

A Graph-Based Source Code Multi-Diff Visualization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Mario Pilz, BSc

Matrikelnummer 01325898

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 29. August 2024

Unterschrift Verfasser

Unterschrift Betreuung



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

A Graph-Based Source Code Multi-Diff Visualization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering und Internet Computing

by

Mario Pilz, BSc

Registration Number 01325898

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 29th August, 2024

Signature Author

Signature Advisor



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



A Graph-Based Source Code Multi-Diff Visualization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Mario Pilz, BSc

Matrikelnummer 01325898

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, 29. August 2024



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Mario Pilz, 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, 29. August 2024

Mario Pilz



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich bedanke mich bei meinem Betreuer für seinen Rat, den er mir während der Erstellung dieser Arbeit gegeben hat. Ebenfalls bedanke ich mich bei meinen Freunden, die sich Zeit genommen haben, diese Arbeit korrekturzulesen.

Desweiteren möchte ich mich bei meinem Betreuer, meinen Freunden, meinen Arbeitskollegen und meiner Familie bedanken, die mich während dieser Zeit ausgehalten haben, für ihre mentale Unterstützung, ohne welcher diese Arbeit vielleicht nie fertiggestellt worden wäre.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I want to thank my advisor for the guidance he gave me during the creation of this thesis. I also want to thank my friends who took their time to proof-read my this thesis.

Furthermore, I want to thank my advisor, friends, colleagues, and family, who put up with me during that time and provided me with the mental support without which this thesis might never have been finished.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Softwareentwickler verwenden Versionskontrollsysteme (VCSs) zur Verwaltung von Versionen und Koordination der Entwicklung. Um Einblicke in die Evolution des Source Codes eines Projektes zu erhalten, wurden Werkzeuge entwickelt: Mining Software Repositories (MSR)-Werkzeuge extrahieren Information aus dem Projektrepository und Visualisierungen machen diese Information für Menschen leichter verständlich. Seit Mitte der 2000er-Jahre sind verteilte Versionskontrollsysteme (DVCSs) vorherrschend geworden. Aufgrund der guten Unterstützung von Verzweigungen und Zusammenführung dieser Systeme haben sich Entwicklungsprozesse etabliert, welche sich auf diese Fähigkeiten verlassen. Der Großteil der Forschung auf den Gebieten von MSR und Visualisierungen konzentriert sich jedoch nach wie vor auf rein lineare Historien.

In dieser Diplomarbeit wird eine dreidimensionale Visualisierung der Historie einer einzelnen Datei in einem VCS-Repository vorgestellt. Das Ziel ist es, einen Prototypen zu entwickeln um die Machbarkeit einer solchen Visualisierung zu demonstrieren und ihren Nutzen beim Verstehen von Codeevolution mit verzweigter Historie zu evaluieren.

Um den Stand der Technik in den Gebieten von MSR und Softwarevisualisierung und seine Mängel im Kontext von verzweigten Historien zu ermitteln, wurde eine Literatur- und Toolrecherche durchgeführt. Aufgrund der Ergebnisse und der persönlichen Erfahrung des Autors mit solchen Werkzeugen, wurde ein Visualisierungskonzept entwickelt und eine Liste von Features vorgeschlagen. Das Konzept wurde durch semistrukturierte Interviews mit Entwicklern evaluiert, wobei Mockups zur Erklärung verwendet wurden. Mit den Ergebnissen wurden die Features für die Entwicklung des Prototypen priorisiert, welcher in einer zweiten Runde von Interviews evaluiert wurde.

Die vorgestellte Visualisierung wurde als nützlich für das Verstehen von Codeevolution in einer verzweigten Historie befunden. Auch die einzelnen Features wurden von den Interviewteilnehmern als nützlich bewertet. Für weitere Entwicklungen wurde mehr Interaktivität gefordert, als der gezeigte Prototyp bot. Integration mit existierenden Entwicklungsumgebungen und Projektmanagementwerkzeugen wird als notwendig erachtet. Desweiteren konnte gezeigt werden, dass eine Umsetzung als Web-Anwendung, ohne Einsatz spezieller Technologien für dreidimensionale Grafik, machbar ist.

Schlüsselwörter: *Softwarevisualisierung, Visualisierung von Codeevolution, Historien-graph, verzweigte Softwareentwicklung, Visualisierungssysteme und -werkzeuge, Softwareentwicklung, Software-Konfigurationsmanagement and Versionskontrollsysteme*



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Software developers have been using Version Control Systems (VCSs) for the purposes of version management and development coordination. To get insights about the evolution of a project's source code, tools have been developed: Mining Software Repositories (MSR) tools extract information from a project's repository and visualizations are used to make the information easier to understand for humans. Since the mid-2000s, Distributed Version Control Systems (DVCSs) have become dominant. Due to their good support for branching and merging, development processes relying on these features have become common. However, the fields of MSR and software visualizations have not yet caught up; most research still focuses on purely linear history.

To address this issue, a three-dimensional visualization of the history of one file in a VCS repository is proposed in this thesis. The objective is to develop a prototype to demonstrate the feasibility of such a visualization and evaluate its usefulness for understanding of code evolution in the context of branched history.

A literature and tool research was performed to determine how the state of the art in the fields of MSR and software visualization is inadequate for dealing with branched VCS histories. Based on the results and the author's own experience of working with such tools, the visualization concept was developed, including a list of proposed features. The concept was then evaluated in semi-structured interviews with developers, using mockups for explanation. The results were used to prioritize the features for the following prototype development. The prototype was then evaluated in a second set of interviews.

The obtained findings indicate that the proposed visualization is purposeful for understanding the evolution of code in a branched history. The individual features were generally rated purposeful by the interview participants. For future developments, the interviewees requested more interactivity than the prototype provided. Integration with existing Integrated Development Environments (IDEs) and project management tools was deemed necessary. Implementing the proposed visualization as a Web application without dedicated technologies for three-dimensional graphics was demonstrated to be feasible.

Keywords: *Software Visualization, Visualization of Code Evolution, History Graph, Multi-Branch Software Development, Visualization Systems and Tools, Software Engineering, Software Configuration Management and Version Control Systems*



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	xiii
Abstract	xv
Contents	xvii
1 Introduction	1
1.1 Problem Description	1
1.2 Aim of the Thesis	2
1.3 Methodology	4
1.4 Structure of this Thesis	5
2 Fundamentals: Version Control Systems	7
2.1 History	7
2.2 Differentiating Characteristics	8
2.3 Terms and Concepts of Version Control	11
3 State of the Art	17
3.1 Mining Software Repositories	17
3.2 Existing History Visualization Tools and Methods	19
3.3 Distinction from Existing Research and Tools	22
4 Conceptual Design	25
4.1 Introduction	25
4.2 Overview	26
4.3 Implementation Considerations	35
5 Semi-Structured Expert Interviews on the Concept	43
5.1 Plan	43
5.2 Results	44
5.3 Summary and Interpretation	54
5.4 Threats to Validity	55
6 Implementation	57
	xvii

6.0	Iteration 0: Setup	57
6.1	Iteration 1: Layout Algorithm for Layered Graphs	59
6.2	Iteration 2: Graph Layout as Part of the Visualization	68
6.3	Iteration 3: Diff Indicators	77
6.4	Iteration 4: Scrolling of File Text	83
6.5	Iteration 5: Simple Graphic Settings	85
6.6	Iteration 6: Selecting Shown Branches	87
6.7	Iteration 7: Code Indentation	89
6.8	Iteration 8: Difference Indications on Long Edges	91
6.9	Iteration 9: Commit Handles and Reference Labels	93
6.10	Summary of the Prototype	94
7	Semi-Structured Expert Evaluation with the Prototype	99
7.1	Plan	99
7.2	Results	102
7.3	Summary and Interpretation	112
7.4	Threats to Validity	113
8	Results	115
9	Conclusion	117
9.1	Future Work	119
A	Questionnaire of the Interviews on the Concept	121
B	Questionnaire of the Prototype Evaluation	141
	List of Figures	151
	List of Tables	155
	List of Listings	157
	List of Algorithms	159
	Glossary	161
	Acronyms	163
	Bibliography	165
	Web References	173

Introduction

1.1 Problem Description

Developers have used VCSs to manage versions of the software they write and coordinate work for a long time. In recent years, DVCSs in particular are being used more often [1] with branches as a tool for multiple developers to work concurrently and merge their work afterwards [1]–[3]. In fact, Barr, Bird, Rigby, *et al.* [1] conclude, that the success of DVCSs stems from their lightweight support of branching and merging. One such DVCS is Git [W1], which, since its introduction in 2005 has become one of the most used VCSs [4].

The version history of a software system can get quite complex and hard to comprehend. Several ways of dealing with this problem have developed, such as, data mining [2], [5] or tools dedicated to querying the history of a VCS repository [6].

Another approach to help developers understand the recorded changes are visualizations [3]. There are several tools available for this purpose. However, the visualizations provided by these tools have a variety of limitations. On the one hand, there are several quite abstract visualizations, that are not very well suited for application in practical software engineering contexts [4].

On the other hand, there are a several techniques for visualizing linear evolution of code [4], but these often do not extend to branches. While there are few examples of tools, that do concern themselves with branches, they do not scale well as size and complexity of the repositories increase [4]. Again other tools, do provide a visualization of the history in graph form, but do not show the changes made in the individual commits [2]. Navigation in these visualizations is also often cumbersome.

With the current state of available tools, it is hard to get an overview of changes in a piece of code in a project, that makes heavy use of branching and merging. However,

as already mentioned, in current software engineering practice, developers often make extensive use of the branching facilities provided by modern VCSs [2]. Therefore, tools and visualizations concerned with only linear histories do not fit current software engineering practices very well.

1.2 Aim of the Thesis

This thesis aims at developing a prototype for a set of visualizations showing the change history in form of differences between the versions in a VCS repository that involves branches, that is, where the history is not linear but forms a directed acyclic graph. Existing techniques for visualizing changes between versions in linear histories as well as visualizations of branched histories will be considered and combined to new ones. The prototype should give a better overview of the evolution of source code on different scales (whole files or selected sections) in a project where branching is used. This thesis aims to answer the following research questions:

RQ1 – How to visualize a graph-based source multi-diff?

The goal for this research question is a concept for a visualization of the history of code in a VCS, which involves the history graph of the repository as well as the changes of the file across versions.

RQ2 a How do experts assess the proposed solution?

This research question aims to find the opinions of experts—software developers working with VCSs—on the proposed concept and the prototype.

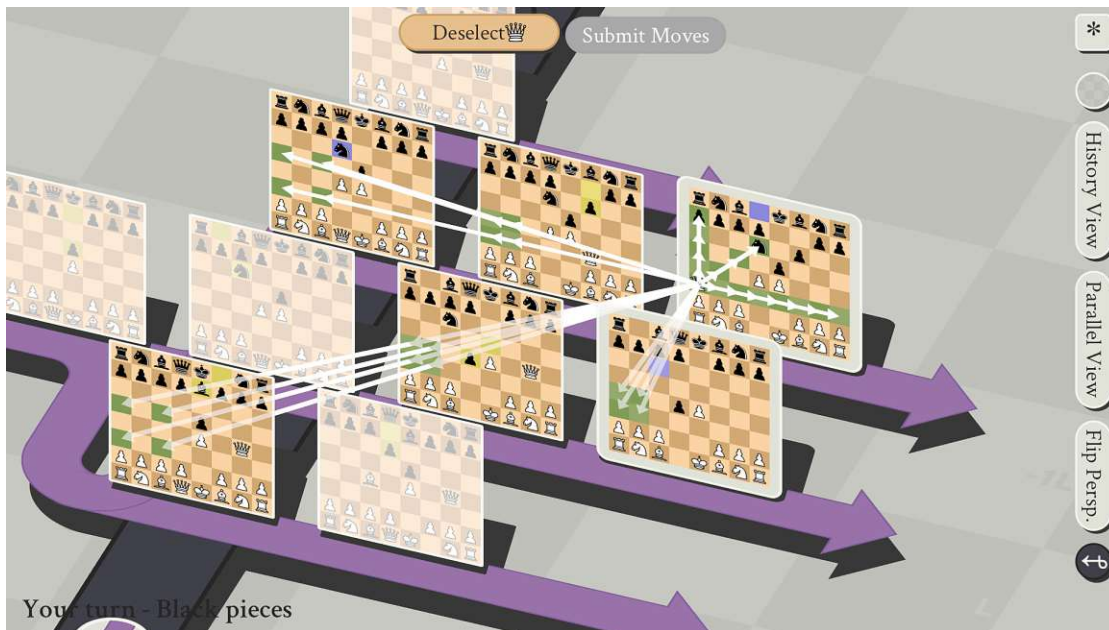
b What expectations and requirements do the experts have for the proposed solution?

The goal of this research question is to reveal expectations that developers have towards a tool that incorporates the concept, so that future improvements can be directed towards the needs of the users.

RQ3 – How purposeful is the proposed visualization method for comprehending the evolution of a source file?

The purpose of the proposed visualization is to help developers understand how the code in a project evolved over time. The aim of this research question is to find whether this goal has been achieved.

Inspiration for this thesis was drawn from the game *5D Chess With Multiverse Time Travel* [W2], in which the classic game of chess is extended by—among other things—non-linear history in form of parallel realities. Each intermediate state of the game is displayed as a board and successive steps in one reality are displayed linearly and connected as a timeline. Each state can be the source of more than one move, in which case a fork in the timeline occurs and a new independent timeline is created, with its own states displayed alongside the existing realities. An example is shown in Figure 1.1.

Figure 1.1: Situation in *5D Chess* [W2]

This model of parallel realities, is very similar to the non-linear histories that branching and merging in a VCS repositories produce. Therefore, it could be possible to create a visualization for code evolution in such a repository similar to that of this game, where branches form parallel time lines, each node represents a commit and the state of a piece of a certain piece of code in that commit is displayed, possibly with further decorations to indicate the concrete changes.

1.3 Methodology

The methodological approach will involve the following steps:

1. **Literature and Tool Research:** Research about the current State of the Art of VCS history visualizations will be performed, both in terms of academic literature and existing tools. Further, a gap analysis, exploring the difference between the State of the Art and the desired goal will be performed.
2. **Conceptual Design:** To answer RQ1, a visualization concept will be developed. The findings of the aforementioned literature and tool research together with the author's personal experiences in software engineering work with VCSs will form the base for this concept. From existing visualization methods, and drawing inspiration from the *5D Chess* [W2] game, the concept will be designed and Mockups will be created to show examples of the envisioned visualization.
3. **Semi-Structured Interviews with Developers:** Semi-structured interviews with experts—in this case developers—will be conducted, to answer RQ2. These interviews will contain questions to rank features on a scale (RQ2a) and open questions, about the expectations and requirements they hold towards the proposed tool (RQ2b). The results be used to prioritize the features for the following prototype development. Before the interviews, a pilot interview will be conducted, to find possible problems with the questionnaire and the way the interviews are executed.
4. **Iterative Prototype Development:** To answer RQ3, a prototype of the proposed visualization will be implemented. The order, in which the features will be implemented, depends on the prioritization obtained by the expert interviews.
5. **Semi-Structured Expert Evaluation:** The prototype, that will have been developed in the previous stage will be presented to developers for evaluation. A demonstration of the features will be given with a real project repository. Then, semi-structured interviews with the participants will be performed, to find whether they that proposed visualization can be beneficially applied in Software Engineering (SE) (RQ3), how they asses the implemented features (RQ2a), and what further expectations they have for future developments (RQ2b). As with the interviews on the concept, a pilot interview will be conducted first, to find potential problems with the questionnaire and the procedure.

1.4 Structure of this Thesis

In the following, the structure of this thesis is summarized.

- **Chapter 2** contains an explanation of the fundamentals of VCSs. An overview of the history of VCSs is provided. Also, different ways to categorize VCSs are discussed. Finally, common terms and concepts are described and compared in different, widely used VCSs.
- **Chapter 3** describes the results of the literature research into the state of the art in the fields of software history visualizations and Mining Software Repositories (MSR).
- In **Chapter 4** the visualization concept is presented. The proposed features are listed in Table 4.1. Furthermore, some considerations for the prototype implementation are discussed.
- The interviews on the concept are described in **Chapter 5**. The plan for the interviews is laid out, followed by the attained results. Furthermore, the results are summarized and interpreted. Finally, identified threats to the validity of the results are listed.
- **Chapter 6** describes the implementation of the prototype. The structure of this chapter follows the iterative nature of the development process.
- In **Chapter 7** the results of the prototype evaluation interviews are discussed in a fashion analogous to Chapter 5.
- In **Chapter 8**, the findings of this thesis with regards to the research questions are listed.
- **Chapter 9** concludes the thesis with a summary of the findings and a prospect for potential future work on the topic.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Fundamentals: Version Control Systems

A Version Control System (VCS) is a tool to manage revisions of a document, typically a piece of software. In this chapter, the history, concepts and terminology of VCSs will be discussed, as well as some differentiating characteristics.

2.1 History

The roots of VCSs can be traced back to tools like the *Librarian* [7]–[9] or *Panvalet* [9]–[11], which were used to manage the source code of programs on the IBM System/360 line of computers. An early VCS in the modern sense was the *Source Code Control System (SCCS)* [12]–[15], which was developed for the MVT variant of OS/360 in 1972 and later ported to Unix. Both Ruparelia [14] and Baudiš [15] name it as the very first VCS. SCCS remained the dominant VCS until it was succeeded by the *Revision Control System (RCS)* [14], [16] in the early 1980s.

With the rise of computer networks, it became possible for development teams to collaborate without working on the same machine. The *Concurrent Versions System (CVS)* was first developed by Dick Grune in the mid 80s for working with his students on the Amsterdam Compiler Kit [14], [17]–[19]. It is a client-server system on top of RCS [14], [18]. After it was released on the internet in 1986 [14], [18]–[20], it became a widely used tool through the 1990s. CVS introduced more new features to VCSs besides the network support, such as branch imports [14]. However it did have its drawbacks, such as poor scalability in large projects and cumbersome management of said branches [14], [21]. Because of this, it was started to be gradually displaced by newer systems by the early 2000s [14], [21]. One example is *Subversion*, which explicitly aimed to be like CVS but address its shortcomings, and inherits its centralized client-server model [14], [W3].

The other trend in VCS technology that started around the year 2000 was the rise of *Distributed Version Control System (DVCS)* [14], [20], [21] (see Paragraph *Distributed Version Control Systems* in Section 2.2.1). When working with a DVCSs, every developer has a full copy of the project repository and there is no real distinction between working copies and a central server repository. One of the earliest DVCSs—if not *the* earliest—was the commercial *BitKeeper* [14], [19], [20], [W4], which was used in the development of the Linux kernel from 2002 to 2005 [14], [W5]. A licensing dispute between the company behind BitKeeper and the Linux kernel developers led to the search for another VCS for use in Linux development [W5], [W6] and sparked the development of two Free and Open Source (FOSS) DVCSs: *Git* [W1], [14], [20], [W5] and *Mercurial* [20], [W7], [W8]. Many other VCSs based on this distributed model had been developed in the early 2000s, such as *Monotone* [15], [W9] and *Darcs* [15], [W10], [22].

Git has since become the most widely used VCS. A survey conducted by the Eclipse Foundation [W11] in 2014 found that roughly a third software developers primarily use Git for version control. A similar yearly survey by Stack Overflow [W12] found that 69.3% of respondents used Git in 2015 [W13], while that number rose to 93.87% in 2022 [W14]. There are a number of web sites that provide hosting services either primarily or solely for Git, such as GitHub [W15], BitBucket [W16], or GitLab.org [W17]. Also, a number of well known FOSS projects have switched to Git from other VCSs [14], for example Perl [W18] (from Perforce) [21], GNU Emacs [W19] (from Bazaar) [W20]–[W22], Python [W23] (from Mercurial) [21], [W24], and FreeBSD [W25] (from Subversion) [W26], [W27].

2.2 Differentiating Characteristics

VCSs can be categorized by a number of aspects of their operation. Among these are the *development model*, the *scope of a version*, and the *concurrency model*. While there are more such characteristics that VCSs can be compared and contrasted by, the following discussion is focused on these three aspects, as they are the most apparent ones to users of the systems and reveal trends in the historic development of VCSs.

2.2.1 By Development Model

The development model describes how developers on a project use the VCS to work on the code and collaborate with each other [15]. The three main development models are the *local* model, the *client-server* model, and the *distributed* model [15].

