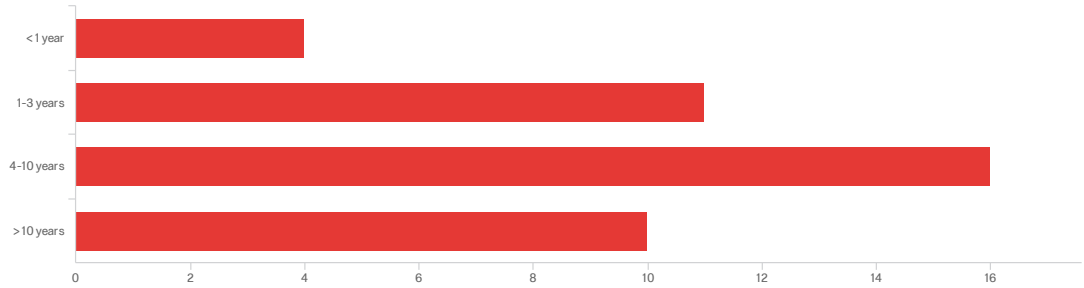


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q5.1 How many years have you been programming?	2.00	4.00	3.40	0.66	0.43	42

#	Field		Choice Count
1	< 1 year	0.00%	0
2	1-3 years	9.52%	4
3	4-10 years	40.48%	17
4	> 10 years	50.00%	21
			42

Showing rows 1 - 5 of 5

Q5.2 - Q5.2 How long have you been working as a professional software developer?

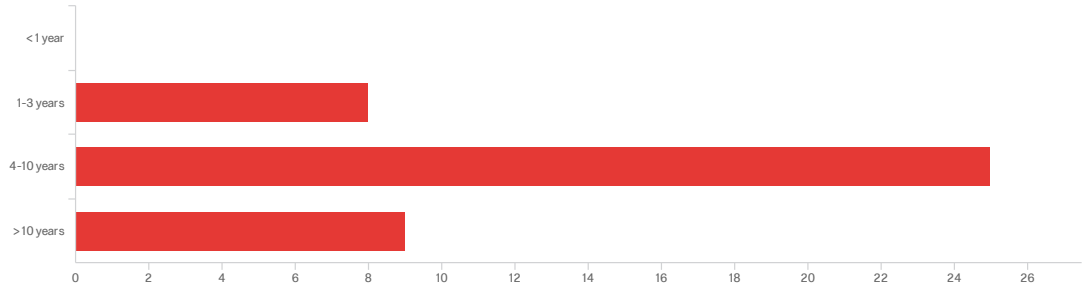


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q5.2 How long have you been working as a professional software developer?	1.00	4.00	2.78	0.92	0.85	41

#	Field		Choice Count
1	< 1 year	9.76%	4
2	1-3 years	26.83%	11
3	4-10 years	39.02%	16
4	> 10 years	24.39%	10
			41

Showing rows 1 - 5 of 5

Q5.3 - Q5.3 How many years have you been using source code version control?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q5.3 How many years have you been using source code version control?	2.00	4.00	3.02	0.64	0.40	42

