Informatics

Visualisierung von getrackten Quellcodeänderungen auf Symbolebene

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

André Mategka, BSc
Matrikelnummer 11809614

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 16. Jänner 2026

Unterschrift Verfasser | Unterschrift Betreuung

Informatics

Visualization of Symbol-Level Code Changes Across Version Control History

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

André Mategka, BSc

Registration Number 11809614

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 16th January, 2026

Signature Author

Signature Advisor

Visualisierung von getrackten Quellcodeänderungen auf Symbolebene

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

André Mategka, BSc
Matrikelnummer 11809614

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 16. Jänner 2026

Erklärung zur Verfassung der Arbeit

André Mategka, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. Jänner 2026

André Mategka

Danksagung

Ich möchte mich bei meinen Freunden und meiner Familie bedanken, einerseits für ihr konstruktives Feedback und andererseits für ihre emotionale Unterstützung über all die Monate, die in diese Arbeit eingeflossen sind.

Zusätzlich möchte ich mich bei meinem Diplomarbitsbetreuer bedanken, der mir die Möglichkeit gab, diese Arbeit umzusetzen, und mich durch sein hilfreiches Feedback und Unterstützung mit organisatorischen Modalitäten durch den Prozess begleitet hat.

Und natürlich möchte ich mich herzlich bei all den Mitarbeitern der TU Wien bedanken, die ihre Zeit geopfert haben, um an den Interviews im Rahmen der Anforderungsanalyse und Evaluation teilzunehmen und mir wertvolles Feedback zu vermitteln.

Acknowledgements

I would like to thank my friends and family, for their constructive feedback but also for their emotional support throughout all the months that went into creating this research work.

Additionally, I want to thank my supervisor for the opportunity to work on this thesis and his support in the form of helpful feedback and assistance with organizational modalities throughout the entire process.

Finally, I want to express my thanks to all the staff at TU Wien who gave their time to participate in the requirements analysis and evaluation interviews to provide me with valuable feedback.

Kurzfassung

Mit dem kontinuierlichen Anstieg der Komplexität von Softwareprojekten geht auch ein Komplexitätsanstieg der zugehörigen Änderungsverläufe einher. Versionskontrollsysteme wie Git stellen eine etablierte toolbasierte Lösung für das Interagieren mit Softwareänderungen dar, verfolgen Änderungen jedoch stets auf Basis von Dateien und deren Zeilen, unabhängig davon, ob es sich bei den Inhalten um Quelltext oder normalen Text handelt.

Dies führt letzten Endes zu Mängeln hinsichtlich der Genauigkeit von Auswertungen, die auf Basis dieser Daten versuchen, das Ziehen von Schlüssen über vergangene Änderungen zu ermöglichen. Tatsächlich zeigen Studien, dass auch heute Entwickler viele bestehende Lösungen größtenteils für unzufriedenstellend erachten und sich stattdessen gezwungen sehen, auf eine manuelle Suche durch die Versionkontrollhistorie zurückzugreifen.

Software Evolution Visualization (SEV) und auf Versionskontrolldaten basierende Data-Mining-Ansätze stellen in der Forschung etablierte Gebiete dar, deren Erkenntnisse Entwicklern wiederholt bei solchen Problemen helfen konnten. Durch Kombination der Forschungsergebnisse auf beiden dieser Gebiete wird im Rahmen dieser wissenschaftlichen Arbeit ein moderner Ansatz zur Visualisierung von Versionskontrollhistorien erarbeitet, der quelltextnah Änderungen auf der Codesymbol-Ebene berücksichtigt.

Unter Zuhilfenahme hochmodernster Konzepte und Technologien wird auf das Untersuchen von Möglichkeiten, solch einen Ansatz durch zielorientierte Literaturrecherche und in Absprache mit Experten auf dem Gebiet der Softwareentwicklung umzusetzen, abgezielt. Die Ergebnisse dieser Arbeit demonstrieren nicht nur die effektive Realisierbarkeit, sondern auch das Potential dieser Klasse von Visualisierungskonzepten anhand von szenariobasierten Evaluationsverfahren.

Keywords: *Software Evolution Visualization, Software Repository Mining, Quellcode-Diffing, Datenvisualisierung, Informationsbedürfnisse in der Softwareentwicklung*

Abstract

As the complexity of software projects increases, so too does the complexity of their historical evolution. Version control systems, such as Git, constitute an established tool solution for interacting with software evolution, yet track changes based solely on files and the lines contained therein, regardless of whether the contents reflect source code or plain text.

This results in inaccuracies that render processing this data to gain further insights into properties of past change events very challenging. In fact, studies show that, to this day, many developers remain dissatisfied with existing solutions, resorting to the tedious alternative of browsing the commit history themselves to extract the information they need.

Software evolution visualization and version control data mining have become well-established research fields over the past decade and have proven to aid developers in addressing such issues time and time again. Combining these two ideas, this thesis presents an approach to software history visualization based on the very building blocks of code, code symbols with associated identifiers.

Utilizing current state-of-the-art concepts and technology, this thesis aims to examine the feasibility and promise of such an approach through a thorough literature review and interviews with experts in the field of software engineering. Its findings demonstrate both the viability and potential of this class of visualization concepts by way of scenario-based treatment evaluation.

Keywords: *software evolution visualization, software repository mining, source code differencing, data visualization, information needs in software engineering*

Contents

CHAPTER 1

Introduction

Due to the complexity of software projects, software development is often quite complicated and fraught with obstacles for managers and developers alike. Whether it is changing requirements, developer reallocation, or large-scale project scope, keeping track of all changes to the source code can prove to be quite the Herculean task.

Fortunately, version control systems like Git enable teams to capture changes in a way that is both semantically meaningful and lends itself to software history inspection and manipulation. However, the sheer volume of change information stored in such systems may also prove more and more unwieldy the further development progresses, to the point that developers frequently resort to manually inspecting sets of changes one by one until they can find the information they are looking for [1], [2].

This thesis fundamentally examines the ways data mining, visualization and interaction techniques based on source code-level change information can be leveraged to address this issue, using current state-of-the-art concepts and technologies in the fields of Mining Software Repositories (MSR) and Software Evolution Visualization (SEV) as well as feedback from experts in the field of software engineering.

1.1 Problem Statement

Given that many developers' information needs relate to code changes in the past (as seen later in Chapter 6), version control information needs to be processed further to compute more goal-oriented data aggregations. While this may simply take the form of metric-based textual summaries, interactive visualizations have proven to provide a highly intuitive interface by which information can be effectively and efficiently dissected. This visualization-based approach has been utilized to address many commonly encountered problems such as finding code experts, evaluating code ownership and identifying change frequency and locality [3], [4].

However, most state-of-the-art visualization approaches used today focus on changes at the level they are reported in by the version control system: files and lines. As such, the resolution of processed changes can be quite coarse and the semantic properties of changes and features, such as which attributes of the program have actually been altered, are largely discarded, so changes are primarily analyzed at the plaintext level.

While this loss of detail might be acceptable for some purposes (such as in the context of more functionally-oriented source code modularization approaches), this can result in misleading change clustering for more intricate modules and data structures as frequently found in object-oriented languages. Moreover, it disregards information that developers may associate with code changes, rendering navigation (e.g., searching, filtering, . . .) based on these change aspects impossible.

1.2 Motivation

As current scientific literature suggests, modern treatments fail to provide a satisfactory solution to these problems, opening the path towards novel solutions based on sub-file-level change information [5], [6]. Consequently, this thesis proposes an interactive visualization based on changes to code on the symbol level.

In their role as addressable elements with an associated identifier, code symbols (as later defined in Chapter 2)—such as packages, type declarations, functions and variables—constitute important building blocks of source code. As features develop, so do the names and properties (e.g., modifiers, type information, body structure, . . .) of these symbols. That being the case, developers often end up associating changes to code symbols with software features, serving as a more fine-grained and intuitive mental model for source code changes.

Consequently, this thesis aims to leverage the unique qualities of symbol information to design and implement a useful and efficient interactive visualization that aids developers in satisfying their information needs related to past changes.

1.3 Research Goals

This thesis aims to examine how visualization-based approaches can be applied to software projects to analyze and monitor changes made over the course of software development by leveraging semantically meaningful symbol-level information in order to provide more specific results which can then be used to identify code characteristics and potential issues.

As a part of this thesis, an architectural process for extracting, storing and querying this fine-grained data was designed and evaluated with requirements from both current scientific research and planned interviews with software developers in mind. To this end, a visualization prototype was developed to serve as an interface and tangible basis for evaluation.

Concretely, this thesis aims to answer the following research questions:

RQ1: What insights can developers gain from symbol-level code history information?

First and foremost is identifying the relevant information needs from developers relating to symbol-level changes. This also serves to elicit information- and visualization-related requirements, as well as to verify which of the intended use cases in the areas of code expertise, code ownership, code hotspots, code rationale and recent activity analysis present the best opportunities for interactive symbol-based visualizations.

RQ2: How can a symbol-based visualization be designed to satisfy the information needs of developers?

Specifically, this thesis is interested in design considerations that need to be taken into account when developing a symbol-based visualization. Expecting that analyzing changes intricately linked to programming language-specific concepts on such a fine-grained level presents unique challenges to visualization design and implementation, it discusses what problems one might encounter and how to address them in a way that leads to a useful and efficient visualization. To this end, a concrete visualization prototype based on an accompanying data mining approach for version control information be implemented.

RQ3: Does the proposed visualization approach lead to the intended benefits for developers?

Finally, determining whether the proposed visualization conforms to developers' use cases and how it compares to the requirements lifted during RQ1 requires a final evaluation. Based on the decisions made during the development of the visualization prototype, this thesis discusses how the proposed concept addresses the problems raised by developers and where room for improvement remains according to the received feedback.

1.4 Structure

The structure of this thesis is largely based on the individual steps necessary to investigate the described problem as well as to design, validate, implement and evaluate a suitable treatment according to the approach by Wieringa [7]:

- Chapter 1—this chapter—serves as an initial summary of this thesis, outlining the main problem, hypotheses and core research goals covered over the course of the following chapters.
- Chapter 2 explains the key terms and concepts used throughout this work, ranging from general ideas in the fields of algorithmics, formal languages, graph theory,

data visualization and version control to more complex topics such as Software Evolution Visualization (SEV) and code differencing.

- Chapter 3 contains a brief summary of existing scientific literary research, particularly in the fields of information needs in software engineering, code differencing and SEV.
- Chapter 4 acts as a review of the current state of the art concerning data mining approaches based on version control and source code, with a focus on current solutions to the problems of efficient file parsing and tree differencing in a temporal context.
- Chapter 5 describes the research methodology of this thesis, including the concrete modalities behind the overall approach and steps taken to ensure correctness and rigor throughout the entire design and engineering process.
- Chapter 6 contains the requirements analysis performed as part of the design process, consisting of a review of relevant scientific literature as well as a summary of the construction and results from the first set of expert interviews.
- Chapter 7 concerns the design considerations made throughout the design process, detailing both the iterative improvement cycles of the proposed visualization concepts and the architectural decisions behind the implemented data mining approach.
- Chapter 8 explains the concrete implementation details behind the analysis framework created to extract the data necessary for the final visualization prototype as well as the visualization itself.
- Chapter 9 serves as the evaluation of the visualization approach, containing both a summary of the results from the second set of expert interviews and a discussion of the research outcomes of this thesis.
- Chapter 10 concludes this thesis with a brief summary of its findings as well as a discussion of the potential for future work in related research areas.

CHAPTER 2

Fundamentals

This chapter is meant to provide a brief overview and explanation of the most important terms mentioned in this thesis. Where certain names are ambiguous, the corresponding description is additionally meant to disambiguate them from any homonyms [1] and related terms.

2.1 Graph and Source Code Structures

This section concerns graphs and related structures that can be used to describe programs in source code form.

2.1.1 Graph Concepts

This subsection is dedicated to common graph concepts, not necessarily related to software development.

Graph. A graph is a mathematical structure used for modeling relationships between entities using the abstraction of nodes (or vertices) connected by edges. A graph's edges can have a defined direction, in which case the graph is said to be *directed*, otherwise the graph is said to be *undirected*. In certain contexts, edges can have additional values associated with them, such as *weights*, which are frequently used to model optimization problems.

Graphs as discussed in this thesis are implicitly assumed to be *simple* (disallowing multiple identically treated edges between two nodes), *loop-free* (disallowing nodes to be connected to themselves) and *connected* (every node is reachable from any other node by following edges).

[1] Words are considered homonyms when they share the same spelling and pronunciation but differ in meaning.

Cycle. A cycle is a sequence of distinct edges which can be followed to go from a node back to itself without repeating other nodes in the process. For directed graphs, a *directed cycle* is a cycle that respects the directions of all followed edges. If a cycle ignores some or all edge directions, it is considered an *undirected* cycle (often just called a "cycle"). Therefore, every directed cycle is also an undirected cycle (when ignoring edge directions) and undirected graphs can only contain undirected cycles.

Tree. A tree is a special type of graph that disallows cycles. The directed equivalent of a tree is often called an *arborescence* in graph theory, and while this is technically the more fitting term for most tree data structures found in software engineering, the term "tree" is frequently used synonymously in many literary works and practical contexts, including this thesis. Similarly, the terms "tree" and "tree graph" usually refer to a *rooted tree*, in which a special node in the graph is declared the *root*.

Trees can be *traversed* by following their edges from the root node, which is said to go in the direction from *parent* to *child nodes*. Depending on the prioritization between *visiting* nodes and traversing their child nodes, the traversal is said to be a *preorder traversal* (if visiting parent nodes before their children) or a *postorder traversal* (if visiting child nodes before their parents). In the special case of *binary trees* (rooted trees where nodes may only have up to two child nodes), there is also a third kind of common traversal called an *inorder traversal*, which visits each parent node after their "left" child node but before their "right" child node.

DAG. A Directed Acyclic Graph (DAG) is a directed graph that—like a tree—disallows cycles, but only directed ones. In other words, nodes in a DAG are allowed to form an undirected cycle but not a directed cycle.

DAGs have the useful property of possessing a *topological sorting*, which is a sequence of the graph's nodes such that, when imagining removing the nodes from the graph in this order, only ever removes nodes without incoming edges. In cases where the edges in a directed graph model dependencies between nodes, this means that the nodes in the graph can be processed very efficiently, in an order that respects their dependencies.

2.1.2 Source Code Structures

This subsection is dedicated to ways source code can be described on an abstract level.

Language. A language in a computer science context can refer to at least two different but related concepts: A *programming language* and a *formal language*. Simply put, a formal language is a set of words, with each word constructed from concatenating elements of the same fixed *alphabet* set. Formal languages are often defined using a *grammar* that specifies *rules* describing how to derive words belonging to the language. Programming languages can be viewed as a subset of formal languages corresponding to input texts that can be read by a parser to produce a program, but the term "programming language" is usually associated with not just the syntactic properties of its corresponding formal language but its semantic properties, as well.

In this thesis, the word “language” primarily refers to formal languages (especially in parsing-related contexts), with the word “programming language” used with a concrete name like Java to refer to programming languages. An exception are the terms “language-specific” and “language-agnostic”, which do, in fact, refer to programming languages in most contexts.

Control flow. The control flow of a program represents the ways by which execution passes between its components or statements. For example, the control flow of a method usually consists of all combinations of the “branches” created by condition checks (e.g., `if` and `while`) that end in the method terminating (e.g., by returning to the caller or raising an exception).

Symbol. The term “symbol” can refer to multiple (sometimes related) concepts, including, but not limited to, logical symbols, alphabet elements, grammar symbols, grammar symbol instances, identifiers and symbol data types. In this thesis, “(source code) symbol” refers to a code element with an associated *identifier* (also known as its *name*) and other attributes such as *position*, *type* and *function signature*. Common types of symbols include packages, type declarations (classes, interfaces, enums, type aliases, derived types, . . .), type members (fields, methods), variables and constants. This definition is closely related to the “symbol object” as used by Aho et al. [8]. In the context of this thesis, “constants” refers to variables that disallow reassignment (e.g., `final` variables in Java).

2.1.3 Graph Structures in Software Engineering

This subsection is dedicated to graph structures that can be used to describe source code.

CST. A Concrete Syntax Tree (CST)—sometimes called a parse tree—is a tree data structure that is produced by a parser as the result of applying the rules of a language grammar, with each node corresponding to a rule in the grammar [8, p. 45]. In some contexts, the CST represents the final parsing result of a program, as is the case when using so-called *CST parsers* like Tree-Sitter [80] and Chevrotain[2].

Due to this low-level nature and lack of any strictly language-dependent structure, CSTs can sometimes be viewed as being language-agnostic, which can prove helpful for applications dealing with multiple programming languages but difficult when trying to extract specific information, as the mapping to “keys” for properties and the interpretation of their values are usually not part of the CST (names and values act mostly as strings that need additional context to process).

[2]https://github.com/Chevrotain/chevrotain, Accessed: Apr. 3, 2024

AST. An Abstract Syntax Tree (AST) is a tree data structure that resembles the final parser output for a specific programming language. In many cases, parsers directly output an abstract syntax tree without fully generating the concrete syntax tree first, while in others, the AST represents a second parsing step resulting from transforming the CST.

In contrast to CSTs, ASTs usually contain more than just purely syntactic code information [9, p. 13] and are therefore language-specific. For example, tokens such as visibility modifiers are typically stored in some kind of "visibility" property, and the values of such properties are usually pre-defined (if not defined as part of an enumerated type altogether). Operations are also normally already nested by order of semantically correct application (operator precedence) and ASTs often omit unneeded intermediate nodes present in the CST or evaluate constant expressions as they are parsed.

In some sense, ASTs constitute a syntactical representation of the source code bound, in part, by the semantics of the programming language. They follow a specific schema, and so, their representation is strongly tied to their respective language—different AST formats are rarely interchangeable.

CFG and DFG. A Control Flow Graph (CFG)—not to be confused with *context-free grammars*, which are also often abbreviated as CFG—is a graph representation of the control flow of a program. Similarly, a Data Flow Graph (DFG) is a graph representation of the data flow of a program (i.e., how values are assigned and passed). These two types of graphs frequently occur in the same context, at least within the field of visualization, due to their collective expressiveness regarding the "behavior" of a program and their utility in debugging and formal verification.

2.2 Source Code Change Tracking

This section concerns the ways software evolution is recorded, analyzed and interpreted.

2.2.1 Version Control System (VCS)

This subsection is dedicated to aspects of capturing source code changes over time.

Version control. Version control refers to the concept of using a program and a persistent data structure to keep track of all or some *revisions* (or *versions*) of certain content, usually a file. Modern Version Control System (VCS) tools are based on the concept of *diffing*, which allows for processing versioning information efficiently by computing the differences to the previous version as opposed to returning the entire after-state file for each change.

Modern version control systems mostly fall into the categories of *centralized* and *distributed* version control systems. Centralized version control systems such as Subversion store the entire version history of a project exclusively on the server, which acts the single source of truth. Distributed version control systems such as Git have the version history of a software repository copied to each client, which retains its own copy of the repository separately from the server. In the context of this thesis, the term “version control” and “version control system” primarily refer to versioning by Git due to it being the by far most widespread version control system.

Differencing. Differencing or *diffing* is the process of computing the differences between two versions of a file, also simply referred to as a *diff*. In the case of Git, all differencing is performed line by line, with its algorithm based on the Longest Common Subsequence (LCS) optimization problem.

Commit. A commit is a set of file differences in the context of a software project. Commits form a DAG by virtue of a pointer that points to zero or more *parent commits*. A commit is called an *initial commit* if it has zero parents and a *merge commit* if it has more than one.

2.2.2 Code Differencing

This subsection is dedicated to differencing techniques related to source code and aspects thereof.

Tree differencing. Tree differencing is the more general term for differencing tree data structures as opposed to traditional text line differencing. It is itself a specialization of the *graph differencing problem*, which in turn is based on so-called *graph isomorphisms* [10] and the concepts of *mappings*, *tree editing costs* and *edit distance* [11]. The process of determining the optimal way to transform one tree into another is called *tree edit script generation*.

Code differencing. Fundamentally, code differencing simply refers to differencing when applied to source code. In the context of this thesis and its referenced works, however, it is more commonly considered a special kind of tree differencing that takes the structure of source code text files as defined by their programming language’s grammar or parsed syntax tree (usually the AST rather than the CST) into account. This can result in more expressive diffs that accurately model changes in program properties as opposed to lines, which often contain text belonging to multiple nodes on the same level (e.g., different parts of a statement or multiple expressions).

Symbol-level change information. Symbol-level change information refers to change information as determined by applying a code differencing procedure to two versions of an AST and only retaining the differences related to code symbols, omitting large pieces of change information that do not affect the semantics of a program in any meaningful way. For example, reformattings are usually not reflected in an AST and reorderings in unordered contexts (such as fields of a class in Java) are not reflected in symbol-level change information (as these have no effect on the properties of the symbols themselves).

LP and ILP. Linear Programming (LP) is an algorithmic approach to solving optimization problems based on linear constraints on unknown real-valued variables. Integer Linear Programming (ILP) is similar to linear programming, but requires all variables to be integers instead. This has important implications for the performance of solving algorithms: LP problems can be solved efficiently (in polynomial time), while ILP problems are generally significantly harder to solve. Still, despite formally still generally being as computationally expensive as a brute-force search, *ILP solvers* have become very efficient in practice over the last decades. Due to the nature of tree differencing problems, optimal edit scripts can be computed using an *ILP formulation* and solver [12].

Refactoring. A refactoring is a type of code change that aims to restructure parts of a program without altering its behavior. Common examples include moves, renamings (which "delete" a semantic code element and insert a "new" one in a similar fashion to moves), code outsourcing, method inlining and supertype extraction.

This thesis defines a refactoring to be an *intra-file refactoring* if all—both source and destination—code positions involved are located within the same file (e.g., renamings), otherwise it is referred to as an *inter-file refactoring* (e.g., extracting a superclass). *Refactoring detection* is the algorithmic concept of identifying refactorings as part of edit script generation. This is a task that can prove to be very challenging because each type of modification allowed during code differencing increases the work required to check all possible modification and refactoring traces.

2.3 Data Analysis and Visualization

This section concerns the terms found in software repository data mining and data visualization.

2.3.1 Mining Software Repositories (MSR)

This subsection is dedicated to concepts related to data mining as applied to software repositories.

MSR. Mining Software Repositories (MSR) is the field of research dedicated to identifying ways to extract information and gain insights from software repositories, such as codebases under version control [13].

ITS. An Issue Tracking System (ITS) is a software that allows developers and managers to keep track of bug reports as well as feature or enhancement requests over the lifetime of a software project. They are often used in conjunction with VCSs to form relationships between task items and corresponding development branches.

CI/CD. Continuous Integration / Continuous Delivery (CI/CD) is a practice that enforces the frequent merging of changes, testing of software state and provision of delivery workflows, usually in combination with a suitably configured VCS code base on a public or company-wide repository hosting platform, where it is used as a simple way to perform regression and feature verification tests in addition to providing an easy automated way to deploy the compiled software artifact to staging and production environments. If deployment to production environments is also automated as part of the overall workflow, CI/CD can also refer to "Continuous Integration / Continuous Deployment", which shares the same acronym.

2.3.2 Software Evolution Visualization (SEV)

This subsection is dedicated to categories of data visualization.

Visualization. A (data) visualization is a way to visually present information in a way that allows the identification of patterns, trends or results from a holistic view based on a set of raw or processed data points. Visualizations often feature ways to interact with the displayed data through options such as filtering, searching, grouping, distorting or zooming.

Time-based visualization. A time-based visualization is simply a type of visualization that uses data with a temporal component as its input. This usually manifests as an additional axis in the visualization, either as a literal time axis or some way to navigate between data points at different points in time or in specific timeframes.

Software visualization. A software visualization is a visualization that depicts some type of information associated with a software repository or artifact. Frequently, these visualizations center around some kind of inherent structure present in source code, e.g., code hierarchies, component dependencies and size or complexity comparisons.

SEV. Software Evolution Visualization (SEV) is a field that finds itself at the intersection of time-based visualization and software visualization, aiming to show how a codebase has evolved over time [14]. The most prominent source of the temporal data component is usually VCS change information. Part of what makes visualizing data in this area so complex is the fact that both software as a structure and the ways in which it is modified over time can be quite difficult to accurately convey at the same time.

CHAPTER 3

Related Work

This thesis includes aspects from research topics in the areas of information needs in software engineering, language parsing, code differencing, software evolution, software history mining, software evolution visualization, as well as general data visualization and interaction techniques.

3.1 Information Needs in Software Engineering

Several studies have been conducted on developers' information needs. The main concern of this area of research is the identification of questions and challenges that arise during software development and providing extension points for further research based on these identified issues. Fundamentally, these are exactly the kinds of issues and ideas areas like Software Evolution Visualization address, and the implications for this thesis will be further elaborated on in Chapter 6.

Fritz and Murphy [15] conducted eleven open interviews with software engineers of the same company across three different locations to elicit frequent development-related questions and what domains they consult to try to answer them, yielding a categorized list of 46[1] questions, which they went on to use in their software history plugin prototype design. In the same vein, LaToza and Myers [16] received 179 free-text responses to their survey at Microsoft with hard-to-answer questions about code, resulting in 371 questions clustered by subject and intent. Begel and Zimmermann [17] also put out multiple surveys at Microsoft focused around hard-to-answer questions, albeit more from the perspective of a data scientist rather than that of a developer, the results of which show a large breadth of questions of interest regarding software.

[1]The actual number of elicited questions was 78, but only 46 of them are listed in the paper due to space constraints. Unfortunately, as of April 2024, the link to the website containing the complete listing leads to a generic "Page not found" error page.

In 2015, Codoban et al. [1] published the results of their survey about the motivations, strategies and challenges when it comes to questions related to software history with participants from both the industry and social media, revealing a well of untapped potential for history-related information needs. These findings were later re-analyzed by Ragavan et al. [2], showing that they are largely consistent with the concepts found in Information Foraging Theory (IFT), which sheds further light on the origins of—and connections between—the responses. McKee et al. [5] performed both interviews and surveys to discover the shortcomings of currently available tools when dealing with merging, merge conflicts and frustration-inducing issues related to software history, highlighting the demand for higher-usability tooling. These same findings were later included in a larger journal article on the topic by Nelson et al. [18], giving further insight into the consequences of increased complexity in the face of inadequate tool support.

Regarding information needs for visualization, Merino et al. [19] conducted a literature review of over 300 papers on software-centric visualization, aiming to identify the characteristics of prevalent visualization techniques, which ultimately demonstrated the importance of change-related information needs and visualizations among software practitioners. Related to this—but from an auditory perspective—is the paper on the visualization approach by North et al. [20] which contains valuable insights into the use of sound to convey information. A paper by McNair et al. [21] additionally includes a summary of questions related to software architecture posed by them as well as researchers, developers and managers in other studies across scientific literature, and shows how they can be addressed using a visualization-based approach.

3.2 Code Differencing

Ever since the introduction of the concepts of graph isomorphisms [10] and edit distance metrics for trees [22], a lot of research effort has gone into how differencing can be applied to structures like common source code representations. Seeing as the data mining approach for this thesis relies on VCS source code history information, fine-grained change detection plays an integral part in its realization.

3.2.1 Tree-Based Differencing

Probably the most fundamental method of differencing source code is tree differencing on the AST or CST structures produced by language parsers. In 1996, Chawathe et al. [23] developed the by now eponymous two-traversal tree edit script generation algorithm with support for tree move operations. This approach was later expanded upon by Fluri et al. [24], who modified the algorithm to perform better on structures like source code representations. These two algorithms form the foundation of both the AST differencing framework GumTree [25] and multi-level refactoring detector RefactoringMiner [26], [27]. Somewhat similar in nature to GumTree is the Shift AST[2] family of tools which form a

[2]https://shift-ast.org/, Accessed: Mar. 25, 2024

code analysis ecosystem focused on ECMAScript source code.

Qingtao et al. [28] also proposed their own AST-based differencing technique that groups changes based on change content, location, author and timestamp. Since many of these approaches rely on similarity metrics, there have also been papers such as the one by Sunghun et al. [29], who investigated the features of functions to determine similarity metrics suitable for accurate function mapping.

There have also been attempts to make code differencing more space-efficient at scale: The LISA framework by Alexandru et al. [30] is one such approach which exploits redundancy in ASTs to analyze the entire history of a VCS repository. Similarly, HyperAST [31] and HyperDiff [32] are tools that employ spatial and temporal compression and use a combination of a DAG and Tree-Sitter [80] CSTs, as well as the findings of the paper by the same authors on co-evolution tracking [33] based on GumTree [25], RefactoringMiner [27] and Spoon [34]. CodeShovel [6] and CodeTracker [35] are both approaches to on-demand symbol-level history mining for individual code symbols based on ASTs with refactoring detection.

3.2.2 Language-Agnostic Differencing

Aside from approaches based on CSTs and language-specific ASTs with or without parser plugin systems, there have also been endeavors to design more language-agnostic differencing algorithms which take advantage of language-agnostic source code representations. Effendi et al. [12] developed so-called "MU graphs" as a language-agnostic intermediate representation for source code and a graph differencing approach using ILP with support for type stubs and type inference for dynamic programming languages to accurately cluster changes and mine static analysis rules. This approach is partially based on code change pattern mining techniques using data and control flow information by Nguyen et al. [36] and Dilhara et al. [37].

There are also a number of tools [38], [39], [40] that rely on the XML-based "srcML" [41] code representation, which aims to enable language-agnostic code analysis and transformation. Counted among them is also srcDiff [42], which uses the well-known LCS algorithm by Myers [43] combined with preorder traversal for more efficient differencing than the traditional AST-based approach.

3.2.3 Other Techniques

Aside from the above techniques which are all inherently based on source code "as is" in CST, AST or graph form, there has also been quite a bit of research into other representations. Hata et al. [44], for example, based their differencing approach itself on the line-based diff capabilities of Git by creating a separate history for individual symbols and their associated code. This was later followed up by the papers of German et al. [45] and Higo et al. [46] who split the data into individual tokens with added classification information—from their srcML origin and structural context, respectively—before applying Git's line-based differencing.

There are also techniques that use in-IDE change recorders [47], [48], [49], [50], [51] to capture more fine-grained code changes as they occur, which, in turn, assists in tasks like code differencing for merge conflict resolution and history refactoring. Other unique approaches include leveraging other programming paradigms: LSDiff [52], for example, uses logic programming to infer changes based on facts derived from source code.

3.3 Software Evolution Visualization

Numerous papers have delved into assistive visualization, from visualization and interaction design in general to more granular classes of visualizations designed for specific types of data sources and information needs. As this thesis is primarily concerned with creating a useful and efficient visualization, the approaches listed here serve as a good guideline for how both visualization and interaction techniques can be employed to achieve high usability and clearly communicate information related to software evolution. A more elaborate exploration of the most important implications for this thesis can be found in Chapter 6 as part of the requirements analysis.

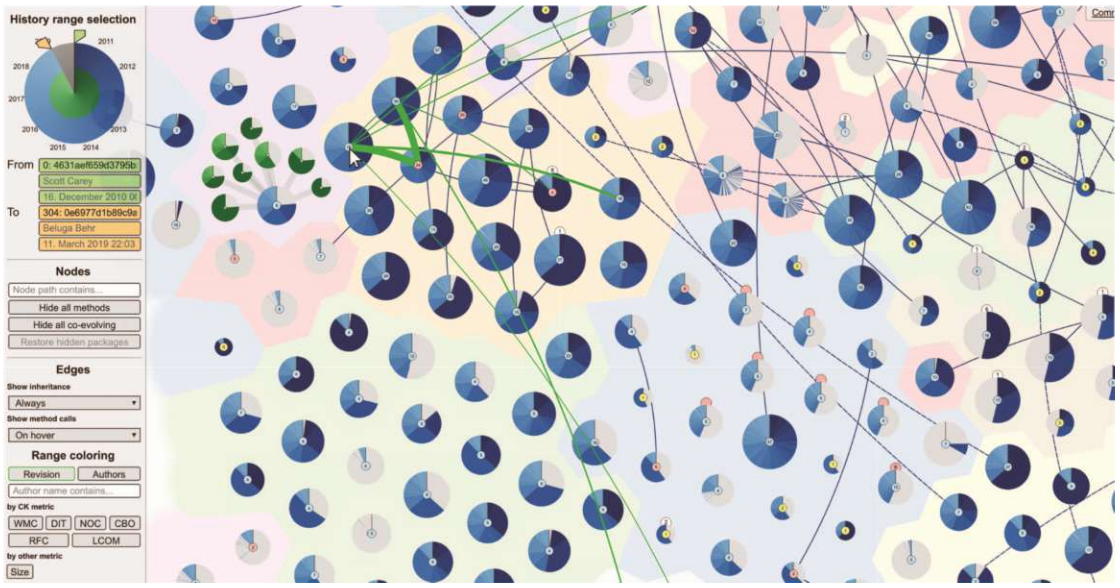
3.3.1 Visualization and Interaction Design

The oft-quoted paper by Keim [53] on information visualization represents a foundation for categorizing visualization approaches using a classification scheme that accounts for data source, visualization technique, as well as interaction and distortion techniques. Maletic et al. [54] also published a paper in which they described pivotal aspects of software visualizations, as well as navigational needs of their users in a similar vein to the interaction and distortion techniques identified by Keim. Fritz and Murphy [15] developed a prototype to showcase how information can be organized relative to different pivot data domains to cater to different information needs. Caserta and Zendra [14] conducted a literature survey in which they classify different types of static software aspects visualizations and their used visual abstractions.

Shahin et al. [3] later introduced a classification scheme for software architecture visualization approaches which has since seen widespread use across other research papers. Similarly, McNair et al. [21] had previously refined a classification approach for software evolution visualization based on target aspects. A literature review paper by Bani-Salameh et al. [55] uses these two classification schemes to categorize a number of visualization approaches and provides a good overview over the prevalent techniques in SEV. The following categories of related works are also very roughly based on the classification scheme by Shahin et al.