Local only When working with a purely local VCS, there is one repository residing on one computer where all development is concentrated. Early VCSs like SCCS [12]–[14], [23] and RCS [14], [16], [23] worked this way. This was a logical and viable approach when most organizations had a single large computer that all users were attached to using terminals. When computer networks and personal computers became more widespread in the 1980s, this model became impractical [14], [18], [23].

Client-Server systems The first type of networked VCS that developed to support distributed development teams, was the client-server model [14]. In this model, there was a clear distinction between the central repository on the server and the client working copies of the developers; for this reason, this model is also often called *centralized* [14], although that term is sometimes used to also encompass purely local VCSs [20]. In a client-server VCS, many operations can only be performed when there is a connection to the server; if the repository is not reachable, work can only continue within certain limits. The first such system was CVS [14], [18], [20], [23], [24], while a later example is Subversion [14], [20], [W3], [23], [24], [W28].

Distributed Version Control Systems A different approach to extend the concept of a VCS to enable distributed development teams is the Distributed Version Control System (DVCS). With this kind of VCS, every developer has a full repository in their working copy [14], [20], [23], [25]. To contrast it with the centralized client-server systems, it also sometimes called a *decentralized* VCS [21].

Like in a local VCS—and in contrast to a client-server system—this repository is completely self-sufficient and it is possible to work in it without any interaction with the outside world [14]. However, like a centralized client-server VCS—and in contrast to a purely local one—it *does* support developers when working independently in different repositories [W29]. This way, a DVCS combines the advantages of both local and the client-server VCSs [25]. The programmers can then combine their work by exchanging history information among their own repositories [20]. Of course, it is also possible for a project to have a de facto central repository that all developers pull from and push to, but still enjoy the advantages of the distributed nature of a DVCS when working locally [20].

2.2.2 By Scope of Versions

Another way to categorize VCSs is on what scope history is kept.

Individual files One possibility is that history is kept on the file level. The VCS keeps an individual change log for each file and each recorded version includes changes only for a single file. The early systems SCCS [12], [13] and RCS [16] work this way [14]. The drawback of this model is that it makes it cumbersome to work on projects that consist of multiple files or even directories [14]; for example it cannot properly deal with file renames [15].

Directory tree On the other end of the scale are the VCSs where a single version history is kept for the whole directory tree in the repository. When a new version is created, it includes the state of all files in the project repository; when an older version is restored, all files—and which files are present at all—are restored to the state of that version. Examples of VCSs that use this model are Subversion [W30], Git [W31], [W32] and Mercurial [W33], [W34]. CVS tries to implement this model [W35], but it has some

limitations as a consequence of being based on RCS [18]. One example is the inability to properly deal with renamed files: Because RCS tracks the history of each file in its own history file, and renaming in CVS is usually done by marking the old file as removed and freshly adding the file under its new name, the history of the file before it was renamed is not easily accessible anymore¹ [W36]. While there are other ways to effectively rename a file which do not “lose” its history, they have other downsides, most notably making it impossible to faithfully restore older project versions from the repository [W37], [W38].

2.2.3 By Concurrency Model

The concurrency model describes the facilities the VCS gives to developers to coordinate their work. The two models that have been used in the past are *locking* and *merging*.

Locking With this concurrency model, a contributor needs to lock a part of the project before starting work on it. The VCS then makes sure, that no other developer makes modifications to this part of the repository [12]–[14], [16]. In other words, locking deals with concurrent changes by preventing them in the first place.

In SCCS [12], [13] and RCS [16] locking at the file level was the primary mechanism for coordinating concurrent development, though RCS also has some support for merging [16]. In the centralized VCSs CVS [18] and Subversion [W39] have some support for file locking, but the fact that multiple contributors might start work on a file in their own working copies, without connecting to the central repository in the meantime, makes it unpractical; the Subversion documentation in particular states, that locking of files is supported but discouraged in favor of merging [W39].

In DVCSs, locking makes little sense, as each developer has their own, full repository as working copy, and they might work independently for any amount of time before they integrate their changes back to a central repository again, if such a thing exists at all.

Merging In contrast to preventing concurrent work by locking, *merging* allows developers to make their changes first and combine them later [14], [20]. This way, programmers working on different computers do not need to worry that they might have to discard work because they could not commit it to the project repository before someone else modifies the same portion of the project [W39]. The merge can often happen automatically, because the changes by the different contributors do not overlap directly or because the merge is handled automatically by one out of a number of sophisticated algorithmic strategies [15]. The downside of this is that sometimes the automatic strategies might fail, in which case the programmer that performs the merge has to resolve the conflicts manually [14].

¹It is of course possible to get the history of a file before the rename by manually asking CVS for the history of the old filename, but this also requires knowledge about when file renames have happened. CVS does not help the user deal with this in any way.

Support for merging in VCSs has developed gradually [14]. RCS has some support for merging, but it is limited to individual files [14], [16]. The merging mechanism in CVS was built atop the RCS merge facility, but—like CVS as a whole—it supports merging of whole project versions rather than just singular files [18]. While Subversion has better support for merging [W39], it did not include metadata about merges before version 1.5 [W40], which means that history information about when merges of which branches happened was lost. The DVCSs Git and Mercurial have more advanced support for branching and merging, with their version history based on a directed acyclic graph (DAG) [15], [25].

2.3 Terms and Concepts of Version Control

In the following section, some concepts common to most VCSs and the associated terminology will be discussed.

2.3.1 Versions

A VCS stores versions of something, typically accompanied by a *log message* [16], [W30], [26]. How this representation of a version in the VCS—and the operation to create one—is called, varies from system to system. In Git, both the version object and the operation to create one, are called “commit”. As an illustration for the wide range of terms used, consider the following non-exhaustive list:

- In his article on RCS, Tichy [16] talks about the system’s “checkin” operation (short “ci”), and variously calls the resulting version either “revision” or also “checkin”.
- In Subversion, the version object is called “revision”, while the operation is “commit” [W30].
- In Darcs, a stored version is called a “patch” [W41] and the operation to create a patch is “record” [W42].

In the following, these terms will be used mostly interchangeably.

2.3.2 Version Identifiers

To be able to reference a specific version later again, it needs to have some identifier. Different VCSs employ different schemes for these version identifiers. SCCS [12], [13] and RCS [16] use hierarchical version numbers akin to the kind of version numbers used to describe public releases of software. CVS [18] also inherits this scheme from RCS. By contrast in Subversion, revisions are numbered using a simple, monotonically increasing sequence number [W30].

The DVCS Monotone [W9] is credited as the first VCS to use the hash of the commit as its version identifier, in particular, SHA-1 [15], [W6], [W43]. Git and Mercurial also

employ this scheme [W31], [W44]. This has the advantage that the integrity of a commit can be verified from its identifier [W29], [W31], [W44]. However, Alwis and Sillito [21] note that it also brings a disadvantage: the commit hashes are long, hard to read and write, and do not carry any other meaningful information, such as the order of versions.

2.3.3 Trunk, Branches and Tags

The terms *trunk*, *branch*, and *tag* are related to the development work flow with a VCS. They are used to identify lines of development and mark important version for later reference. An explanation of those terms and a brief overview of their implementation in Git follows.

Branches Many VCSs have support for the concept of a *branch*. Branches provide support for separate lines of development to occur simultaneously. Scenarios for branches vary; examples are:

- Parallel development of independent features, that are later reintegrated back into the main line of development [16]
- Concurrent work of multiple developers, when using a VCS where directly committing to the same development line from different sources is problematic—such as RCS, CVS and Subversion
- Maintenance of multiple releases of the same software

Basic support for multiple branches has existed as early as RCS [16], where they were also called “customer modifications”. In CVS and Subversion, working with branches was still cumbersome in several aspects, necessitating workarounds in the development process, such as a practice called “bunny hopping” [W45] in Subversion. Over time, support for branches has improved, in particular with the rise of DVCSs, such as Git and Mercurial [15], [21].

Trunk The main line of development is sometimes called the *trunk*. In some VCSs the trunk is an integral part of the system, such as in RCS [16], [18] and CVS [18], [W46]. Subversion [W28]—where branches are just directories—does not have a distinguished trunk by design, but the recommended standard repository layout separates the trunk from other branches [W47], [W48]. Again in others, there is no inherent concept of a trunk at all; the main line of development is just another branch, which is named according to convention. Examples are Git’s “master” or “main” [W32], [W49] and Mercurial’s “default” [W50].

Tags A *tag* is a label that can be put on a commit of the project to give it a name. Using such a tag, the designated version can easily be viewed and referred to later. This is typically used to mark a commit that is used for a release.

Though RCS did not support tags as described here, it had the “freeze” command to bring all files in a project to the same version number, thus making it easier to reconstruct this state of the project later [16]. Named tags in the modern sense were then introduced with CVS [17], [18] and have been present in most VCSs since then, such as Subversion [W28], Git [W1], and Mercurial [W7], to name a few examples.

Git References In Git, branches and tags are implemented using *references*. A Git reference is just a name that refers to a certain commit via its commit identifier—that is, its hash. There are several kinds of references in Git:

- *Branch heads* mark the currently last commit of a branch,
- *Remote branch heads* are used to track the state of branches in remote repositories,
- *Tags*,
- *Stashes*, to temporarily pack a set of unfinished changes away from the working directory,
- *Annotated tags*², which are tags with an additional information text,
- *Notes*², to attach note text to commits.

Because branches and tags serve different purposes, the corresponding reference types—branch heads and tags—are treated differently when checked out: Checking out a tag just brings the working directory to the state of the designated version; a checkout of a branch head marks it as the current branch, and committing then also updates this branch head reference.

Notably, there is no designated trunk in Git; all branches are treated equally by the VCS. If the process of a project relies on the existence of a development trunk, a branch must be chosen to serve this purpose. Often, this is the initial branch created at repository initialization, which is usually named “master” or “main” [W49].

2.3.4 Merging

The process of bringing together two or more diverging series of changes is called *merging* [23], [W34], [W51], [W52]. Merges can be done for different reasons. The most obvious case is to simply merge two diverging branches together. In VCSs which follow a model similar to CVS or Subversion, it is not possible to commit one’s changes when a different developer has already committed to the same development line in the meantime; in this case, it is necessary to merge the uncommitted changes with the newly added commits in the system, before it is possible to commit one’s own work [W39].

²Technically, annotated tags and notes are proper Git objects, which store the note text and the identifier of the referenced object, and the reference itself points to these special purpose objects.

Support for merging has generally improved over time. RCS supported merging of different development lines of the same file and CVS also builds its own merge support upon this [16], [18]. These systems, however, do not track when merges have occurred in any way; after the merge, the resulting changes have to be committed normally and to the VCS it is just another ordinary commit. For example in CVS, if some branch *A* were to be merged into trunk, then a new commit added to *A*, and then again merged into trunk, then the VCS would try to merge *all* changes from *A* into trunk—including those that have already been merged [W46]. This results in unnecessary conflicts. To avoid this, the developers have to know the revisions from the branch that have already been merged and manually instruct CVS to merge only the yet unmerged commits [W46].

Developers using Subversion have dealt with similar scenarios with a technique called “bunny hopping” [W45], where a new branch is created for every merge. Newer versions of Subversion make merging of branches easier to handle by the introduction of *merge tracking*, where Subversion stores the range of merged versions in a metadata property [W40].

In DVCSs, the support for merging is generally thought to be better than in the earlier CVS and Subversion [14], [20], [21]. For example, Git and Mercurial remember merges through *merge commits*, which refer to the heads of all branches involved in the merge as parent commits [W32], [W34]. This way, when doing more merges of the same branches in the future, the system knows which commits have already been merged, avoiding the problems described for CVS and Subversion.

2.3.5 Rebasing

An alternative to merging for combining branches which have diverged, is *rebasing*. While a merge creates one new version that combines the changes of both merged branches, a rebase moves the commits of one branch from the common ancestor version to the head of the other branch instead. Figure 2.1 shows an example, where a branch *B* is rebased onto another branch *A*.



Figure 2.1: Example history before (left) and after (right) rebasing of *B* to *A*.

While in the end both merge and rebase combine the changes of two branches into one, the means and effects on the history are different. In particular, in VCSs with a history structured as a DAG, such as Git, the merge preserves the branched history, while the rebase rewrites part of the history to create a new, linear one.

Because a rebase rewrites history and does not leave neither some kind of mark nor the original version of the history in the history graph, it cannot be seen directly in a visualization of the VCS repository history. As such, the concept of the rebase has no bearing on the topic of this thesis and is not discussed further.

2.3.6 Diffs and Patches

A *diff* is a representation of the differences between two files or versions. The shorthand “diff” comes from the name of the Unix program `diff`, which provides a compact, textual listing of the line-based differences between two files [27], [28]. A variant is the *3-way diff*; the differences of two files to a third file, often used for comparing two different versions with respect to a common ancestor revision [16], [18].

Showing Changes One use of diffs is the presentation of the differences in a human-readable form. The output shown to the user can be a textual description of the changes, directly as produced by `diff` or similar programs, possibly with the addition of color for easier reading. This method is used, for example, by `git diff` [W53]. Alternatively, the differences can be prepared in a more graphical manner, for example by listing both versions side-by-side and indicating the changes between them using colored curves. For instance, the IDE *IntelliJ IDEA* [W54] provides such a view (Figure 2.2).

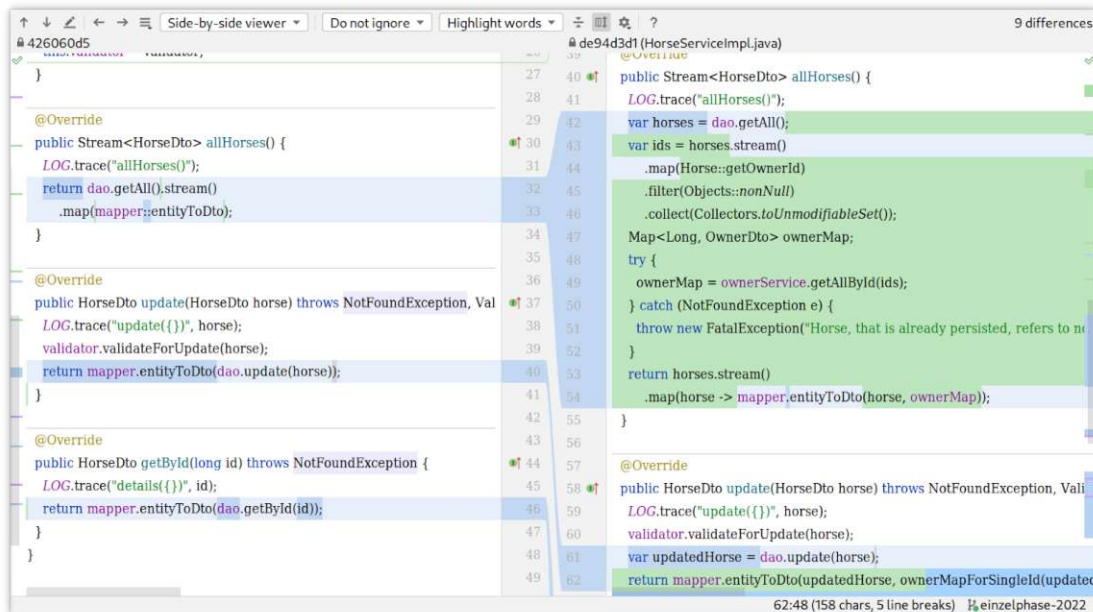


Figure 2.2: IntelliJ [W54] graphically presents a diff. Changed code blocks are marked blue.

Patches Another use case for diffs is as *patches*. A patch is used to bring a file—or a set thereof—from one version to another [18]. Berliner [18] describes CVS’s `cvs patch` command to produce patches for a whole project, which are then to be applied using the `patch` program by Wall [18], [W55]. Similarly, Git makes it possible to export commits as patches using `git format-patch`, which can be reapplied in another repository by `git apply` or `git am` [W56].

Implementation of Merges 3-Way diffs have seen successful application in the context of VCSs as the basis for merges. This has been done since at least RCS [16] and CVS [18], which used the diff3 program for this purpose. As a more modern example, Git still uses the 3-way diff for some of its automatic merging strategies, including the default one [W53].

State of the Art

This chapter gives a brief overview of the State of the Art in the fields of research surrounding the topic of this thesis. First, the wider field of *Mining Software Repositories (MSR)* will be touched briefly, then developments in the more narrow area of visualizations around code history in VCSs will be presented. The chapter concludes with a brief discussion of what sets the proposed concept apart from existing visualization tools and methods.

3.1 Mining Software Repositories

Mining Software Repositories (MSR) is a field of research that aims to extract useful data from software repositories [29], [30]. The extracted data takes on many forms such as information about the evolution of the developed software; statistics; or connections between different aspects, such as commits, issue tracker tickets, and documentation. The resulting data can be used on its own or serve as the basis for a visualization.

In the following, a few examples of recent developments in the field are listed:

Git Branch Analyzer [3] is a MSR tool that reconstructs the workflows of the developers from the commit history. It considers a unit of workflow to be the linear series of commits between a branch and a merge point. It also provides a visual overview over the number of concurrent workflows at any given point in time by showing a graph of the number of branches and merges. Lee, Seo, and Seo [3] noted a disregard for branches in existing repository analysis tools, while they also acknowledged that branching and merging are central tools for concurrent work in FOSS development. In response, they developed the Git Branch Analyzer.

SOFAS Ghezzi and Gall [31]–[34] have proposed *Software Analysis as a Service* [31], [32], a move away from standalone tools for MSR and towards a distributed software analysis

platform. The purpose of this platform is to make individual tools accessible across the internet as services and to make them more easily composable. As an implementation of this concept, they present the RESTful web service *SOFAS* [33]. Furthermore, they show how *SOFAS* can be used to replicate existing mining studies [34].

SOFAS has been applied to the code bases several well-known open source projects [31], including Eclipse [W57], *jUnit* [W58], and projects of the Apache Software Foundation [W59], like *Subversion* [W28].

Unified data source for software analysis Müller, Mahler, Hunger, *et al.* [35] propose a framework based on open source components, for analyzing software several aspects of a software project, including its history. The described framework works in three steps: *data acquisition*, *analysis*, and *visualization*. The data acquisition stage gathers data from different sources—including the project’s VCS—and puts into a graph database, which then acts as the unified data source for the analysis stage. Finally, the results are then shown to the user through a system of views. Using this software stack, Müller, Mahler, Hunger, *et al.* [35] present a proof of concept that analyses and visualizes test coverage data, dependencies between components, static code analysis and code change hotspots.

Candoia Tiwari, Upadhyaya, Nguyen, *et al.* [5] propose a platform for portable MSR tools in the form of “apps”. The goals of this platform are to make building new and adapting existing MSR tools easier, as well as to making distribution and adoption of such tools simpler by acting as an app store. As an example for such a platform Tiwari, Upadhyaya, Nguyen, *et al.* present *Candoia*. In this system, MSR tools are divided into a view, which is built using ordinary web frontend technologies (Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript), and the mining logic, which is written in Domain Specific Language (DSL) based on *Boa* [36]–[38], an earlier language for that purpose.

History Slicing [39], [40] is a technique for querying and exploring software history, developed by Servant and Jones. They define a history slice of a given set of code lines as the set of past commits, in which any of those lines were changed, reduced to the corresponding revisions of said lines; changes in the gathered commits which are unrelated to the selected code lines are not considered part of the history slice. The purpose of history slicing is to reduce the commit graph to a subset which only includes commits and changes relevant to a selected part of the code. The objective is to make the history of the project easier to query and understand.