#	Field		Choice Count
1	< 1 year	0.00%	0
2	1-3 years	19.05%	8
3	4-10 years	59.52%	25
4	> 10 years	21.43%	9
			42

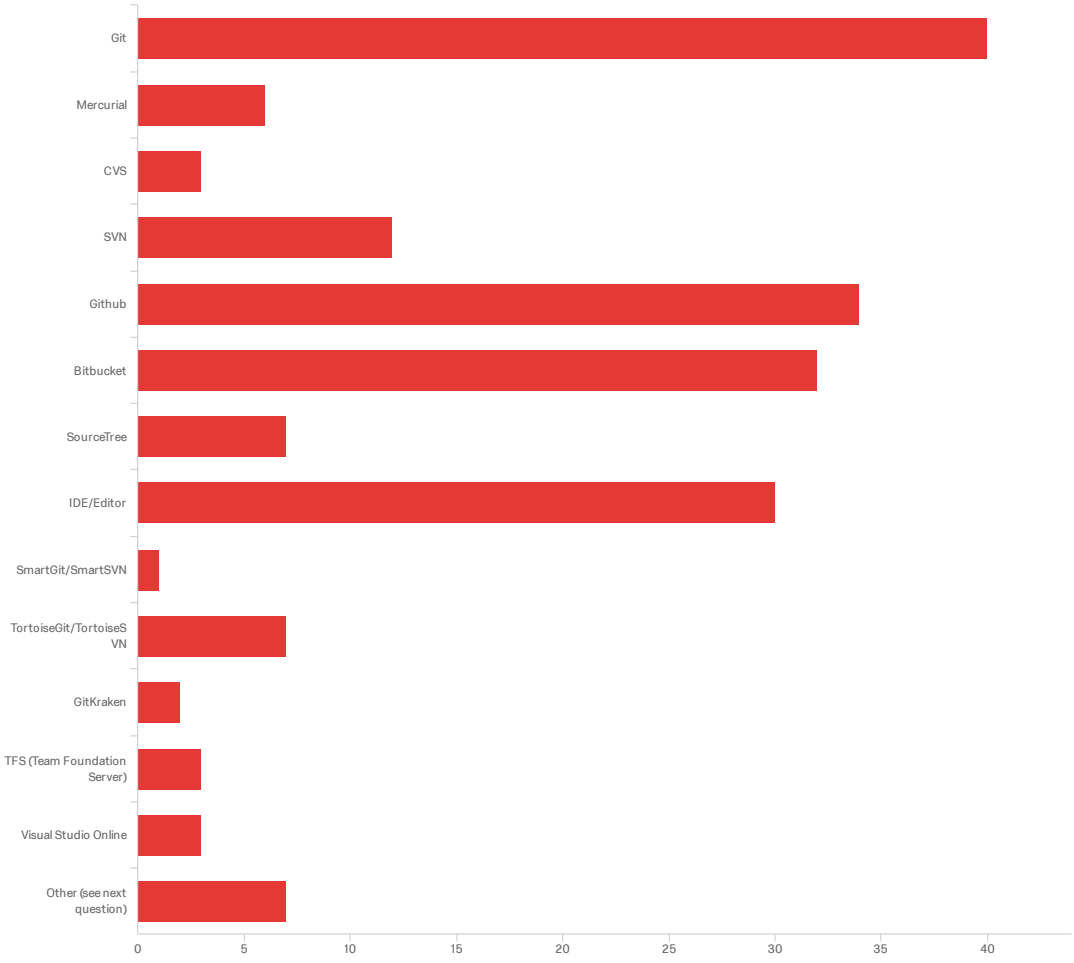
Showing rows 1 - 5 of 5

Q5.4 - Q5.4 What is your current job title?

Q5.4 What is your current job title?
software engineer
Postdoctoral fellow at UBC -- Data Science Institute
Software Developer
Backend Developer
IT Specialist
CTO
Software Developer
Graduate Student
Principal Consultant
Software Developer
IT Consultant/Fullstack Developer
Diplom-Informatiker
PhD student
PhD Student
Platform Engineer
Softaware Developer
Software Developer
Node.js Engineer
Software Engineer
Graduate student
Projektleiter Software & Consulting
PhD Student

Q5.4 What is your current job title?
Senior software product engineer
Assistant Professor
Grad Student
Software Developer
Software Developer / UI/UX Designer
Developer
Software Delevoper
Web Developer
Software Developer Web
Senior Software Developer
senior software consultant and architect
Senior Frontend Engineer
Permanent researcher
Professional Software Developer
Software developer
Software developer
Fullstack Developer

Q5.5 - Q5.5 What version control systems and tools do you use? Please select one or more options.



#	Field	Choice Count	
1	Git	21.39%	40
2	Mercurial	3.21%	6
3	CVS	1.60%	3
4	SVN	6.42%	12
5	Github	18.18%	34

#	Field		Choice Count
6	Bitbucket	17.11%	32
7	SourceTree	3.74%	7
8	IDE/Editor	16.04%	30
9	SmartGit/SmartSVN	0.53%	1
10	TortoiseGit/TortoiseSVN	3.74%	7
11	GitKraken	1.07%	2
12	TFS (Team Foundation Server)	1.60%	3
13	Visual Studio Online	1.60%	3
14	Other (see next question)	3.74%	7
			187

Showing rows 1 - 15 of 15

Q5.6 - Q5.6 If you selected IDE/Editor in the previous question, please specify what IDE/Editor (and if you know, what underlying version control system) you are using. If you selected "Other", please specify what other tools you use.