3.3.2 Graph- and Charts-Based Visualization

This class of visualizations contains techniques that primarily rely on different types of charts or structures akin to directed or undirected graphs, where nodes represent some kind of entities and edges represent some kind of entity relationships. EvoLens by



From 0: 4631aef659d3795b
Scott Carey
16. December 2010 0(
To 304: 0e6977d1b89c9a
Beluga Behr
11. March 2019 22:03
Node path contains...
Show inheritance: Always
Show method calls: On hover
Author name contains...

Figure 3.1: Graph-based visualization by Alexandru et al. [57, p. 16]

Ratzinger et al. [56] is one such graph-based visualization, which sees packages displayed as boxes, classes displayed as nodes, and change coupling between classes displayed as edges with corresponding thickness. The graph visualization by Alexandru et al. [57] visible in Figure 3.1, on the other hand, places nested charts inside the nodes of the graph and uses the edges of the graph for dependency relationships such as inheritance or control flow dependencies. More straightforward is the set of graph visualizations by Guo et al. [58], which render the AST, DFG and CFG for a given piece of source code in one of the three supported programming languages, respectively. Rozenberg et al. [59] proposed RepoGrams, consisting of a collection of "footprint" stacked bar charts for representing code metrics like code ownership or commit localization where the size of each bar corresponds to the size of a specific commit in version control history. Grabner et al. [4] created a visualization ecosystem focused around harvesting information from multiple sources, including VCS history, ITS items and CI/CD results, and show how this data can be combined to yield a more holistic view of the software beyond its source code using mostly charts-based techniques.

3.3.3 Metaphor-Based Visualization

This class of visualizations contains techniques that mainly utilize existing mental models from real-world contexts to communicate information. Perhaps the most well-known visualization technique in this category is the "3D blended city" approach that constructs a "city" from "buildings" whose dimensions and appearance mirror the properties of an element in a collection of data items. Some examples of this include the works by Xie et al. [60] and Dal Sasso et al. [61], which use files and classes for the data represented by the "buildings", respectively. Other approaches that one might also prefer to include in

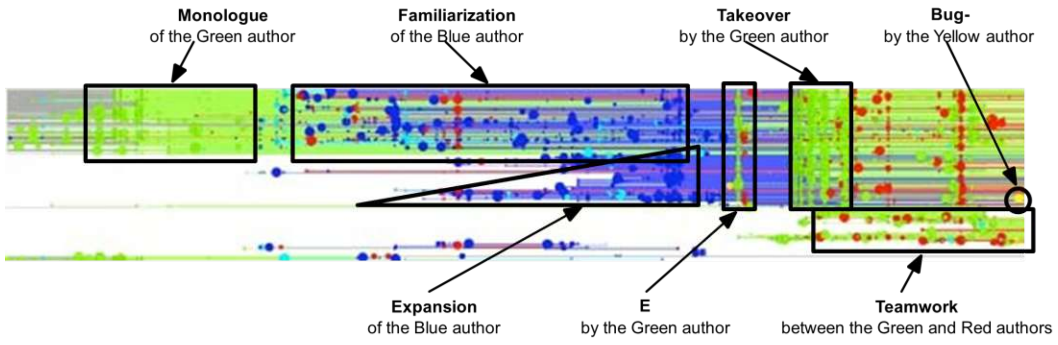


Figure 3.2: Timeline metaphor visualization by Girba et al. [62, p. 4]

the category of graph- and charts-based visualizations focus on timelines, as you would expect to find in a real-world schedule or program overview. Most notable here are the visualizations by Girba et al. [62]—pictured in Figure 3.2—and Kuhn and Stocker [63], which use color coding to represent change authors, as well as the file-level change event view component in the visualization by Yoon et al. [64]. Additionally, there are techniques using a "rewiring" metaphor to represent correlations between two states of nested sequences such as the AST difference viewer by Chevalier et al. [65] shown in Figure 3.3, with identically lettered boxes representing matching code regions.

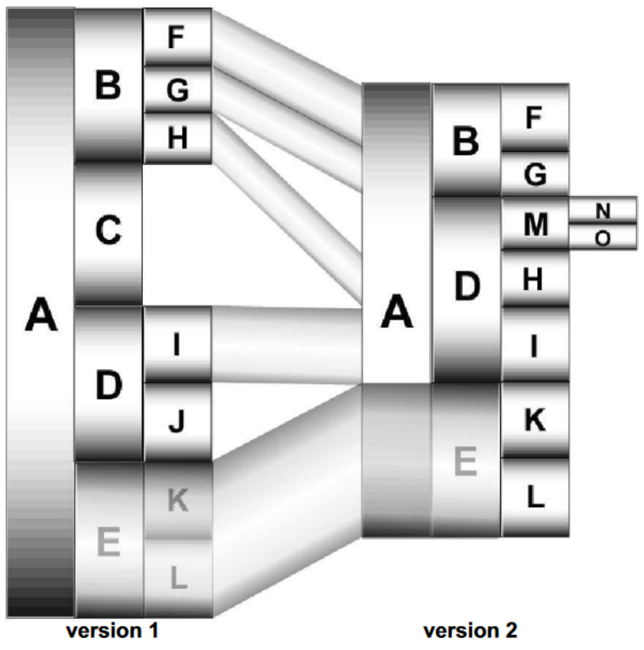
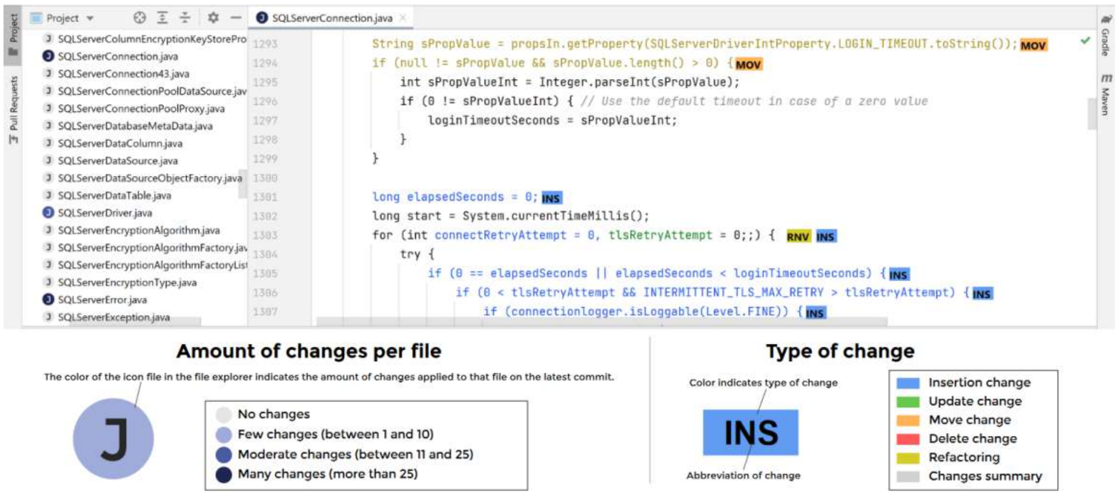


Figure 3.3: Rewiring metaphor as found in the paper by Chevalier et al. [65, p. 93, modified]


Project
SQLServerConnection.java
SQLServerColumnEncryptionKeyStorePro
SQLServerConnection.java
SQLServerConnection43.java
SQLServerConnectionPoolDataSource.jav
SQLServerConnectionPoolProxy.java
SQLServerDatabaseMetaData.java
SQLServerDataColumn.java
SQLServerDataSource.java
SQLServerDataSourceObjectFactory.java
SQLServerDataTable.java
SQLServerDriver.java
SQLServerEncryptionAlgorithm.java
SQLServerEncryptionAlgorithmFactory.jav
SQLServerEncryptionAlgorithmFactoryList
SQLServerEncryptionType.java
SQLServerError.java
SQLServerException.java
Pull Requests
Gradle
Maven
Amount of changes per file
The color of the icon file in the file explorer indicates the amount of changes applied to that file on the latest commit.
J
No changes
Few changes (between 1 and 10)
Moderate changes (between 11 and 25)
Many changes (more than 25)
Type of change
Color indicates type of change
INS
Abbreviation of change
Insertion change
Update change
Move change
Delete change
Refactoring
Changes summary

```
String sPropValue = propsIn.getProperty(SQLServerDriverIntProperty.LOGIN_TIMEOUT.toString()); MOV
if (null != sPropValue && sPropValue.length() > 0) { MOV
    int sPropValueInt = Integer.parseInt(sPropValue);
    if (0 != sPropValueInt) { // Use the default timeout in case of a zero value
        loginTimeoutSeconds = sPropValueInt;
    }
}

long elapsedSeconds = 0; INS
long start = System.currentTimeMillis();
for (int connectRetryAttempt = 0, tlsRetryAttempt = 0;;) {  RNV INS
    try {
        if (0 == elapsedSeconds || elapsedSeconds < loginTimeoutSeconds) { INS
            if (0 < tlsRetryAttempt && INTERMITTENT_TLS_MAX_RETRY > tlsRetryAttempt) { INS
                if (connectionlogger.isLoggable(Level.FINE)) { INS
```

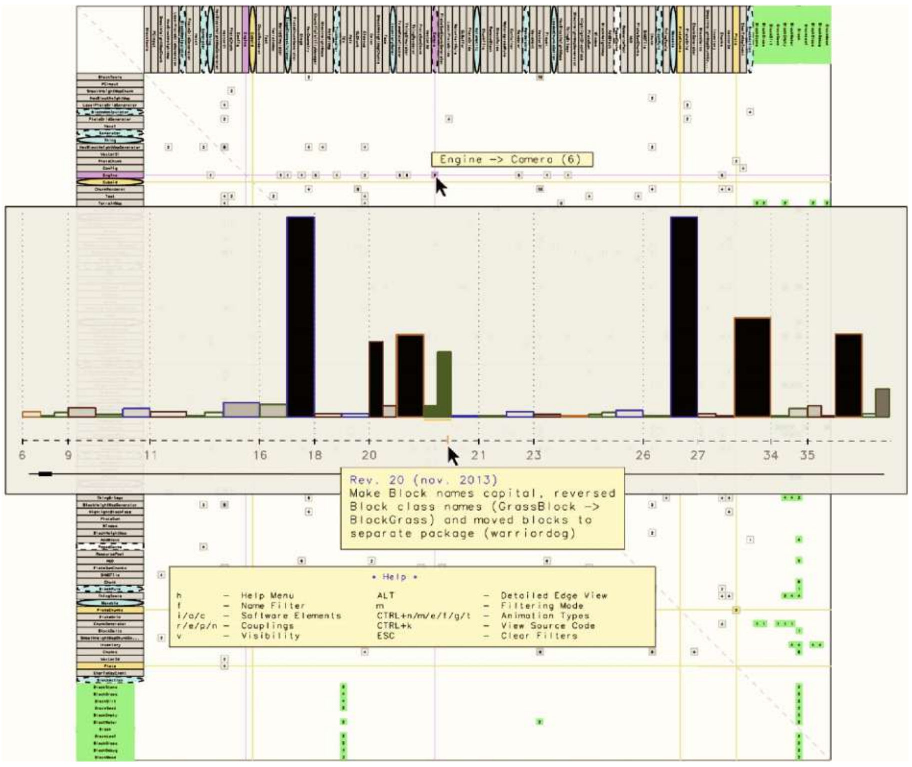
Figure 3.4: Editor overlay visualization by Escobar et al. [71, p. 168]

3.3.4 Editor-Based Approaches

Having experienced a resurgence in recent years are the visualization approaches which prominently feature a file viewer or directly integrate themselves into IDEs. Around 2010, there were quite a few papers such as those by Holmes and Begel [66], Fritz and Murphy [15], as well as Bradley and Murphy [67], which focused around the idea of adding views or overlays directly to the IDE to provide context for the current source code state and the changes that led to it through the perspectives of the commits, ITS items and people involved in those changes.

A number of approaches that followed shifted the focus more towards diff viewing, such as the previously mentioned visualization by Yoon et al. [64] and the web-based AST diff viewer DiffViz by Frick et al. [68] which has adapters for various code differencing backends such as GumTree [25]. The Code Time Machine by Aghajani et al. [69] also allows diff viewing but places more importance on code history exploration by providing a multitude of linked visualizations displaying commit frequency, date timeline, a code metrics diagram and a stacked color-coded text view display for the individual commits of a specified file.

Recent techniques again see the information placed directly in the IDE more often, such as the plug-ins for JetBrains IntelliJ IDEA from Kurbatova et al. [70] and Escobar et al. [71], the latter of which is shown in Figure 3.4. These two approaches both try to integrate the refactoring information from RefactoringMiner [27] into the version control tool windows and text editor, respectively. Escobar et al. in particular go into more detail as for their design decisions and general reception according to a user study, showing the visualization-related advantages and shortcomings of their approach.



Rev. 20 (nov. 2013)
Make Block names capital, reversed Block class names (GrassBlock -> BlockGrass) and moved blocks to separate package (warriordog)

• Help •

```
h        –  Help Menu              ALT                –  Detailed Edge View
f        –  Name Filter            m                  –  Filtering Mode
i/o/c    –  Software Elements      CTRL+n/m/e/f/g/t   –  Animation Types
r/e/p/n  –  Couplings              CTRL+k             –  View Source Code
v        –  Visibility             ESC                –  Clear Filters
```

Figure 3.5: Matrix-based visualization by Rufiange and Melancon [72, p. 142]

3.3.5 Hybrid and Isolated Approaches

Last are the approaches whose main contributions do not neatly fit any of the above categories. As for matrix-based visualizations [3], Figure 3.5 depicts AniMatrix by Rufiange and Melancon [72], which uses bar charts for timeline navigation and a matrix display to show relationships between code elements with a pronounced focus on the use of animation to communicate time-based information. Novais et al. [73] is also a hybrid SEV approach using multiple different kinds of visualization techniques: a treemap view for code structure comprehension, an inheritance tree and a dependency graph. Even more sophisticated in this regard is the approach by Kim et al. [74], whose visualization consists of six linked primary views, which, in turn, consist of two to six secondary components, including timelines, treemaps, lists, word clouds and charts. Finally, there is the previously mentioned paper by North et al. [20] which investigates the effectiveness of sound when paired with version control history visualization.

CHAPTER 4

The State of Version Control History Mining

Seeing as this work deals with the visualization of software histories, investigating ways to extract the necessary change information plays a key part in its conception. More specifically, this thesis is interested in approaches to harvesting semantically meaningful symbol-level change information from VCS histories of projects with Java as their primary programming language. What follows is a more detailed examination of the current state-of-the-art techniques for parsing and code differencing touched upon in Section 3.2.

4.1 File Parsing

Before tackling the problem of code differencing, there is the problem of file parsing. Specifically, the representation and parser used largely impact the granularity of information fed into code differencing platforms, thus also affecting the code difference accuracy. ASTs usually contain helpful, semantically labeled information, whereas CSTs focus more on directly representing the productions of the language grammar as a tree of tokens. While the latter can typically be generated faster due to remaining on a purely syntactical level, its lack of semantically meaningful information about the relationships between its nodes beyond the implied compositional relationship between parent and child nodes can necessitate additional language-dependent parsing to uncover information that would already be included in many AST representations.

4.1.1 AST-Based Approaches

For libraries that produce and analyze ASTs for Java, there is Spoon [34], which is an open-source code analysis suite which (at the time of writing) supports up to Java 20. In addition to code parsing and AST traversal helpers, it also supports AST querying with pattern matching, imperative code transformation, automatic symbol solving, as well as Maven POM declaration files. Under the hood, Spoon uses the Eclipse JDT Parser[1], which can also be used as a standalone parser but uses a different AST representation than Spoon.

There is also JavaParser [81], which is a different AST parser that features code transformation and symbol solving components, and (at the time of writing) supports up to Java 18. Last but not least, there are also AST parser generators which support the use of custom grammars to parse a variety of languages, the most notable in this context being ANTLR [75], which is not only widely used and provides language bindings for Java but also comes with a repository of ready-made grammars, including a lexer and parser grammar for Java 20.

4.1.2 CST-Based Approaches

By far the most popular library on the CST end of the spectrum is the Tree-Sitter [80] parser generator, which—despite still being in a pre-release development state at the time of writing—offers support for a wide variety of markup and programming languages and provides language bindings for Java. Its official Java parser appears to support up to Java version 19 based on the language features it can process without errors. One other noteworthy CST parser implementation for Java is the `java-parser` npm package used by the formatting tool Prettier. Notably, this parser was created using the Chevrotain CST parser generator[2], which does, however, not support Java as a target language for the generated parsers.

4.1.3 Hybrid Approaches

Besides these two general approaches to source code parsing, there are also structures which try to combine the advantages of both: Moderne's "Lossless Semantic Trees"[3] are trees conceived for their OpenRewrite refactoring ecosystem that are structurally similar to ASTs but preserve formatting information—similarly to CSTs—and include type information that would normally be either computed on top of the AST during semantic analysis in compilers and interpreters or appended by symbol solvers with AST parser libraries like Spoon [34].

Incidentally, there also exist so-called "Lossless Syntax Trees"[4], which are based on a concept known as a "red-green tree"[5] that essentially generates an AST-like view *into* a CST, thus also preserving formatting information while retaining more high-level syntax information.

[1]https://github.com/eclipse-jdt/eclipse.jdt.core, Accessed: Apr. 3, 2024
[2]https://github.com/Chevrotain/chevrotain, Accessed: Apr. 3, 2024
[3]https://docs.openrewrite.org/concepts-and-explanations/lossless-semantic-trees, Accessed: Oct. 27, 2025
[4]https://github.com/rust-analyzer/rowan, Accessed: May 4, 2024
[5]https://learn.microsoft.com/en-us/archive/blogs/ericlippert/persistence-facades-and-roslyns-red-green-trees, Accessed: May 4, 2024

4.2 History Mining and Change Detection

Next are the actual algorithms and differencing techniques used on top of the representations mentioned above.

4.2.1 Classical Tree Differencing Algorithms

Most approaches to modern tree-based source code differencing can be traced back to the paper by Chawathe et al. [23], whose algorithm was among the first well-known approaches to compute tree matchings and tree edit scripts with support for subtree move operations. Capturing subtree moves is essential since these are, of course, a frequent occurrence in source code history when moving code across functions or changing the nesting context of statements via control structures, and could be seen as a first step towards refactoring detection. However, the matching part of their algorithm makes the key assumption that each leaf node in either tree has at most one leaf node in the other tree with low update cost. While this assumption may make sense for some markup languages and schemas, it is highly detrimental for computing good matchings for source code representations where similar or even identical nodes are commonplace. An attempt to fix this was made by Fluri et al. [24], whose algorithm circumvents problems related to matchings with multiple viable candidates on either or both sides by employing similarity metrics and greedy maximum-similarity matching. These two algorithms, together with the LCS algorithm by Myers [43], form the basis for most of the code differencing approaches listed below.

4.2.2 GumTree 3.0 and GumTree-Spoon 1.69

GumTree is a commonly used Java-based tree differencing framework written by Falleri et al. [25]. Similar in flexibility to parser generators like ANTLR [75], it can be combined with parsers for all kinds of languages since it is, at heart, a tree edit script generation tool. At the time of writing, the latest plugins for Java parsing use versions of the JDT and JavaParser [81] parsers compatible with up to Java 14. By default, GumTree uses an adapted version of the algorithm by Chawathe et al. for edit script generation and therefore offers limited support for low-level intra-file refactorings such as moving variable declaration sites inside of a class.

GumTree is inherently designed for file-to-file comparisons, so changes on the file-level (e.g., moving a class) and inter-file changes (e.g., moving a method from one class to another) do not get picked up. Additionally, due to being a general-purpose tree differencing tool, the resulting edit scripts consist of actions to “nodes” and “trees” as opposed to actual syntactic code elements, leading to very fine-grained, localized changes.

Using the findings from their paper on hyperparameter optimization for AST-based source code differencing [76], Martinez et al. later developed GumTree-Spoon (gumtree-spoon-ast-diff), which adapts GumTree to work more effectively with Spoon [34] ASTs. In particular, this version leads to improved edit script quality, with changes that more closely reflect the granularity expected from developers. For example, changes may be associated with Java language constructs such as methods instead of plaintext tokens. It also allows further examination of the detected changes using Spoon's APIs. It is worth noting, however, that even this version of GumTree maintains its tendency towards minimum-cost edit script generation due to the underlying optimization scheme. This can result in some unexpected edit script operations whenever multiple categorically identical code elements are involved, as is the case when multiple field declarations, which are commonly found bundled together at the top of class declarations by code style convention, are changed at once. Still, in most cases, the resulting edit scripts are accurate enough to be acted on, be it through a library client or via Spoon.

4.2.3 srcML and srcDiff

The srcDiff tool by Decker et al. [42] offers another approach to language-agnostic file-to-file diffing. It operates on program files in srcML [39] format, which is a largely language-independent XML-based source code representation, and aims to produce more understandable and accurate edit scripts compared to GumTree [25]. In this context, "understandable" can be understood to mean "closer to the developer's intention" as srcDiff uses the LCS algorithm by Myers [43] to perform tree differencing on a level-by-level sequential basis, leading to purposefully suboptimal edit distances and avoiding the performance penalty usually incurred by tree differencing. Additionally, while srcDiff itself only outputs a purely structural diff—closer to a diff for general XML files—without any refactoring change information, it is designed to be partially *refactoring-aware*: For example, the fact that variables could have been renamed is taken into account for node matching purposes. While intriguing, seeing as this project is not listed in the tools section on the official srcML website[6] and no published artifact is mentioned in the paper itself or easily found online, this solution cannot be used as a base or library for the purposes of this thesis.

4.2.4 RefactoringMiner 3.0 and CodeTracker 2.6

RefactoringMiner is a refactoring detection framework for Java developed by Tsantalis et al. [27] and Alikhanifard and Tsantalis [77]. It is publicly available on GitHub and still in very active development, with the paper for the newest release still in its pre-publishing stage at the time of writing. This library uses a matching algorithm inspired by Fluri et al. [24] and can detect a wide variety of medium-level refactorings, including inter-file refactorings as well as more complex changes involving many-to-one and one-to-many mappings. However, with the exception of changes to methods in certain situations, it generally does not allow the retrieval of unmatched elements, necessitating the use of a separate change detector for simple changes like additions and deletions. Changes without semantic implications for the program are also generally ignored, e.g., reordering method parameters is counted as a refactoring (since it usually changes the method signature), whereas reordering methods in a class is not.

[6]http://www.srcml.org/tools.html, Accessed: May 15, 2024

CodeTracker by Jodavi and Tsantalis [35] is a code symbol history extractor based on RefactoringMiner that allows retrieving information on-demand on a per-symbol basis. In other words, given a specific code symbol, it harvests related change information from VCS history using RefactoringMiner for that symbol only. As one might expect, part of the main performance benefit of this approach is strongly tied to the fact that only one symbol is actively tracked throughout the procedure. For example, since a single symbol can only be in a single file or class, a search can be conducted in locations where the symbol was last seen, which can dramatically reduce the number of changed files that need to be examined. As such, this library works best when wanting to know the history of symbols that still presently exist in some form since their discovery only requires parsing the current workspace once. For symbols that have been deleted, however, a library client would need some kind of way to discover them quickly, which involves making sure that they have, in fact, been deleted as opposed to refactored to avoid double-searching or result duplication. Still, even for current symbols, the entire search would have to be invoked for each single one if one wants to precompute change information for all symbols, leading to algorithmic overhead so significant that this type of usage ultimately defeats the purpose of the library. It is worth noting that CodeTracker borrows its change hierarchy and overall approach from CodeShovel [6], which is a method-level history extractor using text-based method body similarity metrics for method diffing that also works on demand. However, CodeShovel is only interested in method symbols, whereas CodeTracker can track other kinds of symbols, as well.

4.2.5 HyperAST and HyperDiff

HyperAST [31] and HyperDiff [32] are part of a fairly new history-aware repository-scale tree representation and code differencing toolset written in Rust by Le Dilavrec et al. Somewhat confusingly and contrary to the name, the HyperAST representation is actually *not* based on ASTs and instead relies on the CST parser ecosystem of Tree-Sitter [80] as well as the Merkle DAGs of Git repositories. More specifically, HyperAST as a data structure acts as an extension of the Merkle DAG from the file-level to a syntactic level: Rather than including only information about individual files, this approach opts to include the contents of CST code elements in the DAG structure, as well. This, in turn, leads to spatial and temporal compression of syntax elements across the repository history as CST nodes get reused whenever possible, both within the same commit and across the entire history. The resulting DAG then basically constitutes something akin to an all-encompassing CST containing all elements from VCS history with as little redundancy as possible.

With HyperAST as the data structure, HyperDiff acts as the tree-based code differencing component. Since the DAG structure contains compressed nodes, this involves decompression when performing a diff between commits. However, since nodes include a hash of their subtree structure, decompression can be performed lazily and is only necessary if two subtrees actually differ. Other than that, the overall approach is mostly an adaptation of GumTree [25] written for Rust. In fact, for evaluation purposes, Le Dilavrec et al.

implemented a port of the GumTree algorithm using HyperAST with eager instead of lazy decompression in Rust. While their results show a large improvement in terms of overall runtime between their fully lazy approach and the Java-based GumTree baseline, both the figures and some brief remarks in the paper [32, pp. 8-9] reveal that the performance gains relative to their Rust-based GumTree implementation are much smaller in comparison, lending credence to the idea that the runtime improvements can largely be attributed to the switch in programming language. Additionally, although not completely clear from the paper or the source code of the published artifacts, HyperDiff seems to require computing the HyperAST first, meaning that significantly more memory space is needed when compared to a sequential commit-by-commit diffing approach. In any case, the fact remains that the published artifacts represent more of a prototype than a ready-to-use library, so while the overall idea is certainly interesting and relevant, it means HyperDiff is not currently suited to being part of the mining approach for this thesis due to a lack of polish and documentation.

4.2.6 Historage and FinerGit

The last class of code differencing techniques that deserves a mention here are approaches that exploit conventional VCSs to perform the differencing for them. Tools like Historage by Hata et al. [44] construct a separate Git repository and corresponding history where fields, methods and constructors are elevated to separate files. The built-in change detection mechanisms of Git are then used to find additions, deletions, moves, renames and modifications of these code element files. A major drawback of this particular approach is that, since Git uses line-based differencing and lines of code often encompass multiple code elements, changes end up getting grouped together and are usually quite coarse-grained, similar to how changes are captured in the original Git repository.

This problem is somewhat rectified in FinerGit by Higo et al. [46], who—similarly to a paper on token-level blame information mining by German et al. [45]—further split each line into separate lines for each token with its corresponding syntactic meaning; for example, pairs of parentheses get prefixed with the name of the structure they belong to, leading to fewer line-based change detection errors. Despite these improvements, this approach still exhibits multiple significant shortcomings: The algorithm is computationally intensive due to large amounts of disk operations, requires Git to perform a complete re-indexing, essentially doubles the amount of project source code on disk and cannot capture related changes such as refactorings that involve partial modifications to multiple code elements, even within the same class. As shown by Grund et al. [6, p. 1518], FinerGit also tends to produce less accurate results than other approaches and runs the risk of more easily running out of memory. All in all, while these types of approaches are certainly unique enough to deserve to be mentioned here, they ultimately appear to be inferior to the previously mentioned techniques due to these key disadvantages, at least for the purposes of this thesis.

4.3 A Note on GitHub's Symbol-Related API Capabilities

Lastly, GitHub's APIs were also considered as a possible avenue to query symbols and their related changes. In recent years, GitHub has added features to provide code navigation assistance with support for various programming languages to their website, such as a side pane with a list of symbols from the currently viewed file[7]. Under the hood, these code navigation features utilize GitHub's Haskell library semantic[8] to correctly parse source code files and extract symbol information. However, while GitHub allows for text-based symbol-related filters in search queries only, none of the captured symbol information appears to be exposed via their official REST[9] or GraphQL APIs[10], rendering this working direction a dead end.

4.4 Summary

All in all, while there are many different approaches to computing the history of code symbols, none of the papers directly deal with computing the history of all symbols over all commits, and the degree of refactoring-awareness differs from library to library. Approaches based on srcDiff [42], HyperDiff [32] and GitHub symbol information have not been taken into consideration due to multiple factors, such as limited availability and documentation. Furthermore, Historage [44] and FinerGit [46] are Git-based solutions which ultimately suffer from having to construct an entire second version control history and relying on Git's innate diffing capabilities.

This more or less leaves GumTree [25] and its surrounding ecosystem with GumTree-Spoon [76], RefactoringMiner [27] and CodeTracker [35] as the remaining options. Implementing the code differencing from scratch based on the algorithms by Chawathe et al. [23] and Fluri et al. [24] would also be a possibility, but since such a solution would presumably pale in comparison to even the simplest alternative of GumTree in terms of accuracy and take quite a bit of time to develop, this avenue was deemed not worth exploring for this thesis. Both GumTree and GumTree-Spoon are primarily designed for file-to-file comparisons and would therefore require additional care to take VCS changes on the file level into account. For the same reason, neither library can detect changes involving moving nodes across files, and refactoring detection remains rather limited. Meanwhile, RefactoringMiner and CodeTracker support taking a VCS history as input and can detect a large variety of refactorings. However, RefactoringMiner does not output any mapping information, meaning simpler changes not classified as refactorings cannot be retrieved, while CodeTracker requires information about the most recent location of each code symbol, which is difficult to determine for symbols that have been removed, and has to traverse the history for each symbol individually.

[7]https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github, Accessed: Apr. 3, 2024

[8]https://github.com/github/semantic, Accessed: Apr. 3, 2024

[9]https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28, Accessed: Apr. 3, 2024

[10]https://docs.github.com/en/graphql/overview/public-schema, Accessed: May 20, 2024

Ultimately, it appears as though a solution not dissimilar to that of Escobar et al. [71] would prove to be the most effective option: using regular AST differencing—for example, GumTree—for simple tree-based changes *in conjunction with* RefactoringMiner for refactoring detection. While running both libraries concurrently certainly incurs some computational overhead, their strengths nicely cancel out each other's weaknesses, resulting in change information on all symbols throughout the entire VCS history with support for complex refactorings to boot.

CHAPTER 5

Methodology

Naturally, in order for this thesis to yield accurate, insightful and actionable research findings, it has to follow proper structural standards for how information is attained, acted upon and evaluated. In the case of this thesis, its research results are primarily based on two artifacts: a symbol-based software history visualization concept and the implementation thereof in conjunction with the data mining and software architecture approach necessary to fully realize it. This section deals with the step-by-step procedure put in place to provide a methodical and goal-oriented process by which to design and evaluate a possible solution to the problem outlined in Chapter 1.

5.1 Research Questions

The research questions listed in Chapter 1 serve as the goals for this thesis. In terms of milestones as part of a greater work, these research questions are modeled after the design science approach by Wieringa [7] in that they map to the problem investigation, treatment design & validation and treatment evaluation phases outlined as part of the design and engineering cycles.

RQ1. First and foremost, designing a treatment requires prior context and analysis. In the case of this thesis, these components are provided by two components: a review of current academic and industrial literature, intended to identify relevant information needs related to symbol-level changes from developers, as well as a survey on information- and visualization-related requirements conducted as part of semi-structured expert interviews, both of which are described in Chapter 6 as part of a requirements analysis.

RQ2. Second is the actual treatment design & validation. For this purpose, visualization mockups were created based on the findings of the literature review. These mockups were then validated through sets of open questions from the expert interview questionnaire in order to gauge interest and get first feedback on the early stages of the proposed visualization approach, a process further explained in Section 6.2. Additionally, any possible design for a symbol-based history visualization must not neglect the underlying data mining approach, the software architecture of which was largely based on a prior analysis on state-of-the-art file parsing and VCS symbol information extraction techniques found in Chapter 4. The entire design process and all design considerations are recorded in Chapter 7.

RQ3. Last is the evaluation of the implemented treatment. This implementation, as described in Chapter 8, uses the findings from the aforementioned evaluation on state-of-the-art tools described in Chapter 4 for the data mining approach and the findings from the requirements analysis described in Chapter 6 for the implementation of a suitable visualization prototype. The resulting artifact was then evaluated using a second round of semi-structured expert interviews, this time centered around the created prototype in the context of scenarios derived from the information needs elicited and evaluated as part of answering RQ1 and RQ2.

5.2 Related Work

For identifying related work as part of literature research in Chapter 3, the following domains were primarily taken into consideration for direct searches, listed here in alphabetical order along with a selection of examined topics:

- Algorithmics: trees, file parsing, program compilation, tree differencing and edit script generation
- Code Differencing: differencing approaches on ASTs & CSTs, refactoring detection, efficient VCS history mining techniques, code representation formats
- Information Needs in Software Engineering: general software engineering information needs, VCS- and software evolution-based information needs, information needs rooted in data visualization
- Software Evolution Visualization (SEV): specialized extraction, visualization and interaction techniques for the domain of software evolution and VCS change information
- Visualization: general visualization and interaction techniques and associated use cases, use of colors and metaphors

Initial research began with the papers by Codoban et al. [1], Grabner et al. [4] and Ragavan et al. [2], then progressed mainly using direct searches, forward reference tracking and backward reference tracking. The primary platforms enabling these search methods were IEEE Xplore[1], the ACM Digital Library[2] and Google Scholar[3] due to their vast collections of scientific works and advanced search capabilities. Where appropriate, searches were also triggered by virtue of repeatedly occurring author names, as was the case with Codoban et al. and Tsantalis et al., for example.

In terms of recency, most included works were published in the year 2015 or later, with some papers as recent as 2024. However, to provide additional context for traditional issues tackled in the field of AST- and VCS-based visualizations, certain historic works dating back as far as 1957 also find mention in this thesis.