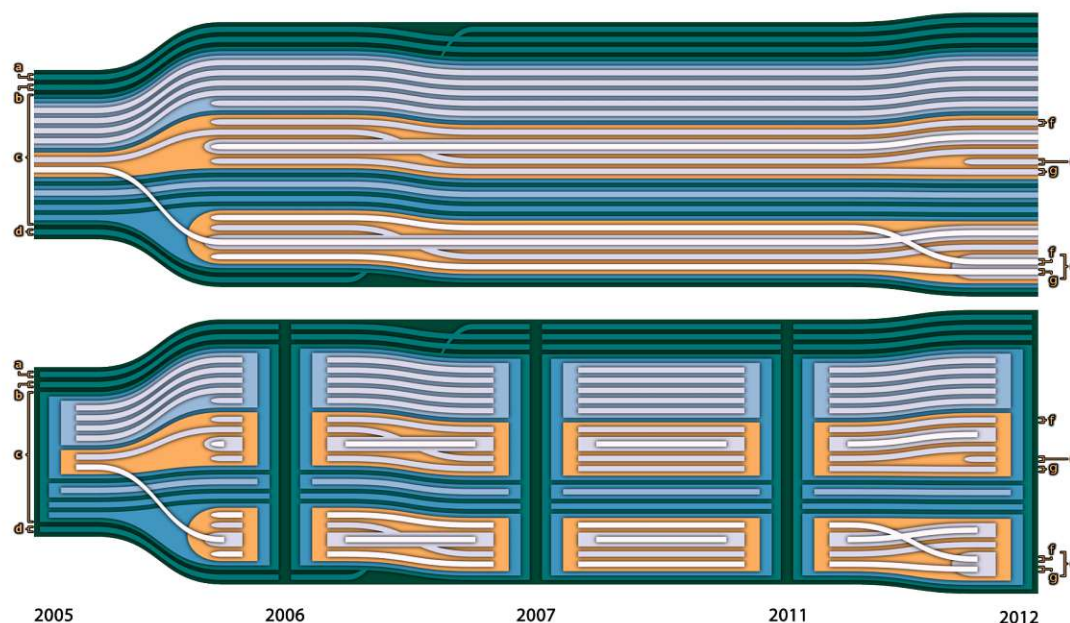


Figure 3.1: Example of SplitStreams [41] visualization

3.2 Existing History Visualization Tools and Methods

There are several visualization methods for evolution of source code and existing tools for analyzing VCS repositories, some of which utilize such visualization methods.

Following is a non-exhaustive list of examples.

SplitStreams [41] is a method for visualizing changes in hierarchically structured data, such as program source code. It utilizes so called “streams” to show creation, deletion, movement of nodes and other changes in the abstract tree structure over time.

Chronos [42] displays a linear timeline, on which the times of commits are marked and connected to a view of a file in that commit. Changes in these commits are highlighted by a different text color, but they are not otherwise visually related to their predecessors and successors. This view can be panned to navigate along the timeline as well as zoomed, to select between more details or a more rough overview. When hovering of a change, Chronos displays metadata associated with this change. A sample screenshot of Chronos can be seen in Figure 3.2.

Code Time Machine [43] is a plugin for the IntelliJ IDEA [W54] Java IDE. Inspired by Apple’s Time Machine, it emphasizes temporal navigation and displays the commit history of a file as a stack of cards, where each card shows the state of the file in one

3. STATE OF THE ART

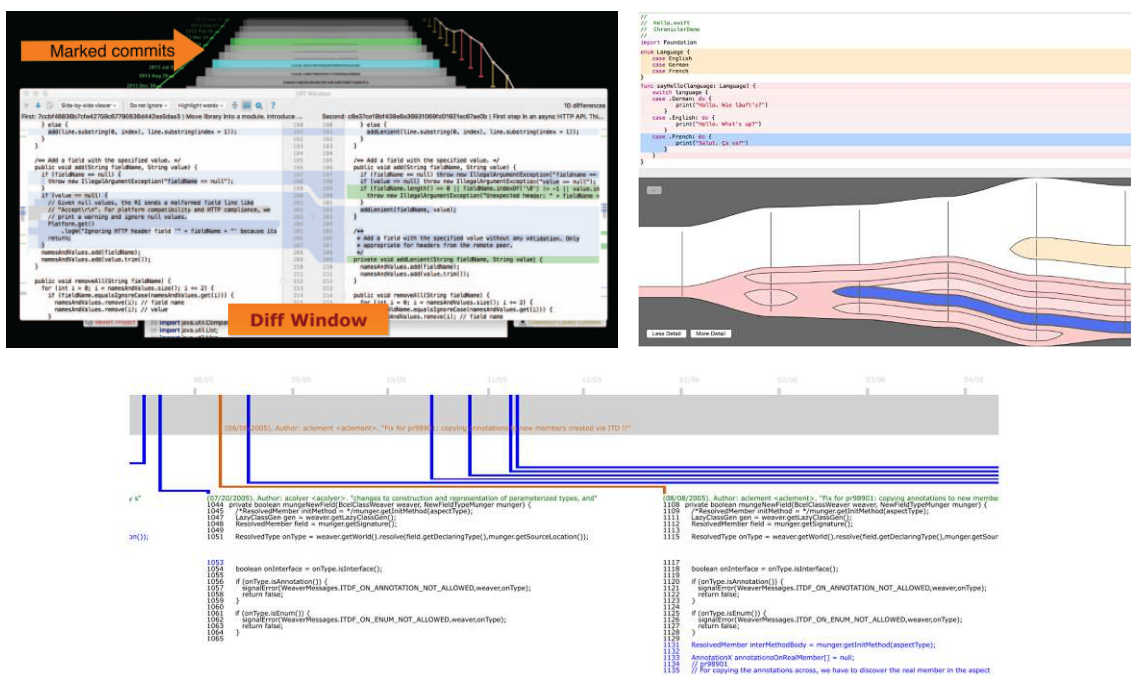


Figure 3.2: Partial sample screenshots of Code Time Machine (top left) [43], Chronicer (top right) [44] and Chronos (bottom) [42].

commit, with timelines on the sides showing commit dates and change numbers of each commit. A sample screenshot of Code Time Machine can be seen in Figure 3.2.

Chronicer [44] aims to give an overview of the structural changes in a file over time. Wittenhagen, Cherek, and Borchers present *Tree Flow*, a change visualization similar to SplitStreams. Chronicer combines this with a view of one selected version of the source code. While earlier deletions or later introductions of structural elements relative to the currently selected commits are visible in the Tree Flow, it is not possible to see what these code pieces are without changing the selected commit. A sample screenshot of Chronicer can be seen in Figure 3.2.

TypeV [45] is a tool for analyzing the histories specifically of Java projects. It utilizes an Abstract Syntax Tree (AST) differencing algorithm—instead of a line-based diff—to extract more domain knowledge, such as usage of methods and changes in types, out of the source code. While it does handle branches to a degree during the diffing of the commits, the employed visualizations present only a linear timeline. An example for the main view is given in Figure 3.3.

VisGi [4] visualizes the commit graph of a given Git repository as a graph, where each node represents a *group*—a linear sequence of commits without branching points—the

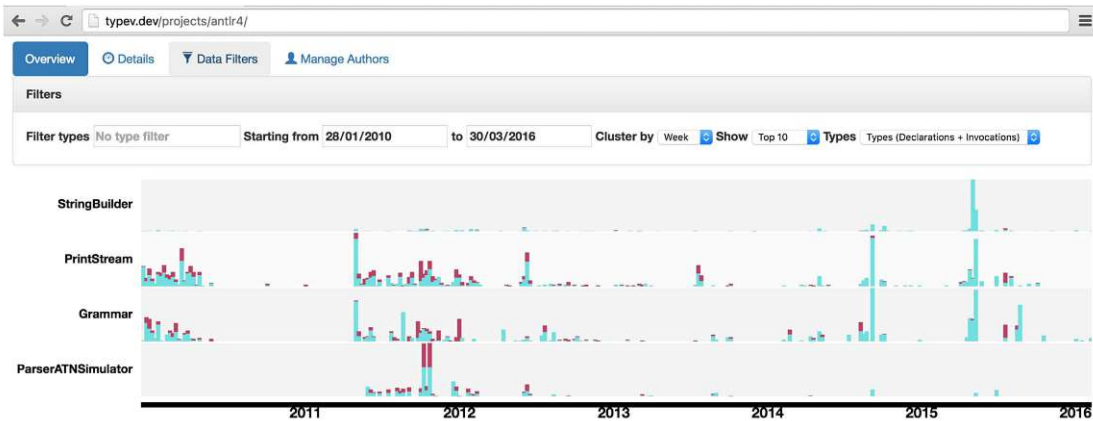


Figure 3.3: Main view of TypeV [45] with a long term change overview.

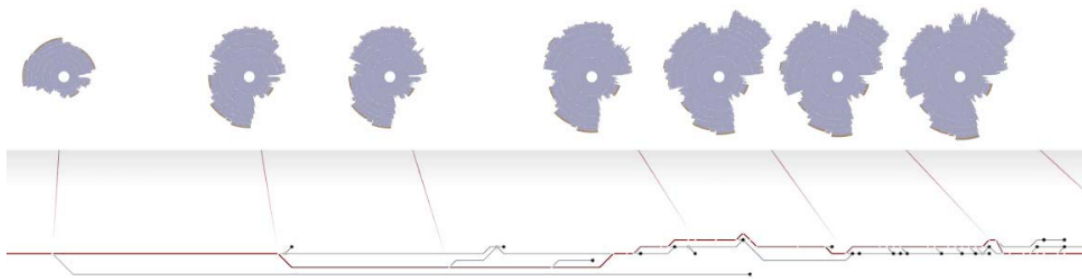


Figure 3.4: History graph of VisGi [4] with sunburst diagrams of the file and directory structure at the given commits.

nodes are placed in a layout based on their commit time and branches are visibly marked. In addition, the directory tree of the repository at a certain commit can be viewed as a sunburst diagram. Such a diagram can also be displayed for every group along a selected path in the history, as seen in Figure 3.4. This makes it possible to get a rough overview of the evolution of the coarse structure of source organization in the project. Elsen [4] notes, that this does not allow to gain much insight into the inner structure of the project, while visualizations of the file contents might be more useful.

GitHub Desktop [W60] is a general tool for working with Git repositories. In terms of history visualization, it provides a way to list the commits of branches and view the changes in those commits as a simple diff. While this enables detailed insight into individual commits, it does not provide a bigger picture of longer commit sequences.

History graph Views showing the history graph of a repository are an established part of the tool sets surrounding VCSs. BitKeeper [W4], an early DVCS, includes the command `bk revtool` to view the history graph of the project [W61]. More modern tools typically display the history graph vertically, with one commit per line, including the

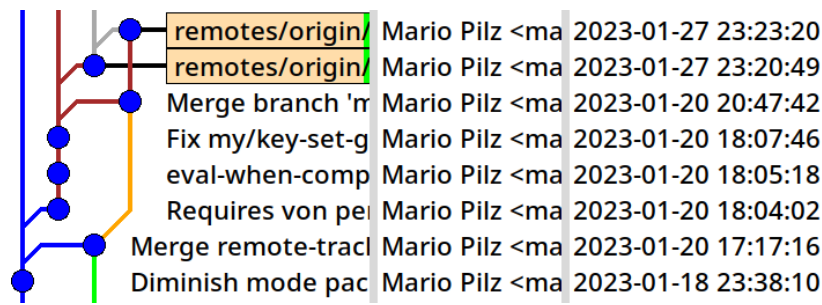


Figure 3.5: Git history graph view, as provided by Gitk [W62].

commit message, author and date. Git [W1] includes *Gitk* [W62] to display the graph graphically in this manner, and `git log --graph` [W63] to do the same in a textual terminal. External tools that provide such a visual representation of the history also exist, such as *GitKraken* [W64] or the Git integration in the IDE *IntelliJ IDEA* [W54]. Comparable tools also exist for Mercurial [W7], such as *HgK* [W65]. An example for such a history graph view is provided as a screenshot of Gitk in Figure 3.5.

3.3 Distinction from Existing Research and Tools

The literature review conducted as part of this thesis revealed that all existing methods and tools for visualization of code history have one or more of the following shortcomings:

- They ignore branches, focusing solely on linear history
- They do not visualize longer sequences of changes, either showing only the differences between two commits, or generally not connecting the changes in successive commits, displaying each one isolated
- They do show evolution over a longer term, but only of the rough structure, without showing the concrete changes.

Visualizations of a file’s change history generally fall into one of two categories: On the one hand, there are visualizations that show changes between different versions but use only a linear timeline. One example is *Chronos* [42]. Similarly, while *Code Time Machine* [43] shows the differences between two versions in more detail than *Chronos*, but not between multiple versions. *Chronicler* [44] shows one concrete revision of a file and indicates which parts have changed over time on a structural basis, but does also not account for branches.

On the other hand, visualizations of the commit graph are restricted to a view where the commits are displayed as small markers, sometimes also showing the version identifier and commit message; the changes are not usually presented as part of this view. An example in research is *VisGi* [4], which shows a plain graph of commits; while it presents

the history of the project with branches and all branching and merging relations intact, the only displayed information of individual revisions is a sunburst diagram of the file and directory structure (Figure 3.4). Many tools used in practice show the history graph of a Git repository vertically with one commit per line, usually accompanied by general information, like the commit message, author and hash. This includes Gitk [W62] and GitKraken [W64]. An example screenshot of Gitk is provided in Figure 3.5.

The default installation of Git includes the command `git log --graph --patch`, which renders the history graph as a character-oriented “picture” in the terminal. In contrast to the aforementioned tools, this also displays the changes made in that commit as a textual diff. While this gives more insight into the historic development of a piece of code, the presentation does not provide a good overview.

In contrast to the listed methods and tools, the aim of this thesis is to develop a concept and prototype of a visualization which:

- shows the whole commit graph with the branch structure intact,
- shows the contents of a file at each version,
- indicates the differences between related versions.

3. STATE OF THE ART

```
* | commit 3d560d64047c99312daa4f10162d4ba0a8181648
  | Author: Mario Pilz
  | Date: Wed Feb 8 23:53:31 2023 +0100
  |
  | Backticks und Forwardticks enclose keys.
  |
  | diff --git a/elisp/init/keys.el b/elisp/init/keys.el
  | index 63a8161..ef46fc2 100644
  | --- a/elisp/init/keys.el
  | +++ b/elisp/init/keys.el
  | @@ -55,5 +55,7 @@
  | (my/key-set-global-except-minibuffer-read-char (kbd ",") (my/make-enclose-command enclose-gdquotes ?" ?"))
  | (my/key-set-global-except-minibuffer-read-char (kbd "\"" (my/make-enclose-command enclose-esquotes ?' ?'))
  | (my/key-set-global-except-minibuffer-read-char (kbd "\"") (my/make-enclose-command enclose-edquotes ?" ?"))
  | +(my/key-set-global-except-minibuffer-read-char (kbd "`" (my/make-enclose-command enclose-backticks ?` ?`))
  | +(my/key-set-global-except-minibuffer-read-char (kbd "`") (my/make-enclose-command enclose-forwardticks ?' ?`))
  |
  | (provide 'init/keys)
  |
  | * | commit 76085aa4cabe6570592d992b0001e9b218101bff
  | | Merge: d74900a b67e77a
  | | Author: Mario Pilz
  | | Date: Wed Feb 8 23:55:06 2023 +0100
  | |
  | | On veldin: merge
  | |
  | | * | commit b67e77aadfdc072c558af5c93d5fb72d9bd7f943
  | | / | Author: Mario Pilz
  | | / | Date: Wed Feb 8 23:55:06 2023 +0100
  | |
  | | index on veldin: d74900a Glasses-Separator = .
  | |
  | | * | commit d74900a9802fc530c2fbf06d5de83eeba0dd9d4b
  | | | Author: Mario Pilz
  | | | Date: Wed Feb 8 23:54:39 2023 +0100
  | | |
  | | | Glasses-Separator = .
  | | |
  | | | diff --git a/custom-vars.el b/custom-vars.el
  | | | index 0bac749..26cc975 100644
  | | | --- a/custom-vars.el
  | | | +++ b/custom-vars.el
  | | | @@ -18,6 +18,7 @@
  | | | '(delete-old-versions t)
  | | | '(display-line-numbers t)
  | | | '(enable-recursive-minibuffers t)
  | | | + '(glasses-separator ".")
  | | | '(global-hl-line-mode t)
  | | | '(indent-tabs-mode t)
  | | | '(inhibit-startup-screen t)
```

Figure 3.6: History graph with patches as shown by `git log --graph --patch`.

Conceptual Design

In this chapter, the concept for the visualization, which was developed as part of this thesis, is described.

4.1 Introduction

In the past, tools to visualize the history of source code concentrated firmly on linear history. For example, the *Code Time Machine* [43] displays the different versions of a file as a linear stack of cards, while the visualization technique *Code Flows* [46] shows the changes of a piece of code over time schematically, but only along a linear timeline.

This is not limited to visualizations: As recent as 2018, Kovalenko, Palomba, and Bacchelli [2] noted, that there is a lack of knowledge about the impact of branching and merging on Mining Software Repositories (MSR). Research about how to deal with non-linear, branched history, as it commonly occurs when working with modern DVCSs, is clearly underrepresented—both in terms of visualization as well as MSR tools.

Also, differences are mostly only shown between one version and its parent, such as in *Chronos* [42], or between two selected versions, such as in [43]. While *Chronicler* [44] utilizes the Code Flows [46] visualization technique to show differences continuously over a longer sequence of commits, it only displays the actual source code text for one of those versions, making the historical overview rather vague and abstract.

In the following, inspired by the way non-linear time is displayed in the game *5D Chess with Multiverse Time Travel* [W2], the concept for a visualization that displays the history of a single file and deals with branched history by displaying it as a graph will be presented. First an overview of the core structure of the visualization will be given, then features and requirements for them will be discussed in more detail. Table 4.1 gives an overview of the features and assigns them short identifiers by which they will be referred to throughout the rest of this thesis.

ID	Feature Description
F.01	View of the whole commit graph in the context of a file
F.02	View of multiple branches (not necessarily all of them) side by side
F.03	Simple indication of changed code pieces between related commits
F.04	Indicate long term structural code change between commits (Tree Flows [44])
F.05	Focusing individual select branches by fading or hiding others
F.06	Folding linear parts of the history, that contain no branching points
F.07	Zooming in, to see a few related commits in more detail
F.08	Zooming out, to get more overview of the historic development with less detail
F.09	Direct comparison of two commits by selecting them
F.10	Direct comparison of two commits with regards to a common base commit (3-way diff)
F.11	Synchronized scrolling of text in all commits
F.12	Selecting a block of code in one commit and highlighting the corresponding block in another commit
F.13	Selecting a block of code in one commit and highlighting the corresponding block over the whole history

Table 4.1: Proposed features for the visualization

4.2 Overview

As mentioned before, inspiration was taken from the game *5D Chess with Multiverse Time Travel* [W2]. In this game, alternate, parallel timelines of the same game of Chess are shown side-by-side, with the full board shown at each point in logical time. It is in principle a directed acyclic graph (DAG), where a node represents a point in logical time, and when a node has multiple successors, multiple different moves have been made from the same point in time, splitting the timeline.

This is similar to how history in DVCSs, such as Git, works. When one commit has multiple direct children, the history is branched, forming multiple parallel timelines from that point on. In contrast, commits with multiple parents merge two or more branches, reuniting multiple parallel timelines into one again. Therefore, the idea behind the visualization of the non-linear history used in the game is also applicable to branched history in a VCS.

As already mentioned, the proposed visualization is about the history of a single file. It utilizes a graph, where each node corresponds to a commit in the history of the VCS repository, to deal with the fact, that the history potentially has branches and is therefore not necessarily linear. The graph is oriented from left to right, where nodes on the left side represent older commits whereas nodes on the right represent newer commits. This ordering is strictly from a logical point of view; a commit that is the ancestor of another is always more to the left than its descendant, while this is not guaranteed for the commit time of any two commits, if they are not directly related.

The node is graphically represented by a text block, that displays the state of the file in question at that commit. The edges of the graph are formed by planes connecting related commits, that is, where one commit is the parent of the other. On these planes, the differences between the file in the two involved commits are indicated by colored areas, that connect a piece of text in the parent commit with the corresponding piece in the child commit. Since this does not only compare two versions but instead shows the differences between all related versions, that are currently shown, the view can be considered a continuous diff over the the whole commit graph. These *difference indicators* are discussed in more detail later.

Showing the contents of source code file typically requires more height in comparison to a quadratic chess board. Also, multiple difference indicators at branching at merging points would need to be laid on top of each other and therefore become unrecognizable. Arranging this graph solely on a flat plane thus would likely lead to an impractical result. Therefore, the layout is done in three dimensions instead, with the commit history being layout out on a plane, while the source code text and corresponding difference indicators are visualized perpendicular to that in the third dimension. Marcus, Feng, and Maletic [47] did a similar shift from two to three dimensions to better utilize the available space in their visualization of large software systems.

Figure 4.1 shows a sketch of the described structure on the example of a short history with three branches, which are all diverged from the same base commit, and two branches are merged in the end.