Q5.6 If you selected IDE/Editor in the previous question, please specify wh...
Qt+Git
I'm confused by the counterpart of this question. I use emacs, mostly. I've used Eclipse, and Sublime, and IDLE for python in the past. I use vim once in a while, but only to shoot myself in the foot or to insert a bunch of "^qq:" at the top of each file. I've only _heard_ of Visual Studio Online. Looks decent, i'd be keen to try it. What underlying version control system ? I'm not sure what that means. Do you mean under which SCM the IDE is developped (I don't know)? for most editors I use, I install plugins for most common SCMs. I have also used CVS, SourceForge, and SourceSafe in the past, but these tools have definitely lost their edge today. I feel ancient now. There's perforce that comes to mind, but it's not on the list.
- VScode uses git - the IDEA suite of IDEs from JetBrains provide an excellent feature for managing current source code changes called "changelists"
IntelliJ (with git)
- Atom (git) - vi - Eclipse (git) - IntelliJ (git)
Vim
Eclipse (git, svn), IntelliJ IDEA (git, svn)
IntelliJ IDEA
Using Eclipse, IntelliJ, Atom, and VS Code all along with git.
Eclipse with SVN plugin. Will change in the next weeks to Bitbucket.
VS Code (various including git) Visual Studio (TFS, Perforce, git)
IntelliJ, Eclipse Git but only through command line
VSCode
- VS Code - IntelliJ
Vim, ctags
VS code, GitLens
Intellij IDEA
Editor: sublime text

Q5.6 If you selected IDE/Editor in the previous question, please specify wh...
PyCharm, Atom
I use multiple IDE, Currently at work I am using Intellij, at home I use VS code.
Eclipse with all the version systems.
P4V and perforce.
IntelliJ
Php Storm
Jetbrains IntelliJ Visual Studio Code
IntelliJ, Visual Studio, Atom
PHP Storm & Visual Studio Code
IBM ClearCase , IBM RTC
IntelliJ
IntelliJ / Sublime (Underlying git)
Terminal and git with various helpers and configs.
IntellyJ IDEA and PHPStorm with Git
IntellijIDEA/Atom/Git
IntelliJ
Editor / IDE: - VS Code with GitLense - various IntelliJ products with both its "Local History" function and the Git integration Other: - The Git CLI itself (not sure if "git" included this)

Q5.7 - Q5.7 Do you have any final comments? Do you have any other ideas for tool support or systems to solve the general problems described in this survey and its scenarios? Is there anything else on your mind?

Q5.7 Do you have any final comments? Do you have any other ideas for tool s...
- informal / flowchart-like documentation is often more helpful than just the code for complex algorithms and structures. - reviewing code with unit/integration tests is easier than than just reviewing code. - we always should know who to ask: next to "git blame" sometimes I would like something like "how did something similar in this company"? (based on semantic similarity of code) often people solve the identical problems again, because they don't from each other. after the first commit a good tool could give a hint "are you looking for...?"
I do like the idea of being able to express rich queries on evolving code. I think that's a powerful concept, and one that could work across repos (or forks of a repo). I was thinking of use cases as I was filling in the survey, and I thought that it would be pretty cool to automatically merge patches that have been fixed in forked repos. If I find a bug in my own code, I could quickly determine whether it affects "descendent" repos, and vice versa. I could also look at a piece of code (a function , a class, a struct, etc.) , and compare it side by side with alternate implementations in other repos. However I think it's too big of a gun for inspecting the localized PR presented as a use case in this survey. I think a use case where a bug was reintroduced might be a more compelling example (it happens a lot!). There are other powerful use cases for a structured historical search during development (and less so patch approval). If I add a parameter to a function I would be interested in seeing what other parts of the code change along with that parameter. E.g.: "show me the code that changes when signature of method X changes". An obvious consequence of changing a signature is to change the callers of the modified function. A less obvious one is adding appropriate logging for the new parameter -- this is often forgotten, and added after the fact. I hope this is going to be a real tool! Would love to try it. I'm thinking even a prototype implementation of this as a wrapper script to the git commands that would be great to have at one's fingertips.
This was a long survey!
Being able to see who revised the method, and comment revisions would be nice.
I like the idea of being able to deep dive into a methods history, this would also be helpful for internal classes or fields that get moved from one file to another, but methods are definitely the best use case.
The described scenario is hard to solve with existing tools from my point of view. However I thought on my situations during the survey and I can't remember that I've needed such detailed information about one method over time. I'm pretty sure that I had these situations in the past but I think they are pretty rare. In other words: it is not something I miss in my daily work but of course it would be very helpful in these rare situations where method tracking is needed.
Merging the call tree with the historical view would be super useful. IDEs do this, for example when you rename a method, but the window is tiny and hard to read. It also only appears for the time when you're doing the refactoring and the historical view is difficult to retrieve.
I don't know method level history is very challenging considering the myriad of programming languages. I personally never had to encounter with the problems mentioned in the survey, may be because the tools that I use don't support this feature.
The history of a method helps to understand a pull request, but I usually don't find the need to do so.
Thinking back, I dig a lot less in code version control systems than I thought. Generally it's to rollback or to find out who did what or at most figure out when something happened and what issues were involved. I'm not sure I would find rich method-level semantics that useful. But, rich meta data that is easier to get at would be helpful.
Use diffs to trace a method through history in a file. Use search to trace a method in different files.