5.3 State-of-the-Art Analysis

While the state-of-the-art analysis for data mining techniques also uses the general approach of Section 5.2, it additionally considers online sources in the form of public VCS software repositories and websites of products or services found in the industry. These were primarily found with a simple Google search using keywords provided by the scientific literature examined as part of collecting relevant related work.

5.4 Requirements Analysis

Both as part of the deeper problem investigation process and as part of the treatment design & validation, a thorough requirements analysis was performed. This analysis consists of two components: the literature review and semi-structured expert interviews, which are meant to give insight into requirements from other scientific works and experts in the field of software engineering, respectively.

5.4.1 Literature Review

The literature review acts as a summary of external findings related to information needs in software engineering and Software Evolution Visualization. It accomplishes this task by comparing the papers collected in Section 5.2 to draw conclusions about commonly occurring problems or questions. These commonalities have been rigorously collated through reading and note-taking, resulting in the question table found in Section 6.1.1 and a textual description of current pain points when dealing with software history or the act of extracting or extrapolating information from it. These questions then form the basis for the perspective-based visualization mockups detailed in Section 7.2.

[1]https://ieeexplore.ieee.org/Xplore/home.jsp, Accessed: Oct. 23, 2025
[2]https://dl.acm.org/, Accessed: Oct. 23, 2025
[3]https://scholar.google.com/, Accessed: Oct. 23, 2025

5.4.2 Semi-Structured Expert Interviews

Using the findings and mockups which resulted from the literature review, the thesis then proceeds to a round of semi-structured expert interviews. In terms of content, these interviews focus primarily on the issues, possible solutions and visualization concepts from the literature review, packaged as a survey questionnaire written using Google Forms[4] with both open and closed questions, with the latter following the Likert scale [78] system for constructing low-bias answer options.

The experts in question were all from the INSO research group at TU Wien. In total, four experts were chosen for the requirements analysis, with one expert for a preliminary pilot interview and three for the actually evaluated interviews, the purpose of the pilot interview being to gather feedback for the structure and wording of the questionnaire as well as the overall interviewing process.

All interviews were conducted via an online video conferencing tool, with the participants' answers being manually entered into the questionnaire fields by the interviewer over the course of the interview. Any questions regarding clarifications were answered whenever the resulting bias was deemed low-risk; for example, terms were readily explained when ambiguities arose, whereas design decisions and intentions were not (since whether or not a design reads correctly is part of the assessment).

The results of the quantitative questions from the interview were compiled for the figures in Section 6.2.2 using Microsoft Excel, then exported as CSV once in a format suitable for LaTeX packages to consume.

5.5 Conceptualization

The initial proposal visualization mockups were all hand-drawn using pen and paper, whereas the perspective-based mockups were written in HTML, CSS and TypeScript with the Vue.js front-end framework so as to facilitate faster transfer when moving from conceptualization to implementation. Naturally, the perspectives themselves were a result of the different categories of information needs as identified in the literature review phase, and the decision to move forward with the search-based design was similarly informed by the evaluation of these concepts through the semi-structured expert interviews conducted as part of the requirements analysis. This ultimately resulted in the final visualization prototype design, which was then realized using the reactivity functionality of Vue.js as part of one half of the implementation. This overall structure intuitively reflects the iterative design process as part of the treatment design & validation.

[4]https://docs.google.com/forms, Accessed: Oct. 23, 2025

5.6 Implementation

The actual realization of the visualization prototype and data mining application likewise took place over several iterations, with each iteration leading to incremental progress on the overall artifact implementation. Iterations took place in roughly three-week intervals, separated by a thesis status meeting and a progress report on the current state of the implementation. It was also through this iterative process that a decision to reduce the scope of this thesis due to time constraints was ultimately reached, more on which is explained in Section 8.1.4. The final prototype was developed using a Docker container setup for simple and environment-independent deployment.

5.7 Evaluation

In the final evaluation stage, the implemented treatment was then judged by participants in a second round of semi-structured expert interviews. Fundamentally similar in structure to the requirements analysis interviews described in Section 5.4.2, also using a questionnaire, online conferencing, pilot interviews and the same number and source of participants, this set of interviews placed a greater focus on a scenario-based expert assessment of the implemented visualization prototype in the context of live demonstrations as presented by the interviewer.

The scenarios for these demonstrations were largely based on the questions and use cases from the literature review corresponding to the search-based visualization concept chosen as a result of the requirements analysis interviews as well as the capabilities of the finished prototype.

The final results from this evaluation were then processed similarly to the first set of interviews for illustration in this thesis, ending with a last interpretation of the quantitative and qualitative feedback received as well as a discussion of the outcomes and findings of this thesis as a whole.

CHAPTER 6

Requirements Analysis

As outlined in Section 5.4, a thorough requirements analysis has been conducted to inform the further course of action for this thesis. This step is split into two tasks, which have been addressed in this order: a scientific literature review and qualitative semi-structured expert interviews.

6.1 Literature Review

As part of the literature review, this section seeks to collate the most important requirements as identified across scientific literature. More specifically, the review corpus consists of the works outlined in Chapter 3, which likewise leads to an investigation across the same three major domains: information needs in software engineering, code differencing and software evolution visualization. Seeing how the requirements analysis step for the domain of code differencing is intricately tied to both the current state of the art and the technical feasibility of specific concepts and features moving into this thesis, it has naturally taken precedence over the other two steps, which more so concern themselves strictly with results from literary surveys, interviews and studies. Its findings and conclusions can therefore be found in the prior Chapter 4.

As for the other two domains, there exists a clearly identifiable overlap which is more or less pronounced when looking over the collected sources, i.e., many papers on software history visualization naturally include sections about the fundamental information needs that serve as the motivation for their proposed concepts, and some works about information needs illustrate ways to go about capitalizing on this in the form of proposed effective graphical representations for the desired information fragments. Thus, the following two sections are divided based on the domain with the overall subjectively perceived stronger focus for particular pieces of information, rather than the section of their respective works as part of Chapter 3, and some papers or concepts may find mention in both sections.

6.1.1 Information Needs in Software Engineering

Searching and navigating through source code history can be quite a difficult task, especially for large codebases. Developers often need to find suitable reversion points during trial-and-error development to be able to undo unsuccessful experiments [1] or gather the information required for merge conflict resolution [5]. Towards this end, developers frequently employ search-space reduction techniques by code region and date to find the information they are looking for [1, p. 4]. However, due to the high information density in VCS information and large amounts of noise in diffs [1, p. 4] coupled with a lack of proper support for recognizing move, rename and reformatting operations [2, p. 1650], relevant changes often tend to disappear in the shuffle. Today's developers most frequently employ the `git log` command in the Git CLI and IDE features like IntelliJ IDEA's history functions, yet surveys such as those by Grund et al. and McKee et al. demonstrate that tool support for software history exploration is something the majority of developers still feels unsatisfied with [6, p. 1513] and has been shown to constitute the third highest unmet need among open-source developers [5, pp. 6-7].

Proof of this issue can be found throughout numerous research papers on questions frequently asked by developers, where source code history plays an integral part in finding the answers developers seek for a significant number of them [1], [2], [5], [6], [15], [16]. A summary of the most recurring questions in this context can be found in Table 6.1, subject to interpretation of intent by the respective authors as well as the author of this thesis. With the exception of the paper by McNair et al. [21], whose questions are based on either what the authors consider typical use cases or lifted from other literary sources, all the listed papers conducted their own surveys and interviews to elicit relevant questions regarding source code evolution—as well as related topics such as code ownership and code hotspots—which were considered difficult to answer or for which tool support was deemed unsatisfactory by respondents.

As the results from the survey by Fritz and Murphy [15] show, questions regarding changes made to source code make up one of the largest categories of developer questions. The paper by LaToza and Myers [16] likewise lists many relevant hard-to-answer questions about change times, contexts, authors, recency, rationale and relationships. It further highlights the frequent need to find a person knowledgeable about the current state of the code, or "code expert" for short, which is corroborated by plenty of other sources (see Q01 and Q02 in Table 6.1), including papers like the one by Begel and Zimmermann [17], which focus more on the managerial aspects of software development. Notably, the reported questions also included questions about code structure, such as code size, dependencies and type relationships (see Q15 and Q20), which is also backed by Codoban et al. [1], whose respondents repeatedly remarked on the importance of change effects on related modules or APIs. In the case of Codoban et al., McKee et al. [5] and Grund et al. [6], important use cases for source code history are more directly highlighted due to the qualitative nature of their conducted interviews, leading to direct developer quotes which paint a clearer picture over what questions are actually being asked and which practical applications would provide the most utility.

	Question	Subject	Sources
Q01	Who is the expert for this code? (Who modified it the most?)	Code expertise	[16, p. 4] [17, p. 17] [1, p. 3] [6, p. 1512] [21, p. 135] [15, p. 178] [66, p. 2]
Q02	Who last changed this code?	Code expertise	[21, p. 135] [15, p. 178]
Q03	How has this code changed over time?	Code history	[16, p. 3] [1, p. 3] [6, p. 1512] [15, p. 178]
Q04	When was this code element created and by whom?	Code history	[6, p. 1512] [21, p. 135] [15, p. 178]
Q05	What changes have occurred in this particular timeframe?	Code history	[1, p. 4] [21, p. 136]
Q06	What code has recently changed?	Recent changes	[16, p. 3] [21, p. 135] [15, p. 178] [66, p. 2]
Q07	What have I recently changed?	Recent changes	[1, p. 3] [6, p. 1512]
Q08	Which recent changes are related to my work?	Recent changes	[1, p. 3]
Q09	Who is working on what?	Recent changes	[15, p. 178]
Q10	Has this code ever been changed?	Code stability	[16, p. 3]
Q11	Which code elements have *not* been changed recently?	Code stability	[21, p. 136]
Q12	Who changed this code?	Change author	[16, p. 3] [1, p. 3] [6, p. 1512] [15, p. 178]
Q13	Why was this code changed?	Change rationale	[16, p. 3] [1, p. 3] [6, p. 1512] [15, p. 178]
Q14	What else changed when this code changed?	Change context	[16, p. 3] [1, p. 6]
Q15	What are the type and dependency relationships of the changed code?	Code context	[16, p. 5] [1, p. 3]
Q16	Which modules are affected by this change?	Code context	[1, p. 4]
Q17	When was this code changed?	Change date/time	[16, p. 3]
Q18	Which code elements were changed frequently?	Code hotspots	[21, p. 135] [15, p. 178]
Q19	Which code elements were changed by the most people?	Code hotspots	[21, p. 135]
Q20	How big is this code?	Code properties	[16, p. 4]

Table 6.1: Frequently asked questions about source code evolution

Despite these lists of frequently asked questions providing a good starting point, alone they only paint a vague picture of how developers would realistically want to go about finding answers for these questions, focusing more on the dimensions (changes, code elements, people, . . .) of the problem. Looking into the respective papers again reveals pivotal aspects essential to improving tool support for software history exploration, with a desire for better usability at the top: Results from McKee et al. [5] suggest that usability is, in fact, *the* most desired improvement when it comes to aiding in conflict resolution, with confusing terminology and unappealing graphical presentation mentioned as contributing factors. Yet another factor that compounds this problem is the amount of noise that developers often need to contend with when trying to find the information they are looking for [1]. Codoban et al. and Ragavan et al. [2] remark that trivial code changes such as changes in whitespace or other reformattings, combined with unsatisfactory support for detecting file moves and renamings, complicate reading diffs and threaten both the validity and utility of metrics such as code ownership, which plays an integral role in answering questions about code expertise.

Furthermore, there exists a perceived lack of useful options for expressive search, filtering and grouping [1], [5], which might otherwise help mitigate these issues to some extent. Tangentially related to this is the problem of missing change context, which appears to be frequently overlooked: Developers want to view changes not *just* in isolation but also as part of a collective of related changes [5] such as changes within the same commit or in a dependency relationship [1]. Ragavan et al. [2] highlight that the frequently encountered lack of detail in commit messages presents a big problem that often necessitates looking for related changes to understand the rationale behind a change. Additionally, works such as the paper by Grund et al. [6] show the demand for a complete software history exploration software with higher-granularity change detection to aid in these endeavors. Notably, the results from the survey by Codoban et al. [1] suggest that developers use information even from uncommitted changes when inspecting source code history, and is, overall, leaning towards recent changes being more important than old changes; however, it is unclear whether inspection of uncommitted source code is indeed a prominent use case since it finds little mention in any of the other sources, even those directly citing their findings.

6.1.2 Software Evolution Visualization

Apart from the more utility-oriented aspects of the prior section on information needs, there are also challenges regarding the representation and interactivity components of usability to consider. The oft-cited paper from Keim [53] offers a good overview of the fundamentals of visualization design for data mining applications. In it, the process of visual data exploration is described as following three general steps: Initially, only an overview of the information should be presented to the user. This “overview” step should enable the efficient initial identification of interesting clusters of information. To further narrow down which clusters to take a closer look at, the second step allows for zooming, filtering and other interaction techniques to hide or highlight different areas of interest.

Finally, the visualization requires some manner of "drill-down" capability to allow the user to actually and properly navigate through the information, again either through means of highlighting or hiding.

Holding it all together is the targeted use of interaction and distortion techniques, which Keim groups into five distinct categories: *Projection* allows for a more natural exploration of multidimensional data by restricting which dimensions inform the layout of the presented information. This technique aligns nicely with the findings of the previous section regarding the use of different perspectives to provide ways for more intuitive information comprehension. *Filtering* techniques make it possible to constrain which data or properties thereof are shown or otherwise influence the way the data is displayed. For these techniques, a distinction can be made between methods to filter by selection and navigation ("browsing") or inclusion and exclusion ("querying"). *Zooming* refers to techniques which alter the fidelity of the information displayed. This encompasses not only including more details for lower-level, "drilled-down" pieces of information but also varying the resolution or granularity of information, e.g., allowing the user to change the unit of measurement for one or more data axes. *Distortion* techniques offer alternative avenues to filtering by making sure that existing detail is not lost through navigation. Instead, additional detail should only be added or highlighted, without replacing the more coarse "overview". Finally, *"Linking and Brushing"* is an interaction technique that applies whenever multiple related visualizations or sub-views are visible at the same time. This inherent connection or relationship between visualizations can be emphasized by sharing some kind of interaction state for identical or closely related data points across components, allowing the data to appear universally instantiated or "linked". A common way to accomplish this in a simple manner is to ensure that data points share the same color across different views, which is what Keim refers to as "brushing".

On a related but more general note, Merino et al. [19] outline several criteria by which visualization papers may be categorized: the motivation for the visualization ("task" and "need"), the visualization's target audience ("audience"), the data to be visualized ("data source"), the means of visualization ("representation" and "medium") and the underlying software solution that generates the data ("tool"). Turning the motivation for the paper on its head, however, reveals that these features may just as well be used to examine and solidify the basis of an in-progress visualization approach, providing some more general points to keep in mind during both the design and implementation phases. Suffice to say, the above techniques and pointers from Keim [53] and Merino et al. [19] in particular have had an effect on this thesis from the ground up, starting at the proposal stage, where the initial draft already took these aspects into consideration. This is further expounded upon in Section 7.1, which delves into more detail regarding early design considerations.

Apart from these more general aspects, there are, of course, more specialized design decisions to consider. Looking at the questions compiled in Table 6.1 from the previous section reveals that frequently sought-after information typically concerns more than one information domain. For example, some questions might place a stronger focus on the source code and its evolution, whereas others put a stronger emphasis on the people working on said source code and their actions throughout development. It therefore stands to reason that compiling all of this information into one single visualization—or, at least, a single *view*—is tantamount to underestimating the complexity of the task at hand and overestimating the amount of information that can be contained by one coherent and intuitive visualization. Indeed, there have been cases where less information has already resulted in severe information overload, such as the network graph visualization by North et al. [20]. It follows that splitting the visualization into distinct perspectives for different use cases is a reasonable approach to take. When put together to form a cohesive whole, this method has been shown to help when faced with a diverse set of fields of interest as inputs to a visualization-based solution [60]. However, this also warrants extra caution, as the issues pointed out by McKee et al. regarding the sharing of terminology and presentational patterns in service of a consistent experience apply just as well to such components within the context of a single visualization.

Regarding the general type of representation, approaches such as that of Yoon et al. [64] have shown the feasibility of timeline-based visualizations for communicating software evolution. Additionally, since VCS history constitutes the primary data source proposed by this thesis, the stronger emphasis placed on the time dimension of data plays another crucial role in reaffirming the pertinence of this type of visualization for the kinds of use cases represented by the questions in Table 6.1, which almost always relate some manner of subject to timed events. The originally intended target medium of a web application is likewise based on the interpreted findings of papers such as those by Holmes and Begel [66], which provide valuable insight into the difficulties of working with limited screen estate for SEV. Considering this medium, the innate relationships between the different kinds of data points in software evolution, as well as the points regarding "Linking and Brushing" outlined by Keim [53], it is also clear that the methodical use of colors can greatly help in conveying and connecting information. Visualizations such as those by Kuhn and Stocker [63] provide a clear example of how colors can be used effectively to provide information at a glance, with participants from the study by North et al. [20] actively remarking on the help provided by the use of color. The latter study additionally demonstrates that, despite its advantages for swift visual communication, iconography needs to be used with care, since using it on its own can be prone to omitting the very information it is meant to convey; for example, some respondents mentioned the difficulty added by solely relying on developers' profile pictures without any indication of their name. Lastly, it is important to note that the study likewise found that one should be careful when handling additional multimedia output such as sound, which appeared to cause disorientation among some participants when used as an integral visualization element.

6.1.3 Literature Review Findings

Summing up, the findings of the literature review provide a clear indication of how to proceed with the semi-structured interview phase of the requirements analysis. Past works reveal that there are numerous questions related to developers' information needs regarding software history, which not only involve a diverse set of aspects and information domains, but have also yet to be addressed in a satisfactory way by a validated approach to SEV. While it has become clear that it is difficult for one visualization to address all these elicited concerns, using approaches such as different perspectives or axes can yield better coverage across these information needs. Regarding current sentiment, common gripes with existing solutions mainly involve lacking navigational options such as search, filter and grouping functions as well as noisy or coarse output due to missing filtering options, insignificant changes and lack of refactoring support, with many developers stating they would favor change detection below the standard file level found in many other pieces of software, such as Git itself. Furthermore, since change interdependencies are also largely ignored, many detected changes lack the supporting context necessary to understand them in the context of the software's history. Lastly, goal-oriented visualizations working towards a solution to these issues should orient themselves after existing nomenclature and representational patterns to improve usability and keep the overall developer experience consistent but take care to avoid common pitfalls such as information overload or overreliance on visual metaphors and iconography over clarifying symbols such as text.

The hypothesis to be confirmed during the semi-structured interviews proposes that symbol-level information can be used to address these concerns in a satisfactory way, leading to an answer to RQ 1 as stated in Chapter 5. The issues discovered during the literature review seem to bolster this hypothesis in a myriad of ways: Since symbols are the very building blocks of software, they are fine-grained, intuitive to grasp in the context of mental models of software (i.e., people naturally associate properties of changes with specific code symbols for the aforementioned reasons), provide clear links to other changes and symbols based on a project's software architecture and its evolution, can be easily identified and searched for, operate on the syntactic level and thus innately ignore whitespace and other reformatting changes, and can be traced with refactoring tools to enable move and rename detection as well as filter out non-essential changes.

Moving forward, this thesis proceeds with an adaptation of the first visualization prototype from the initial proposal, which offers a stronger and more streamlined focus on the time-oriented data aspect of software history as opposed to the other two. To this end, the literature review suggests five different perspectives of interest which are worth further investigating in the interview phase: an exploration-based view which focuses on the history of the software architecture as-is, a search-based view which focuses on finding information related to specific code symbols, a ranking-based view which focuses on finding information by way of comparison using different metrics, a people-based view which focuses on finding changes related to what components different people have contributed to, and a change-based view which focuses on the effects of individual changes on symbols related to the change itself or the change subject's placement in the software architecture or type hierarchy. Section 7.2 goes into more detail about the impact of the literature review on the design of the visualization mockups that were used for the semi-structured expert interviews.

6.2 Semi-Structured Expert Interviews

To cross-verify the findings of the literature review, as well as to shed new light on relevant topics either specific to this thesis or not heavily focused on by the previously discussed papers, a series of qualitative semi-structured interviews was designed and carried out. To this end, a questionnaire consisting of both closed and open questions was formulated, which was refined over the course of the overarching interviewing phase. The final version of the questionnaire can be found in Appendix A.

As outlined in Chapter 5, the overall interviewing process was split into two parts: a pilot interview and three evaluated interviews. The pilot interview saw an expert give their opinion on the structure, quality and clarity of the provided questionnaire, as well as feedback on the overall quality of the survey itself. Parts of the feedback received also stem from personal impressions and interpretations of the concrete interview proceedings, such as encountered misunderstandings and instances of general confusion, rather than explicitly from the interviewee. The response from this first interview ultimately resulted in several changes to the questionnaire and interview process, which were then used for the main set of interviews.

All interviews were carried out individually using an online conferencing tool and lasted for roughly an hour each, with the notable exception of the pilot interview, which took about 75 minutes. The actual questionnaires were filled in by the interviewer in response to the participants' answers, which were usually quick and to the point for many of the more straightforward multiple-choice questions, such as those of the first block, which centered mainly around demographic characteristics. Whenever participants hesitated or sounded unsure when giving their answer, they were subsequently asked about potential perceived ambiguities and to either reaffirm or change their choice as they saw fit. Questions that arose about the ongoing topic—or survey in general—were generally met with short explanations in an attempt to clear misunderstandings, disambiguate used terminology or clarify components of the visualization prototype mockups, whereas questions concerning details such as technical implementation specifics or general sentiment were politely dismissed to avoid drifting into side tangents and causing cognitive biases, respectively.

6.2.1 Interview Design

Following the literature review, the general design goals of the interview questionnaire were defined as follows:

1. Establish the respondents' demographic makeup.
2. Cross-verify the findings and hypotheses found in literature.
3. Validate the vision for the visualization approach to symbol-level change detection.
4. Gauge the utility of different source code history features and interaction options.

Based on these design goals, the questionnaire is split into four major blocks of questions, beginning with a block dedicated to demographic characteristics such as age group, identified gender, and year range of professional programming experience. The purpose of this first block is mainly to establish an overall impression of the group of participants with the intent of forming a basis for the credibility of the survey results.

The second block, titled "Current State of Software History Usage", addresses the second design goal by way of eliciting information about day-to-day use cases for software history, as well as misgivings associated with state-of-the-art solutions. The questions in this block are based on insights stemming from the literature review, consequently asking about the tools used to check source code history, importance and frequency of consulting historical data in the context of software engineering, perceived pain points concerning the experience with modern tooling, perception of "recentness" and relevance of uncommitted changes, views on easily understandable terminology, weight of changes relating to elements absent from the current codebase, as well as overall sentiment regarding change detection granularities below the commonly encountered file level. In general, the questions are arranged in such a way that more specialized topics are often preceded by more generalized topics, to target progressing through the interview incrementally and avoid anchoring effects or consistency bias.

The third block serves as the introduction to both the general visualization concept and the individual perspective-based views as illustrated by the prototype mockups. Each section of this block focuses on a different view, explaining the intent behind the offered navigation options, and asking the respondent to evaluate the perceived usability in addition to the usefulness of commonly found properties of code symbols or changes thereof for the purposes of searching and filtering. Importantly, the first two questions of each section concern the participant's quantitative and qualitative assessment of the specific featured view, respectively, in order to get both an impression of general sentiment and more detailed constructive feedback.

The fourth and final block contains questions regarding general code symbol change comparison as well as overall feedback and improvement suggestions, closing the interview with a brief review of mentioned oddities and points from previous discussions around the individual visualization perspectives.

Answer options for most multiple choice questions follow either a five-point Likert scale [78], although the actual formulation of the option text varies by domain—e.g., importance or frequency—and some questions instead opt for leaving out a neutral option, reducing the range of options to four. Generally, neutral answers were allowed to give participants the ability to express their honest opinion without having to choose whether to lean more strongly in a positive or negative direction; for example, questions centered around "usefulness" all use "Useless" (or "Not useful"), "Rarely useful", "Sometimes useful", "Useful" and "Very useful" as answer options. For the purposes of establishing significance, however, questions about VCS tooling and pain points with current systems instead use "Never", "Rarely", "Sometimes" and "Often"; also in part because an "Always" or "Very often" option would not make much intuitive sense when talking about irregular occurrences in a non-comparative context, whereas never having used a specific tool or encountered a specific type of problem is notably more feasible. Other frequency-based questions do include an "Exclusively" or "Very often" option. The only importance-based question concerns the relevance of code symbols deleted without refactoring and also uses just "Not important", "Slightly important", "Moderately important" and "Very important" for similar reasons.

After the pilot interview, several parts of both the questionnaire and the overall steps of the interview process were changed according to explicit and implicit feedback. Initially, the explanation of the thesis and its topic at the beginning of the interview was significantly shorter and solely delivered through spoken communication. Similarly, the visualization mockups were originally explained incrementally throughout the interview whenever specific aspects of particular views became important to conveying the workings and intent behind their design. In the finalized approach, the interview begins with a more comprehensive—albeit not necessarily more in-depth—explanation aided by a screenshot of a commit page on GitHub. Additionally, following confusion about the fact that all prototype mockups are based on the same visualization concept, the relationship between the different views was explicitly highlighted, and a general explanation of common UI elements was included before delving into the individual visualization views, implicitly leading to parts of the explanation that was originally given at the beginning of the first section on the exploration-based visualization being cut. Thankfully, it became apparent during the main set of interviews that these changes—along with minor adaptations in wording and terminology for increased clarity—indeed led to avoiding all major misunderstandings encountered during the pilot phase.

6.2.2 Interview Findings

This section will briefly review the results of the three main interviews conducted over the course of the requirements analysis phase.

Demographics

Questions 1–3 pertained to basic demographics questions—namely, the participants' age group, declared gender and years of professional programming experience. Important to note is that professional programming experience was verbally communicated to refer to years working on programming projects with stakeholders outside of a self-educational context; in other words, projects for school or one's own university studies were considered to be excluded. As for the results: All three participants belonged to the 25–34 year age group and exclusively identified as male. Participants P1 and P2 both specified 5–10 years of experience, and P3 2–5 years.

Current State of Software History Usage

Questions 4–12 pertained to the current tools and usage behaviors when interacting with software history via version control. When asked about which kinds of VCS they use for source code, all three participants indicated they exclusively use Git. As for which tools they use with or on top of Git, Figure 6.1 gives a concise overview of the given responses. Interestingly, all three respondents primarily use online platforms and code editors for interacting with Git, but differ in how they use Git outside of these methods: P1 uses both UI tools and the command line, P2 exclusively sticks to the command line and P3 rarely uses any such separate tooling, save for dedicated tools in certain cases. Regarding how frequently version control information is checked, both P1 and P3 had to look up information within the last week, answering with "in the last 3 days" and "this week" respectively. P2, on the other hand, last checked in on the state of their project's

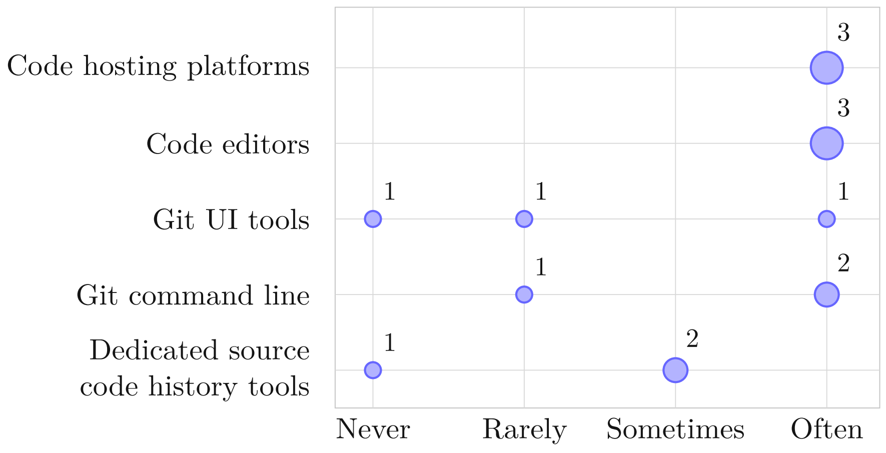


Figure 6.1: Tools used to browse or inspect source code history

software history “earlier this month” (between 2 and 4 weeks ago), clarifying that they were currently keeping to committing rather than inspecting past changes.

On the question of common problems encountered when using tools to read or interact with software history, Figure 6.2 shows which of the provided pain points gleaned from the literature review apply to real-world contexts in the eyes of the interviewees. Clearly, the issue of “inconsistent terminology” seems to be less relevant than expected, with all three remarking on how this did not come up in their daily work very often, if at all. On the other hand, problems regarding difficulties with finding the right change in a series of commits appear to be universally understood; P2 commented on how finding changes in general heavily depends on the team’s “commit discipline” since larger commits usually tend to actually contain multiple independent changes, leading to vague commit messages, which in turn leads to confusion when trying to locate a specific change. The second-largest problem all participants seemed to agree on was related to irrelevant changes: P1 mentioned that existing solutions such as “ignore whitespace” options are insufficient to deal with this problem, explicitly naming modifications such as moving braces and the addition or deletion of trailing commas. In a similar vein, P2 noted that the frequency of reformatting changes would become a more prominent issue in the near future due to the recent rise of AI (referring to the increased use of LLM services for the purpose of composing and changing source code).

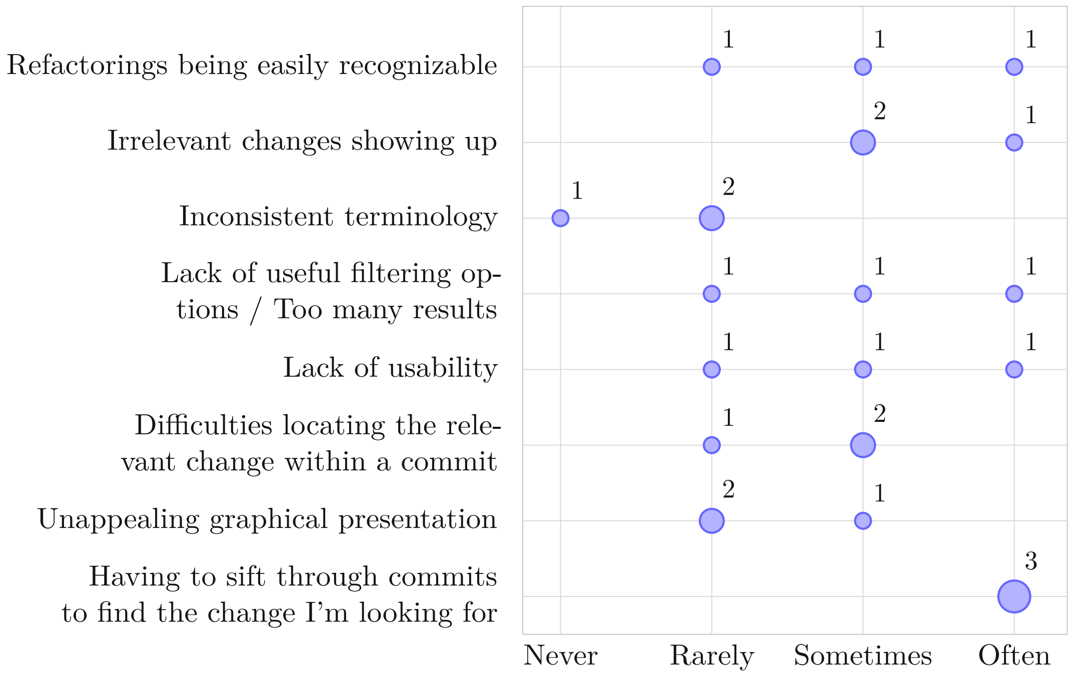


Figure 6.2: Problems when working with software history tools

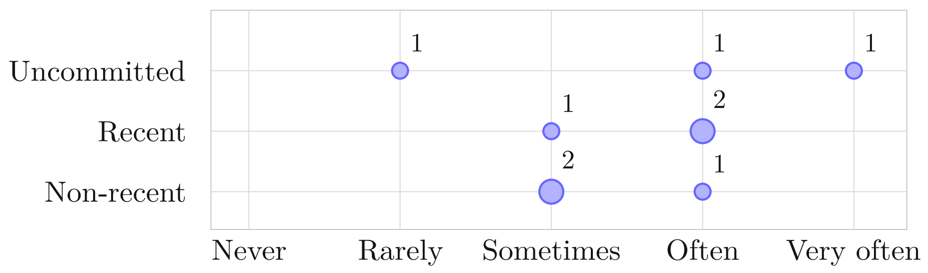


Figure 6.3: Age of changes usually checked

Questions 8 and 9 address the perceived importance of recent changes when it comes to inspecting source code history. Regarding the scale of what constitutes as "recent", all three participants gave very different responses. P1 and P3 both agreed that the rate of progress in a software project, measured by the frequency of commits, should be what decides what is recent and what is not. Similarly, P2 mentioned how the timing of releases often leads to a mental model in which the current release cycle informs the concept of recency. Generally, when it comes to the difference between uncommitted, recent and older changes, the results shown in Figure 6.3 paint a somewhat ambiguous picture. Taking all into account, it would seem as though the difference in overall importance is slight at best, with "Uncommitted" and "Recent" each scoring just one point higher than "Non-recent" in total. Interestingly, the relevance of uncommitted changes appears to be somewhat disputed, seeing as P2 mentioned how they always perform detailed checks before committing, whereas P3 generally only ever checks already committed changes.