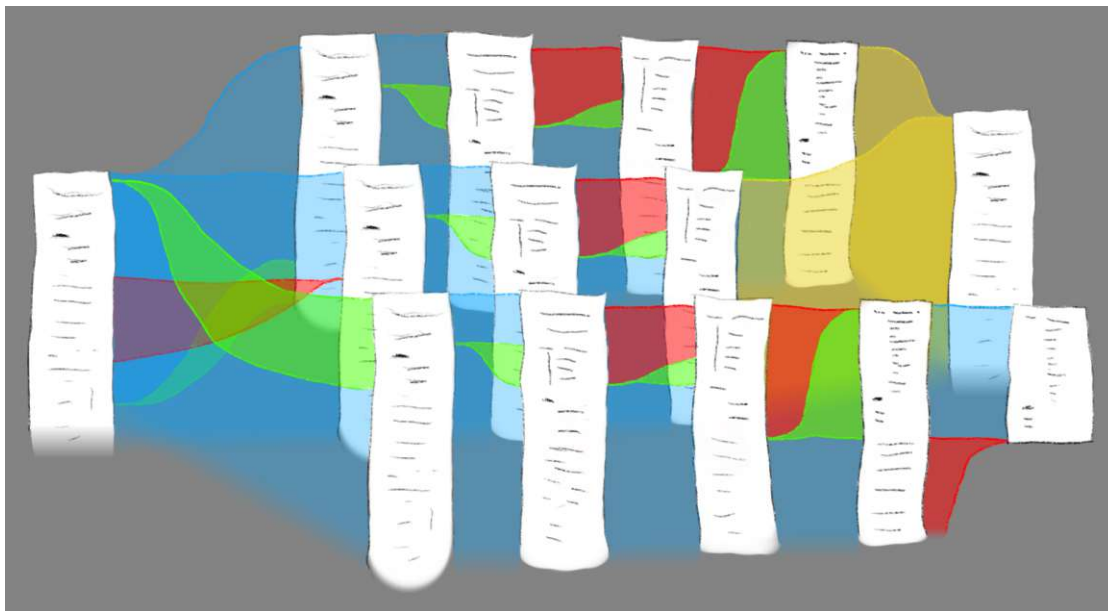


Figure 4.1: Overview of the state of a file in several commits across multiple branches

4.2.1 Zooming

A developer, who tries to get an understanding of the history of code, requires different levels of details at different times. Sometimes a coarse overview over a long time is required, while other times, more detail is required, focusing on short term history instead.

To help in this process, it should be possible for the user to zoom in (F.07), providing a better view onto small portion of the history, as well as to zoom out (F.08), where a bigger part of the history can be displayed while details are harder to identify. This is comparable to how a map can be zoomed to get either more information about a small area or to fit a larger region on the screen at the expense of the detail level or the zooming of a web page by resizing its text, where a smaller text size fits more text on the screen at once while larger text is easier to read.

Similar to the described examples, zooming is done by changing all sizes of the elements in the history graph. Zooming out, which shrinks the dimensions, makes it possible to see more history elements at the same time—both in linear time along one branch as well as across multiple parallel branches—but details in the changes become harder to see. In contrast, zooming in to increase the size of the individual elements provides a higher level of detail but reduces the view to a smaller fragment of the file’s history.

The size of the file’s text—and its readability—is particularly affected by zooming. A very near zoom level makes the text size comparable to how the developer would see it when editing the text in their usual development environment—where the focus is on exactly one version of the file: the current one. In the other direction, zooming out reduces the text size, making it increasingly harder to read, ultimately devolving to small, unreadable dots and lines, with only a schematic view of the code, similar to the minimap feature of VS Code [W66], [W67].

4.2.2 Focusing on Individual Branches

When working with many branches, it might become hard for the user to keep an overview of the part of the history, that they are interested in. In particular, when only a few branches are of interest—or even one—all other branches, that are parallel to these, might become an annoying distraction. Navigation becomes cumbersome, when the branches in question are displayed far apart and the other branches might even block the view on commits, that the developer wants to inspect.

To counteract this, the visualization should aid in keeping the focus on a few selected branches by providing a way to highlight this selected set of branches (F.05). The other branches, that the developer is currently not interested in, are either faded, letting the user see through them, or completely hidden. In the latter case, the branches that are still displayed can then be laid out in a more compact way, as if they were the only ones present in this part of the history.

Figure 4.2 shows an example based on Figure 4.1, with the middle branch focused and the branches in the front and back faded.

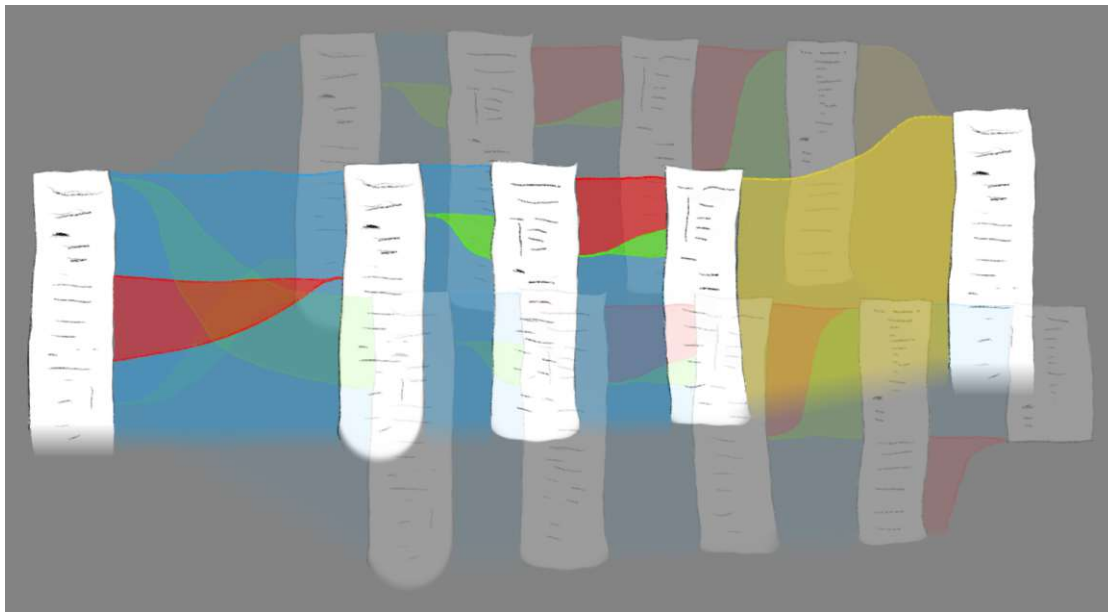


Figure 4.2: Overview with one branch highlighted. The other branches are faded.

4.2.3 Highlighting of Selected Code Pieces

For developers, it can be hard to find the information they require in the history of a larger project. Servant and Jones [39] name information overload as major challenge, that software developers have to deal with today. They propose *History Slicing* as an approach to reduce the amount of information that has to be dealt with, when only the changes of a certain set of lines of code should be shown, to aid in understanding the program and its change history [39], [40]. The idea behind History Slicing can be summarized as showing how a selected set of lines has changed over a selected part of the history.

This method of isolating changes to a small portion of the code can be used to help the developer focus on a specific code block when working with the visualization, that is developed as part of this thesis. The user of the visualization should be able to select a piece of code in one commit and the corresponding lines of code in other commits should then be highlighted, either in one specific commit selected by the user (F.12), or in all commits over the whole history (F.13).

4.2.4 Collapsing of Linear Commit Sequences

Sometimes, the most interesting aspect of the project's history is the branching structure itself. For example, in a typical Git workflow, individual features are implemented in their own branches, which are then merged back to a common development trunk. In such situations, individual commits become less interesting and longer sequential sequences,

where no branching and merging occurs might be distracting for the user by taking away screen space that could be filled with more interesting content.

Lee, Seo, and Seo [3] use this idea to simplify Git history graphs for the purposes of analysis, by reducing them from commits to “workflows”. They consider a purely linear series commits, which have no branch or merge points, to be a single workflow and treat it as a unit for the purposes of building a simplified history graph.

Similarly, Greil [48] provides a *compact view* of the history graph in the context of a tool for increasing the awareness of potential merge conflicts. In this view mode, linear commit sequences are collapsed into *virtual nodes*, which are visually distinguished from ordinary commit nodes. Elsen [4] also uses this approach in *VisGi*, calling the collapsed sequences *groups*. Such a collapsing of node groups into virtual nodes was first proposed for graph drawing in general by Slade [49].

This approach can also be applied to the visualization that is developed in this thesis. Instead of displaying the full commit graph, including file contents, the user should have the option to make the view of the graph more compact. In that case, linear sequences would be shown in a more compact fashion, by only showing the first and the last commit in the sequence explicitly, while the intermediate ones are collapsed into a block. This block would inform the user of the number of hidden commits (F.06).

Figure 4.3 shows an example of this, based on the commit graph seen in Figure 4.1, with three branches, where the leftmost commit is a branching point and the rightmost is a merge, with the commits between them forming linear sequences. In this graph, on the branches in the front and in the back, two and three commits respectively have been hidden. Note that, on the middle branch, nothing has been collapsed in this way, since this branch consists of only three commits in a linear sequence: the first one, the last one, and a single intermediate commit. Displaying this single intermediate commit as collapsed would not result in any spatial gain and therefore should not be done.

4.2.5 Difference Indicators

Between the code blocks that represent commits, the differences between the two versions should be indicated. There are multiple ways, how this can be visualized. In the following, two possible methods will be described: a basic, purely local change indication, that marks points where additions and deletions happened, or the text remained unchanged (F.03), and the more advanced, structured and long-term oriented *Code Flows* (F.04). Both utilize colored curves to mark the positions where the changes happened and that connect the corresponding positions in the two versions.

Basic Difference Indicators

A simple way to indicate differences between two versions shown side-by-side of a text file is to simply show where new text was added, existing text deleted, and where nothing

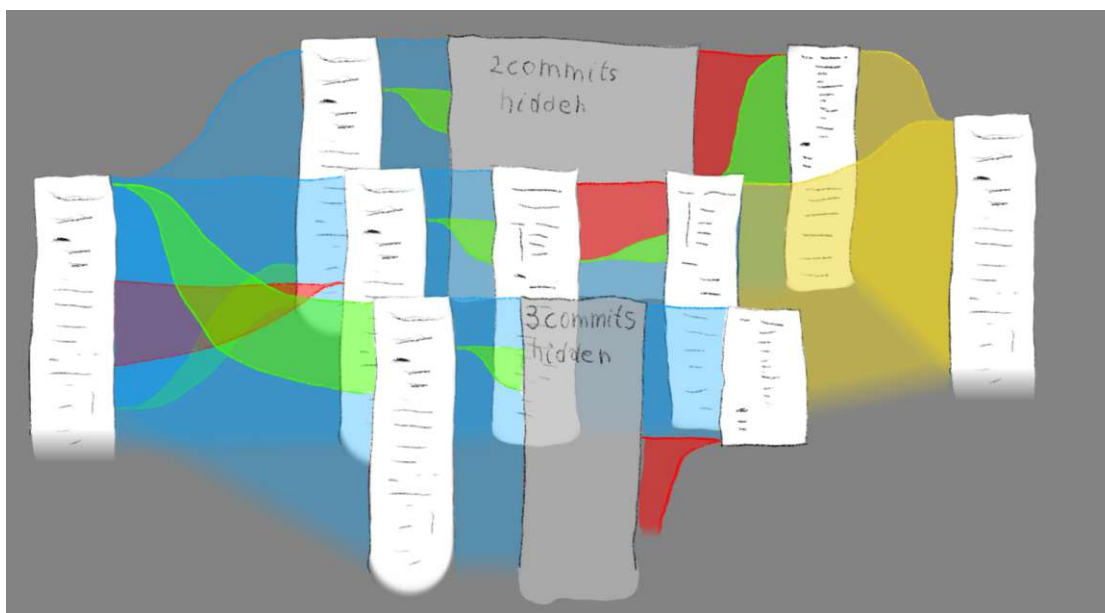


Figure 4.3: Linear sequences of commits are collapsed.

changed. Typically, additions are indicated with green and deletions with red markers (F.03).

This approach is widely used, appearing in many tools that provide a side-by-side diff view, such as *Kompare* [W68]—as can be seen in Figure 4.4—and *GitHub Desktop* [W60], although not all of them draw connecting curves between the two sides, of which GitHub Desktop is also an example.

These indicators can be generated directly from the traditional, line-based diffs, that VCSs like Git typically provide when asked for the differences between two versions. In this case, there are no lines that are *changed*; only additions and deletions exist.

In some cases, a slightly more sophisticated scheme is used, where the tool tries to detect that an existing line or block has only been changed rather than something deleted and something new added. IntelliJ [W54] is an example of such a tool, where besides green additions and red deletions text that was only slightly changed is indicated in blue, with individual, changed words highlighted, as can be seen in Figure 2.2. Autexier [50] presents a similarity-based diff algorithm, that can be used for the underlying difference analysis.

Code Flows Style Indicators

A more advanced scheme to the one described in the previous section is *Code Flows* (F.04). Telea and Auber [46] developed this visual metaphor by improving upon an earlier visualization [51] that was built by analyzing the Abstract Syntax Tree of a C++ program. It represents different pieces of the source text with “tubes” of different colors and shows how these pieces have changed over time. It does this in a structured way.

```

33  this.dao = dao;
34  this.mapper = mapper;
35  this.validator = validator;
36  this.ownerService = ownerService;
37  }
38
39  @Override
40  public List<Horse> allHorses() {
41    LOG.trace("allHorses()");
42    return dao.getAll();
43  }
44
45  // TODO TEMPLATE EXCLUDE BEGIN
46  @Override
47  public HorseDto create(HorseCreateDto horse) throws Validation
48    LOG.trace("create({})", horse);
49    // TODO validation
50    return mapper.entityToDto(dao.create(horse));
51  }
52  // TODO TEMPLATE EXCLUDE END
53
54  @Override
55  public HorseDto update(HorseDto horse) throws NotFoundException;
56    LOG.trace("update({})", horse);
57    validator.validateForUpdate(horse);
58    return mapper.entityToDto(dao.update(horse));
59  }
60
61  @Override
62  public HorseDetailDto details(long id) throws NotFoundException;
63    LOG.trace("details({})", id);
64    Horse horse = dao.getById(id);
36  this.ownerService = ownerService;
37  }
38
39  @Override
40  public Stream<HorseDto> allHorses() {
41    LOG.trace("allHorses()");
42    var horses = dao.getAll();
43    var ids = horses.stream()
44      .map(Horse::getOwnerId)
45      .filter(Objects::nonNull)
46      .collect(Collectors.toUnmodifiableSet());
47    Map<Long, OwnerDto> ownerMap;
48    try {
49      ownerMap = ownerService.getAllById(ids);
50    } catch (NotFoundException e) {
51      throw new FatalException("Horse, that is already persisted");
52    }
53    return horses.stream()
54      .map(horse -> mapper.entityToDto(horse, ownerMap));
55  }
56
57  @Override
58  public HorseDto update(HorseDto horse) throws NotFoundException;
59    LOG.trace("update({})", horse);
60    validator.validateForUpdate(horse);
61    var updatedHorse = dao.update(horse);
62    return mapper.entityToDto(updatedHorse, ownerMapForSingleId);
63  }
64
65  @Override
66  public HorseDto getById(long id) throws NotFoundException {
67    LOG.trace("details({})", id);

```

Figure 4.4: Diff view with additions and deletions in Kompare

When in in a logical block in the source file—such as a class or a loop block—a new child element appears, then this is represented in the visualization by a new flow appearing inside another. Also, when a code block is moved from one part of the file to another between versions, it is shown with the flow connecting the old and the new position of the block, crossing flows that represent other blocks between those position. An illustration of the process as given by Telea and Auber [46] is presented in Figure 4.5.

Similar visualizations are presented by Wittenhagen, Cherek, and Borchers [44] with *Tree Flows* as part of *Chronicler* [44] (Figure 4.6) and later by Bolte, Nourani, Ragan, *et al.* [41], who developed an even more sophisticated scheme called *SplitStreams* to handle more complex hierarchical changes (Figure 4.7).

Although these methods focus on showing the changes in a piece of code over longer time in a structured way mostly or entirely without showing the actual text, they can certainly be adapted for the task, by dividing them into the segments between the versions and using those as the difference indicators between the two corresponding versions of the text of the file.

4.2.6 Synchronized Scrolling

As with any software that displays text, there is only limited screen space to do so and it is very likely, that files will be long enough so that do not fully fill on the screen. In ordinary text editors or document viewers, the obvious solution is to be able to scroll up and down in the document.

The naive way to transfer this concept over to the graph visualization would be to simply display the files in their full length in their text blocks and pan the view as a whole. However, this presents a few complications. Due to the three-dimensional nature of

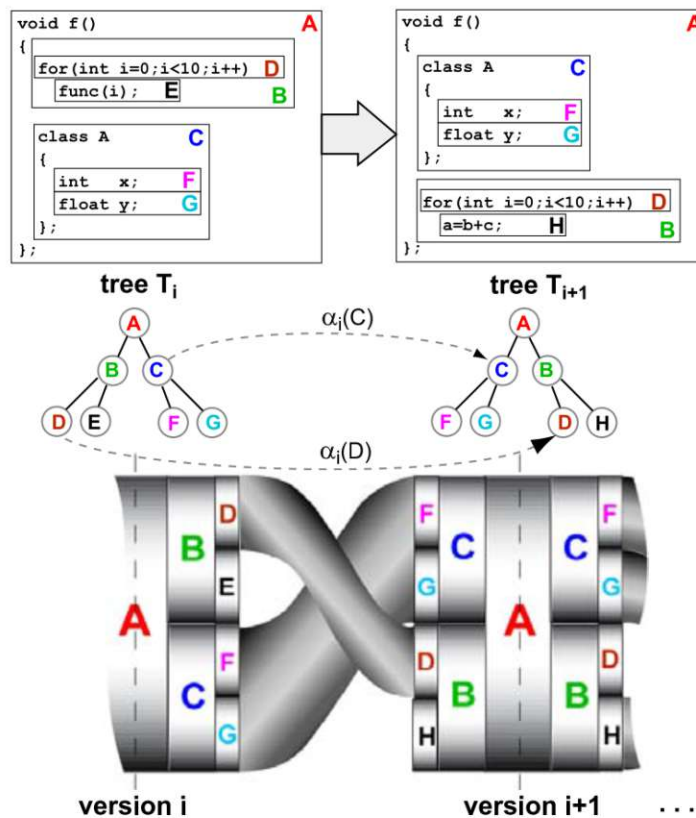


Figure 4.5: How a Code Flows difference visualization represents the difference between two versions based on their AST [46].

the visualization, the text blocks representing the commits, overlap, meaning that it is possible, that only a small part at the top of each commit can be freely seen by the user before being obstructed by the commits placed in front of it. Additionally, some information about a commit will be displayed above its text block.

Therefore the user should be able to scroll the contents of the text blocks. However, since one of the reason to use this visualization is to see the differences between versions, scrolling each block only by itself would not be very useful. Instead, all text blocks—including the difference indicators connecting them—should be scrolled in a synchronized fashion (F.11), so that the user can scroll through the states of the file and their differences across the whole history without having to manually adjust the scroll position of many commit text blocks individually.