Q5.7 Do you have any final comments? Do you have any other ideas for tool s...

Thank you, I realize now that I use VCS purely as an archive. Even with the pull request, the successful execution of the test is more important than the actual change.

No

none

I´d be great to generate a method-, class-, file-history/protocol in a human-readable way. Example: --- Method: ExampleClass::exampleMethodX - aka: DifferentClass::exampleMethodA, MovedIntoAnotherClass::exampleMethodB - created: 2017-06-03 by "john doe" - history: -- 2017-06-03 13:46: Moved to class DifferentClass (see diff) -- 2017-06-10 13:46: Return value has changed to Integer (see diff) -- 2017-06-12 13:46: Added new Argument "newArgument" of Type String (see diff)

Good idea with the semantic code history. But the example with "hasWhitespaceBefore()" differs from my usual code history challenges - in this case, I wouldn't consider the history shown very helpful.

End of Report

Appendix B

Field Study Transcripts

The following pages show the plain results of the field study described in Section 8.

```
### BEGIN: LEGEND ###

Main questions:

Q1: Was this method really introduced in this commit?
Q2: Were there any commits in the history that should not be
    there?
Q3: Do you remember any specific commits that should be there
    but aren't?
Q4: Is the information produced by CodeShovel helpful? Please
    rate from 1 (not useful at all) to 5 (very useful). For this
    question assume there were a good UI for the tool.
Q5: Do you have any further comments? i.e. Do you miss anything?
     For what questions and challenges could you imaging CS to be
     useful?

Participant data:
Q6: How long have you worked as a professional software
    developer?
Q7: How long have you been programming?
Q8: How long have you been using version control?

Result legend:

Q1-Q2-Q3-Q4
(Q1-Q3 are yes or no for each method used in the study)

===
Q5
===

Q6-Q7-Q8
TIME_TAKEN_METHOD_1-TIME_TAKEN_METHOD_2-...

### END: LEGEND ###


============== P1 ==============

yy-nn-nn-4
```

```
===
* It would be interesting to see more details about what changed
     in the body, e.g. in one case an if statement was extracted
    and it would nice to be told about that.
* Multichange type is good because it is good to see that
    multiple things changed in this commit.
* Method-level annotations would be valuable (e.g. @Override)
* More information about refactorings, especially method
    extraction
* Interpretation of changes of call sites would be useful
* Generally really cool thing! It would be great to make
    something bigger out of this. I would really like to see
    method extraction refactorings specifically!
===

4-12-12

============== P2 ==============

yy-nn-nn-3

===

* This definitely needs a UI
* I would want to see the order the other way round from old to
    new
* There is definitely a domain and use cases for this, but I can
    't think of anything specific right now because I'm not a
    developer at the moment (rather a consultant)

20-26-15
2915-2933

============== P3 ==============

yyyy-nnnn-nnnn-4

===
```

A UI with timeline would be great. This way I could see easily
    who introduced a method (accountability).
If a method is very old, I see that many people were involved
    and I may then decide to refactor it.
I could also see in the overview what tickets were associated.
    Then I could find those tickets and use it for better
    documentation.