The last questions in this block, ranging from 10–12, center around opinions on symbol-based software history in general. First, similarly to the prior question about the meaning of recency, respondents were asked to decide on an umbrella term for classes, methods, fields, variables, parameters, and the like. Surprisingly, only one of the three participants—P3—agreed with the common term for these entities found throughout literature—"code symbols". P2 favored the term "code units", and P1 found none of the options provided satisfying, including the option of such an umbrella term not existing. All in all, this would suggest that there is indeed no common colloquially known term for these kinds of entities, so referring to them by name or listing them might prove more effective in overall communication. Importantly, when asked about the concept of change detection more fine-grained than the conventional line diff-based approach, such as on a symbol level, **all participants responded positively**, giving one "Useful" and two "Very useful" ratings, showing the appeal of the general premise of this idea. The final question of the block asks about the importance of fully removed code symbols. While P1 commented that they usually have no need for checking such elements, they still thought that the option for doing so is at least "Slightly important". P2 and P3 both gave a rating of "Moderately important" but likewise agreed that this is a lower bound, and both considered giving a rating of "Very important". P3 went further into detail, describing a workflow where recovering the source code of an erroneously removed code fragment is not too rare of an occurrence.

Symbol Properties and Search Filters

Questions 13–21 pertain to feedback for the developed visualization prototypes. Of these questions, numbers 14 and 16 address sentiment towards the utility of general history-related symbol properties and search filters, respectively. Beginning with symbol properties, Figure 6.4 shows the responses of the interviewees. The only somewhat notable outliers are the "Person who last changed it" and "Number of people who contributed to it" options, with the former receiving slightly more positive and the latter slightly more negative feedback than other options. Concerning the latter option, participants commonly remarked that the number of contributors alone usually tells very little, with the concrete list of contributors constituting more useful information.

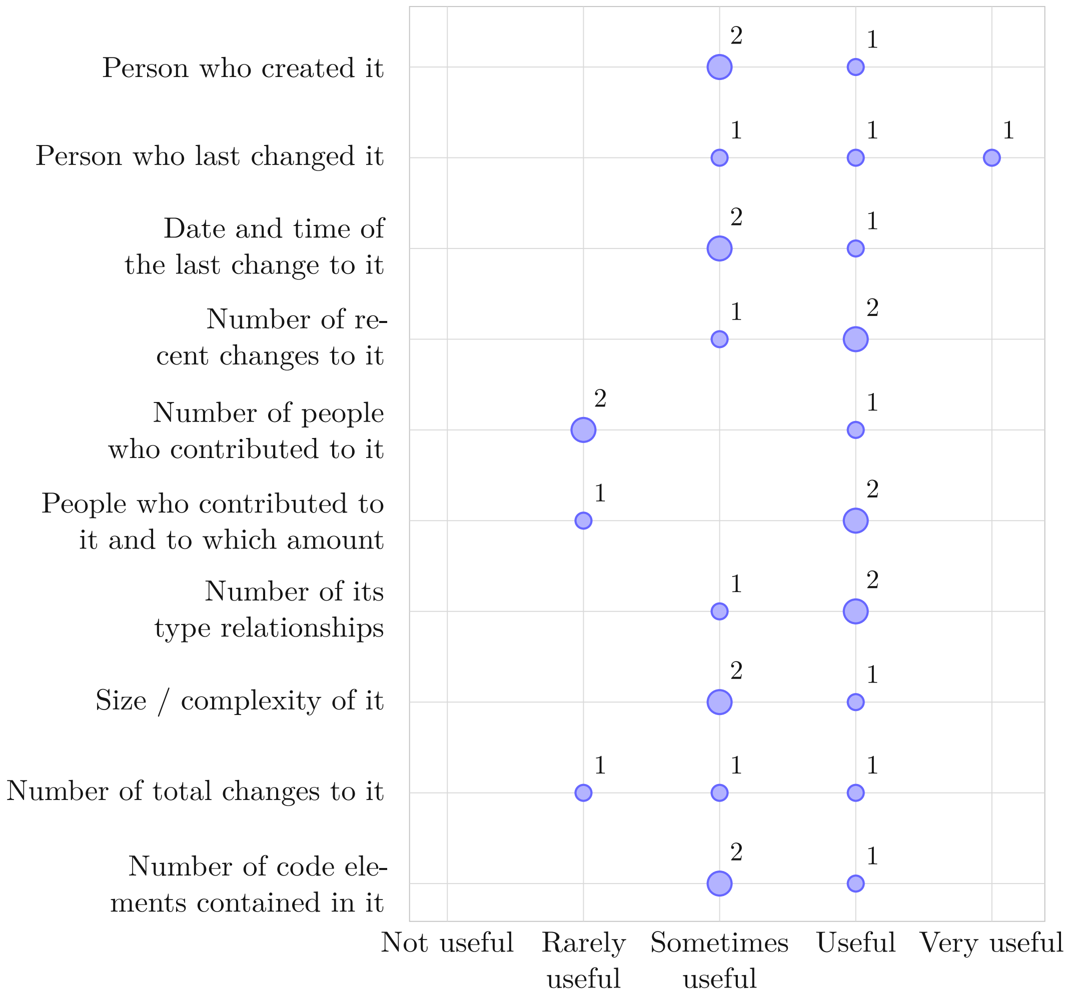


Figure 6.4: Properties of code elements and their perceived usefulness

When it came to search filters and other possibilities for speeding up finding a relevant symbol or change, Figure 6.5 likewise shows two main outliers, being the last two options, "Never changed" and "Changed by a lot of different people", which were both universally received as being somewhat situationally useful but still necessary for a handful of use cases. Among the most common additionally requested options were "In a specific package" and "Matching a supplied query", which were all mentioned by every participant. P3 went into further detail, noting that a declarative query language which allows matching by path, name, type relationships such as "implementing a specific interface" and visibility in a similar fashion as XPath would be very useful.

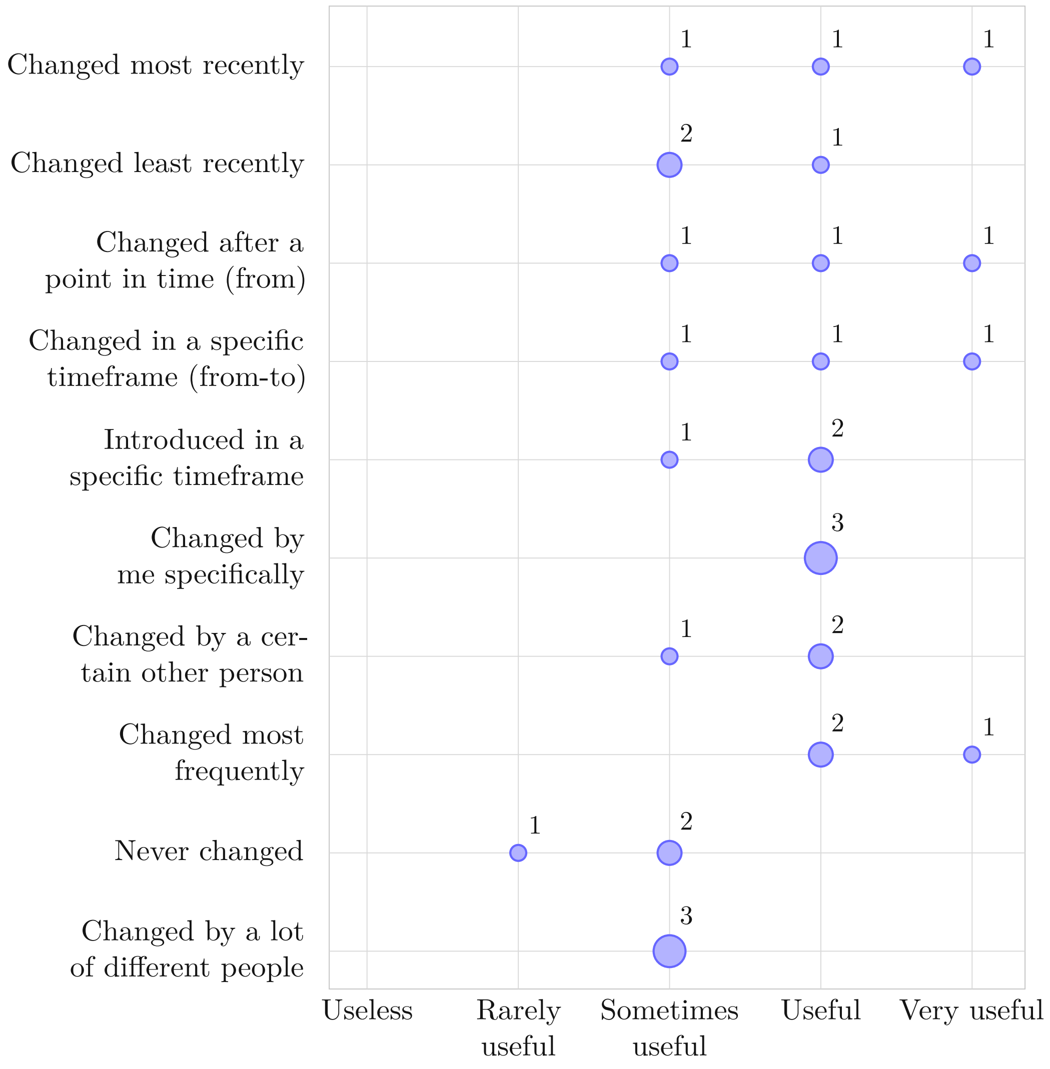


Figure 6.5: Search options and their perceived usefulness

Visualization Concepts

Finally, all remaining questions in the third block revolve around the concrete visualization prototypes as described in Section 7.2. Figure 6.6 illustrates the general sentiment towards the individual views. During the interviews, it became very clear that the search view represented the most desired conventional interface through which to interact with symbol-level change information. All three participants held the opinion that search is by far the most important and fundamental functionality for a visualization concept like this. In contrast, the list view was largely seen as a lesser counterpart to the search view. P2 remarked on how this prototype functioned the way he would imagine the search view to work when no search term is entered. Adding to this, however, P3 mentioned how such a "global" view would likely become confusing in the presence of a history with many commits and symbols. When it comes to the change-to-change view, P1 and P2 both thought that while this represents an ambitious idea that could prove quite useful in practice, the actual technical implementation and approach to clustering dependent changes in a semantically accurate way was perceived to be very hard or even infeasible. Additionally, P1 initially had difficulties identifying what the view actually does and what the common link between the displayed symbols was. Whether the change-to-change view depicts a specific class or a specific instance of a change was also largely seen as unclear.

Most of the criticism received applies to general decisions on what to show and how to represent it in the visualization. Despite many points only being brought up in the context of specific views, the similarities between the views as part of a general visualization concept ultimately implied that the same issues would arise in the other views, as well. First and foremost was confusion about the iconographic elements of the visualization. P2 and P3 both had difficulties with parsing the meaning of the icons shown, both those

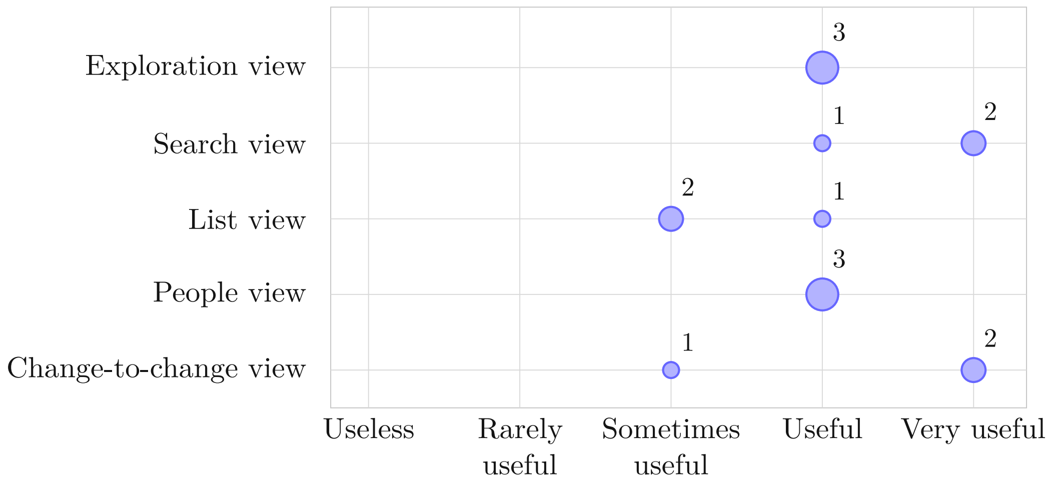


Figure 6.6: Usefulness of visualization concepts

present besides each symbol entry in the left-hand list and the icons for the different kinds of change events on the timelines. Second, the question of how changes would be displayed when multiple instances fell into the same timeframe on the timeline was raised. P1 in particular noted that this could become quite confusing and was unsure about whether or not the idea of "stacking" changes would work.

In terms of missing information, P1 mentioned they would have liked some kind of indication of where the displayed symbols are actually located within the codebase, or even just what package or class they belong to. P3 continually emphasized the importance of additional information such as a diff in a dedicated pop-up when wishing to inspect the change events displayed on the timeline in further detail. For missing features related to the core visualization concept itself, the only comment came from P2, who would have enjoyed being able to compare symbols at different change events even for non-adjacent symbol states, e.g., to determine the aggregated differences for a symbol over multiple change events. The "Compare" feature brought up at the end of the questionnaire was generally received as a good idea, with all three participants giving it a rating of "Useful".

Overall, general feedback given at the end of the interview was by and large very positive. P1 remarked on how such a tool could be very useful for observing progress in student projects at the university, such as evaluating whether each student of a group regularly contributes to their project. P3 said the design of the visualization was quite nice and praised the amount of thought that must have gone into the current iteration.

CHAPTER 7

Conceptualization

Needless to say, the realization of this thesis involves many different planning and design steps. This chapter is dedicated to the results of the iterative processes of these steps; in other words, it serves as a summary of any challenges that occurred during planning as well as the evolution of—and reasoning behind—the design decisions made to accommodate new insights gained from previous steps.

7.1 Initial Proposal

Starting at the thesis proposal stage, the initial design goals were intentionally based on broad appeal. While it was clear that there was a significant demand for works delving into more sophisticated syntax-aided Software Evolution Visualization, the issues which were hypothesized to benefit from symbol-level code information were both complex and diverse. Without conducting thorough literary research beforehand, the initial design goal was therefore to demonstrate the viability and benefits of this approach from a more generalized perspective based on the more commonly encountered problems such as the lack of high-granularity changes and change discovery as well as possible intersections of such low-level change information with other data sources, such as developers' change ownership. This resulted in a trifecta of possible working directions for the visualization, which were all included in the thesis proposal presentation.

7.1.1 Flow-Based Visualization

The first of the original concepts is based on the idea of "flows" as found in so-called Sankey diagrams [79]. Sankey diagrams are a kind of flow diagram in which the transfer, intake and outtake routes of resources are modeled with stripes of varying thickness depending on some metric, providing a clear picture of the nature of the relationships between two corresponding sets of data points, which may represent the before and after states of a

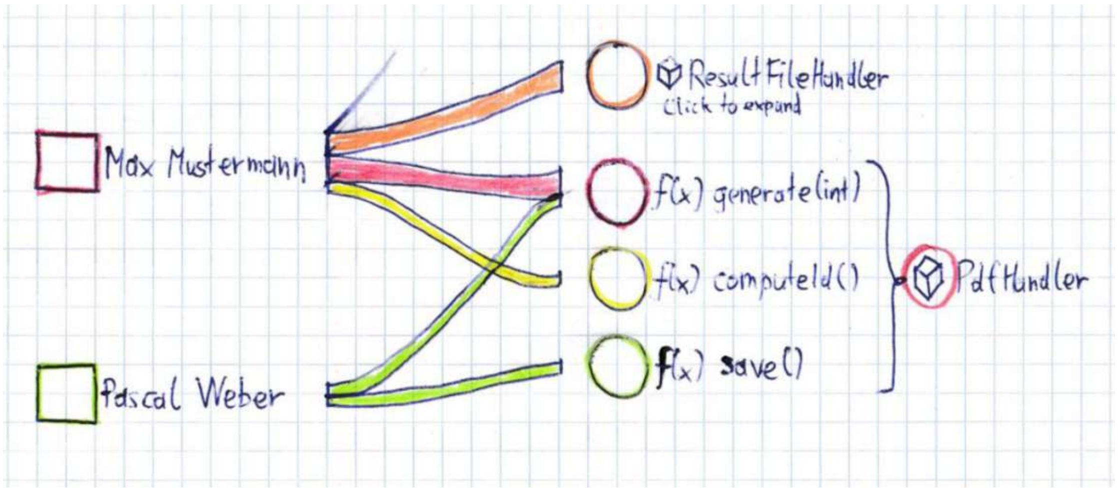


Figure 7.1: Sankey diagram-based concept

resource-dependent process (e.g., energy production/consumption diagrams), belong to the same domain at different states (e.g., voter flows between political elections) or depict a correlation between data points of different but related domains (e.g., composition trees). In fact, one of the presented prior works by Chevalier et al. [65] shown in Section 3.3 used a very similar visual metaphor to show changes between two versions of code, and other like-minded approaches can be seen in VCS diff viewers built into IDEs.

As seen in Figure 7.1, this mainly manifested in a concept from the third of the just-mentioned categories of use cases for Sankey diagrams, demonstratively showing the relationship between change authors and code elements in this case. As such, the connecting stripes represent the relative total change ownership for each pair of data points, with the absence of such a relationship—e.g., Pascal Weber not having touched the `computeId` function—naturally resulting in the absence of a connecting stripe. To further emphasize the “amount” of changed code and follow the examples seen in Chapter 3, the stripes are additionally colored in a red-to-green color palette, with red representing the largest and green the smallest size. Similar logic applies to the contours of the developer and function nodes: Here, red represents the subject or object with the largest amount of total changes and green the subject or object with the smallest amount. In order to reduce clutter, code elements are innately grouped according to their container hierarchy (e.g., functions in classes, which are found in packages, which are part of other packages, . . .) and container elements may be expanded to show additional detail for the child elements contained therein. This was directly inspired by the “distortion” techniques mentioned by Keim [53], in the sense that coarser information is still preserved when drilling down: The `PdfHandler` node remains visible despite having been expanded. Other touches also found in the other approaches are profile pictures for developers, icons to highlight the different types of code elements and a text prompt that acts as a hint to click on nodes for further details.

Seeing as version control history consists of time-dependent data, such a visualization would logically utilize only a fraction of data at a time, as opposed to showing all changes at once, which could result in quite the extreme information overload for larger projects. More precisely, the part of software history displayed may depend on restrictions on either the time axis itself—e.g., a given start and end date or a starting commit—or the relationships of the resources displayed; for example, one could imagine only listing the changes of a subset of developers or only those which affect a certain subset of code elements. Notably, this kind of mapping-based visualization also indirectly serves to show several other categories of information for the selected data subset, such as where multiple developers worked in the same code areas, which symbols have been edited by a large number of people, and—if viewing changes from a specified timeframe—how focussed development efforts were at the time (i.e., "Did most developers only touch a handful of components or did everyone have a hand in everything?"). Aiding in finding relevant information is the increase in granularity afforded by symbol-level code history, which avoids the introduction of unwanted connecting stripes due to insignificant changes (e.g., formatting fixes) while also allowing greater precision for determining the aforementioned change "amount", which would otherwise likely use metrics such as LOC counts that may be vague as far as judging developers' contributions goes.

7.1.2 Orbit-Based Visualization

The second of the original three concepts was heavily inspired by astronomy and the idea of "orbits". Seen in Figure 7.2, this more playful metaphor views code elements akin to planets, around which the developers—which can be likened to spaceships—orbit. The planets themselves can be considered to be part of "systems" according to both code hierarchy and co-change evolution history, i.e., placing code elements often edited together in some sense (e.g., frequently edited in close succession, often changed as part of the same commit, edited by developers with high activity similarity, . . .) closer together. Importantly, the developer nodes leave behind a trail of their trajectory throughout the codebase, with any components touched resulting in a "stop" of their "tour"; in other words, the trail will closely pass or orbit around code elements edited by the respective person. Just like how the code elements retain the same type of red-to-green contour as in the flow-based design, the trails themselves are also colored using the same palette: Here, red signifies more recent activity, while green indicates old activity.

Although the illustration may not depict it clearly, the trail is also implied to shrink in size the further in the past the activity lies. Additionally, while not strictly based in any physically accurate context, this visualization is intended to take some semblance of inertia into account: If developers spend more time on a certain code element, then not only does their icon linger around the code element for longer but the color gradient and size of its corresponding trail gets "compressed" around that code element, as well. In other words, the trails of developers may not only differ in length but also in how their respective sizes and gradients *change along their lengths*. This is one of the reasons this concept in particular can be said to place a much stronger emphasis on the qualities

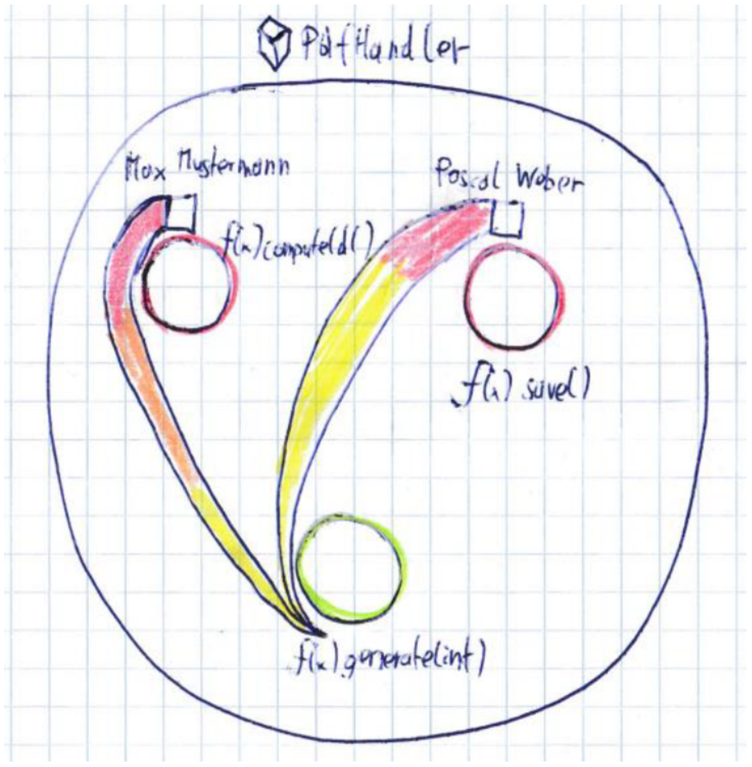


Figure 7.2: Concept based on developers touring around code elements in orbits

or modalities of development progress in a literal sense—i.e., how codebase change is progressing over time—than the other two designs. In this sense, it is also the only concept to more directly deal with the idea of higher-order changes, or "changes in change behavior", if you will. After all, this is directly reflected in both the number and the color gradient (or size) change of trails.

Arguably, this form of data representation is the most heavily time-oriented of all three concepts since the timeframe shown can affect not only the direct trajectory of the developer nodes but the quality of their trails, as well. For this reason, this visualization can only really work when applied to a continuous section of history, as "jumps" would make the information too difficult to efficiently parse. Therefore, what this concept promises in versatility, it loses due to its limitations; for example, search- or filter-based use cases would require a thorough overhaul as they just do not nicely align with this core idea. Still, in terms of benefits from symbol-level change information, it can be said that this concept could only really work when paired with high-granularity data, as the visualization would otherwise suffer from loss of expressiveness and accuracy, resulting in more sporadic node movements and less logical planet/system distances due to insignificant changes being picked up and less precise co-change detection, respectively.

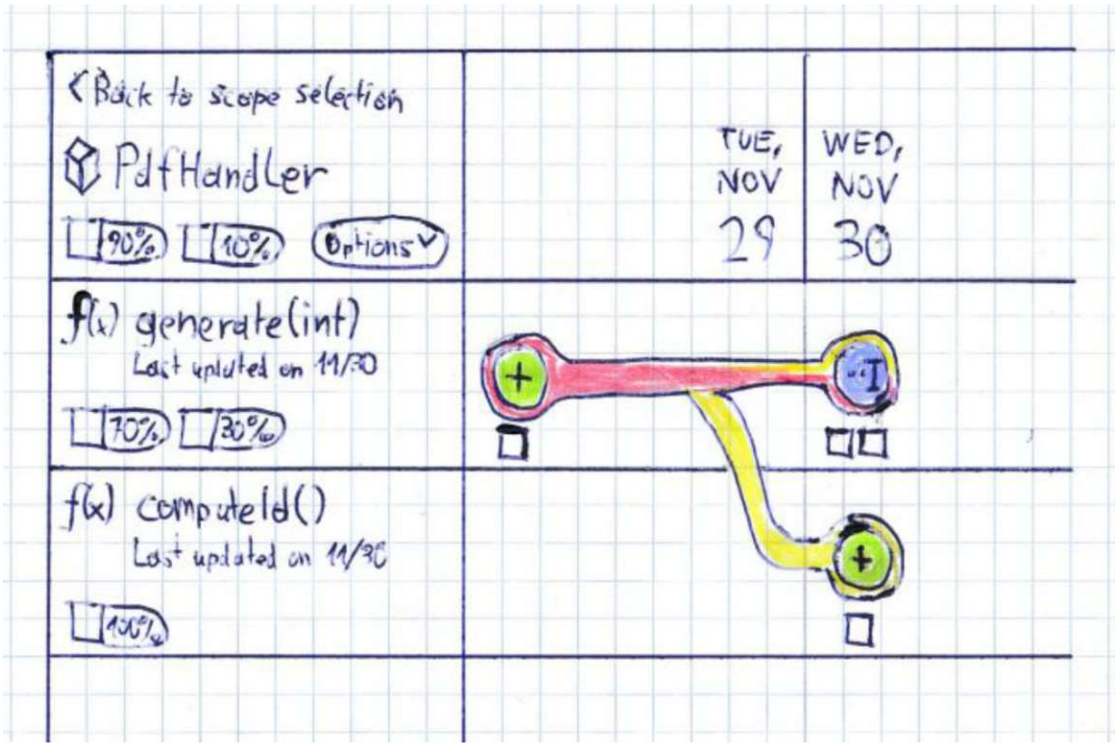


Figure 7.3: Concept based on a 2D matrix correlation view

7.1.3 Matrix-Based Visualization

The third and final concept uses a more traditional matrix view, directly depicting the overlaps between two kinds of data points. In the form shown in Figure 7.3, it directly correlates code elements with past dates to show exactly when which symbol was changed, how, by whom, and more. The change events themselves are represented as nodes whose background color and icon correspond to the type of event: For example, the green nodes symbolize the event of symbol creation, whereas the blue node on November 30 is meant to show that the function `generate(int)` was renamed on that day. Below each event node, there are squares meant to represent the change authors involved with each event through their profile picture. Not only does this make it easy to see who collaborated with who within a certain timeframe when looking at a specific row in the matrix, it also allows for practical stacking of events: Since the rename event has two change authors listed, it can be assumed that the function was, in reality, renamed twice by two different people on the same day. Furthermore, the change events along the timeline of the same symbol are connected via a colored stripe, the color of which is here intended to serve as yet another indication of the change authors involved, here assigned to colors red and yellow, with gradients forming as a result of immediate change ownership and stripes also being able to have two colors at once should two developers collaborate as they did here at the rename event on November 30.

The connecting stripes also serve the additional purpose of acting as a more immediately visible representation of change events that involve more than one symbol: The forking of the stripe here, ending in a rename and a creation event indicate that part of the function `generate(int)` was extracted into its own function named `computeId()`; since the end in the row of the first function results in a rename event as the next change, it is safe to say that—apart from the extraction refactoring—`generate(int)` remained unchanged in all but name. The row headers for the symbols on the left further show the time of the last update and the total change ownership percentage as seen through the lens of symbol-level diffing in the form of chips that once again show the profile pictures of the developers involved.

While this is possibly the most straightforward visualization approach among the three, this simplicity significantly benefits this concept in both readability and versatility. After all, matrix-based visualizations innately exhibit similarities to tables and spreadsheets, with users likely requiring little effort in adjusting their mental models to understand the data and relationships therein displayed here. On the other hand, this type of visualization is extremely adaptable for any number of needs: Changing any properties of the data axes—symbols and dates in Figure 7.3—such as filtering, ordering, grouping, coloring and changes in granularity alone opens up a magnitude of data mining use cases revolving around searching within or analyzing software history. This directly addresses many of the interface and interaction techniques outlined by Keim [53]. Moreover, the axes themselves may be switched out to address other aspects, such as which symbols were changed by which change authors. Suffice to say, the other key strength of this approach lies in its capability of taking advantage of high-granularity change data and refactoring detection: This leads not only to more precise change information to display but also to less noise due to insignificant changes and more human-friendly change representation when it comes to refactorings.

7.2 Prototype Mockups

After the first round of the requirements analysis phase, which involved a literature review (seen in Section 6.1), several observations made it clear that only one of the aforementioned concepts was flexible enough to accommodate the use cases extracted from the relevant papers: the matrix-based visualization. Many of the questions from Section 6.1.1 refer to specific change *events* made *to* code elements *by* some person, and far from the majority of elicited information needs rely solely on metric-based changes such as code ownership, which would have otherwise been more of an indication in favor of the flow-based visualization concept. Moreover, the diversity of the change subjects involved (code symbols, people, changes themselves, . . .) and the more pronounced focus on the time axis of events would have made it difficult to adapt either of the other possible options to display the required information in a succinct and visually intuitive manner.

As a result, a web prototype using the Vue.js web app front-end framework based on the matrix visualization concept was created to allow for more concrete mockups for use during the interview phase of the requirements analysis as well as getting a better grasp of the visual practicality and challenges of this approach. It should be noted that despite using a web app framework, Vue.js was mainly used to dynamically update the data displayed in these mockups—since adjusting all the relevant HTML elements for the timelines in particular would have been quite tedious—rather than serving as a means for creating a true first fully interactive prototype. The symbol and event icon set used in all these mockups is from the Atom Material Design Icons plugin by Boukhobza et al.[1], which is a popular open-source icon set and plugin for the JetBrains suite of IDEs; the PSI and UI icons in particular are mainly used throughout these mockups for symbol and change event types, respectively. In terms of UI components, the mockups are based on the Bulma CSS framework[2]. The general layout and main components—such as the row headers, column headers, timelines, chips, and so on—are handcrafted using CSS and the well-known flexbox layout system. These mockups also use Linus Torvalds's name and profile picture from his public GitHub profile[3] for demonstrative purposes.

As for visual cues taken from the literature review phase of the requirements analysis, these design concepts offer several options to sort, filter and zoom changes, although this is not immediately apparent from the mockups themselves. Besides the obvious sort and filter options for rows, the timeline button is intended to act as a trigger for the settings menu for column resolution (commits / days / weeks / months), timeline change event filters as well as coloring options for event spots and timeline stripes. Moreover, the pronounced use of colors can be observed in multiple areas: On the timeline itself, creation events are colored green, deletion events red, syntax-only changes (e.g., a renaming) blue and changes with semantic implications (e.g., signature or type changes) amber, alluding to the traditional traffic light color trifecta of green-yellow-red, which is used as a visual metaphor for the creation-modification-deletion lifecycle of code symbols. Furthermore, a red-to-blue color gradient is used for the code ownership chips on the left, with blue representing little involvement or activity and red representing strong coupling or collaboration, which is based on the classic hot-to-cold visual metaphor. Even on a grayscale level, the increased brightness of secondary text such as function signatures and field types is intended to provide additional information without significantly detracting from the actual symbol name while also remaining dark enough to remain reasonably legible. As noted by North et al. [20], developers' profile pictures are sometimes insufficient to identify the respective user; however, since people's names can get quite long and multiple developers may contribute to the same change event, this design tries to find a compromise by including the names in popovers when hovering over profile pictures instead of showing all names outright. Additional measures, such as color-coding change authors using a border around the profile pictures, are also being taken into consideration for the final design, pending feedback from the interview phase of the requirements

[1]https://github.com/AtomMaterialUI/a-file-icon-idea, Accessed: Nov. 3, 2024
[2]https://github.com/jgthms/bulma, Accessed: Nov. 4, 2024
[3]https://github.com/torvalds, Accessed: Jul. 11, 2024

analysis. This section will now proceed with a more detailed look at the different views that resulted from the questions elicited during the literature review.

7.2.1 Exploration-based Visualization

First is the exploration-based view shown in Figure 7.4, which is geared toward browsing- and hierarchy-based use cases. It allows the user to navigate through code elements along their code hierarchy and view the change history of all symbols with the same parent, similarly to how one might navigate through packages or folders in a tree view, as found in nearly all popular modern graphical IDEs. This familiarity in navigational context aids in discovering known symbols quickly and allows one to relate changes from related symbols—in this case, symbols with the same parent element—without the need for further input, although the ability to further constrain which elements should be shown through means of sorting and filtering remains. This approach is also notable for its capability for summarization, seeing as the history of container elements additionally serves the purpose of guiding the user toward interesting results while navigating.

Representing the original vision behind the matrix-based visualization concept, this perspective alone already allows one to answer many of the questions posed in Table 6.1 from Section 6.1: the row headers show code ownership percentages of individual symbols (Q01), the last event on a symbol's timeline denotes the last known change (Q02), each symbol's history is visible through its timeline (Q03), the creation date can be read from the start of a symbol's timeline (Q04), which changes occurred to sibling symbols in a certain timeframe is immediately apparent (Q05), unchanged symbols feature no change events after creation (Q10), change events are always accompanied by change authors (Q12), additional change information may be found within change event details or sibling symbols' histories (Q13), the dates of modifications are always visible in the column headers (Q17) and code complexity may be extrapolated from the number of change events and change authors (Q20). Many of these information needs are also covered by other perspectives, but the ease of navigating along the codebase's natural structure makes the process of finding the symbols one wants answers for simpler with this kind of visualization in many cases.