Optimally this would be achieved by making the scroll distances of the connected commits dependent of the changes between them, so that for as much of the displayed changes as possible both versions are shown, but this leads to long chains of cascading scroll adaption—rather than one simultaneous scrolling event for all commits—so that the

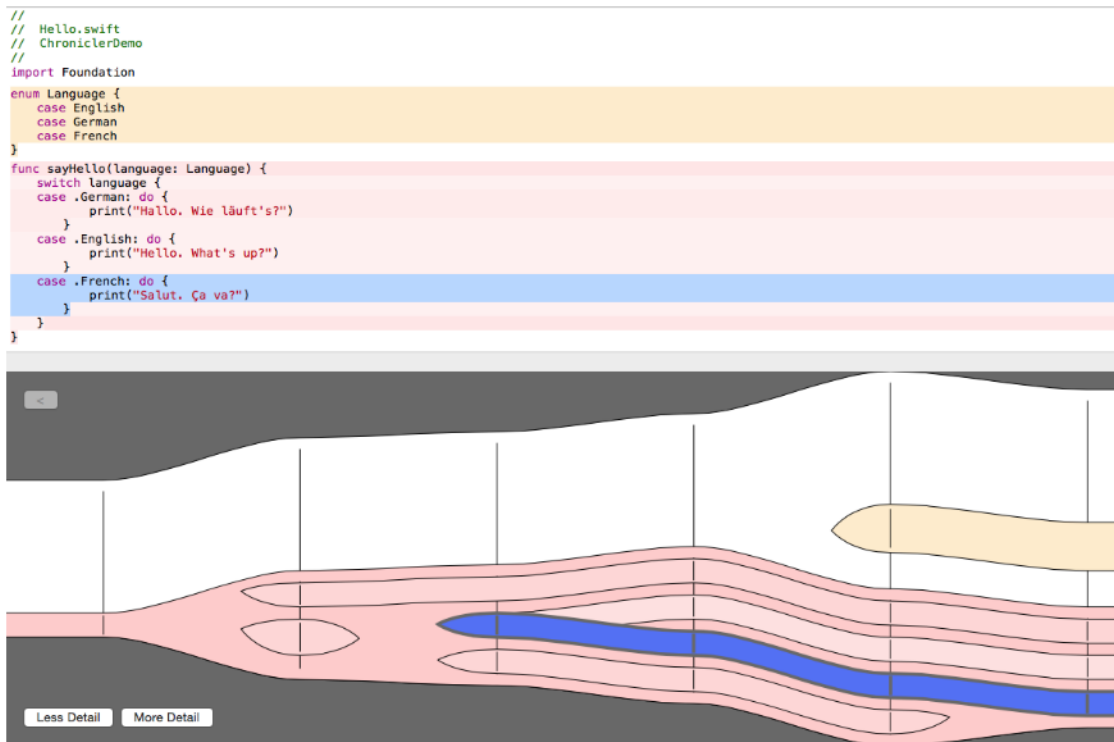


Figure 4.6: Tree Flows in the bottom view of Chronicer [44]

animation would likely take quite some time to settle down. Furthermore, it may even be impossible to do in general, since cycles in the commit graph—subgraphs where first a fork and later a merge happens—might lead to situations, where a scroll event in a commit on one branch would lead to different optimal scrolling positions for a commit on the other branch when viewed from one direction—from the fork point earlier in the history—than from the other—from the merge point later on. This could lead to situations, where two adjacent version have seemingly random scroll positions to each other, or even never ending loops of scroll adjustments, potentially rendering the tool unusable when it occurs. Therefore it was decided to implement a behavior, where commit text blocks will be scrolled for the same amount.

4.2.7 Comparing of Selected Commits

An ubiquitous feature of VCSs is the ability to compare any two revisions of the project in the repository. Graphical frontends and IDEs often expand upon this feature by providing functionality to show the two versions of a file side by side. As was already discussed in Section 4.2.5, such tools typically indicate and highlight the changes between the commits, and corresponding lines on both sides are visually connected or kept aligned.

A variation of this is the three-way diff view, in which two versions are compared against a common base version, which can be either a common ancestor of the two or a later

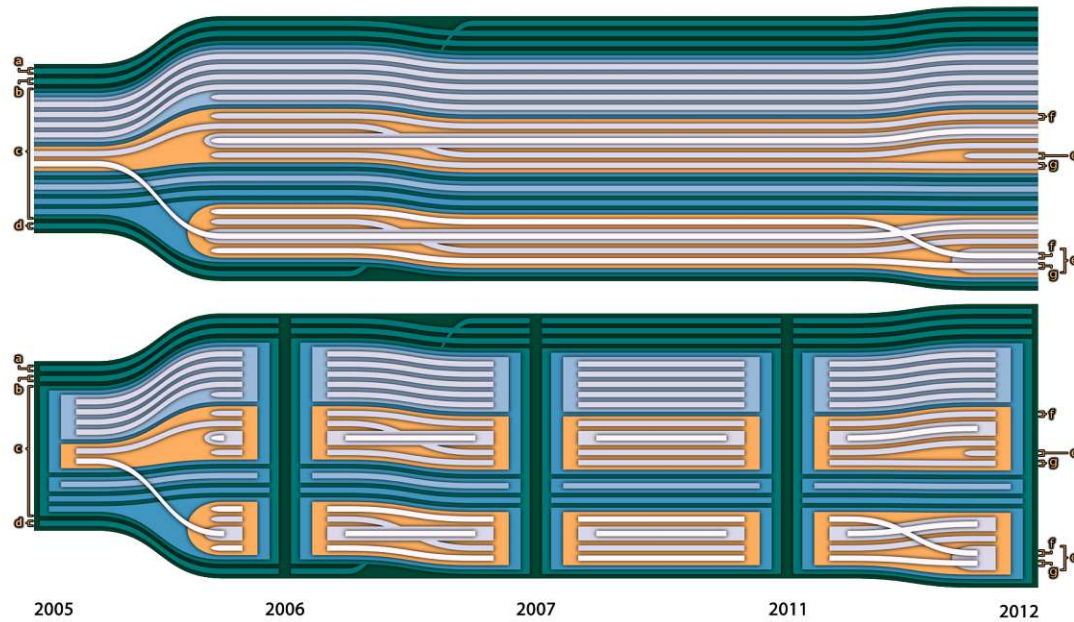


Figure 4.7: Example of SplitStreams [41]

merge, which in turn has both of the two versions in question as an ancestor. An example of the latter variation is provided by IntelliJ [W54], as demonstrated in Figure 4.8.

For a visualization tool, that already shows the commit graph of the repository, the source code of an individual file in those commits and indicates the differences between adjacent revisions, adding the ability to directly compare otherwise unrelated versions seems close at hand. As such it should be possible for the user of the visualization to select any two versions and compare them side-by-side (F.09). Additionally, it should be possible to optionally select a third base version, and show a three-way diff (F.10).

4.3 Implementation Considerations

During the phase of developing the concept, several experiments were done to explore if and how the visualization can be implemented. This section gives a brief overview over the considered ideas.

4.3.1 2D or 3D?

While the visualization itself is clearly three-dimensional, it was not clear if an implementation would need to utilize proper 3D technologies like WebGL [W69], or if a two dimensional layout with a few tricks to simulate three-dimensionality would be enough.

4. CONCEPTUAL DESIGN

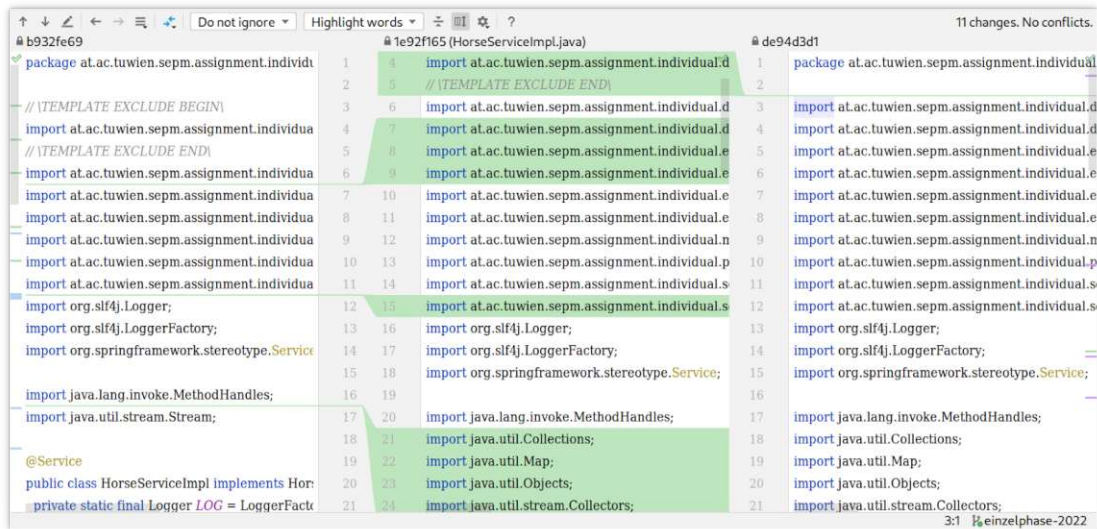


Figure 4.8: Example of a three-way diff view in IntelliJ [W54] of two versions against a merge.

The latter has the advantage, that the web browsers standard functionality for writing ordinary web pages—HTML markup, CSS styling and the Document Object Model (DOM) manipulation Application Programming Interface (API)—can be used to produce the whole visualization. *Z*-order can be used to tell the browser’s rendering engine which elements should be displayed above others and shadows can further give the viewer a sense of spatial distances of elements, that are drawn on top of each other. Together, these can be utilized to implement the displaying of the branch layers. With modern CSS, it is also possible to assign 3D transformations to elements, which could be used to visually bridge the gap between layers for the connections between commits. However it was not clear immediately, how the transformations of multiple elements interact when rendering and how the HTML DOM structure further affects this process. For example, two rotated divs, that should intersect each other might instead be rendered as one on top of the other if the two elements are not contained in the same parent element. Further investigation was needed to determine this.

WebGL on the other hand would have the advantage of proper three-dimensional projection, eliminating all these concerns. However, it would instead raise questions on how to display the two dimensional parts of the visualization. The text blocks for the code and colored curves for the difference indicators are straight forward in two dimensional HTML, but it is less clear how to do produce something like that in WebGL. One possibility might be to use a kind of off-screen HTML, that is rendered into a texture, which can then be applied to an object in the 3D space. But this is clearly not trivial to implement.

For the sake of simplicity, it was decided to make a prototype to investigate if a two dimensional, plain HTML and CSS based approach for the graphical presentation, is enough for the purposes for this thesis, or if the additional effort of dealing with 3D technologies is indeed necessary.

4.3.2 Simple Two-Dimensional Layout

The first tested approach, is to align the logical structure of the HTML DOM with the idea of laying out branches as layers in the three-dimensional space. Each timeline—that is, each branch—is represented by a DOM element. Inside this element, the individual commits are stacked from left to right in order from older to newer. This is achieved by making the timeline elements flex boxes, that arrange their children in a row. Multiple timelines are then stacked on top of each other, with horizontal and vertical offsets to be able to see the branches behind the ones in the front. To further give a sense of three-dimensionality, shadows on the elements, that represent the code in a commit, are used so that layers in the front stand out better from ones further behind.

Handling the connections between commits is more involved. Since two connected commits might lie in different timelines—because branches diverge or merge—connections need to be able to connect commits in different layers. Such connections can themselves not unambiguously assigned to one layer. The chosen solution for this issue is to logically put the connection into the layer of the commit on its left side. If multiple connections originate from the same commit, a container element is used to wrap all these connection elements and overlay them with each other.

To achieve the illusion, that this element connects to a commit at a different layer, the element is rotated, skewed and scaled using three-dimensional CSS transformations.

While this approach has the advantage, that the structure, that is used to create the layout, follows the logical structure of timelines in the history, issues arise with the way the commit connections are displayed. Since a connection starting from one commit and another connection starting from a commit behind the former are put into different parent elements, that lie on top of each other—they have a different Z -index—the connections themselves are also treated as one absolutely in front of the other. For many cases this is not an issue. However, if those connections cross each other logically, the result looks unnatural. Figure 4.9 shows such a case. In this example, the green connection originating from the leftmost commit (represented by a simple white strip) and the red connection crossing it are rendered as if one was completely behind the other. The whole section, where the elements are overlapping, is displayed with the same color, although it should be different on the left side of the crossing point that on the right side. As a result, the illusion of proper three-dimensionality is broken.

4.3.3 Two-Dimensional Layout with Flat Structure and Variables

To address the problems with Z -order and overlapping connections, the second approach uses a flat logical structure with more manual positioning and transformations.



Figure 4.9: Wrong appearance of crossing connections in simple, 2D layout

Instead of building the layout with flex boxes that are positioned absolutely, all elements individually positioned using absolute coordinates. To simplify this, much of the necessary calculations are done directly in CSS. The basic dimensions, such as width of the horizontal gap between commits or the vertical offset between two timelines, are defined as global variables in the style sheet. The commit elements each have their logical position—the number of the timeline and the point in this timeline—assigned to CSS variables in their style attribute. The associated style in the style sheet then uses these variables to calculate the physical position—`left` and `top`—of the element using the `calc` function. A transformation, that translates the element in the Z direction appropriate to its timeline is also applied. While this does not have an immediately visible impact, it is necessary for the correct three dimensional appearance in conjunction with the connections.

The connection elements are handled similarly, but involve more complexity. Each such element has a style attribute, that assigns its start and end position—both consisting of the timeline layer and the position inside the respective timeline—to CSS variables. Based on this data, the position and transformation are calculated. The transformation consists of a translation in the Z direction according to the element's center point, a rotation to achieve correct overlapping effects, and a skew to simulate the optic effect of a three dimensional rotation. The details of the calculation are discussed below.

Calculations for the Transformations of Connections In order to correctly display the connections between commit blocks, the rotation angle of the edge and its length need to be calculated. Figure 4.10 illustrates the relationships between the start position, end position and rotation angle of the edge in a right-angled triangle. While start position S and end position E are known, the rotation angle δ needs to be calculated. This rotation angle δ is then used for both the rotation and skew transformations, while the hypotenuse h is required to find the width of the connection element.

There are several ways to obtain length and rotation angle of the edge. Let w be the width of the gap between the start and end position and d the depth distance, then the hypotenuse h of the triangle forms the absolute distance between the points. One way to calculate h is by application the Pythagorean theorem:

$$h^2 = w^2 + d^2 \quad (4.1)$$

$$h = \sqrt{w^2 + d^2} \quad (4.2)$$

This has the disadvantage of involving a square root. This posed a problem for the goal of doing all calculations in CSS, since, at the time of writing, web browsers do not support a square root function in CSS [W70]; however, there is support for trigonometric functions in CSS [W71], [W72]. The length of the hypotenuse h can alternatively be calculated by application of the sine law if the rotation angle δ is already known:

$$\frac{h}{\sin 90^\circ} = \frac{d}{\sin \delta} \quad (4.3)$$

$$h = \frac{d}{\sin \delta} \quad (4.4)$$

Conversely, the rotation angle δ can be expressed through the sine law:

$$\frac{\sin \delta}{d} = \frac{\sin 90^\circ}{h} \quad (4.5)$$

$$\sin \delta = \frac{d}{h} \quad (4.6)$$

$$\delta = \arcsin \frac{d}{h} \quad (4.7)$$

If h is to be calculated via the sine law too, this obviously creates a cyclic dependency. However, as is demonstrated with circle c in Figure 4.10, the tangent of δ is related to d and can thus alternatively be obtained with the arc tangent function:

$$w \tan \delta = d \quad (4.8)$$

$$\tan \delta = \frac{d}{w} \quad (4.9)$$

$$\delta = \arctan \frac{d}{w} \quad (4.10)$$

With this, both the width of the connection element h and the rotation angle δ can be calculated directly in the CSS style sheet.

4.3.4 Connections that Indicate Differences

The initial idea for drawing the difference indicators between two versions, was to utilize the Canvas API of JavaScript [W73]. This API provides a programmatic interface for drawing complex graphics based on, for example, basic shapes and Beziér curves inside

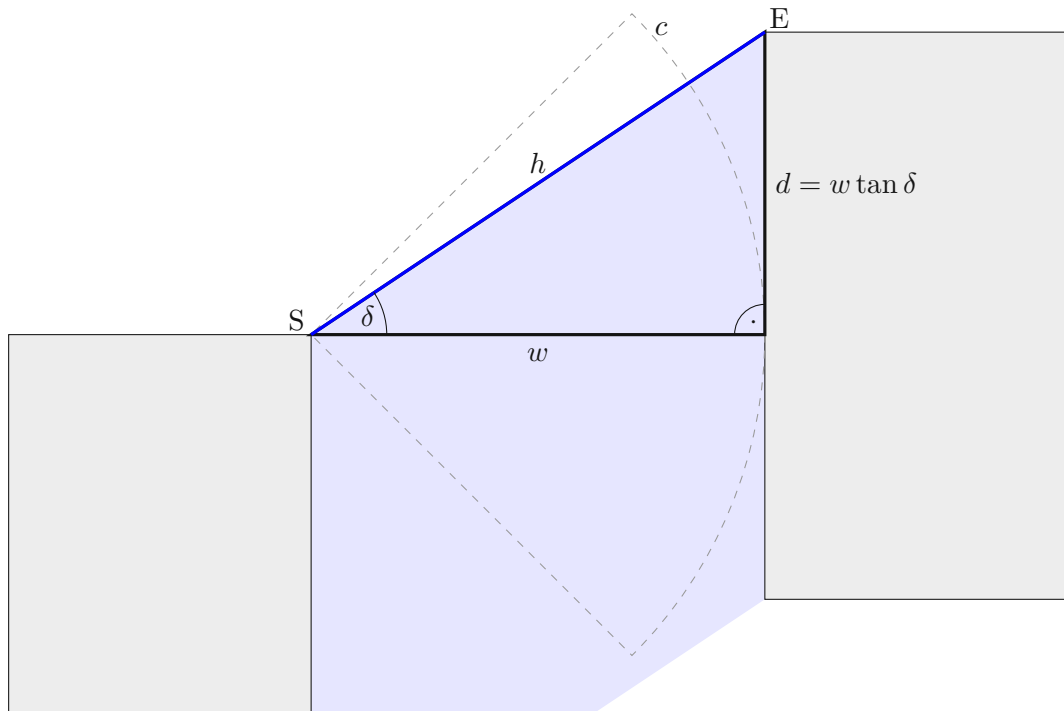


Figure 4.10: Relationships between the corner points of two commit blocks (gray) and the connection block (blue) width h and rotation angle δ . The side d corresponds to the tangent of δ in the circle c of radius w with its center at S .

an HTML `<canvas>` element. However, this has the severe drawback of producing pixel graphics, with the size of a logical pixel in the canvas not necessarily coinciding with the actual pixel size of the display device. In particular, when the user uses the browser functionality for zooming or scaling the page content, the logical pixel size of the canvas is scaled, rather than the resolution. Because of this, the canvas-drawn graphics become pixelated when zooming in this way.

Instead, Scalable Vector Graphics (SVG) was deemed as better suited for this task. Since SVG are vector graphics, they scale gracefully and therefore using the browser zoom does not degrade the display quality.

To indicate the differences between the code in the left and the right commit, connections are displayed in three colors: blue to connect unchanged code block in the two versions, red to indicate where lines from the first commit are deleted in the second, and green to indicate newly inserted lines. The areas are represented as a *path* SVG, where the curved lines are produced using cubic Beziér curves. Figure 4.11 provides a visual demonstration.

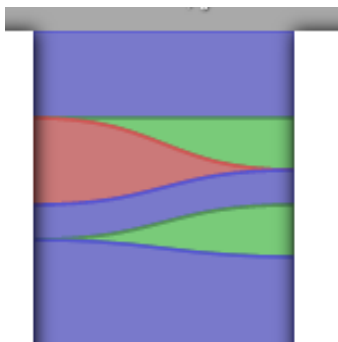


Figure 4.11: Difference indicators produced with Beziér curves in SVG.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Semi-Structured Expert Interviews on the Concept

Before implementation of the prototype for the proposed visualization was started, a set of semi-structured interviews was performed, to get data from experts working in SE with the aid of VCSs if they think that the proposed visualization with the envisioned features satisfies the postulated information needs and if they think that the tool is useful in the process of working with non-linear history.

In this chapter, the planning for and results of these interviews are presented.

5.1 Plan

The interviews were performed with the aid of a questionnaire form. The questionnaire is structured into four sections:

1. **Demography**, in which some data about the interviewee's person and experience in the field are raised for comparison
2. **Information Needs**, where the interview participants are asked about their opinion of different kinds of information during an SE process
3. **Importance of Features**, where all features of the prototype are listed and the interviewees are asked to rate the individual features for their importance
4. **Purpose of the Visualization as a Whole**, in which the interviewee's general impression of the visualization as presented and closing remarks are gathered.

The individual questions are listed together with the results in Section 5.2; for the questionnaire as used during the interviews, see Appendix A.

5.1.1 Pilot Interview

Before the interviews were conducted, a pilot interview was done. This purpose of this special interview was to identify and eliminate flaws in the questionnaire and overall interview procedure. The data from the pilot interview is not included in the evaluation results.