===

2-10-8
3158-3650-4293-4040

============== P4 ==============

yny-nnn-nnn-5

===

I really like your tool! It is much more efficient than the
    tools I am using. But a good UI is definitely a requirement
    for practical usage!

In one case the introduction commit was in fact just a copied
    method. It might be good if the tool would follow the method
    where it was copied from.

===

3-9-8
1866-1872-1036

============== P5 ==============

yyy-nnn-nnn-5

===

It is really interesting to follow such histories. If a method
    has a shitty name and is renamed to another shitty name, on

kann trace that. However, if it was never renamed, this is proof that developers followed conventions from the beginning .

If this tool had a UI it would be extremely useful because you can really focus on moves and other refactoring operations that would not be tracable with conventional Git history. But a UI is definitely a must.

Can you give this tool to me? I could see a potential every-day use. At least when we focus on sprints. It could also be useful for code reviews: how often was a method changed and why? In our team we have a big eye on commit history.

It would also be very useful to extend this tool with metrics. We could derive where there is potential for refactoring. If a method is changed very frequently, there is danger that the method is unstable. This often has consequences.

Statistics would be interesting: e.g. what is before a feature and what is after a feature? Where might I need refactoring?

===

20-35-20
2651-2352-2664

============== P6 ==============

yyy-nnn-nnn-4

===

In one case, nethod was copied from other class. History stops when copied. It would be nice to trace the method where it was copied from.

What are cases where there is only a parameterchange but no bodychange? Maybe something was done wrong if such changes are found?

This would be helpful if I want to find out why we changed
    something and the documentation isn't good enough.

I don't really see it being too useful for code reviews because
    these are about the actual changes.

For the UI: what has changed? Timeline of what has changed.
    Statistics of changes are not so important, I am more
    interested in the concrete changes. I am interested in
    commits.

===

4-10-8
2033-1968-2180

============== P7 ==============

yy-nn-nn-4

===

I can really imagine that this tool can be helpful! If you're
    new to a project and you see a method, its history can really
     help you. I wouldn't have to click through commits. It'd be
    really helpful to have a method-dedicated history. This way
    one can learn more about the codebase in an easy way.

===

1-8-1
781-810

============== P8 ==============

yyn-nnn-nnn-5

===

This tool can be useful you notice that something is going wrong somewhere. You can better debug and analyze why does this bug occur and what complications does it have? What was the reason why this code came to be? A method history can help here. We already do what this tool (i.e. its algorithm) is doing, we just do it manually. It would be really cool to have this functionality in the IDE.

===

10-17-14
4122-1500-1400

============== P9 ==============

yyy-nnn-nnn-5

===

We have methods and big refactorings in our services. I can well imagine that it could be useful to trace their histories. Because sometimes when we start with a new microservice and it receives more responsibilities and business logic, we often split classes into multiple. There's really lots of stuff happening. If then there is a time gap of, say, 10 weeks between sprints, things become really hard to debug. Imagine you have a bug on production with a state that is 10 weeks old. Now you have to create a hotfix and you also want to have the logic on your "develop" branch. Often you can't cherry-pick--you'll have to know where the method moved and from where. If I want to create a hotfix on HEAD as well as on the release branch I have to know what happened to the method. Here this tool would be very helpful! Because you never know where it is!

It is especially interesting to track history through refactorings. Other tools like IntelliJ and git-log don't help us here. Other than refactorings, I think the available tools are sufficient for us. But we often have scenarios where things completely dissolve and things are refactored

big. You'll ask "what kind of monster is this?". This tool looks at the file itself plus the context outside of the file . There is no context switch to reach your goal. Because otherwise I'll end up at "grep". Switching of tools is the most costly activity.

===

13-17-10
814-634-401

============== P10 ==============

yyy-nnn-nnn-4

===

This is very useful for refactorings, legacy code, code understanding for code you're not used to and code reviews. I would also think on the architecture level: I want to trace where a class was introduced and from where did the methods come and why are they there. Source code history is very important if you're new to a codebase.