7.2.2 Search-based Visualization

Second is the search-based view shown in Figure 7.5, which is aimed toward finding specific code symbols more easily, such as cases where only the name of a symbol is known and where deep directory or package structures would make navigating the code hierarchy tedious. It can therefore be seen as a natural counterpart to the exploration-based visualization that focuses less on the tree-like sibling and parent-child relationships and more on the *direct* navigation to a specific point of interest. To aid in future discovery and provide context for the exploration-based view, the header lines for each code symbol additionally include their location in the code hierarchy. The occurrences of the search term within the search results also gets highlighted with an underline text decoration.

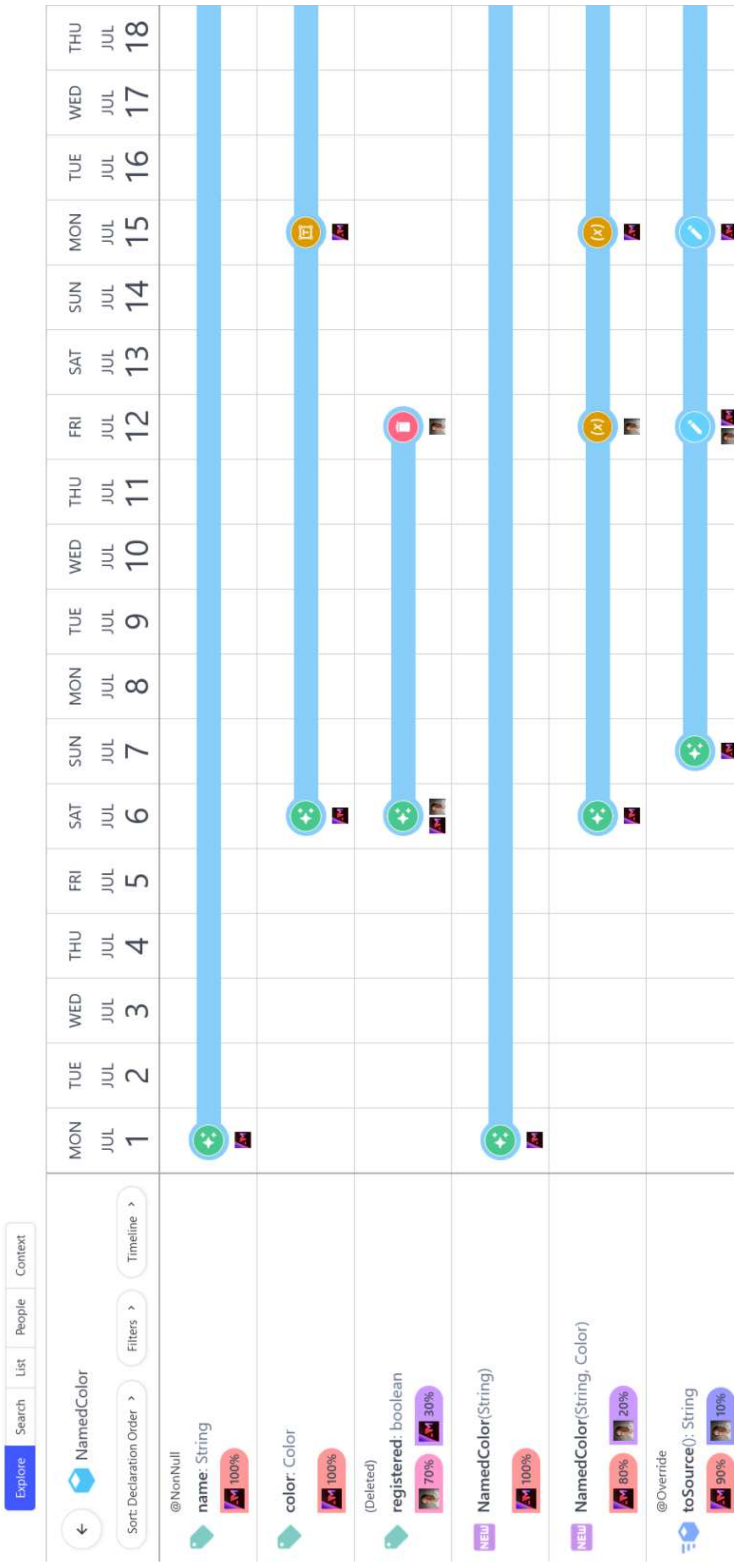


Figure 7.4: Mockup of a visualization that lets users browse through the code hierarchy

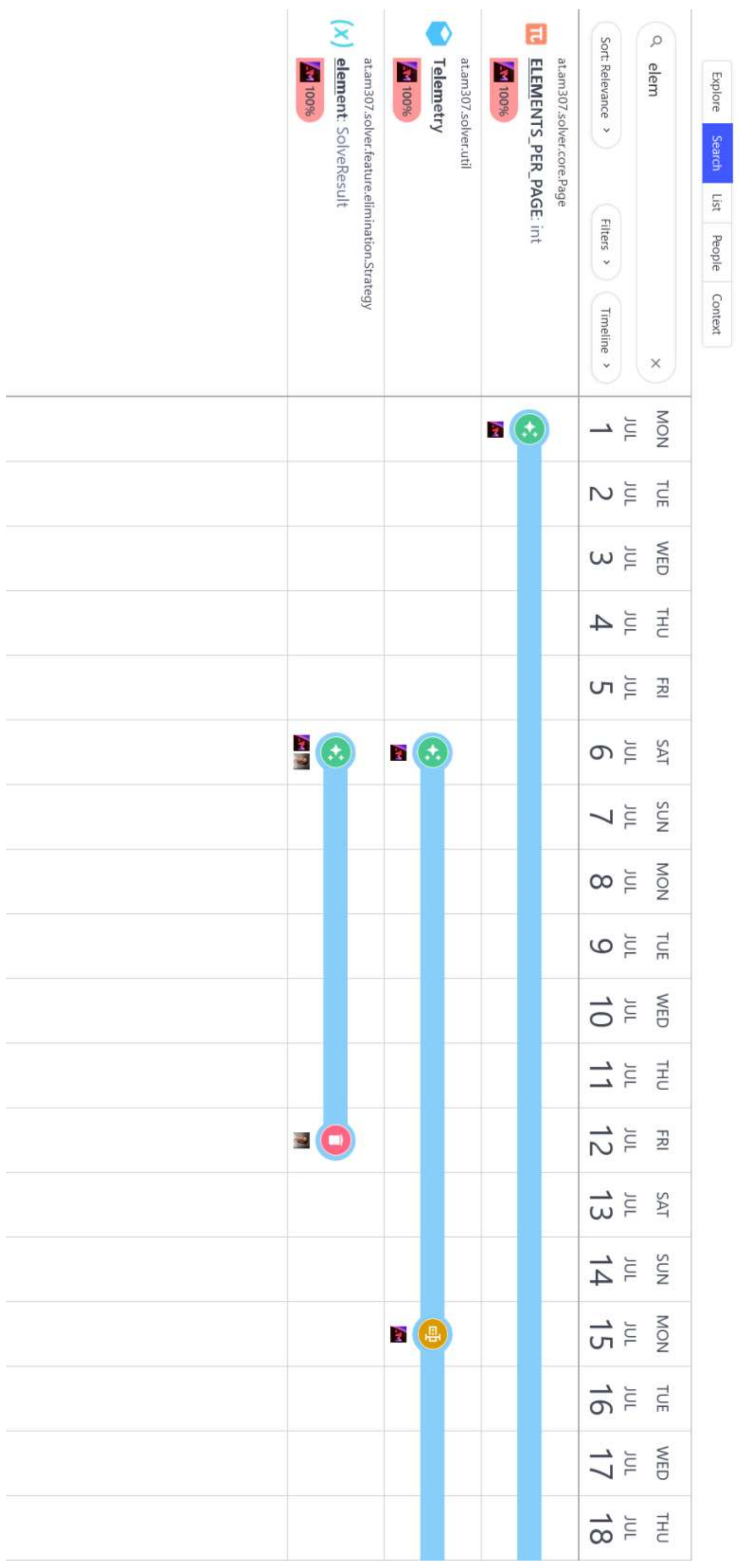


Figure 7.5: Mockup of a visualization that enables search for specific symbols and changes

Due to their similarities, this visualization can be used to answer mostly the same questions as the previous; however, the sorting and filtering options do take on a more important role as means to speeding up discovery since the pool of possible search results naturally encompasses all symbols from across the entire codebase by default.

7.2.3 Ranking-based Visualization

Third is the ranking-based view shown in Figure 7.6, which serves as a global ranking across all code symbols, not dissimilar to the concept of the search-based view but without a search term to filter elements by. It follows that this visualization is most effective when—as the name suggests—the user is interested in the top- or bottom-ranked elements when sorted by some criterion, optionally with an applied filter if only interested in specific types of symbols.

As it turns out, several of the questions identified in Section 6.1 directly address use cases that depend on this kind of global ordering capability. Notably, questions regarding recent changes (Q06), including those concerning subsets thereof such as recent changes made to symbols of interest (Q08), might be answered using such a visualization. Inversely, stale elements (Q11)—symbols which have not seen any recent activity—can be found by the same token simply by reversing the sort order. Lastly and perhaps most obviously, rankings using metrics other than recency are also possible, aiding in endeavors such as finding the most frequently modified (Q18) code elements or those touched by the largest number of people (Q19).

7.2.4 Person-based Visualization

Fourth is the person-based view shown in Figure 7.7, which exchanges the rows previously used to represent code symbols for a list of change authors. Intuitively, this also changes the timelines in each row to denote developers joining and modifying the codebase. To avoid redundant visual clutter, the small change author lists beneath each change event exclude the person the row belongs to. Furthermore, the chips under the name of each row header now illustrate the amount of collaboration between pairs of people; more specifically, the relative amount of change events where another person changed the same symbol within a close timeframe, which is why the percentages listed are not identical despite involving the same people.

Unsurprisingly, this visualization aims to answer questions from Section 6.1 related to people and the changes authored by them, such as what everyone on the team is currently working on (Q09). It can also act as a quick refresher on what the user themselves was last working on (Q07) and who has been working in the same code areas as oneself (Q08). Other metrics, such as the aforementioned collaboration statistics, rough extent of activity in certain timeframes, how many people someone is actively collaborating with as well as commit behavior can also be gleaned quite easily and make identifying potential problems such as developer deadlocks—e.g., two people being blocked by the progress of each other—more straightforward.

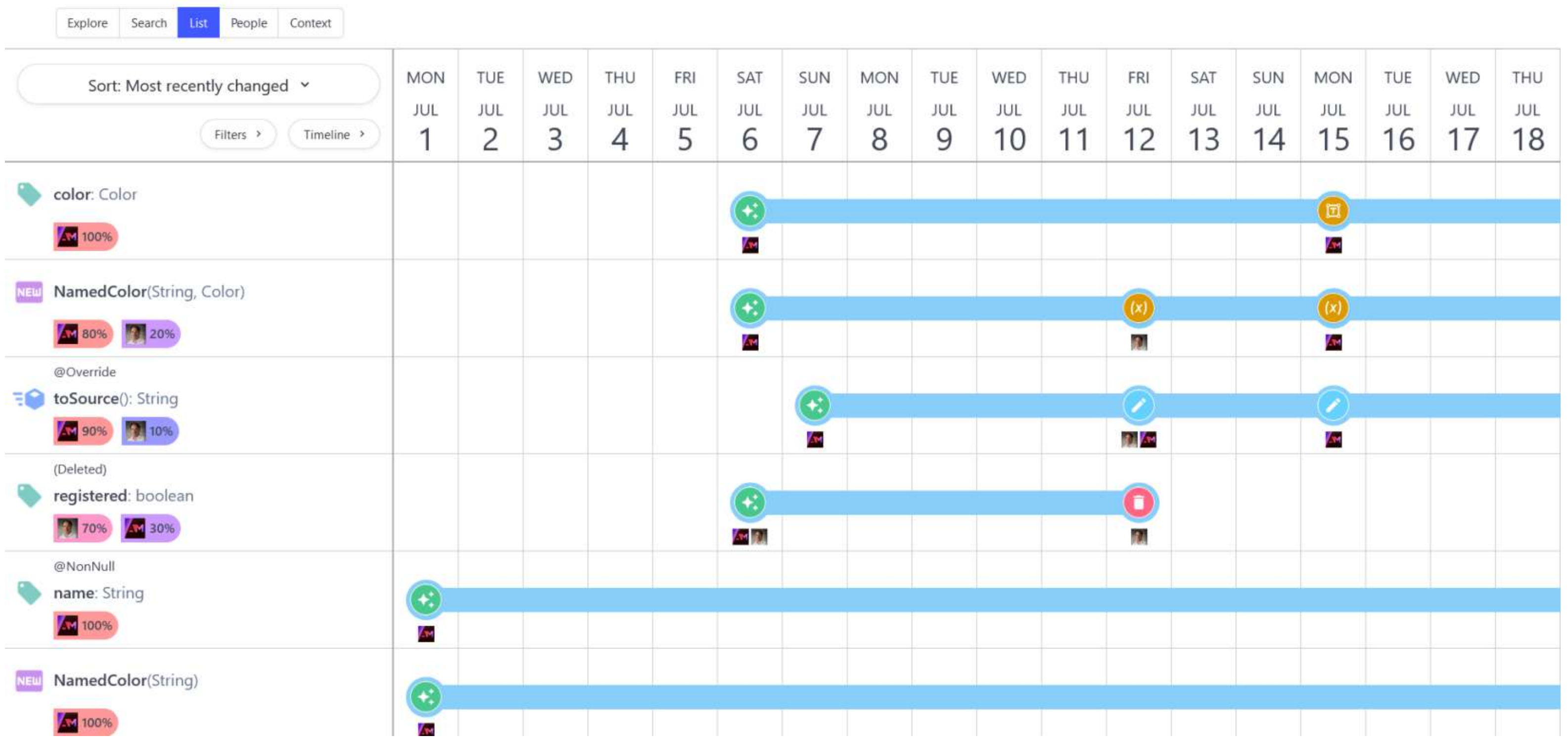


Figure 7.6: Mockup of a visualization that displays a ranking list of code elements

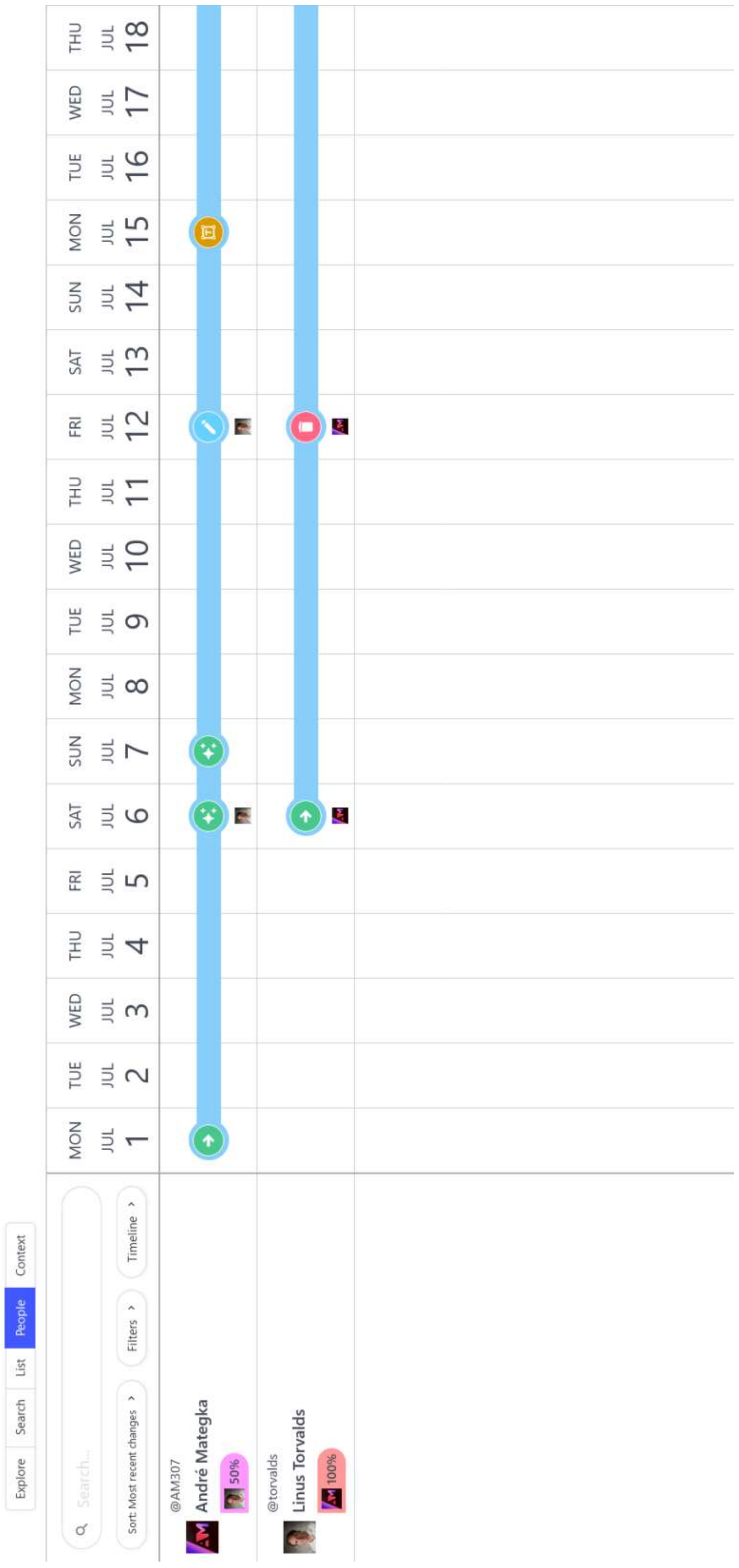


Figure 7.7: Mockup of a visualization that summarizes changes per person

7.2.5 Context-based Visualization

The fifth and final visualization mockup is the context-based view shown in Figure 7.8, which centers around exploring the implications of changes beyond just the individual change itself. Indeed, this perspective's rows are based on code symbols related to a specific change, with the respective subject-of-change symbol being the first row and the column of the observed change being highlighted in the timeline view. In the example depicted in the figure, a deletion change event for a field called `registered` is being outlined. Besides the history of the field itself, affected components both inside and outside the parent code element with related changes are also listed, although external changes only get counted if the respective change modified the public interface of the parent element or the subject of change itself. The container—in this case, a class—is also listed among the downstream changes since modifications to elements other than those directly affected by the main change may also be of interest when trying to decipher code rationale.

The reasoning for this perspective relates to questions from Section 6.1 which called for more detailed exploration of type relationships and other dependencies (Q15) as well as modules affected by certain changes (Q16). Hoping to provide additional context beyond the raw VCS diff and concentrating on the most affected elements (such as those relying on affected interfaces), its main aim is to shed new light on both code rationale and coupling between elements within the codebase.

7.3 Software Architecture

To implement the general SEV approach as outlined in Section 4.4, several factors need to be taken into consideration. The overall software architecture was originally conceived during the literature review phase of the requirements analysis, as it became clear that the components and integrations would have to meet very similar requirements regardless of which visualization views or finer details would be deemed necessary after the interview phase. Ultimately, the structure still ended up being dramatically changed after the interview phase due to circumstances outlined later in this section.

7.3.1 First Draft

For the initial version of the software architecture, the application is split into five components: a visualization frontend, a web backend, an analysis framework, a refactoring detection server, and a graph database to persist output information. All of these components communicate via different kinds of protocols and data exchange formats, or are directly integrated into each other, as is the case with the web backend and analysis framework. To coordinate the efficient deployment of these components, the containerization software Docker is used alongside the Docker Compose service and definition format.

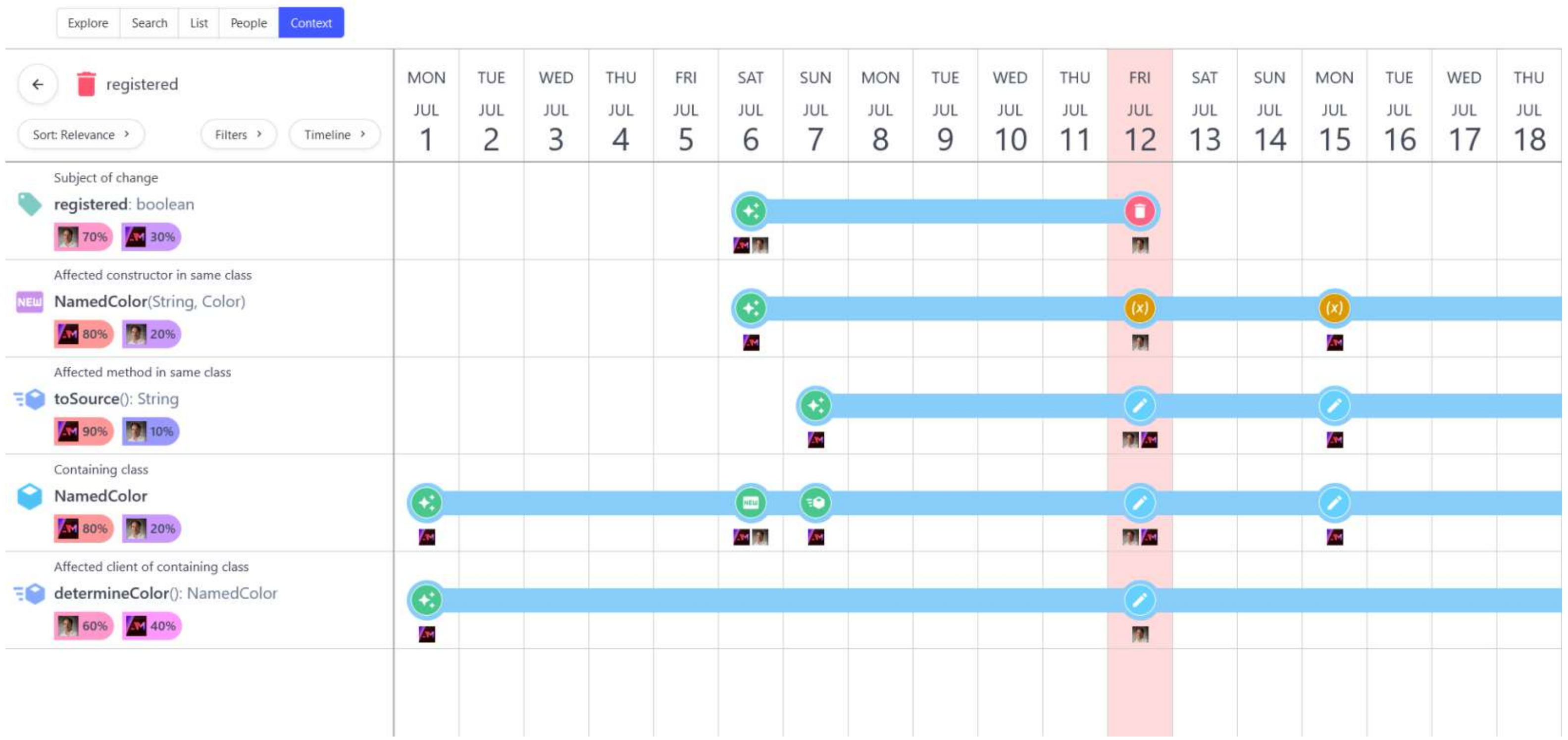


Figure 7.8: Mockup of a visualization that shows related elements for a given change

To start with, the web frontend serves as the host for the visualization application. Largely in part due to the fact that Vue.js was used to build the visualization prototypes, it is also featured here as the main frontend framework alongside the UI component library Bulma[4], which was likewise initially used for prototyping, because of the convenience of code reuse as well as general usability. For efficient bundling and development server hosting, the frontend uses Vite as the preferred option over Webpack due to performance benefits useful for faster iteration and great integration with Vue.js since it was created by the same developer. Modern type-checking and stylesheet capabilities are enabled by TypeScript and Sass, respectively. Additionally, dotenv is used for seamless operation both locally and inside a Docker container.

The web frontend communicates with the web backend via a traditional REST-like API served over HTTP. There were originally considerations to include WebSocket capabilities for a live display of indexing progress, but this idea was later discarded in favor of a simpler interface between the two components. Otherwise, the web backend uses the popular Java-based Spring Boot to manage its HTTP server. Since the analysis framework uses Java to utilize the AST differencing packages made available only for Java and other JVM-based languages, this enables the analysis framework to be directly included as a Maven dependency in the web backend, essentially separating the two components only for reasons pertaining to cleaner abstraction and separation of concerns. This leaves the Spring backend acting primarily as an intermediary between the visualization frontend, the analysis framework and the database. Because of the hierarchical nature of code symbols, a graph database was chosen for data persistence; in this case, Neo4J, mainly due to being the core graph database supported by the Spring team via its Spring Data Neo4J package.

The analysis framework is the package containing the business logic for handling the core symbol change information data mining and aggregation processes. It accomplishes these tasks through the use of GumTree-Spoon [76], which can parse Java files and determine AST node mappings to be used for change detection. Notably, the analysis framework is only used for the analysis of new VCS changes, as it defers any data persistence responsibilities to the web backend component, which stores symbol-level change information in the graph database.

For inter-file refactoring detection, the tool RefactoringMiner [77] is used. However, due to conflicts with Maven dependencies between GumTree-Spoon and RefactoringMiner, the latter is run in a separate JVM instance as a standalone application to avoid difficulties pertaining to dependency resolution and the usage of multiple class loaders. This refactoring detection server uses the open-source RPC library gRPC to communicate with the analysis framework component of the other Java application. An RPC solution in particular was chosen because of the increased communication efficiency afforded by binary encoding using Protobuf and streamlined encoding/decoding functionality.

[4]https://github.com/jgthms/bulma, Accessed: Nov. 4, 2024

7.3.2 Second Draft

Due to complications that arose during the implementation stage (see Chapter 8), leading to a reduction in scope, the final software architecture design ended up being a more simplified version of the initial draft. Specifically, the design now consists of two components: the web frontend and the analysis framework. Data persistence has been simplified to file-based storage to simplify serialization such that only one object mapping model is required. Structurally, the role of the web frontend remains unchanged compared to the first draft, except that data is now retrieved from a static JSON file instead of being fetched from the web backend via HTTP. This JSON file represents the result of an indexing process from the analysis framework, which now acts as a standalone Java application. Further details on the final serialization scheme can be found in Section 8.1.3. The analysis framework itself expanded in size to the point that it now consists of subcomponents of its own, handling commit diffing, symbol tree extraction, symbol differencing and serialization, respectively. The concrete interactions between these subcomponents are outlined in Section 8.1.

7.4 Final Visualization Design

Based on the feedback from the interview stage of the requirements analysis described in Section 6.2.2, several improvements have been made to the visualization design. The visualization, which can be seen in Figure 7.9, has been based on the original prototype of the search-based view from Section 7.2.2, chosen because of its very positive reception—in fact, it was deemed the most important view out of all the prototypes.

Looking at the general navigation, the top bar features buttons for navigation through the time axis, with “Previous days” and “Next days” loading in the data for the closest past and future timeframe, respectively, always shifting the view by the maximum number of displayable days (18 in this case). The “Today” button was added due to the perceived importance of recent commits, making it easier to navigate back to the present day after inspecting change events further in the past. On the left, sort and filtering options have been disabled due to scope reductions (see Chapter 8). As part of the search result row headers, a description of the kind of symbol (package, class, method, variable, . . .) has been added before the symbol path to address the feedback about unclear or confusing symbol icons. Additionally, each row header now features a button with the creation date of the corresponding symbol, which navigates to the shown date on the time axis when clicked. To distinguish deleted symbols from those still existing, the name of deleted symbols is displayed in a reddish hue instead of plain black, and the highlights showing the positions where the current search query matches the symbol name use wavy underlines instead of straight ones. The buttons at the bottom of the search results pane allow navigating between pages of search results.

On the main timeline view, change event stacks now display the types of symbol property changes on that day with separate pills, sorted from most to least significant, e.g., changes affecting public interfaces such as type (“TYP”) or visibility (“VIS”) changes will be

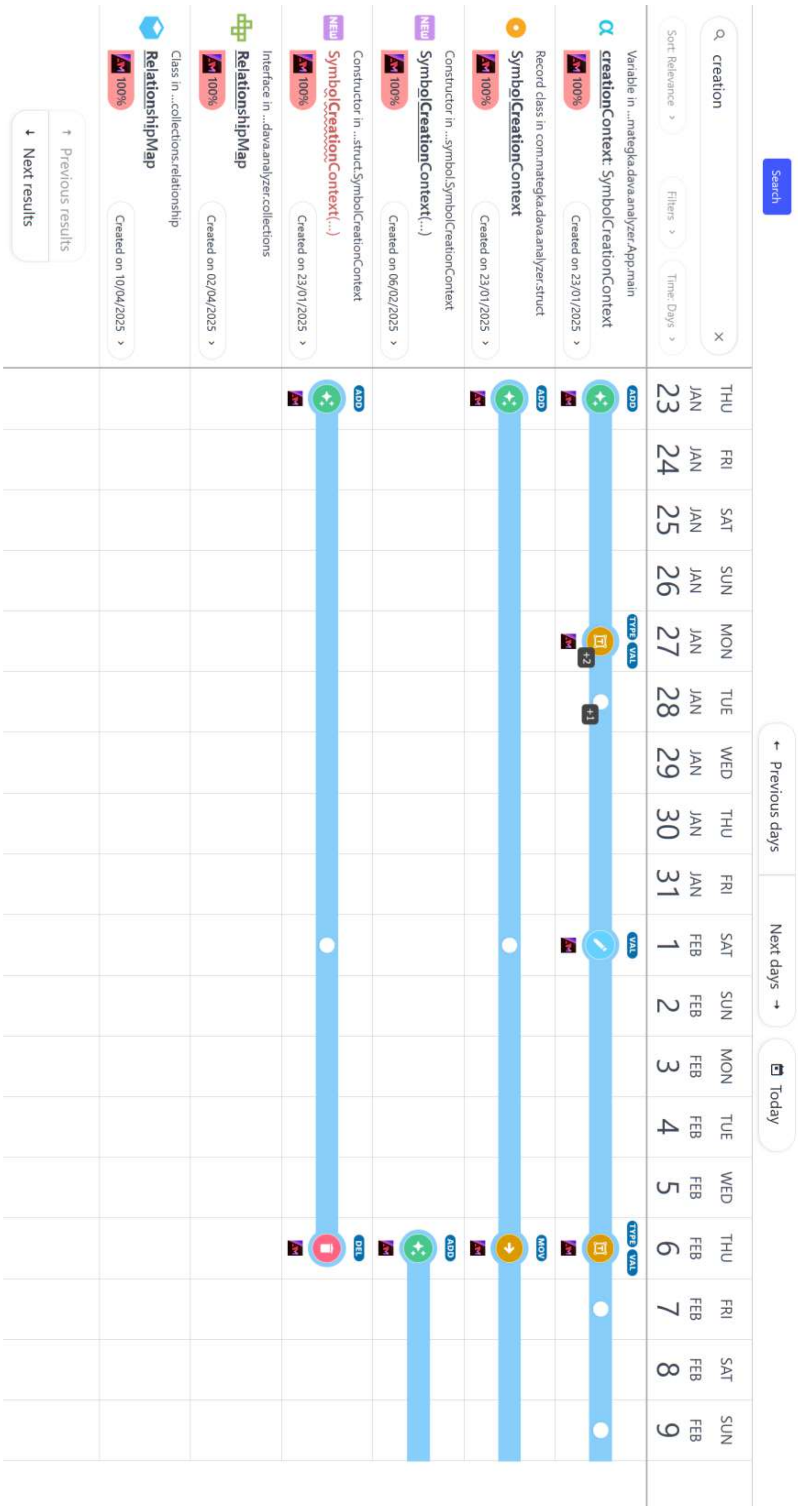
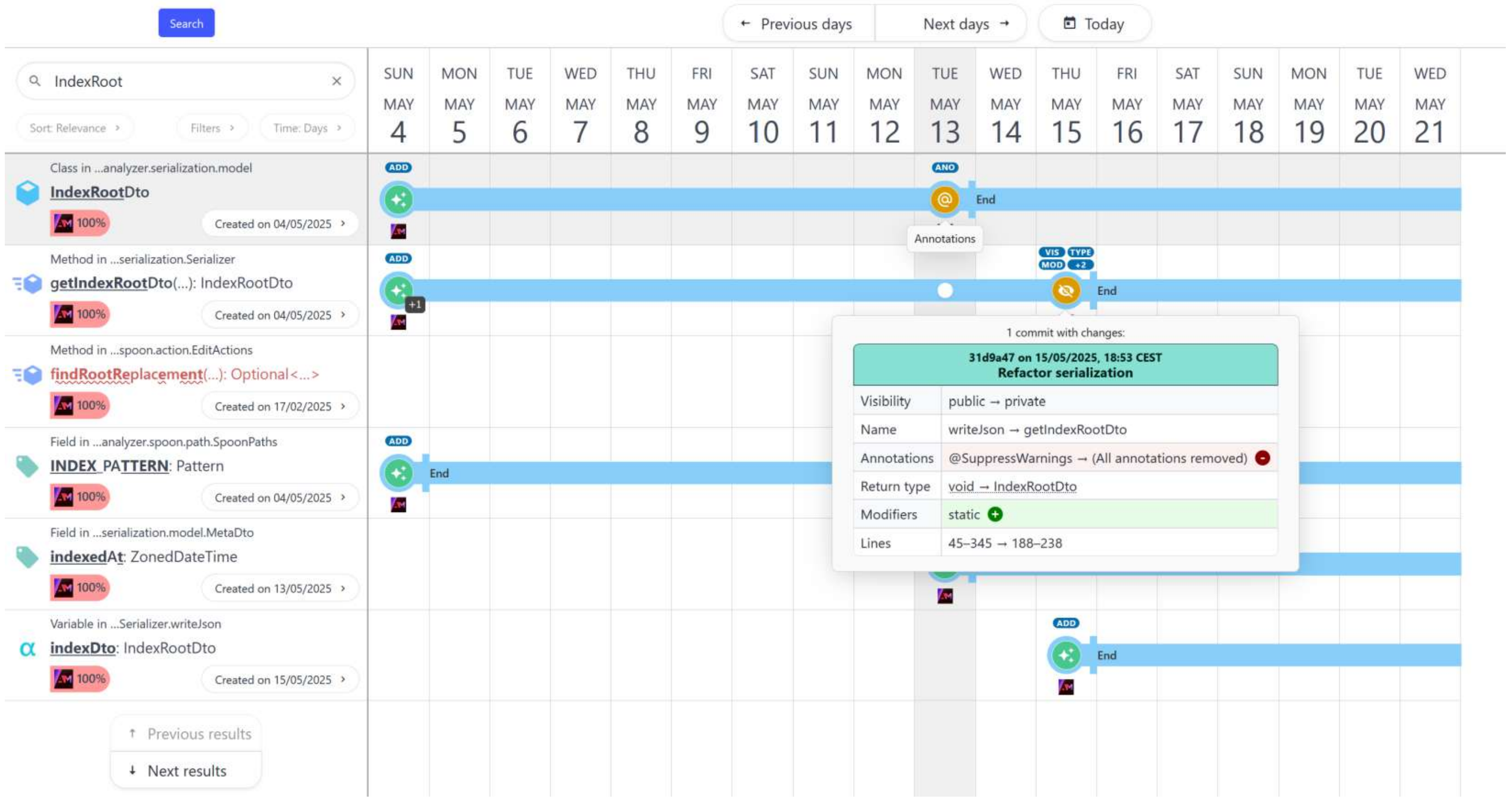


Figure 7.9: The final visualization design (idle state)

displayed before more contained changes such as updates in initial value ("VAL"). If the number of pills exceeds four, only the first three will be displayed, and excess pills will be represented by a "+X" pill (X being the number of additional pills). These pills are additionally meant to aid in the visual parsing of the change event icons since the pill listed first will always correspond to the displayed icon. Not all changes are represented by pills, however; minor but still relevant changes such as line number updates—which allow the user to locate the symbol declaration in its file—are not highlighted in any special way. An exception to this are days with exclusively minor changes, which are displayed as stacks with a smaller white dot without an icon, illustrated by, for example, the last two visible change event stacks for the "creationContext" variable symbol. The boxes with the "+2" and "+1" texts that can be seen to the lower-right of the change event stacks are used to indicate the number of additional change events for that stack, i.e., the number of additional commits with relevant symbol property changes.