The pilot revealed that the structure of the questionnaire section **Importance of Features** was potentially unclear. The interviewee was confused by the fact that the graphics that illustrate the feature description were placed below the question itself; since the graphics appeared in the questionnaire after the answer field—and therefore later in the reading flow—they could easily be overlooked, leading to confusion and problems understanding the question. For clarity, the structure for questions using explanatory graphics in that questionnaire section was changed after the pilot interview to the following format:

- First, the short feature description
- A longer text explanation, if necessary
- The illustrative graphics with brief text notes
- Finally, the answer field, including the short description again

With this format, both the textual explanation and the corresponding illustrations appear before the answer field in the flow of the questionnaire and therefore cannot be overlooked as easily.

5.2 Results

Three interviews have been performed. In this section the results of these interviews are listed and visualized. Median and average of the answers are often listed for completeness, even though they may not be particularly meaningful for only three data points.

5.2.1 Demography

First, the interview participants were asked some questions about themselves to gain some insight about their experience in the field of SE in general and working with VCSs in particular. They were also asked to list tools they have used in the context of their work with VCS repositories.

Experience in the field of SE was given as 7, 8, and 10 years, with a median of 7, average $8.\bar{3}$. When asked about their *experience working with VCSs*, the interviewees responded with 5, 10, and 15 years, with an average of $9.\bar{3}$ and median 8. The answers are visualized in Figure 5.1. Interestingly, one participant had worked quite a few years more with VCSs (15) than they experience with SE (7).

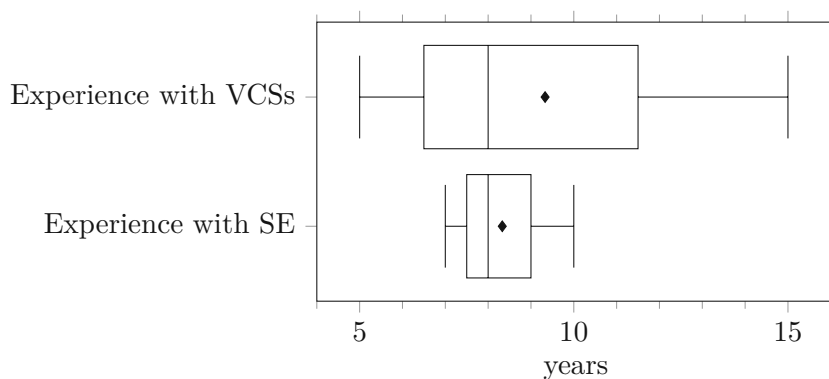


Figure 5.1: Experience of the interview participants. Note that, due to the small sample size of 3, the quartiles are not meaningful.

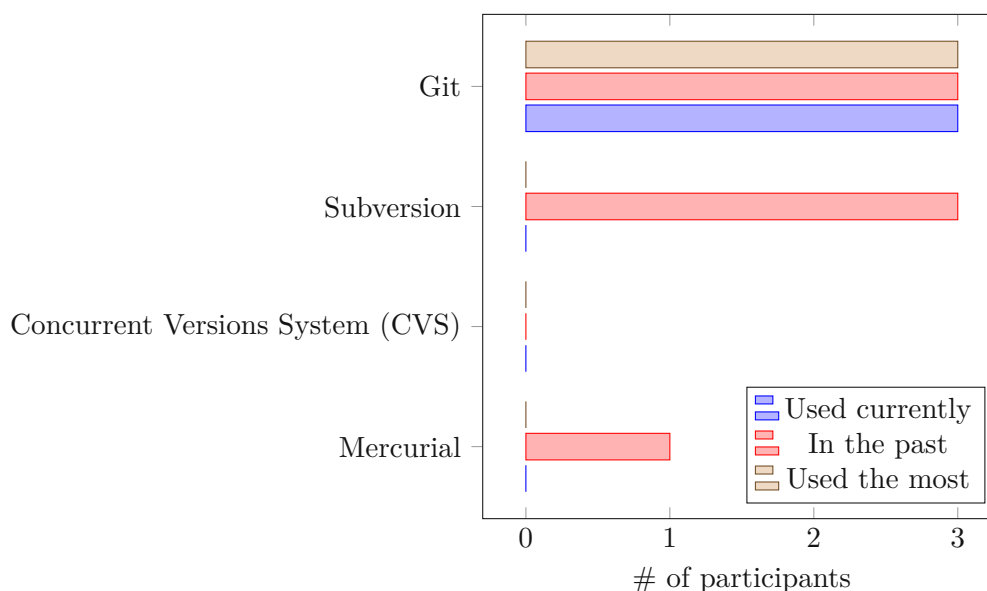


Figure 5.2: VCSs used by the interview participants.

Usage of different VCSs was gathered in three sub-questions: which VCSs the participants use currently, have used in the past, and have used the most. To the question *Which VCS do you use currently?* all participants answered solely Git (see Figure 5.2). Likewise, all participants gave Git as their *most used VCS*. For *VCSs used in the past* all interviewees responded again with Git, but also Subversion, and one participant had used . None of the participants listed CVS, which was also explicitly listed as an option, or added another VCS to the list.

Alternative frontends to the VCSs—that is, other than the default Command Line Interface (CLI)—which the participants had used in the past were:

- GitKraken [W64] (*2 times*)
- SourceTree [W74]
- TortoiseGit [W75]
- Git GUI¹ [W1]
- gitk¹ [W62]
- Fork [W76]
- Integration in IntelliJ [W54] (*2 times*)

Repository mining tools are programs with which more information about the history of a project can be extracted from the VCS. When the interviewees were asked which such tools they had used, the list of responses was short; one of the three interviewees even said they had not used such a tool at all. Overall, only two repository mining tools were named:

- Gitinspector [W77]
- Fork [W76]

Visualizations The final question of the demography section was which visualization tools the interviewees had used to aid them in their work with VCSs. Each participant listed one tool:

- Commit graph in IntelliJ [W54]
- Fork [W76]
- Data from a custom script [30] visualized with Apache Superset [W78]

5.2.2 Information Needs

The goal of the second segment of the interview was to gather data on the information needs of developers. All questions in this section were to be answered as a rating of importance on a scale from 1 to 5, where 1 meant *unimportant* and 5 *very important*.

Overview over non-linear history and visualization The first question in this segment was *Is it important to get an overview over possibly non-linear history of a project's source code?* The responses of the interviewees were uniformly on the *important* side; one participant gave a rating of 4, the other two answered 5, giving a median of 5 and an average of 4.6̄ (see Figure 5.3).

Related to the former question is *Do visualizations help in getting an overview over the history?*² The given ratings were even more strongly towards the *important* side;

¹Git GUI and gitk are part of the default Git distribution, but are still considered an alternative frontend here, as they are different from the standard CLI of git.

²A scale from *unimportant* to *very important* is of course not a good fit for a question about the *helpfulness* of visualizations; this was a mistake that was realized late. However, during the interview no misunderstandings arose and none of the participants seems to have noticed a discrepancy.

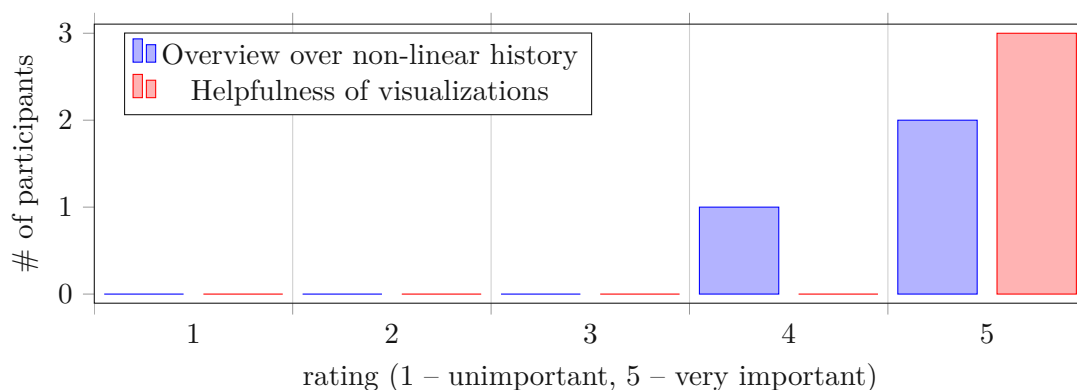


Figure 5.3: Ratings on the importance of overview over non-linear history and the helpfulness of visualizations for that purpose.



Figure 5.4: Ratings about the importance to be able to see changes to a piece of code over the history.

all interviewees answered with 5, trivially giving both a median and average of 5 (see Figure 5.3).

Changes to the same piece of code Two questions were asked about the importance to be able to see how a piece of code changes over the course of development. The first of those questions was about the *changes in different versions* of the project. The answers here also tended towards the *important* end of the scale; two participants gave a rating of 4, while one responded with 5, resulting in an average of $4.\bar{3}$ with a median of 4.

The second question of the two questions was about the *changes across branches*. The results here were more mixed. Two interviewees gave the same rating as to the previous question: 4 and 5; the third interview participant responded with 2. The median rating is therefore 4 and the average $3.\bar{6}$. The results to both questions are visualized in Figure 5.4.

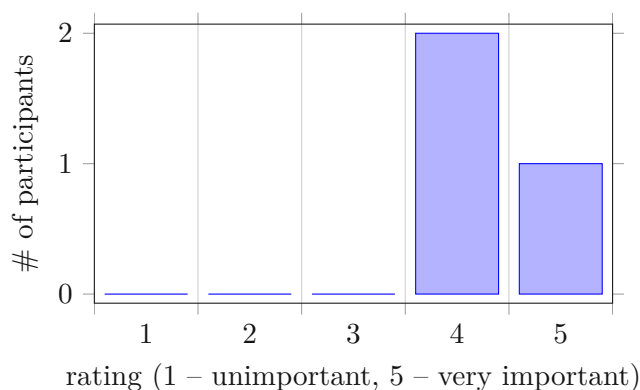


Figure 5.5: Importance of simple and direct navigational control.

Direct navigation The final question on the topic of information needs was how important it is to the interview participants for the navigation through the history to be direct, that is controlled via simple scrolling and panning inputs without having to manually enter a target version to navigate, give textual commands, and so on. The responses were again more towards *important*, with two participants rating it as 4 and the third as 5 (see Figure 5.5), giving a median of 4 and an average of $4.\bar{3}$.

5.2.3 Importance of Features

In the third segment of the interview, the features which were outlined as part of the concept (see Table 4.1) were listed. Each feature was presented in the following format:

1. A short description of the feature
2. A longer textual explanation, describing the feature in more detail
3. Graphics and sketches to further illustrate the feature, with brief textual notes
4. A reiteration of the short description, for context
5. The answer field with a rating selection on a scale from 1 (*not important*) to 5 (*very important*).

View of graph and multiple parallel branches The *view of the whole commit graph in the context of a file* (F.01) is the central feature of the visualization and was also listed as the first. The answers given by the participants were in the range from *neutral* to *very important*: one participant rated the graph view with 3, the other two interviewees with 5 (see Figure 5.6). The median of the answers was 5, the average $4.\bar{3}$.

Closely related is the *view of multiple parallel branches side by side* (F.02), which received the same ratings from all participants as the previous feature: 3 by one participant, 5 by the two others, with a median of 5 and an average of $4.\bar{3}$ (see Figure 5.6).

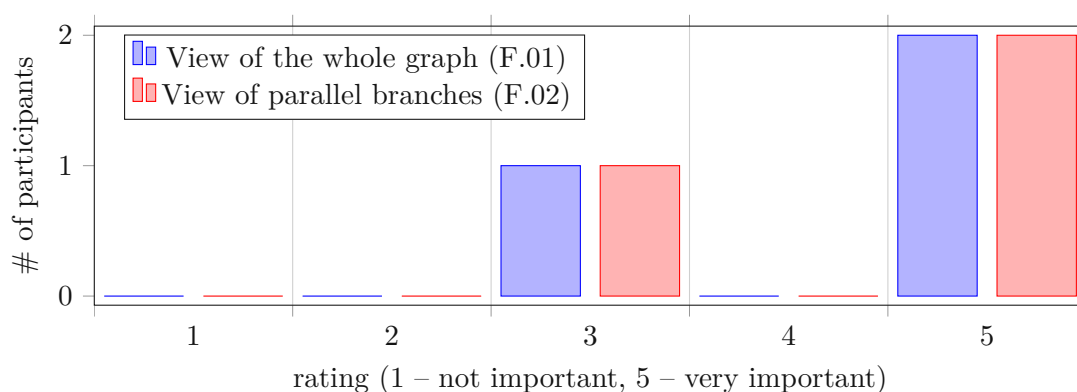


Figure 5.6: Importance of the overview of the whole graph (F.01) and parallel branches (F.02).

Difference indications Two variants of indications of text changes between two related versions were presented. The first of these were the *simple indications* (F.03), which mark additions and deletions of text between versions. This feature was universally rated *important*: all three participants gave a rating of 5, trivially yielding both a median and average of 5.

The second difference indication variant that was presented were *long term structural changes based on Code Flows* [46] (F.04)³. Compared to the former difference indication variant, the ratings of this variant were more neutral: two interviewees rated it with 4 while the third participant gave a 3, resulting a median rating of 4 and an average of $3.\bar{6}$. The results for both questions are visualized in Figure 5.7.

The difference in the ratings is striking; despite the *Code Flows* based variant giving more information about the change in code structure, it was rated as less important. The reason might be familiarity: the “simple” line-based difference indications are a well established way to show differences between file versions and can be found in, for example, IntelliJ [W54] and Kompare [W68]. In contrast, more structural code change visualizations like *Code Flows* [46] and *Tree Flows* [44] have been presented in the context of academic research, but to the knowledge of the author of this thesis no widely available tool implements such a difference visualization at the time of writing.

Focusing of individual branches according to the user’s selection while other branches are faded out or hidden completely (F.05) was rated as *important* by all interview participants: one participant gave a rating of 4, the other two interviewees rated the feature with 5. The median rating was 5, the average $4.\bar{6}$. The results are visualized in Figure 5.8.

³Due to an error, the screenshot shown to the participants during the interview was actually of *Tree Flows* [44], a related and very similar, but different change visualization.

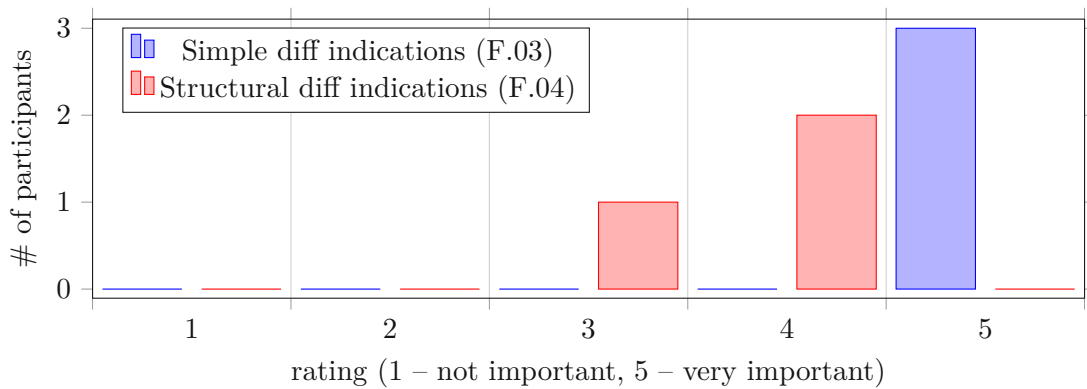
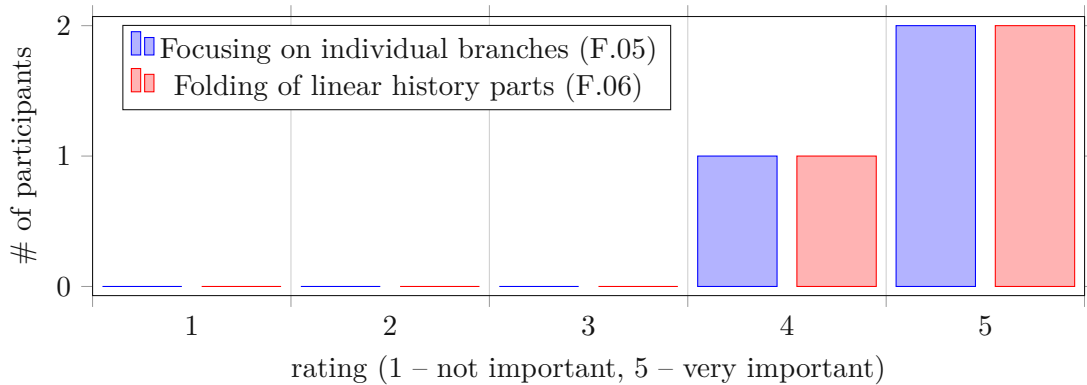


Figure 5.7: Importance ratings of difference indications (F.03, F.04).

Figure 5.8: Ratings for *Focusing on individual branches* (F.05) and *Folding of linear history parts* F.06

Folding of linear parts of the history that contain branching points (F.06) was rated like Feature F.05: one rating of 4 and two of 5, with the median rating at 5 and an average of $4.\bar{6}$ (see Figure 5.8). In fact, this similarity extends beyond the overall result: each individual interviewee gave the same rating for Feature F.06 and Feature F.05. This might be because both features are about changing which parts of the history graph to show. On the other hand, it might be just a coincidence, even more so due to the small sample size.

Zooming the view of the history graph was split into two features: *zooming in* for more detail (F.07) and *zooming out* for more large scale overview (F.08). Both features got the same rating from all three participants; one rating of 3, one of 4 and one of 5, yielding an average of 4 and a median of 4. The results are visualized in Figure 5.9.

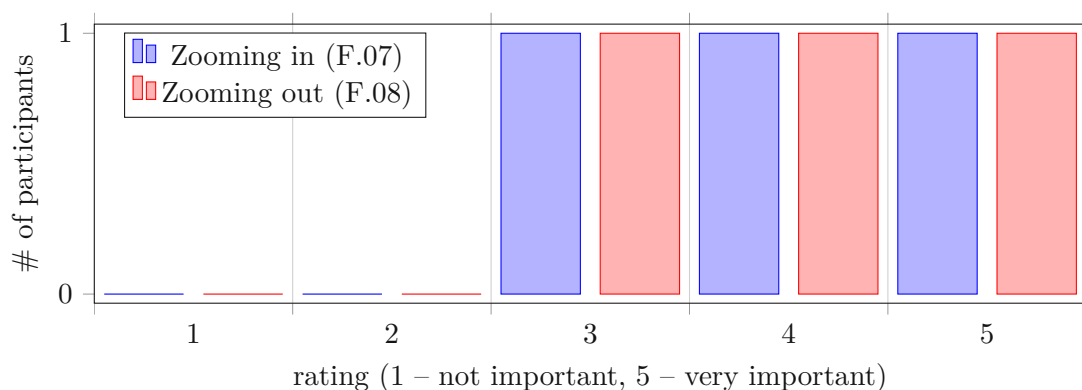


Figure 5.9: Importance ratings for zooming the view of history graph (F.07, F.08)

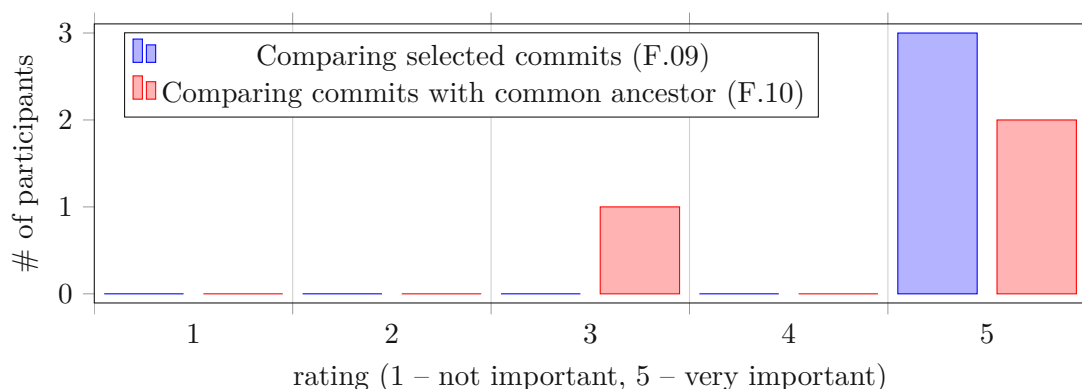


Figure 5.10: Importance ratings for direct comparison of selected commits (F.09, F.10)

Direct comparison of selected commits was also presented in two variations: *comparison of just the selected commits* (F.09) and *comparison with regards to a common ancestor commit* (F.10). The results are graphically presented in Figure 5.10. The simple two-commit variant (F.09) was rated as uniformly *very important*: all three interviewees gave the feature the highest rating of 5, trivially giving both a median and an average of 5.