===

6-13-13
2044-1228-914

============== P11 ==============

yyy-nnn-nnn-5

===

It is important that the tool also tracks method-level annotations. It doesn't do that yet. Especially @deprecated is important. It's interesting how crazy a method has blowed up and the became deprecated at some point. I didn't know that it was renamed at some point before I started at the

company.

I can imagine that this can be useful! In my example, the "BlackListFilter" was introduced and then removed again. It's interesting to see how the last commit before the introduction and the last commit containing the method look like. I can imagine that I would want to see a method-level history. When we introduce functionality and it matures for a few years and then you are confronted with it and you notice: "wow this has really grown. But why has it grown so much?". Functionality is tied to methods and that's why method-level history is helpful. Also in order to see how things grow; to see how all these things are added. Using this history you can see that changes that were made a long time ago aren't good any more now. You can discover design problems! Things become more complex and then it becomes hard to follow what's going on.

It would also be useful for refactoring. If this tool were in the IDE and there were a good UI with this history we might use this tool on an every day basis. But the UI and integration are a requirement because otherwise it's too hard to handle.

We had to learn the hard way how to write commit messages that really help.

##
This participant told me one day after the study that he would have used my tool for tracing a test method: which assertions that once were there were removed and why. You can't find anything to answer this question even with good commit messages and history. Nothing! Except you click through all commits.
##

===

2-10-7
773-2021-1775

```
============== P12 ==============

yyy-nnn-nnn-4

===

This tool is certainly useful. With the last method, for example
    , it was interesting how extremely bloated it became. You can
     learn why that came to be. Why does this method do so much
    now although it didn't really do much in the beginning? What
    was then intention at the time it was created and what's the
    intention now? It was 4 lines in the beginning and now it's
    25?

It would be good to now more about what changed in the body.
    Otherwise all the infos I would want were covered: renaming,
    parameter changes, etc. Really cool, really useful! A good UI
     would be important though.

===

2-9-8
894-631-1696

============== P13 ==============

yyyy-nnnn-nnnn-3

===

You can see when things were introduced and removed again later.
     Even if there's no UI the information is useful, esp. if
    something was moved or renamed. You can see the evolution:
    when was something introduced, where does it come from. Some
    things you can achieve using common git tools but there is
    definitely room for improvement. Where did refactorings
    happen?  It would be nice if duplicate code would be
    recognized: what methods do already exist?
```

The UI should have a timeline where you can click on commits to see changes etc. In general, I think there are few cases where this tool is very helpful and many cases where there are only body changes and in those cases the default git tools are sufficient as well.

===

3-9-9
1944-8053-3896-3048

============== P14 ==============

nyy-nnn-nnn-5

===

Histories are very helpful for onboarding: where do methods come from, how did they come to be? Esp. with complex methods this is helpful. Git blame isn't useful here because formatting commits destroy everything. Also for code reviews this is helpful if you don't know much about the code. But the integration (IDE or Bitbucket) is important. My review of 5 is only valid if the tool is integrated.

===

3-7-6
817-1413-714

============== P15 ==============

yy-nn-nn-3

===

I can't really imagine a concrete case where this is useful. But I haven't worked with big projects yet. I could imagine that it's helpful with bigger codebases. Most often I introduced the methods myself. It's more interesting than useful. The

body changes should be more detailed.

===

1-1-1

============== P16 ==============

yn-nn-nn-5

===

This is very useful! Because you tend to trust some commits more blindly than you should and others you have doubts and it's unjustified. To track methods is really good because you have the context available. You see changes that are specific to this method.

Here's an analogy: if I watch a series and it's a new season, I want a summary of the previous season. With the combination of good commit messages and this tool you discover the current status of knowledge in your team really well!

It's propably hard to show this information well in a UI, esp. when you change files. You get used to trace objects in the IDE, e.g. by clicking on a method. The UI should definitely be directly where things happen (i.e. IDE). If I need this tool then it should definitely be in the IDE because if it's in the pull request somehow then the pull request is too rich/bloated. If I want to see the code I go to the IDE. I don't want any third resources that take me out of my workflow, that's why this must all be in one place.

===

11-19-17
985-1466