Figure 7.10 shows how the visualization displays additional information when interacting with changes on the timeline view. Hovering over any cell of the timeline view highlights the respective row and column in a slight shade of gray to make comparisons with other search results or stacks on the same day slightly easier. Hovering over a change event stack displays a textual representation—e.g., "Annotation"—of the main type of change, similarly to the firstmost pill. Clicking on any change event stacks opens a popover box listing all relevant commits and their changes in a diff-like view. Initially shown commit information includes the commit's shorthand SHA hash, date, time, timezone and summary message. If the commit message includes a description part in addition to the summary part, the summary text will be highlighted, and hovering over it displays the description text. Each commit includes a table of symbol property changes, with the first column showing the property names and the second column showing the change in property value. For single-value properties such as visibility, the diff consists of the before and after values separated by a right-facing arrow. For multi-value properties such as annotations or modifiers, added and removed values will be displayed alongside a circled green plus and red minus, respectively; removing the last value for any such property will result in an additional notice text such as "(All annotations removed)", "(No longer initialized)" or "(No longer generic)". More specific information about the displayed symbol values is once again available upon hovering over any more complex value; for example, the hover text for types will reveal their fully qualified name, and the hover text for longer initialization values will show the full initialization expression. Symbol creation and deletion events always include *all* symbol properties (and their values).

Finally, Figure 7.10 also shows the newly added end markers, visible—in this example—after the last change event stack for each displayed symbol. These end markers indicate the end of the change history for the corresponding symbol, so no new change events will appear on the symbol timeline beyond this point. This not only makes it clearer which change events are the last in a symbol's current history but also highlights symbols that have never changed by placing the end marker directly after their creation event, as is the case with the "indexDto" variable symbol shown in the example.



1 commit with changes:

31d9a47 on 15/05/2025, 18:53 CEST
Refactor serialization

Visibility	public → private
Name	writeJson → getIndexRootDto
Annotations	@SuppressWarnings → (All annotations removed)
Return type	void → IndexRootDto
Modifiers	static
Lines	45–345 → 188–238

Figure 7.10: The final visualization design (hover and active states)

CHAPTER 8

Implementation

This chapter covers the details of the concrete implementation of the final visualization application as described in Section 7.4. The implementation is split into two large components: the analysis framework used to extract the symbol version history information and the interactive visualization frontend to filter and display this information.

8.1 Version Control History Mining

The analysis framework itself consists of multiple subcomponents that largely interact with each other along a pipeline-like structure where each subtask uses the output of all prior subtasks within the same task—in this case, the same commit—as input, although raw VCS information is commonly replaced with more concise structured objects before being passed on. Several subtasks themselves consist of a kind of pipeline where work is subdivided even further in much the same way.

8.1.1 Terms and Setup

Before continuing on to the summary of the main symbol history extraction process itself, this section covers the general setup and important terminology and semantics used throughout the next section. To begin with, the main application starts with taking a path within a Git repository as well as a Git revision expression (e.g., a branch name, tag name, commit hash or `HEAD`) as input. It then constructs the skeleton of a final output `History` object with `Strand` objects that will be mutated inside the main loop.

Strands

Strands are the fundamental building block used for symbol analysis to know how to store, override and discard currently tracked workspace state information. At their core, strands are parent–child ancestry chains of commits such that only the first commit in a strand can have more than one parent and only the last commit in a strand can have more than one child. This essentially means that a strand must always start with a merge or initial commit, end with a pre-merge or branch head commit, and otherwise contain only linearly dependent commits. By this definition, all Git histories can be uniquely and deterministically subdivided into strands. The important property of these commit chains that is exploited here is the fact that—save for commits added later that might necessitate reindexing—no commit (or strand) can depend on the workspace state of any commit other than the final commits of strands, allowing the intermediate workspace states of any non-final strand commits to be overwritten, greatly reducing the memory footprint of the analysis in the process.

Within a `History`, each `Strand` is assigned a unique ID number, which will be simply referred to as a "strand ID" from here on. Additionally captured in the `History` object are the parent–child relationships between strands (as defined by the strands corresponding to the parent of the start commit and children of the end commit) in the form of a DAG as well as the mapping from each commit *to* their associated strand. For simplicity's sake, `Strand` objects are also where the `CommitDiff` objects containing the actual symbol-level change information will be stored.

Bare and Embedded Symbols

To understand the structure of this information, the traits of the `Symbol` class and its objects must first be explained. This class is used to capture two different but related concepts, here called *bare* and *embedded* symbols. Bare symbols represent the state of a symbol as a standalone entity, without any ties to a commit, other symbols within the same AST or related symbols across commits. They therefore capture exclusively local information about the symbol in question, such as name, visibility or initialization expression for field symbols. The way this information is stored as `Property` records is later described in Section 8.1.2.

Embedded symbols, on the other hand, represent a symbol as it exists in a commit workspace and historical context, and therefore contain additional information such as their parent symbol (e.g., the class symbol for a method symbol), their originating commit's hash and a `SymbolKey` that uniquely identifies this symbol across the entire history using a combination of a strand-unique symbol ID together with the previously mentioned strand ID. Note that the complete symbol hierarchy information is not stored within individual embedded symbols (for example, they do not contain any information about *child* symbols), but rather, externally in a symbol tree as part of a symbol workspace, explained later in this section. Additionally, embedded symbols contain information about relationships between symbols, more specifically predecessor symbols; for example, symbols that are "merged" after processing a merge commit often result in the after-state symbol containing links to (the keys of) all symbols from the before state with a `PrdRole` (short for "predecessor role") of `MERGED`. Ultimately, symbols themselves are always diffed using their "bare" properties but stored for use in future commit diffs with their complete embedding information.

Symbol-Level Commit Diffs

The aforementioned change information stored in a `CommitDiff` can be divided into four distinct types: additions, successions, deletions and updates. The addition set simply contains a copy of embedded symbols newly introduced in the commit in their initial then-current state. The deletion set similarly contains the embedded symbols deleted in the commit. However, in contrast to added symbols, deleted symbols simply have their before state “moved” instead of copied since their state will not be mutated afterward; even if another symbol with the same name, properties and relationships later reoccurs, it will be given a new symbol ID and therefore have a different `SymbolKey`.

The update set contains `SymbolUpdate` instances for all actual symbol changes, i.e., changes where the symbol exists both before and after the change. Since a commit can have multiple parents, the same symbol can receive multiple updates in each such commit, up to one for each parent. Additionally included are the `SymbolKey` and commit hash information for both the source symbol (the “before state”) and the target symbol (the “after state”). The actual change information about modifications made to symbol properties is stored as a hash map indexed by property keys. For more details about the structure of properties and their associated data structures, see Section 8.1.2. Updates also come with a set of so-called update flags, which are used to signal modifications that sometimes do not affect the properties of a symbol, which is primarily used for the disambiguation of the relationship between symbol path changes and symbol moves, e.g., a symbol can change its path without having moved relative to its parent and—less commonly—be moved even without changes to its symbol path (such as when the parent symbol gets replaced by a different symbol with the same name or Spoon path fragment).

Successions

Succession indicators are, as the name implies, mainly used for tracking the predecessor–successor relationships discussed earlier. More specifically, the succession set of a `CommitDiff` instance tracks the so-called *pure* successions that occur when two symbols are in a predecessor–successor relationship without any actual changes having taken place. The reason for this inclusion has to do with the concept of the strand-based locality of symbols: To ensure that the primary history analysis only requires a single pass (i.e., no commit is processed more than once), symbols are “copied” when transitioning from one strand to the next whenever no changes take place. Symbols are then given the strand ID of the child strand, ensuring that an unmodified symbol “exists” on all strands it is a part of, just with a different strand ID. This additionally simplifies the tracking of time-oriented symbol relationships by allowing symbols to pass through multiple strands with the same symbol ID (the one initially given to the closest common ancestor) as long as they are not deleted or merged with another symbol not originally belonging to the same symbol (therefore having a different symbol ID) at a merge commit. In this sense, successions, pure or impure, are ultimately a way to keep track of which `Symbol` objects belong to the same *abstract* symbol, i.e., the symbol as a unique semantic entity as defined by change events.

Symbol Workspaces

The final concept worth explaining here, before delving into the main change information extraction process, is the `SymbolWorkspace`. A symbol workspace represents a complete symbol hierarchy of the Git worktree at a specific commit. As already mentioned, it contains the tree data structure consisting of nodes with `Symbol` values that allows navigating from a child to a parent symbol, as well as navigating between freshly-parsed bare symbols which have not had their `ParentProperty` set yet. Additionally, each symbol workspace acts as a "cache" of sorts for parsed mappings and Spoon AST instances, which are reused when the workspace of a parent commit is compared to that of one of its child commits. To be more specific, it stores the `CtCompilationUnit` instances returned by Spoon, indexed by the relative path (with the project root path as the base) of the corresponding file. The tree nodes of main file declaration `Symbol` instances (e.g., a top-level class) corresponding to each file are indexed using a similar mapping. Since diffs output by GumTree-Spoon result in AST differences based on Spoon's `CtElement` objects, each workspace also contains a mapping from string representations of Spoon's `CtPath` objects to the tree node corresponding to the symbol located at this Spoon path, thereby connecting nodes from a Spoon AST to symbols in the symbol tree.

8.1.2 Symbol History Extraction

Similarly to the way strands are first detected when constructing the initial skeleton for the `History` object, the extraction process mainly consists of a single loop over the repository commits in reverse topological order, guaranteeing that parent commits will always occur before any of their child commits. The general structure of each iteration is as follows:

1. Find all relevant Git worktree files
2. Retrieve the parent commits' strands' symbol workspaces
3. Create a file-to-file mapping from each parent commit to the current commit
4. Construct the target symbol workspace for the current commit
5. Produce a symbol-level diff and store it
6. Replace the symbol workspace for the current strand with the target symbol workspace
7. Delete unneeded symbol workspaces of completed strands

This workflow is designed to work for initial commits (with no parents), regular commits (with one parent) and merge commits (with multiple parents) alike.

General Iteration Procedure

The first step is very simple, as it just involves identifying which files from the worktree of the current commit are relevant for the following diffing process. For this thesis, this means that only files with the extension "`.java`" will be scanned for changes. Likewise, retrieving the parent workspaces is done via a lookup in a `HashMap`. No special handling is required here as—per the iteration order—any commit currently processed is guaranteed to have already had its parent commits processed in previous iterations. For initial commits, the list of parent workspaces is left empty.

In the next step, JGit's `DiffFormatter` is used to determine changes between each parent worktree and the current worktree. The result is a many-to-many file mapping between $\langle \text{parent index}, \text{source path} \rangle$ tuples and target path values, with the JGit difference information stored as relationship data. Git additions are marked by their source path being `null` and deletions by their target path being `null`. This mapping is then augmented with mappings for unchanged files, which, by construction, are exactly all files left unmapped from the current worktree for each individual parent. The consequently created edges carry no additional Git diff data, being left `null`. Besides these "unchanged-type mappings", the file mapping additionally serves to extract *strict* additions and deletions. Strict additions are all files added that do not exist in any parent worktree; in a way, this makes these files "true" additions since there is no before state that could be used for copying or diffing. As a sidenote, this implies that additions for initial and regular commits are always strict (since there is at most one parent). This distinction is only relevant for additions, however, since deletions always pertain to individual source symbols, which are associated with a specific parent.

The target symbol workspace generation and symbol-level diffing are explained in more detail in the following sections. Relevant, for now, is the fact that the resulting `SymbolMapping` is destructured and then stored as a new `CommitDiff` in the `Strand` object for the current commit afterward. The workspace for the current strand also gets remapped to the newly created target symbol workspace, freeing the previous workspace object to be garbage-collected. This replacement is one of the boons enabled by the strand destructuring of the Git commit history, as only the final commit in a strand can have any dependents outside of the strand it is closing. Finally, once all child strands of a strand have started processing, the workspace of that strand is no longer of any use and can be freed for garbage collection, as well. To that end, each strand is associated with a countdown initialized to its number of dependent strands that gets decremented for each starting commit of a child strand. When the counter for a strand reaches zero, it is removed from the strand-ID-to-workspace `HashMap`.

Target Workspace Construction

The first step towards symbol-level diffing comes in the form of constructing the target symbol workspace containing the symbol tree of the current commit. Generally, this involves using Spoon to parse the source code of all source code files, then using a preorder traversal of the resulting AST to capture the local properties of its symbols and construct the symbol tree. For this purpose, the content of each source code file is loaded into a `VirtualFile`, which is then parsed into a `CtCompilationUnit`. The compilation unit's main declaration serves as the base for the symbol tree located under the respective package symbol in the symbol tree. Package symbols for the symbol tree are not created in bulk beforehand, but rather, entire package symbol hierarchies are dynamically created for each missing package when adding the symbol hierarchies of files to the symbol tree according to the package declaration in the compilation unit. The so-called `Symbolizer` is used to derive the file symbol trees from main declarations. Construction takes place in two phases: The element capture phase and the property capture phase.

In the element capture phase, the AST subtree rooted at the main declaration is efficiently parsed using a form of recursive tree parsing. At each AST node, the procedure first checks if the node in question is a terminal node, i.e., cannot contain any descendants relevant for the construction of the symbol tree and is itself not a valid symbol tree element; recursion stops if this is the case. Otherwise, it checks the node itself to determine whether or not it constitutes a symbol declaration of some kind; if this is the case, the `CtElement` and the current `parent` element are added to the output, and the `parent` element for all child AST node traversal is set to this element. To ensure this process works without friction, the traversal starts with a virtual root element as the initial `parent` value that is later stripped from the completed element tree by returning its first child, which is expected to be the element corresponding to the main type declaration itself, instead. Whether or not any element–parent tuples are added to the output, as long as the current node is not a terminal node, the procedure checks which child nodes are worth descending into—i.e., *could* contain nodes relevant for the element tree—and issues recursive calls for each child node in order. The element–parent pairs resulting from the initial call to the recursive function are then used to build the element tree, which is easily accomplished thanks to the fact that the recursion followed a preorder traversal, i.e., tree nodes can be constructed knowing that parent elements already exist as nodes in the element tree. Importantly, at this stage, the tree still consists of `CtElement` AST nodes that have yet to be turned into proper bare symbols.

In the property capture phase, this element tree is then converted into a symbol tree using one-to-one mapping. For this purpose, each element's local properties are extracted from the element, determined by their `CtElement` subtype as evaluated using dynamic type checking. Which properties are captured here ultimately decides which information will later be available for diffing and displaying. It is worth reiterating that all properties captured at this stage are *local*, i.e., only require the immediate context (the current element and its parent in the element tree) to be defined, such as name or visibility.

All properties are stored in a type-safe way, using its own hierarchy of custom `Property` classes to ensure each property is associated with values suitable for that property; for example, a `VisibilityProperty` must be instantiated using a `Visibility` enum value. Each `Property` subclass is additionally fitted with a `PropertyKey` annotation, which is a unique string identifier used for storing properties in maps, diffing and serialization. Finally, the symbol tree resulting from this phase is attached to the correct package symbol node in the target workspace's symbol tree.

Symbol Properties

The property capture procedure collects a wide array of information, including general element attributes such as name or line range, enum-set-based attributes such as the list of modifiers, list-based attributes such as enum constant arguments, and type-based attributes such as declared type parameters.

Properties that are always set include the flags property, which uses an `IMPLICIT` enum constant to indicate that the symbol does not syntactically exist in the source code, as well as the Spoon path and simple path properties. The Spoon path is the unique identifying AST path of the symbol's corresponding code element as returned by Spoon. As a result, it is used throughout the symbol history extraction process as a universal identifier for symbols whenever they need to be located within the `CtCompilationUnit`. Notably, since Spoon's path creation methods turned out to be quite computationally expensive when applied to an entire symbol tree's worth of code elements, seemingly always traversing the full path from the AST root node and creating objects at each step, the analysis framework uses a memoization mechanism to avoid recomputation of path segments for previously encountered code elements within a commit. In contrast to the Spoon path, the simple path property more accurately captures a human-readable version of the symbol's path. It starts out identical to the Spoon path but replaces named path segments with just the code element's name and discards positional path segments entirely since the absolute index of child elements is usually not essential for understanding their locations within the code hierarchy. In this sense, the simple path of a symbol provides a more intuitive notion of a location within a code hierarchy at the cost of no longer being unique, e.g., in the case of shadowed variables within the same method. Properties that are *almost always* set include the line range and name properties, which are only unset in special cases like implicit elements (e.g., record fields and methods) and the root package, respectively.

Properties that are set where corresponding information is available include annotations on the symbol, visibility, modifiers, (return) type, declared type parameters and type bounds, enum constant arguments, initial value, supertypes (extended classes or interfaces) and realizations (implemented interfaces). While expressions are captured as string code snippets from the argument AST elements, type usages are captured with their entire structure, i.e., generic types, upper type bounds, lower type bounds and the use of generic type parameters are all recursively included in the property value.

Symbol Tree Reuse and Error Handling

In order to avoid having to reparse parts of the symbol tree corresponding to unchanged files, symbol subtrees from parents are reused wherever possible. Generally, this means that for each file symbol subtree, as long as a parent commit exists for which there exists no Git change information, that parent commit's symbol tree can simply be inherited (within a strand) or copied (when switching strands). This is very useful for intra-strand changes or for merge commits, where it is not unusual to find at least one participating branch (usually the target branch) where a large number of files remain unchanged. An exception to this rule is made for the `ROOT` package node, which is always inherited as-is since it contains no useful information and primarily acts as a common parent for all top-level package symbols. Since subtrees copied using this technique can later skip diffing entirely, this ensures that no symbol merging can take place between `ROOT` symbols for single-initial-commit Git histories. An added side effect of this mechanism is some degree of robustness when it comes to (parsing) errors: The last known state of the file symbol subtree will simply be silently propagated until the file becomes parseable again, which will automatically be tested for after each subsequent modification to the file in question. This avoids the problem of symbols being "deleted" when the file becomes unparseable and "reintroduced" once the file is fixed.

Symbol Diffing

After the target workspace has been constructed, all that remains is to diff it with the symbol workspaces of all parent commits. Symbol diffing again takes place in multiple stages: package mapping, file symbol mapping, symbol addition and deletion capture, symbol augmentation and diffing, as well as symbol addition and succession post-processing.

The package mapping stage involves simply mapping the package subtrees of each parent's symbol trees to the target symbol tree. In this context, "package subtree" refers to the maximal subtree of a symbol tree rooted at the `ROOT` package node consisting only of nodes with package symbols. `ROOT` package symbols are always mapped to each other. Other package symbols get paired up whenever their parent package symbols have been matched and their Spoon path is identical.

The file symbol mapping procedure tries to establish mappings for all sub-package-level symbols, i.e., symbols correlating to main type declarations and their descendants. If a file remains unchanged relative to some parents, those parents' file symbol subtrees are directly matched without any diffing taking place. Note that this behavior for unchanged files applies to all "unchanged parents" as determined during target workspace construction, not just the parent the tree was copied from as part of symbol tree reuse. Otherwise, if a file is associated with an addition or modification change event as per the file mapping, GumTree-Spoon's `AstComparator` is used to generate mappings corresponding to a raw AST diff. Notably, this raw diff matches Spoon's `CtElement` nodes, not the symbols themselves; however, the corresponding symbols can be easily retrieved from the respective symbol workspaces using the `CtPath` index mentioned earlier in Section 8.1.1. From this AST diff, the procedure only considers any matches between elements where both the old element is a defined symbol in the parent tree and the new element is a defined symbol in the target tree. Any matches from a non-existent symbol to a defined symbol and vice versa will simply result in an unmatched symbol that will later be interpreted as an addition or deletion (relative to that specific parent), respectively.

Since GumTree-Spoon sometimes ends up matching main type declarations to nested type declarations or other elements, which is often considered undesirable, the matching procedure ensures that main type declarations between parent and target files are always matched to each other, discarding any conflicting matches from the raw diff. Additionally, GumTree-Spoon also does not match so-called *implicit* elements. This primarily affects diffs involving record fields and methods, which, while present in the AST, do not appear in the source code and are thus considered implicit. To ensure that source-code-level type conversions such as turning a record class into regular class and vice versa (without changing the interface of the class) result in explicit and implicit symbols being treated as "continuations" of their counterparts instead of as additions or deletions, implicit symbols (the symbols correlating to the implicit elements in question) are additionally attempted to get matched to unmatched symbols from the other side, matching by Spoon path first and $\langle \text{parent spoon path}, \text{Spoon node type}, \text{simple name} \rangle$ tuple second.

Symbol addition and deletion capture determines which symbols are strict additions (added relative to all parents) and deletions (either absent or deleted relative to all parents). The symbol augmentation and diffing phase is the most involved phase of the entire symbol diffing process. Generally, change processing occurs for each symbol in the target symbol tree according to a preorder traversal. This way, processing can rely on parents of target symbols having already been processed by the time their children get their turn. Note that deletions do not require any additional handling since they are not present in the target tree and have no actual change information to be processed in addition to the deletion itself, which has already been recorded in the previous phase. Inside the loop, each target symbol is assigned its symbol and strand IDs, although the latter will always be equal to the ID of the current commit's strand, even in the case of pure successions. If a symbol is a strict addition, it is assigned a new unique symbol ID and processing ends since additions—just like deletions—have no actual *change* information to process; in the `CommitDiff`, where non-succession additions will eventually end up, they are implicitly understood as change events where *all* their symbol properties count as "new" or "changed".

If a target symbol's source symbols from the parent commits all share the same symbol ID, then that target symbol simply inherits this symbol ID. This is always the case for intra-strand commit sequences (by virtue of only having one parent as per the definition of a strand) but also occurs when a symbol remains mapped throughout multiple strands, only to get merged back together at a merge commit. In that case, there is no "actual" symbol merging taking place as the symbols merged together share the same origin and therefore belong to the same abstract symbol. Within a strand, symbols get updated in-place, otherwise—when switching strands—a succession takes place and the source symbols are declared as `DIRECT` predecessors. If a target symbol's source symbols have conflicting symbol IDs, the symbol receives a new unique symbol ID. Source symbols without any changes are linked as `DIRECT` predecessors (pure successions) and those with changes as `MERGED` predecessors (impure successions).

For mapped pairs of source and target symbols with changes, symbol property diffing follows next. Diffing on properties is performed in a universal way across all properties via an equality check, effectively resulting in the distinction between removed properties, unchanged properties, added properties and changed properties. Of these, the final diff only contains changed properties with their new value (since you can derive whether or not the property existed before from the change history for the visualization) and properties with the `null` value as a stand-in for the removal of properties (which is possible since all removable properties are non-nullable). This diff is then used to instantiate a `SymbolUpdate`, which additionally has update flags—for the symbol being `RENAMED`, having its `BODY_UPDATED`, being `MOVED` in general, being `MOVED_WITH_PARENT` and being `REPLACED` with a different type declaration—set where applicable.

The final phase, the symbol addition and succession post-processing procedure, checks the symbols marked as additions by the symbol addition and deletion capture step, and separates them into true additions (e.g., symbols that may have been added at a merge commit) and successions (e.g., symbols that resulted from merging parent symbols). These addition and succession symbols are then copied as embedded symbols and returned in the final overall `SymbolDiff` output, together with the symbol update and deletion sets from previous steps.

8.1.3 Serialization

Once all commits have been processed, the resulting diff results from the `Strand` objects is then serialized to JSON using Jackson. At this point, all symbols are given new unique symbol IDs to reflect their corresponding abstract symbol, i.e., all input symbols in a predecessor–successor relationship are mapped to belong to the same output symbol. For the diff results themselves, the properties of symbols get denormalized so that every captured state of a symbol contains all the symbol's then-up-to-date property values and a list of property keys to indicate the properties that have changed (or have been removed).

Symbols are also given a `keys` array of smaller objects containing important attributes that are used for search indexing, e.g., all the names the symbol was ever known under. For lookups with temporal restraints, each of these "keys" also includes start and end dates for the validity of the key from a chronological perspective; note that this might not necessarily align with a key's "perceived" validity if the respective symbol has been modified in parallel-running strands. These symbol states are then grouped by ISO 8601 `YearMonth` (`YYYY-MM` representation) for fast index-based lookups on the visualization frontend. Data on commits is included as a separate array on the top-level of the primary JSON object, and any prior occurrences of commit hashes now point to an element in this array instead.

8.1.4 Challenges

Over the course of the implementation, there were several complications due to difficulties implementing the actual change extraction. Initially, the analysis framework was based around the actual AST diff returned by GumTree-Spoon. While the current version still utilizes the mappings between code elements returned as part of the diff, the actual edit operations themselves are not used to create the symbol diff, instead diffing the Spoon properties of the elements through the `Property` objects returned by the property capture component of target workspace construction. Many of the properties of this approach came from the operations taking place on the AST but needing to be projected onto the symbol tree, which led to complications owing to the fact that the symbol tree is itself a graph abstraction of the original AST: determining which operations were actually relevant and how the changes affected relationships between symbols or compounded on each other. There were also a lot of cases where the returned AST diff would simply see the entire type declaration removed and an entirely new tree of elements inserted, causing inaccuracies when it came to the detected changes.

Refactoring detection was also made more difficult due to dependency conflicts between GumTree-Spoon and RefactoringMiner [77], which, because of problems with conflict resolution through Maven, would have meant the two would have to run in two different JVM instances and use some form of RPC or HTTP to communicate. Additionally, even when only using the simple refactoring detection offered by GumTree-Spoon, it remained unclear how one could partially update the project-wide AST without having to reparse the entire project, as there seemed to be no documented way to accomplish this, despite being necessitated by the step-by-step nature of VCS-backed commit-by-commit diffs and a desire for an efficient approach that exploits the redundancies exposed by knowing which files are changed by a commit and which remain unchanged.

Ultimately, these difficulties led to the minimum viable implementation of the analysis framework exceeding the originally intended scope for the purposes of this thesis, which resulted in a reduction in scope regarding the refactoring detection and visualization interaction options. The final prototype uses file-based mappings as a basis for symbol mapping, meaning all refactorings returned by GumTree-Spoon are, as implemented, constrained to taking place within the same file. For the visualization, this meant that filtering and sorting options would be left for future work, although the requirements analysis of this thesis provides a good picture of how these functionalities could be effectively implemented.

8.2 Visualization Frontend

The visualization frontend itself is written using primarily Vue.js and TypeScript. It consists of two major components: the change information querying service and the visualization. When the web application is initially launched, the change information querying service first indexes the JSON output from the analysis framework, then makes the resulting index available for the visualization, which then applies post-processing to all objects for the presentation layer.

8.2.1 Change Information Querying

The visualization frontend uses Fuse.js[1] for its search indexing. At startup, the `keys` of all symbols are entered into the index as keys for their respective symbol. When the user triggers a search, Fuse.js first uses this runtime index to search for key matches. Since the match score returned by Fuse.js measures distance, the returned match score for search results is first inverted ($s_{\mathrm{real}} = 1 - s_{\mathrm{fuse}}$), then followed by a small penalty for deleted symbols to ensure currently still existing symbols appear first in the search results for similarly strong matches. To avoid flooding the visualization with weak-match symbols, only items above a threshold match score of `0.8` are included in search results. When looking up change events for particular symbols in a particular timeframe, the `YearMonth`-indexed states of the symbol are retrieved according to the `YearMonth` values of the start and end date, which are later filtered by the visualization based on the concrete displayed days.

8.2.2 Visualization

In the visualization component, several reactive values are used to keep track of the end date to display (the date span specifying how many days can be displayed is hard-coded), the `DateObject` records for all displayed days, the `YearMonth` values corresponding to the current date range, the list of all search results, the start index of the displayed search results, the list of displayed search results itself and the `Cell` objects displayed in the timeline view. All dates used to refer to days on the timeline throughout the visualization frontend use date normalization, setting the time component to midnight in the local time zone to allow for easier date comparisons. This is different from the `DateObject` type, which is used for records that contain the individual components of a day used for the column headers, namely weekday (`MON`–`SUN`), month (`JAN`–`DEC`) and day of the month (1–31).

When a search is triggered, debouncing is used to avoid lookups in the Fuse.js index on every input. Afterward, search results are converted to `SymbolElement` records, which use a symbol's kind property and modifiers to determine which icon and kind text (e.g., "abstract class") to display. Typed symbols such as fields, variables and methods additionally have type text generated, which is truncated to fit into the width of the search

[1]https://www.fusejs.io/, Accessed: Oct. 9, 2025

results, e.g., `List<AVeryLongType>` might get truncated to `List<...>`. Similarly, the path of each symbol is truncated from the end to only include the list of last package names that can be displayed, e.g., “`com.example.very.long.packagename`” might get truncated to “`...long.packagename`”.

Searching or changing the cell region—either by changing the page of search results or the date range to display—causes the `Cell` records for the timeline view to be re-computed. This is done by first extracting all relevant change events to display, then grouping this data by symbol and date. For each cell with change events, the accumulated states then get assigned a `CellEventCategory` corresponding to the kind and severity of the most drastic change; for example, a cell with only change events affecting line ranges is categorized as `MINISCULE`, change events with renamings or type changes will cause the category to be at least `MAJOR`, and symbol additions or deletions cause the category to be `ADDED` or `DELETED`, respectively. It is this category, alongside the kind of the most drastic change (e.g., `RENAMED`), which affects the displayed cell color and icon. Miniscule-category cells are those that get displayed as only a smaller change event dot on the timeline. The kind values associated with each type of change are additionally used for determining which “pills” to display above the change event dot, sorted by their associated severity and always starting from the change event which was used to decide the cell color and icon.

Based on the creation and (optional) deletion event of each search result symbol, each cell is also assigned `starts` and `ends` truth values, which are used to determine whether the colored strip on the timeline should start, end, continue (if both are `true`) or be absent altogether (if both are `false`). The last-change indicator displayed after the last change event for still existing symbols is triggered by the `last` truth value, which is computed in the same step. Since the number of cells displayed on each view update is large, columns and cells use string-based hashing to allow Vue.js to only re-render columns and cells that change, which significantly improves responsiveness due to the usually large amount of empty or continuation cells. Special care has gone into making sure that cells with identical appearance but different underlying change information still get re-rendered.

When it comes to the details popover that gets triggered when clicking on a change event dot, the visualization computes the list of properties to display (e.g., all for addition and deletion events, only updated properties for modification events) and refers to a mapping of property display functions for how to render values of individual properties. This allows each property to be rendered using a suitable human-readable representation; for example, list-type properties such as placed annotations or implemented interfaces get treated as lists of type values, and simple properties such as symbol kind or initial value use additional text formatting to ensure readability (“Enum constant” instead of `ENUM_CONSTANT`) and conciseness (shortening to “(Longer expression)” if too long), respectively.

Where full information, such as the fully qualified name or full initialization expression, is hidden or truncated, an `abbr` text is additionally set. Property values with an `abbr` text are underlined via CSS to indicate that hovering over them reveals additional information.

The data returned by the property display functions is also used to determine how to render the before–after diff of property values: Changes to simple properties will simply see the old and new value separated by a right-facing arrow, while changes to list-type properties are additionally diffed using an LCS algorithm to make sure individual values retain their relative order and value changes (e.g., added or removed annotations) are inserted at the right indices (these changes are displayed with background highlighting as well as green plus and red minus signs in the popover).