The 3-way variant (F.10) involving a common ancestor version was rated very similarly: Two of the three interview participants rated it with 5 again while the third interviewee rated it with a neutral 3. The median was 5, the average of the ratings was $4.\bar{3}$.

Synchronized scrolling of text in all commits was rated with a hinge towards the *important* side: The highest rating of 5 was given twice, a neutral rating of 3 once, resulting in an average rating of $4.\bar{3}$ and a median of 5. See Figure 5.11 for a visual presentation of the results.

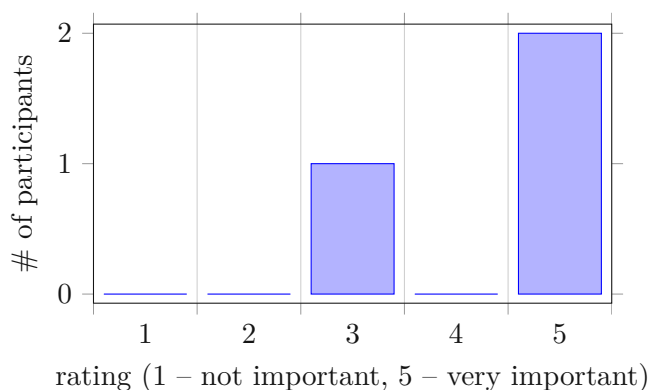


Figure 5.11: Importance ratings for synchronized scrolling of commits (F.11)

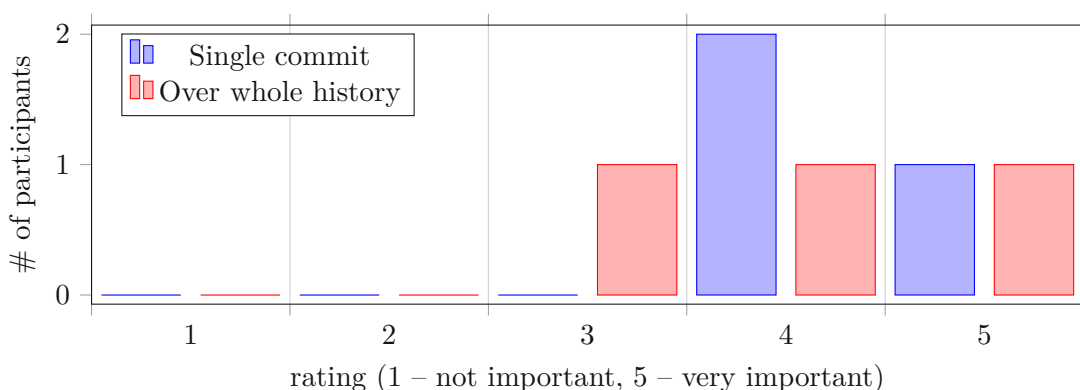


Figure 5.12: Importance ratings for highlighting a selected block in other commits (F.12, F.13)

Selected block highlighting The final two features presented to the interviewees were two variants of highlighting a selected code block in other commits: *Highlighting the corresponding block in one other selected commit* (F.12) and *highlighting the corresponding block over the whole history* (F.13). The interview participants' ratings are visualized in Figure 5.12.

The feature variant of highlighting the corresponding block in a single selected commit (F.12) was rated uniformly on the *important* side of the scale: The three interviewees gave two ratings of 4 and one of 5. The average rating was $4.\bar{6}$ and the median 5.

The second variant of the feature where the code block is highlighted over the whole history graph (F.13) was in comparison rated a bit more neutrally but still with a clear hinge towards *important*. Two participants gave the same rating as they did to the other feature variant: 4 and 5; the third participant rated this feature neutrally with 3. The average rating was 4, as was the median.

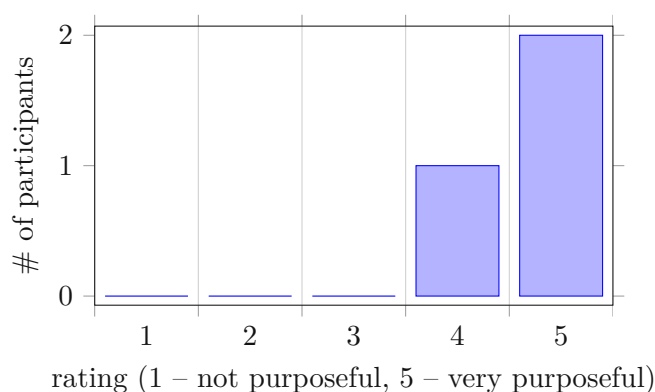


Figure 5.13: Ratings of the purposefulness of the proposed visualization

5.2.4 Purpose of the Visualization as a Whole

The final segment of the interview was concerned with the interviewees opinion on the purposefulness of the visualization as a whole, as well as concluding remarks. The format was a mix of rating questions on a scale of 1 to 5 and open questions.

Purposefulness of the visualization The question on the purposefulness of the whole visualization as proposed was to be answered with a rating on a scale from 1 (*not purposefulness*) to 5 (*very purposeful*). The answers were uniformly on the *purposeful* side. One participant rated the visualization with 4, the other two gave it 5, the highest rating (see Figure 5.13). The median of the ratings was 5, the average $4.\bar{6}$.

Overview over code evolution The interview participants were asked about their opinion on the helpfulness of the visualization to get an overview over the evolution of code. This was to be rated on a scale from 1 (*not helpful*) to 5 (*very helpful*). All answers were towards the *helpful* end of the scale: two ratings of 5 and one of 4 were given, with an average of $4.\bar{6}$ and a median of 5 (see Figure 5.14).

Teaching of Software Engineering practice For the final rating question in this interview questionnaire, the interviewees were asked if they think that the visualization could be utilized in teaching SE practice. The helpfulness for this purpose was to be rated on a scale from 1 (*not helpful*) to 5 (*very helpful*). The answers were mixed, as can be seen in Figure 5.15: The highest rating given was 4, which was answered twice; the third answer was 2. The median of all answers was 4, the average $3.\bar{3}$.

Final remarks The conclusion of the questionnaire were two open questions for final thoughts from the interviewees. First, the participants were asked if they had in mind *features that were not presented but which they would want to see implemented*. Only one participant listed a single feature: *The ability to change the colors used in the visualization*. This could be used to avoid problems with color blindness.



Figure 5.14: Ratings for the helpfulness for getting an overview of code evolution

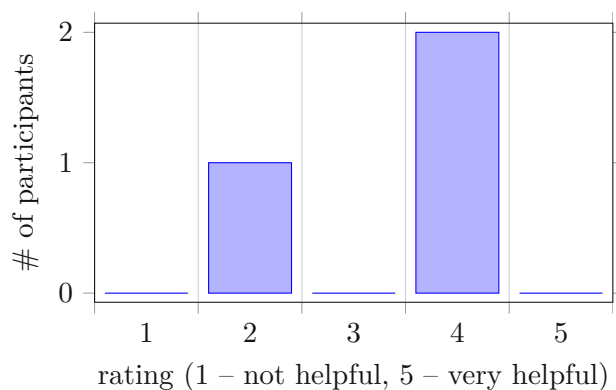


Figure 5.15: Ratings of the helpfulness in the teaching of SE practice

The closing question to the participants was if they had any *further remarks*. This was likewise answered by only one participant, who noted that *possibly high effort is required to implement the proposed visualization*.

5.3 Summary and Interpretation

The proposed visualization was overall well received by the participants of the interviews. For all proposed features both the average and the median were towards the *important* side of the scale both. The most *important* rated features were the indications of differences between commits in the simple addition and deletion based fashion (F.03) and the direct comparison of two selected commits (F.09). Both of those features received the highest rating of 5 (*important*) from all three interviewees. In contrast, the least important feature was the difference indications based on code structure (F.04). The lowest rating this feature received was the neutral rating of 3, with an average of $3.\bar{6}$.

The visualization as a whole was also clearly rated as *purposeful* by all interviewees. They also thought it to be *helpful* for getting an overview over the evolution of the code in a

project. Only the *helpfulness* for the purposes of teaching SEs practice was rated more neutrally with an average rating of 3.3̄. This was also one of only two questions in this questionnaire which received a rating lower than 3, namely 2 (towards *not helpful*).

The information needs which were listed were also all thought to be *important* by the participants. They only gave ratings of 4 or 5, with the single exception of *being able to see changes to the same piece of code across branches*, which received one vote of 2 (on the *not important* side).

5.4 Threats to Validity

Small Sample Size Three developer participated in the interviews on the concept. This small number of data points is far from a statistically relevant size. It is possible that the small number of interview participants biased the interview results. However, the purpose of the interview was to find out if the concept is worth exploring further, but was not developed enough yet for a large scale evaluation.

Influence due to the Interviewer The interviewer might have some influence on the perception and decisions of the interviewees. This could be in different forms, such as suggestive phrasing or a bad demonstration of the prototype. While it was attempted to avoid this by minimizing the interaction between interviewer and interviewee during the filling in of the questionnaire, this factor cannot be fully eliminated, especially during the demonstration phase of the interview.

Bias due to Phrasing The phrasing of the questions in the questionnaire and the explanations of the prototype could be ambiguous, leading to misunderstandings which could have possibly biased the answers of the participants. Furthermore, the phrasing could be otherwise suggestive and influence the responses more directly. A pilot interview was performed to find such problems in the wording, however in cannot be fully ruled out that such influences were still present in the evaluation interviews.

Implementation

The topic of this chapter is the development of the prototype. The development process is described in a chronological succession of *iterations*, in which a certain feature from the concept or other part of the prototype was implemented or extended, until the final state of the prototype, which was used in the evaluation (see Chapter 7 Semi-Structured Expert Evaluation with the Prototype), was reached. Theoretical foundations and complexities in the implementation are explained. In the final section the whole process and its result are summarized and a list of the features which were included—and which were not—is given.

6.0 Iteration 0: Setup

This section discusses the architecture of the prototype and the technologies used. It also describes the extraction of history information from the software repository.

6.0.1 Architecture

The prototype is built as web application consisting of two components: a frontend and a backend.

The frontend is a web page built as a *Single Page Application (SPA)*. Its purpose is to implement the actual visualization, displaying the commit graph, file contents and other information related to the history stored in the repository. The history data is obtained from the backend via HTTP through the Fetch API provided by the Web browser.

The User Interface (UI) framework *Svelte* was used to simplify the development process compared to using the bare DOM API. For styling of standard UI elements, *Bootstrap* [W79] was employed. The concrete selection of these particular frameworks is of little consequence and is purely due to the author's preference.

As programming language, *TypeScript* was selected because of the static type checking it provides over JavaScript.

The backend exposes a REST interface which provides the repository history data that the frontend requires. It is implemented in the programming language *F#* [W80] on top of *.NET 7* [W81] using the *ASP.NET* [W82] framework and the library *LibGit2Sharp* [W83].

The main concern for the technology selection of the backend was to be able to extract the history data from the Git repository in a way as simple as possible. The following functionalities were required:

- Listing all references in the repository (branches, tags, ...)
- Listing all commits reachable from a reference and their metadata
- Getting the contents of a file in the repository at a certain commit
- Creating a diff of two different versions of a file

LibGit2Sharp provides all these functions. *ASP.NET* was then chosen as the Representational State Transfer (REST) framework, because it is the most widely known web service framework for *.NET* [W84] and the programming language was selected due to the author's preference.

6.0.2 Extracting Data from the Repository

First, most of the backend functionality was implemented. A REST endpoint was implemented, that returns the commit graph for a given repository and file in that repository. The paths for the repository and the file are passed as query parameters.

The graph is structured as sets of nodes and edges. The nodes represent the commits. Each node includes the commit ID, the set of IDs of the parent commits and the text of the file in that commit. The edges represent the connections between a commit and one of its parent commits. One such edge includes the ID of the predecessor commit, the ID of the successor commit, and the differences in the file between the two versions. In this initial implementation, the differences were returned to frontend in the form of a patch, formatted the same way as the command `git diff` would present it to a user.

The graph is constructed in two steps: getting the commits from the repository and creating diffs for connected versions. The commits are obtained by listing all commits that are reachable from any reference—that is, from any branch head, remote branch head, tag or stash mark. This functionality is already provided by the Git access library. The commits are then converted from the internal format of the library to the structure described above. The edges are then constructed by getting diffs between each commit and each of its parents, and packing the result into the edge structure described above. Diffing the commits is done via the Git access library.

To enable simple retrieval of specific nodes and edges, the corresponding commit and diff objects are collected into dictionaries, where the key for the commits is their ID and the key for the diffs is a tuple of their predecessor and successor commit IDs.

The REST endpoint returns the resulting graph in the form of lists of commits and diff objects in the HTTP response.

6.1 Iteration 1: Layout Algorithm for Layered Graphs

At the center of the visualization lies the structure of the commit graph of the repository. Positioning the nodes and edges of the graph is therefore a central aspect of the prototype.

Features and problems tackled in this iteration were:

- The logical positioning of commit graph nodes and routing of edges (related to Feature F.01 and Feature F.02)

6.1.1 Problem and Theory

The commits and their parent-child relations form an acyclic, hierarchical graph [52]. A commit comes logically after all its parent commits. These before-after relationships determine at which level in the hierarchy a commit is placed. Root commits, that represent the start of the history in the repository, are at the very top of the hierarchy. On the second level are the commits that have only such root commits as parents. On the level below that are commits that have parents only in the second and first levels and so on. A directed graph laid out based on such a hierarchy is also called a *layered graph* [53].

To be consistent with the terminology employed by the relevant literature, the edges corresponding to parent-child relations are considered to go from the parent commit node to the child node. Therefore, an *inbound* edge of a node is an edge where the commit is the child, while an *outbound* edge is an edge where the commit is the parent. Due to this, the concepts of a *predecessor* and *successor* node in a directed graph are aligned with the *predecessor* or *ancestor* commit and the *successor* or *descendant* commit, respectively, in a Git history.

Sugiyama, Tagawa, and Toda [54] present an algorithm for automatically creating a layered layout of a directed graph. The algorithm works in four steps:

- **Create a layering**
The nodes of the graph are grouped into layers based on their relations formed by the directed edges.
- **Minimize edge crossings**
The nodes in the layers are assigned an order with the goal of minimizing edge crossings.

- **Improve readability**

Connected nodes in adjacent layers might still be rather far apart from each other and “long” edges that span multiple layers might not be straight due to the positions of the nodes. To improve the readability of the graph and straighten out long edges, the nodes are moved to other positions in the layer, nearer to the nodes that they are connected with, without changing the order within the layer. This process might create holes between the nodes of the same layer.

- **Display the graph**

The logical graph structure is rendered in a two-dimensional, graphical form.

The first three stages of the algorithm were implemented as part of this iteration and are discussed in more detail in subsections 6.1.2 to 6.1.4. The fourth stage—the graphical presentation—was implemented later in Iteration 2. For intermediate testing, the graph was drawn in a simple SVG-based graphic in the traditional top-to-bottom layout—rather than the left-to-right timeline-like layout that will be used for the purposes of the visualization prototype.

Over time, similar four-stage algorithms based on the one by Sugiyama, Tagawa, and Toda, were presented by, among others, Gansner, Koutsofios, North, *et al.* [55], Bastert and Matuszewski [53] and Eades, Lin, and Tamassia [52]. They often only change one stage of the algorithm to perform better on average for certain conditions or adapt the output to certain aesthetic criteria. Since the graph layout is not the focus of this thesis and the goal is to implement a prototype for the visualization as a proof of concept, these variations were not considered and are not discussed any further.

Another variation is to make the graph more compact by bundling edges [56], [57]. Elsen [4] utilizes such a method in *VisGi*. Since the visualization presented here uses the edges to show differences between connected versions, such a bundling of edges does not make sense in this context.

6.1.2 Stage 1: Create Layering

In the first stage of the layered graph layout algorithm, the nodes of the graph are assigned to layers. If connected nodes are not placed on adjacent layers, the edge is split into pieces, each between two adjacent layers. Dummy nodes are inserted on the intermediate layers as placeholders and used as the endpoints of the pieces. The result forms a chain of edges that connects the original nodes on its ends. This way, the following steps of the algorithm only need to consider adjacent layers and do not need a global view on the graph.

In general, solving this problem is not trivial. If the graph has cycles, those have to be removed or otherwise dealt with beforehand [52], [55]. Gansner, Koutsofios, North, *et al.* [55] have shown, that one automatic way of eliminating cycles—the reversal of certain edges while minimizing crossings—is NP-complete [55]. Garey and Johnson [58] have shown, that even finding the minimal crossing number of a graph is NP-complete.

However, for the purposes of this thesis, this is not an issue. Since the graph to be laid out is the commit graph of a Git repository, it is guaranteed to be free of cycles. Therefore no prior detection and handling of cycles needs to be implemented.

Furthermore, due to the prototypical nature, the solution is not required to be optimal. Therefore, each node can simply be placed the earliest layer it can be in according to the edges connected to it, even if that sometimes results in a sub-optimal layering. One situation in which this could potentially arise is when the repository has multiple root commits. In that case some root commits might be able to be placed in layers other than the first, which could, in turn, make it possible to move other commits to later layers, reducing the potential for crossing edges. In practice however, the vast majority of software repositories have only one root commit.

The implementation produces two lists as result: a list of layers and a list of edge sets `layerEdges` where the elements of `layerEdges[i]` are edges between `layers[i - 1]` and `layers[i]`.

Before starting the layering process itself, the implementation sets up four helper structures:

- **remainingCommits:** The set of commit nodes, that have not yet been assigned to a layer. Initially, all commits are in this set.
- **remainingParents:** A dictionary from commit IDs to edges. It associates every commit node c with those of its inbound edges e where the corresponding parent of e has not yet been assigned a layer.
- **children:** A dictionary from commit IDs to edges. It maps every commit node to its outbound edges.
- **lastLayerEdges:** The set of edges whose predecessor node has been placed in a previous layer while the successor node has not yet been assigned to a layer. This set actually contains only edges where the predecessor is in the previous layer. The reason for that is that long edges that span over layers are split into pieces that only connect adjacent layers. Initially, there are no such edges, as no layers have been processed yet. Thus this helper structure is initialized to the empty set.

After that, the following steps are repeated until `remainingCommits` is empty:

- **Apply BUILD NEXT LAYER**
The procedure `BUILD NEXT LAYER` (Algorithm 6.1) is applied with the current values of the helper structures as argument. It produces three sets: a set of nodes L representing the new layer, a set of edges E between the new layer and the previous, and a set of edges O where the predecessor is in the new layer and the successor has not yet been assigned a layer.

- **Add new layer**
 L is appended to `layers` and E is appended to `layerEdges`.
- **Update helper structures**
 The helper variable `lastLayerEdges` is set to O . All nodes in L are removed from `remainingCommits` and from all elements of `remainingParents`.

Algorithm 6.1: Building the next layer

```

Procedure: BUILD NEXT LAYER
Input: remainingCommits, remainingParents, children, lastLayerEdges
Result: nodes, edges, edgesTowardsChildren

  /* Step 1: Find nodes for this layer */
1 let nodes  $\leftarrow$   $\{c \in \text{remainingCommits} \mid \text{remainingParents}[c] = \emptyset\}$ 
  /* Step 2: Edges, where child is in this layer, end here */
2 let edgesTowardsParents  $\leftarrow$   $\emptyset$ 
3 forall  $n \in \text{nodes}$  do
4    $\left[ \begin{array}{l} \text{lastLayerEdges} \leftarrow \{e \in \text{lastLayerEdges} \mid \text{child of } (e) \neq n\} \\ \text{edgesTowardsParents} \leftarrow \\ \text{edgesTowardsParents} \cup \{e \in \text{lastLayerEdges} \mid \text{child of } (e) = n\} \end{array} \right.$ 
  /* Step 3: Find edges starting at this layer */
6 let edgesTowardsChildren  $\leftarrow$   $\{c \mid n \in \text{nodes}, c \in \text{children}[n]\}$ 
  /* Step 4: Split edges, that span over this layer by creating dummy nodes */
7 forall  $e \in \text{lastLayerEdges}$  do
8    $\left[ \begin{array}{l} \text{let } d \leftarrow \text{new dummy node} \\ \text{nodes} \leftarrow \{d\} \cup \text{nodes} \\ \text{edgesTowardsParents} \leftarrow \{\langle \text{parent of } (e), d \rangle\} \cup \text{edgesTowardsParents} \\ \text{edgesTowardsChildren} \leftarrow \{\langle d, \text{child of } (e) \rangle\} \cup \text{edgesTowardsChildren} \end{array} \right.$ 
12 return (nodes, edgesTowardsParents, edgesTowardsChildren)
  
```

The procedure BUILD NEXT LAYER produces a set of nodes `nodes` that form the next layer, as well as two edge sets: `edgesTowardsParents` consists of the edges between the current layer and the previous one, and `edgesTowardsChildren` contains the edges where the predecessor node is a member of the newly produced layer. The latter set is passed to the procedure as `lastLayerEdges` when the procedure is later called for the next layer. The procedure is outlined as pseudo-code in Algorithm 6.1. The pseudo-code uses set notation because it is convenient, well known, and order is irrelevant for all involved structures; the TypeScript implementation uses lists and objects where appropriate.