CHAPTER 9

Evaluation

This chapter covers the evaluation of the visualization approach as described in Chapter 7 and implemented as summarized in Chapter 8 in addition to the evaluation of this thesis as a whole, with an interpretation and discussion of the results as well as an examination of the threats to internal and external validity.

9.1 Scenario-based Expert Evaluation and Semi-Structured Interviews

As outlined in Chapter 5, the overall process for the evaluation of the visualization approach was carried out in a similar manner to the interviews conducted as part of the requirements analysis phase, with the aim of validating the final prototype and discussing the general sentiment towards this area of research after multiple rounds of usage scenario-based demonstrations. In terms of methodology, this meant that the expert interviews were divided into one pilot interview and three evaluated interviews, with the pilot interview once again serving as a way to gather feedback for the design of the questionnaire, usage scenarios, and general interviewing process. For the final version of the evaluation questionnaire, see Appendix B.

The interviews were carried out individually using an online conferencing tool and lasted for roughly 40 minutes each, with the exception of the pilot interview, which took about 50 minutes. It is worth noting that the evaluation was carried out with a slightly different group of respondents compared to the requirements analysis phase, consisting two returning participants and one new interviewee.

9.1.1 Scenario and Interview Design

As already touched on above, the questionnaire for the evaluation phase was designed with three key goals in mind:

1. Establish the respondents' demographic makeup and VCS usage.
2. Validate the visualization prototype via demonstration and discussion based on predetermined usage scenarios.
3. Discuss overall thoughts on the visualization and the general outlook on symbol-based SEV as a whole.

Like before, this naturally leads to the questionnaire being split into three parts, each corresponding to one of the goals just listed. The initial block centered around demographic characteristics was included due to the differing respondent group, expanded with a short question about general VCS usage to provide additional context for the second and third block answers.

The second block revolves around the aforementioned usage scenarios. Each usage scenario section consists of three parts: a question about the general use case, a live demonstration of how the visualization addresses the use case and a closing question about the perceived usefulness of the visualization in this context. The first question in particular serves the important purpose of validating the use case itself before the eventual judgment on the efficacy of the visualization. The closing questions are always accompanied by a brief discussion on the perks and drawbacks of the visualization from the participant's perspective after the live demonstration. Both questions utilize Likert-scale answer options: "Not very useful" / "Somewhat useful" / "Useful" / "Very useful" and "Poor" / "Fair" / "Good" / "Excellent" for usefulness and performance, respectively.

The usage scenarios themselves are derived from the requirements analysis phase, largely based on the findings of the literature review from Section 6.1, specifically the core question categories related to symbol-based change search, seeing how it was the search view which was ultimately chosen for the implementation (see Section 7.4). Specifically, the use cases chosen for the scenario-based evaluation center around code evolution ("What has changed?" and "Has it changed?"), the creation and change timeline ("When has it changed?" and "When was it created and deleted?"), change motivation ("Why was it changed?") and change authorship ("Who has changed it?" and "Who created or changed it last?"). The concrete data for the live demonstration was based on the output of the analysis framework when being fed itself as the input, with slight adjustments made to showcase the final usage scenario with multiple authors.

The third block poses two key questions regarding the concept of symbol-based visualization: First, it asks the participant about the general perceived usefulness of the visualization following the usage scenarios, given the same answer options as before

("Not very useful" / "Somewhat useful" / "Useful" / "Very useful"). During the actual interview, respondents are urged to again consider the subjective relevance of the provided usage scenarios, the visualization's performance in these scenarios and the difference in presentation, interaction methods and change granularity by virtue of changes being tracked on the symbol level. The second question addresses the overall sentiment towards symbol-level visualization as a field of research, framed as a question of interest, with the answer options "Not interested", "Slightly interested", "Interested" and "Very interested". Participants are encouraged to reflect on the advantages and disadvantages of symbol-level change visualization in general as well as the potential for future developments in this working direction (contrasted with traditional change detection and visualization as known through Git). Finally, the questionnaire ends with a last call for comments from the interviewee.

It is worth mentioning that the pilot interview—despite its longer runtime—did not lead to any significant changes to the questionnaire or usage scenarios. The usage scenarios in particular went through multiple iterations even before the pilot interview, leading to multiple adaptations such as slight changes to the wording of the central scenario questions and a shift from three more involved scenarios (roughly 10 minutes in length) to four simpler ones (roughly 7 minutes in length). Nevertheless, the fact that the (by the pilot interview) then-current interview process, usage scenarios and questionnaire seemed to work well in practice served as an implicit kind of feedback in much the same way as the explicit feedback seen in the requirements analysis interviews.

9.1.2 Evaluation Results

This section serves as a brief review of the results of the three primary interviews conducted over the course of the evaluation phase.

Demographics

Questions 1–3 pertained to many of the same basic demographics questions seen in the requirements analysis—namely, the participants' age group, declared gender and years of professional programming experience. Once more, it is important to highlight that professional programming experience in this context refers to years working on programming projects with stakeholders outside of a self-educational context. Identical to the requirements analysis, all three participants belonged to the 25–34 year age group and exclusively identified as male. Participants P1 and P2 both specified 2–5 years of experience, and P3 5–10 years (which, for P1 and P3, was in line with their previous answers). The additional question (number 4) regarding the general frequency of using tools to inspect or browse code history was universally answered with the highest option, "Often".

Usage Scenarios

Following next are the previously mentioned four pairs of questions about the usefulness of the presented use cases and performance of the visualization in the corresponding contexts, one pair for each usage scenario, which will here be referred to as the “what” (S1), “when” (S2), “why” (S3) and “who” (S4) scenarios for simplicity’s sake. Figure 9.1 shows the perceived usefulness for each use case as judged by the interviewees. It would appear that the participants had a clear preference for the first and third scenarios pertaining to the “what” and “why” questions, respectively. This can also be seen in the verbal responses from the interview: P1 remarked that the “when” is often less important than the “what” and primarily useful for edge cases (where it is, however, deemed *very* useful). Expressing a similar sentiment, P2 stated that the fourth scenario regarding the “who” is generally very interesting but often only of limited use. For comparison, Figure 9.2 shows the perceived performance of the visualization prototype. Overall, the absence of “Poor” and “Fair” ratings indicates that the visualization indeed adequately addresses the presented use cases. Feedback was overall very positive, with P1 remarking that the “why” scenario worked pretty well and P2 saying the “who” scenario was excellent with nothing further to note.

Recurring feedback included wishes for collapsible timeline regions in the absence of changes and adjustable timeline granularity as well as comments on the reliance of the “why” scenario on the expressiveness of commit messages. However, on the topic of the “why” scenario specifically: While every respondent noted something to the effect that change rationale can be notoriously difficult to glean from version control information (a hypothesis which the literature review from this thesis supports), they all seemed to agree that, given sufficient commit discipline, the visualization works well in helping to extract this rationale information where available.

Overall Thoughts

The final questions, numbered 13–15, address the overall thoughts on the visualization in particular as well as symbol-based visualization in general. The results from the questionnaire are shown in Figure 9.3, where both of these aspects have been combined for easier comparison. In line with feedback for the visualization’s usefulness in individual use cases, all participants agree that the developed prototype proved to be “very useful”. P1 mentioned that the approach looks very promising, while P2 praised the visualization’s intuitiveness. P3 remarked that it definitely seems more useful than Git in many regards and that structural changes are presented in a more understandable manner than a traditional line-based diff. Regarding the overall interview, P2 felt that the presented use cases were all very sensible. General interest in further research into symbol-level visualization was above average across the board, with P1 and P2 noting that additional filter and search options would present a significant improvement for the visualization prototype for future work.

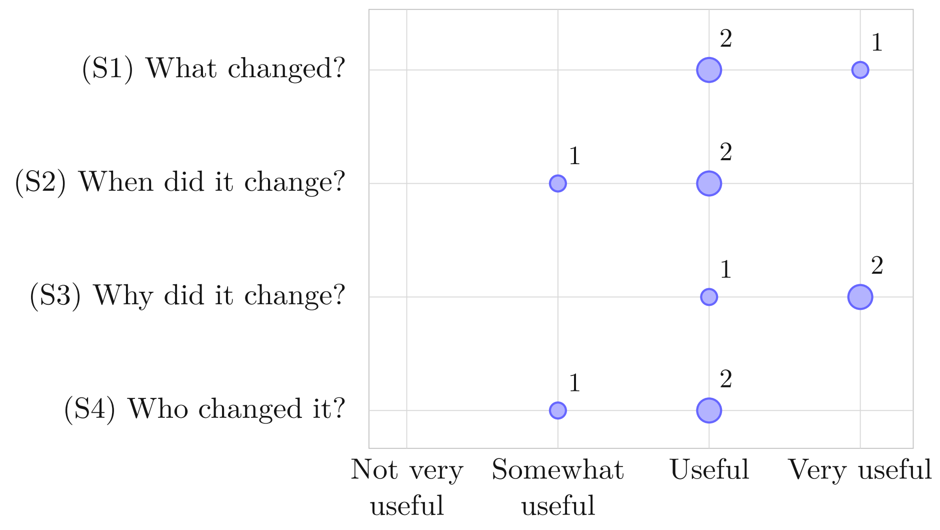


Figure 9.1: Perceived usefulness of software evolution information

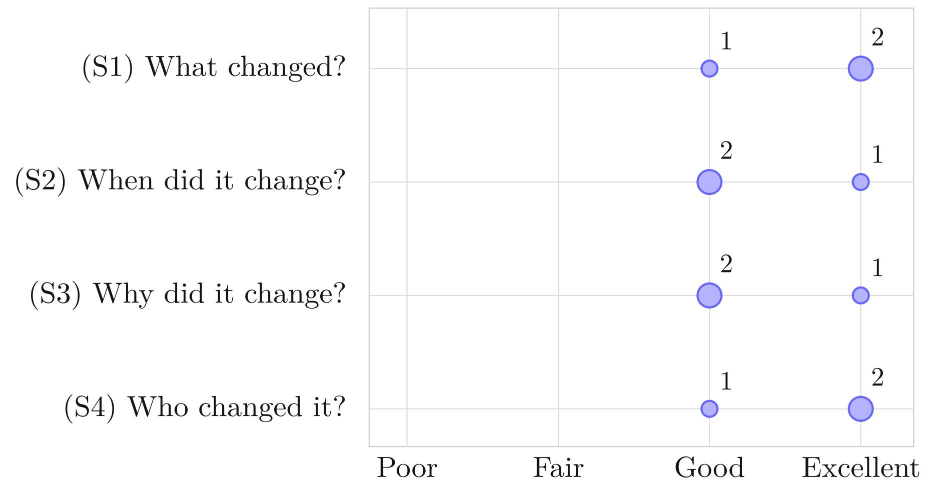


Figure 9.2: Perceived performance of the visualization prototype

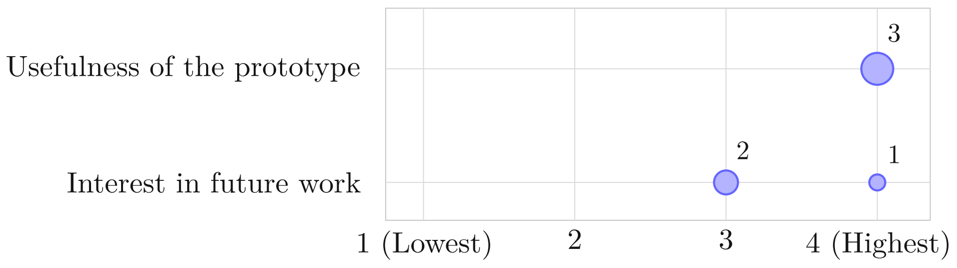


Figure 9.3: Verdict on the current state of symbol-based visualization

9.2 Discussion

This section discusses the results of this thesis throughout its chapters, primarily focusing on the evaluation interview results while also including findings from the state-of-the-art VCS history mining analysis, the requirements analysis and the implementation where appropriate.

9.2.1 Interpretation of Results

Ultimately, the evaluation interviews ended with universal positive feedback, with Figure 9.3 showing both that the developed visualization prototype sufficiently demonstrates the potential of symbol-based history processing and visualization methods and that unambiguous interest in future work in this field indeed exists. Based on the concrete feedback received, it appears that the overall presentation-layer structure of such an approach does manage to communicate change information on a more granular level in an intuitive-to-navigate and readable way.

Most of the constructive criticism from the participants addresses the advanced filter and timeline zoom options which were originally planned to be included prior to the scope reduction described in Section 8.1.4. However, this feedback serves as a validation of the findings from the literature review and requirements analysis interview results in addition to providing a clear direction for future work to add onto this thesis. In any case, the concluding thoughts of the interviewees demonstrate that even in this limited-scope visualization prototype, there is enough functionality present for it to provide an immense help in finding useful historical change information, confirming the thesis's hypothesis about the merits of symbol-based history visualization.

9.2.2 Examination of Research Questions

As stated in Chapter 5, this thesis aims to answer three research questions. What follows is a breakdown of each research question and the answer provided by this thesis, as can be found in previous chapters.

RQ1: What insights can developers gain from symbol-level code history information?

This first research question pertains to the utility of symbol-level SEV and its advantages over line-based approaches. For this purpose, a literature review and requirements analysis interviews were conducted in order to elicit the state of current research and determine the use cases as well as possible working directions for this approach. As per the findings in Chapter 6, there indeed exists a need for solutions to finding accurate information regarding software code history which can only be uncovered using sub-file-level change detection, such as capturing changes to code elements or refactoring detection. This resulted in a list of twenty relevant questions commonly found in scientific literature which closely relate to changes to code components such as symbols and VCS change information as captured by systems like Git, as well as an analysis of the reasons for a lack of suitable solutions, which ultimately resulted in a set of requirements that was then validated through semi-structured interviews with experts in software engineering. These interviews sufficiently showed the demand for symbol-based history processing tools—particularly, search-based visualization applications—and validated many of the use cases related to problems frequently stated in contemporary works, such as difficulties with finding relevant change information within complex version control histories.

RQ2: How can a symbol-based visualization be designed to satisfy the information needs of developers?

This second research question pertains to how the information needs identified in the requirements analysis can be properly addressed by a visualization-based approach. This question is addressed in two parts: feasibility and design considerations. First, Chapter 7 and Chapter 8 of this thesis go into the concrete challenges unique to symbol-based history processing faced when trying to realize such a concept. With little related work (see Chapter 3 and Chapter 4) available to look to for visualization-based approaches based on efficient symbol-level history information mining techniques, it was initially unclear how feasible an actual implementation might be. As a result, this thesis set out to create both an analysis framework capable of harvesting this information and a visualization prototype, thereby effectively demonstrating possible directions in software architecture, implementation techniques and ways to transform the harvested information into a format suitable for an interactive visual user interface. Second, Chapter 7 delves into more detail regarding challenges faced when creating said user interface, based on the findings from the literature review and requirements analysis interviews alike, discussing design considerations along all steps of the conceptualization process, iteration by iteration, until finally coalescing in the final visualization prototype.

RQ3: Does the proposed visualization approach lead to the intended benefits for developers?

This third and final research question pertains to whether or not a visualization yielded by this approach can actually result in a useful abstraction that delivers on the potential benefits outlined by answering RQ1. For this purpose, the final visualization prototype was validated through a second round of semi-structured expert interviews by way of live demonstrations in usage scenarios based on the use cases previously exposed in the requirements analysis. The results clearly reveal very positive overall sentiment towards both the concrete visualization prototype developed and symbol-based SEV as a whole, cementing symbol-based software history mining and visualization as a promising area of research with significant potential for future work.

9.3 Threats to Validity

This section addresses the threats to the validity of the results presented in the requirements analysis and evaluation chapters.

Questionnaire-induced biases. While the interview questionnaires underwent multiple revisions in order to be constructed as neutral as possible, it cannot be ascertained whether the concrete wording, framing or structure of the questions could have influenced participants' responses to lean toward more negative or positive answers.

Inconsistency-induced biases. As differing sets of participants were used for the requirements analysis and evaluation interviews, the ability to draw any causal relationships between the two is impeded. At the same time, the fact that there is some overlap between the two groups violates assumptions about the independence of the two results.

Contamination bias. The overlap between the participant groups for both sets of interviews and the fact that all participants stem from the same organization at TU Wien lead to an inability to rule out external factors such as communication between interviewees after the requirements analysis and prior to the evaluation phase.

Number of participants. Both the requirements analysis and evaluation interviews used a sample size of just three. This number of participants is too small to be able to generalize the results of this thesis to broader populations.

Participant selection bias. The fact that all respondents stem from the same organization at TU Wien causes the observed results to not be generalizable, as this would require random sampling from a broader pool of software engineers, including those not at TU Wien.

CHAPTER 10

Conclusion

This thesis saw the methodical investigation of a design problem in the research area of Software Evolution Visualization. The initial stages began with a thorough review of current scientific literature and state-of-the-art concepts and techniques. Utilizing this knowledge, this thesis proceeded with semi-structured expert interviews to validate its initial findings and hypotheses and gather additional actionable feedback that was later used to refine a visualization concept from a pen-and-paper proposal to a functional full-stack prototype.

The requirements analysis revealed several insights into information needs in software engineering, Mining Software Repositories and Software Evolution Visualization alike, concluding that these areas are ripe with potential regarding the use of symbol-level change information for software history visualization approaches, as shown by the large number of comments regarding pain points with existing solutions.

Along the entire process, this work included design considerations, implementation challenges and possible avenues for realizing a suitable visualization concept. All of these findings culminated in a treatment evaluation that more than adequately demonstrated not just the viability of symbol-level visualizations, but also the potential for further enhancements and interest in continued work in this direction within the field of MSR and SEV research.

10.1 Future Work

Due to time constraints and the challenging nature of AST-level code differencing and refactoring detection, not all of this thesis's original ideas could be brought to fruition during the design and implementation stages. Ultimately resulting in a reduction of scope, the requirements analysis and evaluation interview findings nevertheless demonstrate possible visualization, interaction and distortion avenues for displaying and navigating symbol-level change information. As such, many of the design considerations already outlined in the requirements analysis provide simple extension points to this work, primarily in the areas of extended search, filtering and zoom functionalities.

Similarly, other possible visualization concepts designed but not realized as part of this thesis, such as the exploration-, person- and change-based perspectives, garnered very positive responses from interviewees and present other possible types of symbol-based visualizations worth exploring in future work.

In terms of functionality, the originally intended solution saw the use of the RefactoringMiner library [27], [77] for more involved inter-file refactoring detection, something that was foregone in the final implementation, which instead relied on GumTree [25] and Spoon [34] for intra-file refactorings such as method moves and Git for file-level refactorings such as top-level class renamings. Naturally, more expressive and accurate refactoring detection is another aspect that may yet be improved upon as part of future research.

The empirical expert interviews provide an additional opportunity to gather more information regarding information needs in software engineering in the context of changes on the symbol level by way of increasing the breadth and number of survey participants to further substantiate or refute this thesis's findings.

Lastly, as the treatment evaluation shows, there exists growing interest in future developments, so altogether different approaches in the realm of AST- or CST-based data mining approaches in conjunction with software history visualization are expected to attract further valuable insights into these examined topics, as well.

List of Figures

List of Tables

Acronyms

ACM Association for Computing Machinery. 31

AI Artificial Intelligence. 46

API Application Programming Interface. 24, 27, 36, 68

AST Abstract Syntax Tree. 8, 9, 14, 15, 17, 19, 21–23, 25, 28, 30, 31, 68, 74, 76, 78–81, 83, 95, 96

CFG Control Flow Graph. 8, 17

CI/CD Continuous Integration / Continuous Delivery. 11, 17

CLI Command Line Interface. 36

CSS Cascading Stylesheets. 32, 59, 86

CST Concrete Syntax Tree. 7, 9, 14, 15, 21, 22, 25, 30, 96

CSV Comma-Separated Values. 32

DAG Directed Acyclic Graph. 6, 9, 15, 25, 74

DFG Data Flow Graph. 8, 17

HTML Hypertext Markup Language. 32, 59

HTTP Hypertext Transfer Protocol. 68, 69, 83

IDE Integrated Development Environment. 16, 19, 36, 54, 59, 60

IEEE Institute of Electrical and Electronics Engineers. 31

IFT Information Foraging Theory. 14

ILP Integer Linear Programming. 10, 15

INSO Industrial Software. 32

ITS Issue Tracking System. 10, 17, 19

JSON JavaScript Object Notation. 69, 82–84

JVM Java Virtual Machine. 68, 83

LCS Longest Common Subsequence. 9, 15, 23, 24, 86

LLM Large Language Model. 46

LOC Lines of Code. 55

LP Linear Programming. 10

MSR Mining Software Repositories. 1, 10, 95

npm npm (formerly Node Package Manager). 22

POM Project Object Model. 22

PSI Program Structure Interface. 59

REST Representational State Transfer. 27, 68

RPC Remote Procedure Call. 68, 83

SEV Software Evolution Visualization. 1, 4, 11, 13, 16, 20, 30, 31, 40, 41, 53, 66, 88, 92, 93, 95

SHA Secure Hash Algorithm. 71

UI User Interface. 44, 45, 59, 68

VCS Version Control System. 8, 10, 11, 14, 15, 17, 21, 25–28, 30, 31, 36, 40, 44, 45, 54, 66, 68, 73, 83, 88, 92

XML Extensible Markup Language. 15, 24

XPath XML Path Language. 49

Scientific References

[1] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, „Software history under the lens: A study on why and how developers examine it“, in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Bremen, Germany: IEEE, Sep. 2015, pp. 1–10, ISBN: 978-1-4673-7532-0. DOI: `10.1109/ICSM.2015.7332446`

[2] S. S. Ragavan, M. Codoban, D. Piorkowski, D. Dig, and M. Burnett, „Version Control Systems: An Information Foraging Perspective“, *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1644–1655, Aug. 2021, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: `10.1109/TSE.2019.2931296`

[3] M. Shahin, P. Liang, and M. A. Babar, „A systematic review of software architecture visualization techniques“, *Journal of Systems and Software*, vol. 94, pp. 161–185, Aug. 2014, ISSN: 0164-1212. DOI: `10.1016/j.jss.2014.03.071`

[4] J. Grabner, R. Decker, T. Artner, M. Bernhart, and T. Grechenig, „Combining and Visualizing Time-Oriented Data from the Software Engineering Toolset“, in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, Madrid: IEEE, Sep. 2018, pp. 76–86, ISBN: 978-1-5386-8292-0. DOI: `10.1109/VISSOFT.2018.00016`

[5] S. McKee, N. Nelson, A. Sarma, and D. Dig, „Software Practitioner Perspectives on Merge Conflicts and Resolutions“, in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai: IEEE, Sep. 2017, pp. 467–478, ISBN: 978-1-5386-0992-7. DOI: `10.1109/ICSME.2017.53`

[6] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, „CodeShovel: Constructing Method-Level Source Code Histories“, in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2021, pp. 1510–1522. DOI: `10.1109/ICSE43902.2021.00135`

[7] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer, 2014, ISBN: 978-3-662-43838-1 978-3-662-43839-8. DOI: `10.1007/978-3-662-43839-8`

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools* (Always learning), Second edition, Pearson new international edition. Harlow: Pearson, 2014, ISBN: 978-1-292-02434-9.

[9] R. Nystrom, *Crafting Interpreters*. Daryaganj Delhi: Genever Benning, Jul. 2021, ISBN: 978-0-9905829-3-9.

[10] P. Kelly, „A congruence theorem for trees“, *Pacific Journal of Mathematics*, vol. 7, no. 1, pp. 961–968, Mar. 1957, ISSN: 0030-8730, 0030-8730. DOI: 10.2140/pjm.1957.7.961

[11] P. Bille, „A survey on tree edit distance and related problems“, *Theoretical Computer Science*, vol. 337, no. 1, pp. 217–239, Jun. 2005, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2004.12.030

[12] S. D. B. Effendi, B. Çirisci, R. Mukherjee, H. A. Nguyen, and O. Tripp, „A Language-agnostic Framework for Mining Static Analysis Rules from Code Changes“, in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, ISSN: 2832-7659, May 2023, pp. 327–339. DOI: 10.1109/ICSE-SEIP58684.2023.00035

[13] K. Chaturvedi, V. Sing, and P. Singh, „Tools in Mining Software Repositories“, in *2013 13th International Conference on Computational Science and Its Applications*, Jun. 2013, pp. 89–98. DOI: 10.1109/ICCSA.2013.22

[14] P. Caserta and O Zendra, „Visualization of the Static Aspects of Software: A Survey“, *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 7, pp. 913–933, Jul. 2011, ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.110

[15] T. Fritz and G. C. Murphy, „Using information fragments to answer the questions developers ask“, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, Cape Town South Africa: ACM, May 2010, pp. 175–184, ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806828

[16] T. D. LaToza and B. A. Myers, „Hard-to-answer questions about code“, in *Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '10, New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 1–6, ISBN: 978-1-4503-0547-1. DOI: 10.1145/1937117.1937125

[17] A. Begel and T. Zimmermann, „Analyze this! 145 questions for data scientists in software engineering“, in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: Association for Computing Machinery, May 2014, pp. 12–23, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568233

[18] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, „The life-cycle of merge conflicts: Processes, barriers, and strategies“, *Empirical Software Engineering*, vol. 24, no. 5, pp. 2863–2906, Oct. 2019, ISSN: 1573-7616. DOI: 10.1007/s10664-018-9674-x

[19] L. Merino, M. Ghafari, and O. Nierstrasz, „Towards Actionable Visualisation in Software Development“, in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, Raleigh, NC, USA: IEEE, Oct. 2016, pp. 61–70, ISBN: 978-1-5090-3850-3. DOI: 10.1109/VISSOFT.2016.10

[20] K. J. North, A. Sarma, and M. B. Cohen, „Understanding Git history: A multi-sense view“, in *Proceedings of the 8th International Workshop on Social Software Engineering*, Seattle WA USA: ACM, Nov. 2016, pp. 1–7, ISBN: 978-1-4503-4397-8. DOI: 10.1145/2993283.2993285

[21] A. McNair, D. M. German, and J. Weber-Jahnke, „Visualizing Software Architecture Evolution Using Change-Sets“, in *14th Working Conference on Reverse Engineering (WCRE 2007)*, ISSN: 1095-1350, Vancouver, BC, Canada: IEEE, Oct. 2007, pp. 130–139, ISBN: 978-0-7695-3034-5. DOI: 10.1109/WCRE.2007.52

[22] K. Zhang and D. Shasha, „Simple Fast Algorithms for the Editing Distance between Trees and Related Problems“, *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989, Publisher: Society for Industrial and Applied Mathematics, ISSN: 0097-5397. DOI: 10.1137/0218082

[23] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, „Change detection in hierarchically structured information“, in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD ’96, New York, NY, USA: Association for Computing Machinery, Jun. 1996, pp. 493–504, ISBN: 978-0-89791-794-0. DOI: 10.1145/233269.233366

[24] B. Fluri, M. Wursch, M. PInzger, and H. Gall, „Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction“, *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2007.70731

[25] 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*, ser. ASE ’14, New York, NY, USA: Association for Computing Machinery, Sep. 2014, pp. 313–324, ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642982

[26] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, „Accurate and efficient refactoring detection in commit history“, in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg Sweden: ACM, May 2018, pp. 483–494, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180206

[27] N. Tsantalis, A. Ketkar, and D. Dig, „RefactoringMiner 2.0“, *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, Mar. 2022, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2020.3007722

[28] Qingtao Jiang, X. Peng, Hai Wang, Z. Xing, and W. Zhao, „Summarizing Evolutionary Trajectory by Grouping and Aggregating relevant code changes“, in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada: IEEE, Mar. 2015, pp. 361–370, ISBN: 978-1-4799-8469-5. DOI: 10.1109/SANER.2015.7081846

[29] Sunghun Kim, Kai Pan, and E. Whitehead, „When functions change their names: Automatic detection of origin relationships“, in *12th Working Conference on Reverse Engineering (WCRE'05)*, Pittsburgh, PA: IEEE, 2005, 10 pp.–152, ISBN: 978-0-7695-2474-0. DOI: `10.1109/WCRE.2005.33`

[30] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall, „Redundancy-free analysis of multi-revision software artifacts“, *Empirical Software Engineering*, vol. 24, no. 1, pp. 332–380, Feb. 2019, ISSN: 1382-3256. DOI: `10.1007/s10664-018-9630-9`

[31] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, „HyperAST: Enabling Efficient Analysis of Software Histories at Scale“, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–12, ISBN: 978-1-4503-9475-8. DOI: `10.1145/3551349.3560423`

[32] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, „HyperDiff: Computing Source Code Diffs at Scale“, in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, San Francisco CA USA: ACM, Nov. 2023, pp. 288–299, ISBN: 9798400703270. DOI: `10.1145/3611643.3616312`

[33] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jezequel, „Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions“, in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Luxembourg: IEEE, Sep. 2021, pp. 206–216, ISBN: 978-1-66542-882-8. DOI: `10.1109/ICSME52107.2021.00025`

[34] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, „SPOON : A library for implementing analyses and transformations of Java source code“, *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, Sep. 2016, ISSN: 00380644. DOI: `10.1002/spe.2346`

[35] M. Jodavi and N. Tsantalis, „Accurate method and variable tracking in commit history“, in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore: ACM, Nov. 2022, pp. 183–195, ISBN: 978-1-4503-9413-0. DOI: `10.1145/3540250.3549079`

[36] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, „Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns“, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada: IEEE, May 2019, pp. 819–830, ISBN: 978-1-72810-869-8. DOI: `10.1109/ICSE.2019.00089`

[37] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, „Discovering repetitive code changes in python ML systems“, in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 2022, pp. 736–748, ISBN: 978-1-4503-9221-1. DOI: `10.1145/3510003.3510225`

[38] J. Maletic and M. Collard, „Supporting source code difference analysis", in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, Chicago, IL, USA: IEEE, 2004, pp. 210–219, ISBN: 978-0-7695-2213-5. DOI: 10.1109/ICSM.2004.1357805

[39] M. L. Collard, M. J. Decker, and J. I. Maletic, „Lightweight Transformation and Fact Extraction with the srcML Toolkit", in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Williamsburg, VA, USA: IEEE, Sep. 2011, pp. 173–184, ISBN: 978-1-4577-0932-6. DOI: 10.1109/SCAM.2011.19

[40] B. Bartman, C. D. Newman, M. L. Collard, and J. I. Maletic, „srcQL: A syntax-aware query language for source code", in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria: IEEE, Feb. 2017, pp. 467–471, ISBN: 978-1-5090-5501-2. DOI: 10.1109/SANER.2017.7884655

[41] M. L. Collard, J. I. Maletic, and A. Marcus, „Supporting document and data views of source code", in *Proceedings of the 2002 ACM symposium on Document engineering*, ser. DocEng '02, New York, NY, USA: Association for Computing Machinery, Nov. 2002, pp. 34–41, ISBN: 978-1-58113-594-7. DOI: 10.1145/585058.585066

[42] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, „srcDiff: A syntactic differencing approach to improve the understandability of deltas", *Journal of Software: Evolution and Process*, vol. 32, no. 4, e2226, Apr. 2020, ISSN: 2047-7473, 2047-7481. DOI: 10.1002/smr.2226

[43] E. W. Myers, „AnO(ND) difference algorithm and its variations", *Algorithmica*, vol. 1, no. 1, pp. 251–266, Nov. 1986, ISSN: 1432-0541. DOI: 10.1007/BF01840446

[44] 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*, ser. IWPSE-EVOL '11, New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 96–100, ISBN: 978-1-4503-0848-9. DOI: 10.1145/2024445.2024463

[45] D. M. German, B. Adams, and K. Stewart, „Cregit: Token-level blame information in git version control repositories", *Empirical Software Engineering*, vol. 24, no. 4, pp. 2725–2763, Aug. 2019, ISSN: 1573-7616. DOI: 10.1007/s10664-019-09704-x

[46] Y. Higo, S. Hayashi, and S. Kusumoto, „On tracking Java methods with Git mechanisms", *Journal of Systems and Software*, vol. 165, p. 110 571, Jul. 2020, ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110571

[47] T. Omori and K. Maruyama, „A change-aware development environment by recording editing operations of source code", May 2008, pp. 31–34. DOI: 10.1145/1370750.1370758

[48] E. Kitsu, T. Omori, and K. Maruyama, „Detecting Program Changes from Edit History of Source Code“, in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, Bangkok: IEEE, Dec. 2013, pp. 299–306, ISBN: 978-1-4799-2144-7 978-1-4799-2143-0. DOI: `10.1109/APSEC.2013.48`

[49] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama, „Historef: A tool for edit history refactoring“, in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada: IEEE, Mar. 2015, pp. 469–473, ISBN: 978-1-4799-8469-5. DOI: `10.1109/SANER.2015.7081858`

[50] Y. Nishimura and K. Maruyama, „Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History“, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita: IEEE, Mar. 2016, pp. 661–664, ISBN: 978-1-5090-1855-0. DOI: `10.1109/SANER.2016.46`

[51] K. Maruyama, S. Hayashi, and T. Omori, „ChangeMacroRecorder: Recording fine-grained textual changes of source code“, in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy: IEEE, Mar. 2018, pp. 537–541, ISBN: 978-1-5386-4969-5. DOI: `10.1109/SANER.2018.8330255`

[52] M. Kim and D. Notkin, „Discovering and representing systematic code changes“, in *2009 IEEE 31st International Conference on Software Engineering*, Vancouver, BC, Canada: IEEE, 2009, pp. 309–319, ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070531`

[53] D. Keim, „Information visualization and visual data mining“, *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 1–8, Mar. 2002, ISSN: 10772626. DOI: `10.1109/2945.981847`

[54] J. Maletic, A. Marcus, and M. Collard, „A task oriented view of software visualization“, in *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*, Paris, France: IEEE Comput. Soc, 2002, pp. 32–40, ISBN: 978-0-7695-1662-2. DOI: `10.1109/VISSOF.2002.1019792`

[55] H. Bani-Salameh, A. Ahmad, and A. Aljammal, „Software evolution visualization techniques and methods - a systematic review“, Jul. 2016, pp. 1–6. DOI: `10.1109/CSIT.2016.7549475`

[56] J. Ratzinger, M. Fischer, and H. Gall, „EvoLens: Lens-View Visualizations of Evolution Data“, in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Lisbon, Portugal: IEEE, 2005, pp. 103–112, ISBN: 978-0-7695-2349-1. DOI: `10.1109/IWPSE.2005.16`

[57] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall, „Evo-Clocks: Software Evolution at a Glance“, in *2019 Working Conference on Software Visualization (VISSOFT)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 12–22, ISBN: 978-1-72814-939-4. DOI: `10.1109/VISSOFT.2019.00010`

[58] Y. Guo, S. Bettaieb, Q. Hu, Y. L. Traon, and Q. Tang, *CodeLens: An Interactive Tool for Visualizing Code Representations*, arXiv:2307.14902 [cs], Jul. 2023. DOI: 10.48550/arXiv.2307.14902

[59] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, et al., „Comparing repositories visually with repograms“, in *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin Texas: ACM, May 2016, pp. 109–120, ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901768

[60] X. Xie, D. Poshyvanyk, and A. Marcus, „Visualization of CVS Repository Information“, in *2006 13th Working Conference on Reverse Engineering*, Benevento, Italy: IEEE, 2006, pp. 231–242, ISBN: 978-0-7695-2719-2. DOI: 10.1109/WCRE.2006.55

[61] T. Dal Sasso, R. Minelli, A. Mocci, and M. Lanza, „Blended, not stirred: Multi-concern visualization of large software systems“, in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, Bremen, Germany: IEEE, Sep. 2015, pp. 106–115, ISBN: 978-1-4673-7526-9. DOI: 10.1109/VISSOFT.2015.7332420

[62] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, „How Developers Drive Software Evolution“, in *Eighth International Workshop on Principles of Software Evolution (IWPSE’05)*, Lisbon, Portugal: IEEE, 2005, pp. 113–122, ISBN: 978-0-7695-2349-1. DOI: 10.1109/IWPSE.2005.21

[63] A. Kuhn and M. Stocker, „CodeTimeline: Storytelling with versioning data“, in *2012 34th International Conference on Software Engineering (ICSE)*, Zurich: IEEE, Jun. 2012, pp. 1333–1336, ISBN: 978-1-4673-1066-6 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227086

[64] Y. Yoon, B. A. Myers, and S. Koo, „Visualization of fine-grained code change history“, in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, San Jose, CA, USA: IEEE, Sep. 2013, pp. 119–126, ISBN: 978-1-4799-0369-6. DOI: 10.1109/VLHCC.2013.6645254

[65] F. Chevalier, D. Auber, and A. Telea, „Structural analysis and visualization of C++ code evolution using syntax trees“, in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, Dubrovnik Croatia: ACM, Sep. 2007, pp. 90–97, ISBN: 978-1-59593-722-3. DOI: 10.1145/1294948.1294971

[66] R. Holmes and A. Begel, „Deep intellisense: A tool for rehydrating evaporated information“, in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR ’08, New York, NY, USA: Association for Computing Machinery, May 2008, pp. 23–26, ISBN: 978-1-60558-024-1. DOI: 10.1145/1370750.1370755

[67] A. W. Bradley and G. C. Murphy, „Supporting software history exploration“, in *Proceedings of the 8th Working Conference on Mining Software Repositories*, Waikiki, Honolulu HI USA: ACM, May 2011, pp. 193–202, ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985469

[68] V. Frick, C. Wedenig, and M. Pinzger, „DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts“, in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid: IEEE, Sep. 2018, pp. 705–709, ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00081

[69] E. Aghajani, A. Mocci, G. Bavota, and M. Lanza, „The Code Time Machine“, in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, Buenos Aires, Argentina: IEEE, May 2017, pp. 356–359, ISBN: 978-1-5386-0535-6. DOI: 10.1109/ICPC.2017.6

[70] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, et al., „RefactorInsight: Enhancing IDE Representation of Changes in Git with Refactorings Information“, in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ISSN: 2643-1572, Nov. 2021, pp. 1276–1280. DOI: 10.1109/ASE51524.2021.9678646

[71] R. Escobar, J. P. S. Alcocer, H. Tarner, F. Beck, and A. Bergel, „Spike – A code editor plugin highlighting fine-grained changes“, in *2022 Working Conference on Software Visualization (VISSOFT)*, Limassol, Cyprus: IEEE, Oct. 2022, pp. 167–171, ISBN: 978-1-66548-092-5. DOI: 10.1109/VISSOFT55257.2022.00026

[72] S. Rufiange and G. Melancon, „AniMatrix: A Matrix-Based Visualization of Software Evolution“, in *2014 Second IEEE Working Conference on Software Visualization*, Victoria, BC, Canada: IEEE, Sep. 2014, pp. 137–146, ISBN: 978-1-4799-6150-4. DOI: 10.1109/VISSOFT.2014.30

[73] R. Novais, C. Nunes, C. Lima, E. Cirilo, F. Dantas, A. Garcia, et al., „On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation“, in *2012 34th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Jun. 2012, pp. 1044–1053. DOI: 10.1109/ICSE.2012.6227115

[74] Y. Kim, J. Kim, H. Jeon, Y.-H. Kim, H. Song, B. Kim, et al., „Githru: Visual Analytics for Understanding Software Development History Through Git Metadata Analysis“, *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 656–666, Feb. 2021, ISSN: 1941-0506. DOI: 10.1109/TVCG.2020.3030414

[75] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd. Pragmatic Bookshelf, 2013, ISBN: 978-1-934356-99-9.

[76] M. Martinez, J.-R. Falleri, and M. Monperrus, „Hyperparameter Optimization for AST Differencing“, *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4814–4828, Oct. 2023, arXiv:2011.10268 [cs], ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2023.3315935

[77] P. Alikhanifard and N. Tsantalis, *A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools*, arXiv:2403.05939 [cs], Mar. 2024. DOI: 10.48550/arXiv.2403.05939

[78] R. Likert, S. Roslow, and G. Murphy, „A Simple and Reliable Method of Scoring the Thurstone Attitude Scales“, *The Journal of Social Psychology*, vol. 5, no. 2, pp. 228–238, 1934, Publisher: Routledge. DOI: 10.1080/00224545.1934.9919450

[79] A. B. W. Kennedy and H. R. Sankey, „The thermal efficiency of steam engines. report of the committee appointed to the council upon the subject of the definition of a standard or standards of thermal efficiency for steam engines: With an introductory note. (including appendixes and plate at back of volume).“, *Minutes of the Proceedings of the Institution of Civil Engineers*, vol. 134, no. 1898, pp. 278–312, Jan. 1898. DOI: 10.1680/imotp.1898.19100

Online References

[80] M. Brunsfeld, A. Hlynskyi, A. Qureshi, P. Thomson, J. Vera, P. Turnbull, et al. „Tree-sitter/tree-sitter: V0.22.2“, DOI: `10.5281/zenodo.10827268` [Online]. Available: `https://zenodo.org/records/10827268`

[81] D. van Bruggen, F. Tomassetti, R. Howell, M. Langkabel, N. Smith, A. Bosch, et al. „Javaparser/javaparser: Release javaparser-parent-3.16.1“, DOI: `10.5281/zenodo.3842713` [Online]. Available: `https://zenodo.org/records/3842713`

Appendices

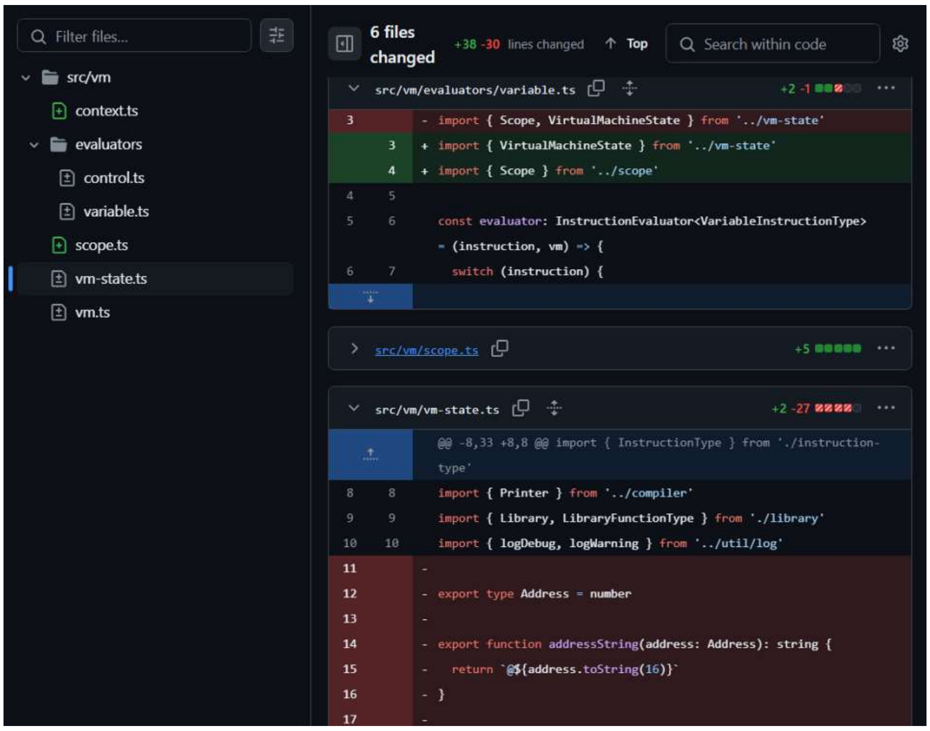
APPENDIX A

Requirements Analysis Questionnaire

Information Needs in Software Engineering

* Indicates required question

Introduction



```
- import { Scope, VirtualMachineState } from '../vm-state'
+ import { VirtualMachineState } from '../vm-state'
+ import { Scope } from '../scope'

  const evaluator: InstructionEvaluator<VariableInstructionType>
  = (instruction, vm) => {
    switch (instruction) {
```

```
@@ -8,33 +8,8 @@ import { InstructionType } from './instruction-type'
  import { Printer } from '../compiler'
  import { Library, LibraryFunctionType } from './library'
  import { logDebug, logWarning } from '../util/log'
-
- export type Address = number
-
- export function addressString(address: Address): string {
-   return `@${address.toString(16)}`
- }
-
```

Demographics

1. 1. What is your age? *

Mark only one oval.

- ◯ 18 - 24 years
- ◯ 25 - 34 years
- ◯ 35 - 44 years
- ◯ 45 - 54 years
- ◯ 55 - 64 years
- ◯ 65+ years

2. 2. What is your gender? *

Mark only one oval.

- ◯ Male
- ◯ Female
- ◯ Prefer not to say
- ◯ Other: ____________________

3. 3. How long have you been professionally working in programming (in years)? *

Mark only one oval.

- ◯ < 1 year
- ◯ 1 - 2 years
- ◯ 2 - 5 years
- ◯ 5 - 10 years
- ◯ 10 - 15 years
- ◯ 15 - 20 years
- ◯ 20 - 25 years
- ◯ > 25 years

Current State of Software History Usage

4. 4. How frequently are you using the Git Version Control System for source code? *

Mark only one oval.

- Never
- Rarely
- Sometimes
- Often
- Exclusively

5. 5. How often do you use these tools to browse or inspect source code history? *

Mark only one oval per row.

	Never	Rarely	Sometimes	Often
Code hosting platforms (GitHub, GitLab, ...)				
Code editors (IntelliJ, VSCode, ...)				
Git UI tools (GitHub Desktop, Sourcetree, ...)				
Git command line (git log, git blame, ...)				
Dedicated source code history tools (CodeShovel, CodeTracker, ...)				

6. 6. When did you last use source code history of any kind to uncover or verify information? *

Mark only one oval.

- ◯ In the last 3 days
- ◯ This week (4-7 days ago)
- ◯ Last week (8-14 days ago)
- ◯ Earlier this month (15-30 days ago)
- ◯ Last month (31-62 days ago)
- ◯ Earlier this year (63-365 days ago)
- ◯ Not in the past year (366+ days ago)
- ◯ I have never had to check source code history

7. 7. When working with software history tools in the past, I have had problems with... *

Mark only one oval per row.

	Never	Rarely	Sometimes	Often
Refactorings being easily recognizable (e.g., method move)	⬭	⬭	⬭	⬭
Irrelevant changes showing up (e.g., reformattings)	⬭	⬭	⬭	⬭
Inconsistent terminology (e.g., unfamiliar/unusual wording)	⬭	⬭	⬭	⬭
Lack of useful filtering options / Too many results	⬭	⬭	⬭	⬭
Lack of usability	⬭	⬭	⬭	⬭
Difficulties locating the relevant change within a commit	⬭	⬭	⬭	⬭
Unappealing graphical presentation	⬭	⬭	⬭	⬭
Having to sift through commits to find the change I'm looking for	⬭	⬭	⬭	⬭

8. 8. When would you normally call a change "recent"? *

If your answer is "it depends": What circumstances would you say make you more likely to call a change "recent"?

9. 9. When you check source code history, how old are the changes you usually need to check? *

"Uncommitted" does legitimately refer to changes that are not yet part of **any** commit. In this context, *unpushed* changes count as "recent", not "uncommitted".

Mark only one oval per row.

	Never	Rarely	Sometimes	Often	Very often
Uncommitted	◯	◯	◯	◯	◯
Recent	◯	◯	◯	◯	◯
Non-recent	◯	◯	◯	◯	◯

10. 10. What umbrella term do you use to refer to packages, classes, methods, fields, variables, parameters and the like collectively? *

Mark only one oval.

◯ I don't think an umbrella term exists
◯ Code symbols
◯ Code elements
◯ Other: ______________________

11. 11. When checking source code history, support for inspecting more fine-grained code elements (e.g., methods and fields as opposed to files/classes) is... *

Mark only one oval.

◯ Useless
◯ Rarely useful
◯ Sometimes useful
◯ Useful
◯ Very useful

12. 12. How would you rate the importance of the history of code elements that have been **fully removed**? *

Note that "fully removed" does not include scenarios where elements that have been removed and incorporated elsewhere within the same change (refactorings) such as moves, renamings, reorderings or inlined methods.

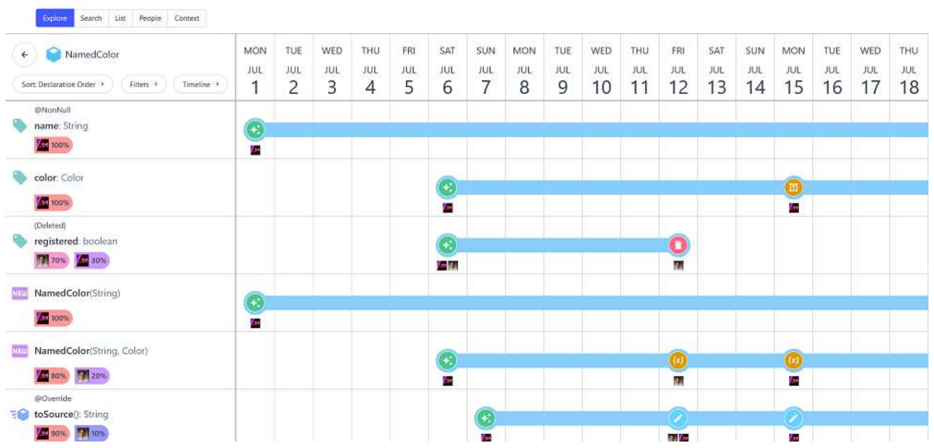
Mark only one oval.

- ⬭ Not important
- ⬭ Slightly important
- ⬭ Moderately important
- ⬭ Very important

Visualization Mockups - Introduction

The next sections will show you mockups of a visualization prototype showing the history of code elements. The images in question **depict the same visualization** but differ in how elements can be discovered and the data displayed.

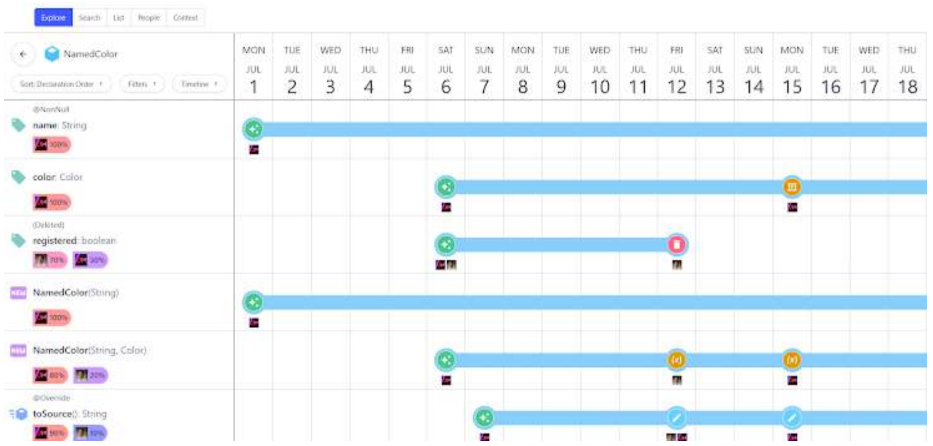
General layout



Exploration-based Visualizations for Software History

13. 13a. How useful would you find a visualization that lets you navigate through the tree of code elements and shows the history of elements within a shared context? *

"Code elements within a shared context" could be, for example, classes in a package, methods and fields in a class, parameters and variables of a method, etc.



Mark only one oval.

- ◯ Useless
- ◯ Rarely useful
- ◯ Sometimes useful
- ◯ Useful
- ◯ Very useful

14. 13b. What do you think of the above visualization concept for navigating through code elements and showing their change history? *

15. 14. When looking at a code element (e.g., a package, a class or a method), which of its properties do you find the most useful? *

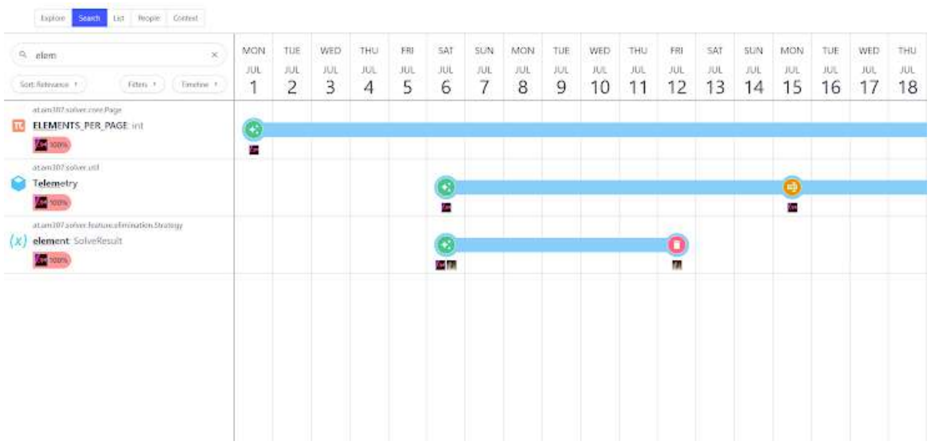
Mark only one oval per row.

	Not useful	Rarely useful	Sometimes useful	Useful	Very useful
Person who created it	◯	◯	◯	◯	◯
Person who last changed it	◯	◯	◯	◯	◯
Date and time of the last change to it	◯	◯	◯	◯	◯
Number of recent changes to it	◯	◯	◯	◯	◯
Number of people who contributed to it	◯	◯	◯	◯	◯
People who contributed to it and to which amount	◯	◯	◯	◯	◯
Number of its type relationships (e.g., subtypes, usages, references)	◯	◯	◯	◯	◯
Size / Complexity of it	◯	◯	◯	◯	◯
Number of total changes to it	◯	◯	◯	◯	◯
Number of code elements contained in it (e.g. classes in a package,	◯	◯	◯	◯	◯

members in a
class)

Search-based Visualizations for Software History

16. 15a. How useful would you find a visualization that acts as a search for code elements * and shows their history?



Mark only one oval.

- ⬭ Useless
- ⬭ Rarely useful
- ⬭ Sometimes useful
- ⬭ Useful
- ⬭ Very useful

17. 15b. What do you think of the above visualization concept for searching for code * elements and showing their change history?

18. 16. When searching for a code element, I hold the following opinion of these filter options: *

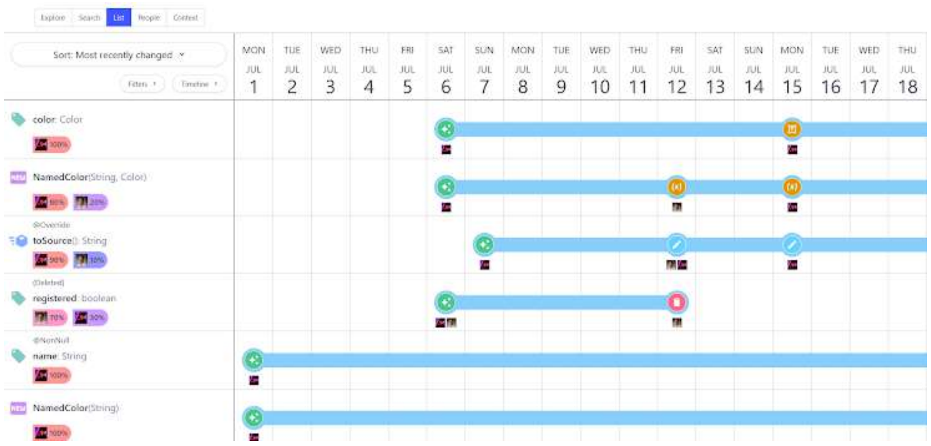
Mark only one oval per row.

	Useless	Rarely useful	Sometimes useful	Useful	Very useful
Changed most recently	◯	◯	◯	◯	◯
Changed least recently	◯	◯	◯	◯	◯
Changed after a point in time (from)	◯	◯	◯	◯	◯
Changed in a specific timeframe (from-to)	◯	◯	◯	◯	◯
Introduced in a specific timeframe	◯	◯	◯	◯	◯
Changed by me specifically	◯	◯	◯	◯	◯
Changed by a certain other person	◯	◯	◯	◯	◯
Changed most frequently	◯	◯	◯	◯	◯
Never changed	◯	◯	◯	◯	◯
Changed by a lot of different people	◯	◯	◯	◯	◯

Ranking-based Visualizations for Software History

19. 17a. How useful would you find a visualization that shows all changes sorted by criteria such as change recency or frequency? *

Change recency ... "What was last changed?"
Change frequency ... "What was changed the most or most often?"



Mark only one oval.

◯ Useless

◯ Rarely useful

◯ Sometimes useful

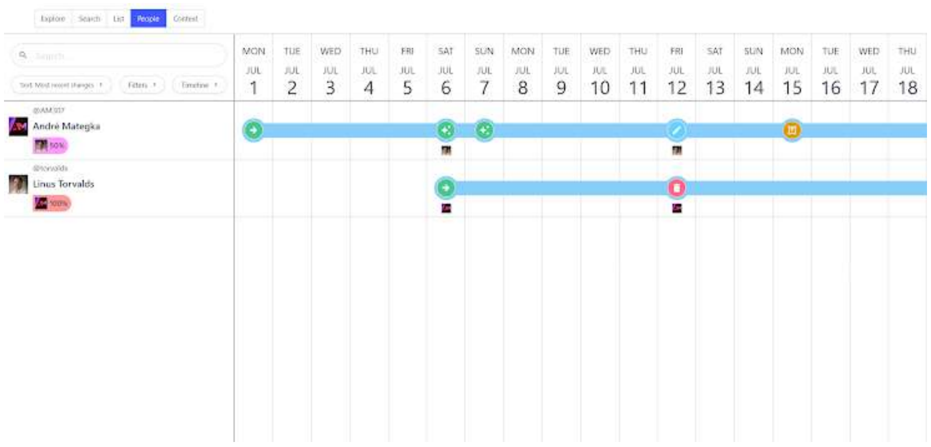
◯ Useful

◯ Very useful

20. 17b. What do you think of the above visualization concept for exploring a sorted list of code elements and their change history? *

Person-based Visualizations for Software History

21. 18a. How useful would you find a visualization that shows the changes made to code elements grouped by person? *



Mark only one oval.

- ◯ Useless
- ◯ Rarely useful
- ◯ Sometimes useful
- ◯ Useful
- ◯ Very useful

22. 18b. What do you think of the above visualization concept for exploring people and their authored changes? *

____

____

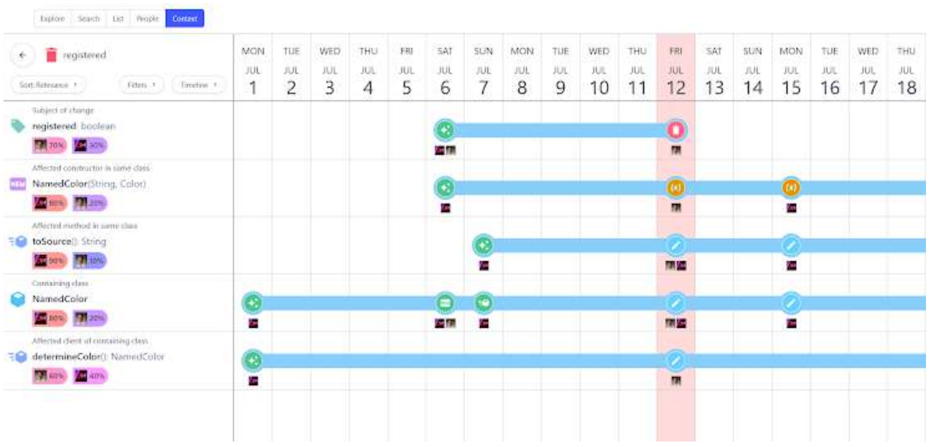
____

____

____

Change-to-Change Visualizations for Software History

23. 19a. How useful would you find a visualization that shows the code elements related to a specific code element change? *



Mark only one oval.

- ⬭ Useless
- ⬭ Rarely useful
- ⬭ Sometimes useful
- ⬭ Useful
- ⬭ Very useful

24. 19b. What do you think of the above visualization concept for exploring the code elements related to a specific change? *

Closing

25. 20. How useful would you find a "Compare" list feature which allows you to show changes for previously selected code elements? *

This notion is a bit difficult to describe. If you have visited technology search or ranking websites before, you may recognize this feature (e.g., phone comparison on GSMArena, product comparison on Geizhals, ...).

The intention would be to allow you to more easily compare the changes made to sets of code elements which do not usually occur together (e.g., packages with different parent packages, non-adjacent search results, ...).

Mark only one oval.

- ◯ Useless
- ◯ Rarely useful
- ◯ Sometimes useful
- ◯ Useful
- ◯ Very useful

26. 21. Are there any other features you would add or improve on? *

Also, if there is any general aspect you find particularly important, feel free to mention it here.

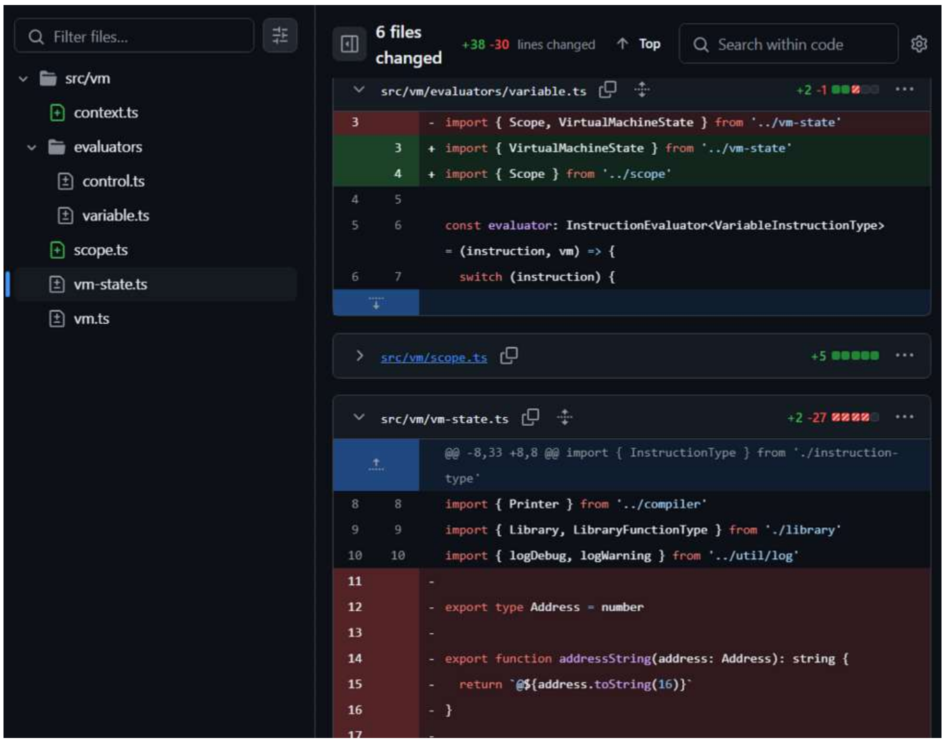
APPENDIX B

Evaluation Questionnaire

Visualization of Symbol-Level Code Changes

* Indicates required question

Introduction



```
- import { Scope, VirtualMachineState } from '../vm-state'
+ import { VirtualMachineState } from '../vm-state'
+ import { Scope } from '../scope'

  const evaluator: InstructionEvaluator<VariableInstructionType>
  = (instruction, vm) => {
    switch (instruction) {
```

```
@@ -8,33 +8,8 @@ import { InstructionType } from './instruction-
type'
  import { Printer } from '../compiler'
  import { Library, LibraryFunctionType } from './library'
  import { logDebug, logWarning } from '../util/log'
- 
- export type Address = number
- 
- export function addressString(address: Address): string {
-   return `@${address.toString(16)}`
- }
- 
```

Demographics

1. 1. What is your age? *

Mark only one oval.

- ◯ 18 - 24 years
- ◯ 25 - 34 years
- ◯ 35 - 44 years
- ◯ 45 - 54 years
- ◯ 55 - 64 years
- ◯ 65+ years

2. 2. What is your gender? *

Mark only one oval.

- ◯ Male
- ◯ Female
- ◯ Prefer not to say
- ◯ Other: ______________________

3. 3. How long have you been professionally working in programming (in years)? *

Mark only one oval.

- ◯ < 1 year
- ◯ 1 - 2 years
- ◯ 2 - 5 years
- ◯ 5 - 10 years
- ◯ 10 - 15 years
- ◯ 15 - 20 years
- ◯ 20 - 25 years
- ◯ > 25 years

4. 4. How frequently would you say you use tools **to inspect or browse source code history**? *

Examples:

- GitHub, GitLab, ... (Code hosting platforms)
- IntelliJ, VSCode, ... (Code editors)
- git log, git blame, ... (Git command line)
- GitHub Desktop, Sourcetree, ... (Git UI tools)
- CodeShovel, CodeTracker, ... (Dedicated source code history tools)

Mark only one oval.

- ◯ Never
- ◯ Rarely
- ◯ Sometimes
- ◯ Often

Scenario 1: Code Evolution Discovery

5. 5. Being able to uncover **what changes, if any, have been made to a particular class/method/field/variable/... over time** is... *

Mark only one oval.

- ◯ Not very useful to me
- ◯ Somewhat useful to me
- ◯ Useful to me
- ◯ Very useful to me

6. 6. How would you describe this visualization's ability to help you understand **what changes, if any, have been made to a particular class/method/field/variable/... over time**? *

Mark only one oval.

- ◯ Poor
- ◯ Fair
- ◯ Good
- ◯ Excellent

Scenario 2: Creation and Change Timeline Discovery

7. 7. Being able to uncover **when a class/method/field/variable/... was created, changed or deleted** is... *

Mark only one oval.

- ◯ Not very useful to me
- ◯ Somewhat useful to me
- ◯ Useful to me
- ◯ Very useful to me

8. 8. How would you describe this visualization's ability to help you understand **when a class/method/field/variable/... was created, changed or deleted**? *

Mark only one oval.

- ◯ Poor
- ◯ Fair
- ◯ Good
- ◯ Excellent

Scenario 3: Change Motivation Discovery

9. 9. Being able to uncover **why a class/method/field/variable/... was changed** is... *

Mark only one oval.

- ⬭ Not very useful to me
- ⬭ Somewhat useful to me
- ⬭ Useful to me
- ⬭ Very useful to me

10. 10. How would you describe this visualization's ability to help you understand **why a class/method/field/variable/... was changed**? *

Mark only one oval.

- ⬭ Poor
- ⬭ Fair
- ⬭ Good
- ⬭ Excellent

Scenario 4: Change Authorship Discovery

11. 11. Being able to uncover **who made changes to a class/method/field/variable/..., including who created or last changed it**, is... *

Mark only one oval.

- ⬭ Not very useful to me
- ⬭ Somewhat useful to me
- ⬭ Useful to me
- ⬭ Very useful to me

12. 12. How would you describe this visualization's ability to help you understand **who made changes to a class/method/field/variable/..., including who created or last changed it**? *

Mark only one oval.

- ◯ Poor
- ◯ Fair
- ◯ Good
- ◯ Excellent

Closing

13. 13. **In general**, how **useful** did you find this kind of visualization concept based on the demonstrated use cases? *

Mark only one oval.

- ◯ Not very useful
- ◯ Somewhat useful
- ◯ Useful
- ◯ Very useful

14. 14. **In general**, how **interested** are you in the possible capabilities of this kind of visualization concept now that you have seen it in action? *

Mark only one oval.

- ◯ Not interested
- ◯ Slightly interested
- ◯ Interested
- ◯ Very interested

15. 15. Do you have any closing remarks or suggestions?

This content is neither created nor endorsed by Google.

Google Forms