The procedure works in four steps:

- **Step 1** gets the set all non-dummy nodes that will be placed in the current layer. This set consists of all nodes n , for which hold that all parents of n have already been assigned to some earlier layer. To find these nodes, simply searching the helper structure `remainingParents` for entries with no elements is sufficient, since this structure is updated every time after a new layer has been built.
- **Step 2** separates the edges in `lastLayerEdges` into two groups: those that end in the current layer and those that span over it. Edges, whose successor node is in `nodes`, end here and are moved to `edgeTowardsParents`. The remaining edges in `lastLayerEdges` span over the current layer and are handled in step 4.
- **Step 3** adds all edges that start at the current layer to `edgesTowardsChildren`. The helper structure `children` was built for this purpose; the edges in question can be retrieved from that structure by simply making a lookup for all current members of `nodes`.
- **Step 4** is concerned with handling long edges and dummy nodes. All edges, that are still in `lastLayerEdges`, start at a node, that has been placed in an earlier layer while their end node has not been assigned a layer yet and will thus be placed in a later layer. These edges span over the current layer and need to be split. For that purpose, a new dummy node is created and added to `nodes`, and two resulting halves of the edge are added to `edgesTowardsParents` and `edgesTowardsChildren` respectively.

The result of this stage of the algorithm is a layering of the graph, where the nodes in the layers have no particular order. The assignment of nodes to layers is not changed in the remaining steps.

6.1.3 Stage 2: Reduce Edge Crossings

The goal of the second stage of the layered graph layouting algorithm is to minimize the number of edge crossings [54]. This is achieved by reordering the nodes in the layers.

Garey and Johnson [58] have shown this problem to be NP-complete, so finding an optimal solution is in general considered to be intractable [49], [55], [59]. Multiple approaches to tackle this problem have been proposed. Sugiyama, Tagawa, and Toda [54] describe two methods: *penalty minimization* and the *barycentric method*, a heuristic approach that tries to position every node such that it lies in the center of all nodes, that it is connected with. Dresbach [60] propose a stochastic heuristic. Healy and Nikolov [61] employ Integer Linear Programming (ILP) to optimize the number of edge crossing. Slade [49] put forward a method, that can handle additional constraints, which makes it suitable when the particular application domain has specific requirements with regards to the layout.

For the prototype, a procedure based on the barycentric method as described by Sugiyama, Tagawa, and Toda [54] was implemented. The method consists of two phases.

Phase 1 consists of the *Down-Up-Procedure*, which performs a down sweep on the layered graph, followed by an up sweep. In a down sweep, each layer's nodes are ordered with regards to the layer above it, beginning with the second layer from the top. An up sweep works analogously in the other direction, by ordering the nodes of a layer with regards to the next layer below, beginning with the second layer from the bottom. This procedure is applied in multiple iterations until either one iteration does not change the order anymore or a predetermined maximum number of iterations has been performed.

The ordering of the nodes in a layer is determined by their barycenter with regards to one neighboring layer. In the down sweep, that is the upper neighbor, while in the up sweep, it is the lower neighbor. Sugiyama, Tagawa, and Toda [54] formulate the barycenter calculation using a matrix. The matrix represents the presence of edges between the nodes of two adjacent layers. In this matrix, the rows represent the nodes of the upper layer and the columns the nodes of the lower layer. Each position a_{ij} of the matrix represents the presence of an edge between the node i of the upper and node j of the lower layer; when there is an edge between these nodes, then $a_{ij} = 1$, otherwise $a_{ij} = 0$. The row barycenters B^r and the column barycenters B^c , are then given by the equations 6.1 and 6.2, where V_u refers to the set of nodes in the upper layer, V_l refers to the set of nodes in the lower layer, and $x(k)$ represents the current position of node k in its layer.

$$B_i^r = \frac{\sum_{j \in V_l} a_{ij} x(j)}{\sum_{j \in V_l} a_{ij}}, \quad i \in V_u \quad (6.1)$$

$$B_j^c = \frac{\sum_{i \in V_u} a_{ij} x(i)}{\sum_{i \in V_u} a_{ij}}, \quad j \in V_l \quad (6.2)$$

During the down sweep, the nodes of the lower layer are reordered according to the column barycenters, while in the up sweep, the row barycenters are used to reorder the nodes of the upper layer.

Phase 2 reorders nodes that have the same barycenter. If the last executed part of phase 1 was a down sweep, then a phase 2 down sweep performed, where the order of nodes that have the same column barycenter is inverted. Likewise, if the last phase 1 sweep was an up sweep, a phase 2 up sweep is done using the row barycenters.

The implementation used for the prototype slightly differs from described procedure in details. In phase 1, after each layer, the number of edge crossings both before and after the reordering are calculated. The result is only taken, if the number of crossings after the reordering is less than before it. After each down sweep of phase 2, that did change the order of some nodes, both a down and an up sweep of phase 1 are performed. As

soon, as one sweep does not yield a change, the procedure is terminated. The maximum number of iterations used for both phase 1 and 2 is 10.

The result of this stage is an ordering of the nodes in the layers of the layered graph, with the goal of minimizing crossings of the edges. The node order obtained here is not changed in later stages of the algorithm.

6.1.4 Stage 3: Improve Readability

The goal of the third stage of the graph layering algorithm is to maximize the *readability* of the graph. This is done by adjusting the position of the nodes inside a layer without changing their order. The criteria for readability that Sugiyama, Tagawa, and Toda [54] list are:

- Straightness of edges
- Closeness of connected vertices to each other
- Balanced layout of in- and outgoing edges of a node

This process may create holes between the nodes of the same layer. Sugiyama, Tagawa, and Toda [54] describe two methods: one based on quadratic programming and the heuristic *priority layout method*. The latter was chosen for the prototype. It has some similarities to the barycentric method of stage 2: it works in up and down sweeps and it employs the concept of the barycenter.

The initial position that is assigned to each node is its order number, as it was assigned in stage 2. Similar to stage 2, there are two procedures that improve the position of nodes: DOWN and UP. The DOWN procedure improves the position of the nodes of a layer with respect to their connections to the next layer above; analogously the UP procedure does so with respect to the next layer below. They are applied to the individual layers in the following order, where 1 is considered the top-most and n the bottom-most layer:

1. DOWN for the layers from 2 down to n
2. UP from layer $n - 1$ up to layer 1
3. DOWN again from some layer t down to layer n , where t is some predetermined layer number 2 and $n - 1$.

The DOWN procedure works as follows: First, the nodes of the layer are given a priority. For non-dummy nodes, this is the number of connections they have to the layer above; dummy nodes are assigned the highest priorities, such that they are processed first. Then each node is placed, in the order of their priority, so that each node is placed as close as possible to its upper barycenter, no two nodes have the same position, and the node order

of the layer, that was determined in stage 2, stays unchanged. UP works the same except with respect to the layer below and the lower barycenter. The upper and lower barycenter are defined essentially the same as the column and row barycenter, respectively, in stage 2.

The implementation has some slight variations in comparison to the description given by Sugiyama, Tagawa, and Toda [54]. For example, because the nodes should be drawn aligned to a grid and edge chains corresponding to a branch in the Git history should optimally be completely straight, all layers are given the same width w_{\max} —the same number of positions for nodes. Naturally, this width needs to be bigger than the maximum number of nodes in a layer. For this purpose a user-adjustable width factor $u_{\text{width}} \geq 1$ was introduced and the width of the layers calculated by the following formula:

$$w_{\max} = \lceil u_{\text{width}} \cdot \max(\{|L| \mid L \in \text{layers}\}) \rceil \quad (6.3)$$

During a down sweep, the node positions of the layers are improved by applying the procedure DOWN1 to each layer, with respect to the layer directly above. For the up sweep the corresponding procedure UP1 is used, which works analogously but with respect to the layer below. The DOWN1 procedure is outlined as pseudo-code in Algorithm 6.2 The nodes are positioned in the order of their priority. Dummy-nodes are assigned a priority that is 1 plus the number of nodes in the other layer. This satisfies the requirement posed by Sugiyama, Tagawa, and Toda [54] that the dummy nodes should be processed first. The nodes are assigned a position that is as near as possible to the barycenter within the following additional constraints:

- The position must be smaller than the maximum layer width w_{\max} .
- The position must be less than any already placed node with higher order number.
- The position must be higher than any already placed node with lower order number.

The latter two conditions ensure that the node order after position improvement is still the same order that was assigned in stage 2.

Because the layer width w_{\max} can be wider than the number of nodes in the layer, it is possible that, after the up and down sweeps have been performed, the nodes are aligned more to the right, that is, the smallest node position across all layers is bigger than 0. Because this would lead to an unnecessary gap between the graph and the edge of the viewport, the minimum node position is subtracted from the position of each node at the end of the stage.

Algorithm 6.2: Improving node positions of one layer during a down sweep

```

Procedure: DOWN1
Input: nodesOfLayer, maxWidth
Result: nodePositions

  /* Step 1: Assign initial position ranges to nodes. */
1 let  $o_{max} \leftarrow \max \{ \text{order number of } (n) \mid n \in \text{nodesOfLayer} \}$ 
2 let positionRanges  $\leftarrow \emptyset$ 
3 forall  $n \in \text{nodesOfLayer}$  do
4   let  $o_n \leftarrow \text{order number of } (n)$ 
   /* Initial lower bound of position range is the nodes order number, so all
   nodes to the left still can fit in any case. */
5   let  $l \leftarrow o_n$ 
   /* Initial upper bound of position range leaves minimal space for all nodes to
   the right. */
6   let  $u \leftarrow \text{maxWidth} - (o_{max} - o_n)$ 
7   positionRanges[ $n$ ]  $\leftarrow [l, u]$ 

  /* Step 2: Assign positions in order of priority. */
8 let nodePositions  $\leftarrow \emptyset$ 
9 forall  $n \in \text{nodesOfLayer}$  in descending order of priority ( $n$ ) do
   /* See equation 6.2 for definition of column barycenter  $B_n^c$  */
10  let  $p \leftarrow \text{clamp}(\text{round}(B_n^c), \text{to} \leftarrow \text{positionRanges}[n])$ 
11  nodePositions[ $n$ ]  $\leftarrow p$ 
   /* Adjust position ranges of remaining nodes. */
12  forall  $k \in \text{nodesOfLayer} \mid \text{unset}(\text{nodePositions}[k])$  do
13    let  $r \leftarrow \text{positionRanges}[k]$ 
    /* A minimal distance  $d$  must be reserved based on the order numbers,
    otherwise nodes with order numbers in between might have no space left
    */
14    let  $d \leftarrow \text{abs}(\text{order number of } (k) - \text{order number of } (n))$ 
15    let  $u \leftarrow \min(\text{upper bound}(r), p - d)$ 
16    let  $l \leftarrow \max(\text{lower bound}(r), p + d)$ 
17    positionRanges[ $k$ ]  $\leftarrow [l, u]$ 

```

The result of this stage is an improvement in the positioning of the nodes, which are aligned to a grid. The edges, particularly long edges spanning multiple layers, are straightened to make the graph more easily readable. The logical part of the layouting process is now complete; the last stage is only concerned with the graphical display of the graph.

6.1.5 Stage 4: Displaying of the Graph

The final stage of the layered graph layouting algorithm, as presented by Sugiyama, Tagawa, and Toda [54]. The version that was implemented in this iteration displays the graph in the traditional top-to-bottom manner, with nodes being displayed as small circles and edges as lines. Dummy nodes are not shown at all. The final implementation, that lays out the graph as described in Chapter 4 was done in iteration 2. The only purpose of the version, that was implemented in this iteration, was to test the first three stages of the algorithm and is therefore only described briefly. It was implemented before stages 2 and 3.

The implementation was based on SVG. The nodes are drawn as circles, the coordinates of which are derived from the position of the node in the layered hierarchy: the Y-coordinate is calculated from the number of the layer, the X-coordinate from the position inside the layer. The edges are drawn as lines, where the start and end coordinates are taken from the positions of the corresponding nodes. To make the nodes distinguishable from each other, the node circles were given their commit hash as title, which is displayed when the mouse pointer is hovered above the circle.

The result was a simple dot-and-line graph display. It served to test the three other stages of the graph display algorithm before implementing the final layout as described in the concept. Figure 6.1 shows two examples.

6.2 Iteration 2: Graph Layout as Part of the Visualization

In the previous iteration, the first three out of four stages of the graph layouting algorithm were implemented, with a simple implementation for the drawing stage that was purely for testing the logical layout process. In this section, the final implementation of the fourth stage for the purposes of the prototype is described.

Features and problems tackled in this iteration were:

- The graphical layouting of commit nodes (related to Feature F.01 and Feature F.02)
- The graphical layouting of commit graph edges as individual segments between layers (related to Feature F.01 and Feature F.02)

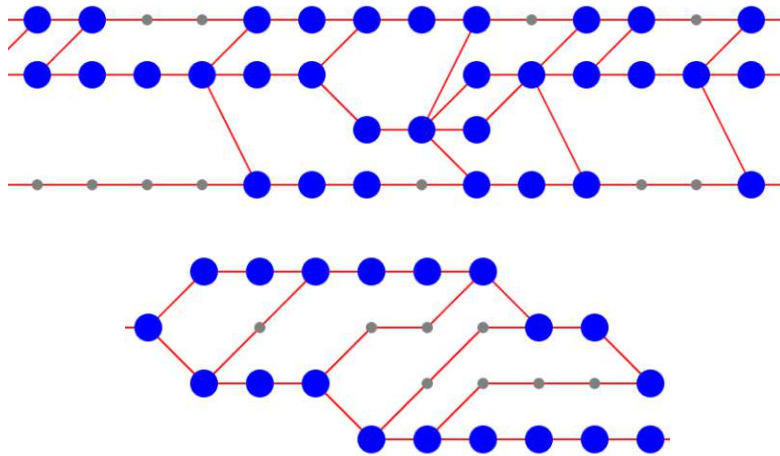


Figure 6.1: Simple visualization of two results of the graph layering. The blue circles represent commits, the gray dots are dummy node, the red lines are the segmented edges. The graphs are shown rotated from vertical to horizontal.

6.2.1 Overview

The display stage can be summarized as follows:

- Take the layered graph that was produced by stage 3 as described in Section 6.1.4
- Lay the commits out on a grid, where the commits are displayed as blocks with text.
- Connect the commit blocks corresponding to their parent-child relations in the history graph. The connections are displayed as translucent blocks.

Traditionally, layered graphs are laid out from top to bottom such that each layer forms a line. In the context of this visualization however, the graph is laid out from left to right such that layers are displayed as columns with the root commits being drawn as the left-most layer.

To maintain consistency with the terms used in the field of layered graphs, and reduce potential confusion, the word “*layer*” will be used to refer to a group of nodes arranged in the same column, which corresponds to a layer in the logical, layered graph. When referring to a horizontal line in the displayed graph, the term “*plane*” will be used instead of “*layer*”.

The implementation of the display stage is structured in two phases: placing the commit nodes and connecting them with edges. A discussion of these phases follows.

6.2.2 Commit Nodes

In the first phase, commits are laid out in a grid, with earlier commits to the left and later commits to the right. Commits that are in the same layer—and therefore from a logical perspective at the same time—are put on different planes. A single branch thus appears as a horizontal chain of connected commits. Parallel branches are shown as a vertical stack of such commit chains.

To achieve the appearance of a three-dimensional space, planes in the back are positioned towards the top, while the planes in the front are placed further below at a fixed offset, overlapping the planes behind them. The commits blocks in each plane also have a slight offset on the horizontal axis compared to the commits on adjacent planes. This is done so that the code in the commit blocks on planes further back is not completely hidden by the commit blocks in the front and to simulate a slight viewing angle relative to the vertical layers.

Ordinary commit blocks contain as text the content of the file at the time of that commit over a white background. To better set off the text from the surroundings, the block has a thin border colored in a subtle gray. Also, the blocks cast shadows on the background to enhance the illusion of three-dimensionality. Dummy node blocks are styled differently from proper node blocks. As they are technically not nodes and but part of an edge, they are displayed the same way as edges.

The implementation of the layout was done as much as possible in pure CSS. The goal was to keep the implementation simple. Doing the necessary calculations in JavaScript would mean, that the calculations would have to be redone each time some variable changes, and the corresponding layout properties would have to be updated manually. In contrast, CSS is a declarative language and the browser automatically evaluates the expressions when appropriate, even when the modern CSS function `calc()` is used to embed calculations directly into the style sheet.

First, a few common constants were defined to make the calculation expressions easier to read and understand, and to make it easier to tweak the values later on. These were:

- the horizontal gap between adjacent commit block in the same plane (d_h)
- the vertical gap between the upper edges of adjacent planes (d_v)
- the horizontal offset between commits in the same layer of adjacent planes (Δ_h)
- the width and height of the commit blocks (w and h respectively)

The position of a commit block is calculated as a function of the numbers of the layer and the plane the commit is part of in the layered graph. Let l be the layer number and p the plane number, with both numbers zero-based, then the coordinates x and

y of the upper-left corner of the commit block—in CSS the properties `left` and `top` respectively—are calculated as

$$x = l(w + d_h) + p\Delta_h \quad (6.4)$$

$$y = pd_v \quad (6.5)$$

In addition to the two-dimensional position properties, the Z -index and a translation in Z direction are also defined. This is to ensure, that overlapping commit blocks are displayed with the ones in the front that are also logically foremost ones, as well as to make the commit blocks interact correctly with the diff connection blocks described in Section 6.2.3. The Z -index, which determines the Z -order of elements as part of ordinary HTML layouting, is set to 1000 times the plane number, where plane 0 is the one furthest in the back. The factor 1000 is added, so that there is enough space to place the diff connections between the commit layers, as the Z -index must have an integral value.

For the Z -translation, the concrete value chosen is not particularly important, as the Web browser uses a parallel projection to bring the 3D-elements to a plane. As a consequence the Z -position has no effect on the way the element is rendered by itself; its only purpose is to make elements obscure and intersect correctly with each other. To achieve that, it is only required to keep the Z -positions of the elements consistent relative to each other. Therefore, the Z -translation was chosen to be equal to the Y -position of the element.

```

--gdv-node-position-top: calc(
  var(--gdv-commit-vdist) * var(--gdv-node-plane)
  + var(--gdv-graph-margin)
);
width: var(--gdv-commit-width);
height: var(--gdv-commit-height);
left: calc(
  (var(--gdv-commit-width) + var(--gdv-commit-hdist)) * var(--gdv-node-layer)
  + var(--gdv-commit-offset-per-plane) * var(--gdv-node-plane)
  + var(--gdv-graph-margin)
);
top: var(--gdv-node-position-top);
/* Z-index can only be an integer. If anything else is assigned it is rounded.
 * Therefore, apply a factor of 1000, to mitigate problems with z-order between commit
   ↪ nodes and diff edges */
z-index: calc(var(--gdv-node-plane) * 1000);
transform: translateZ(calc(var(--gdv-node-position-top)));

```

Listing 6.1: CSS for positioning commit graph node blocks

The corresponding CSS is provided in Listing 6.1. The value for the `top` property is factored out into the variable `--gdv-node-position-top` because it is also used in the transformation. Additionally, a `transform-style` of `preserve-3d` is required on the common

6. IMPLEMENTATION

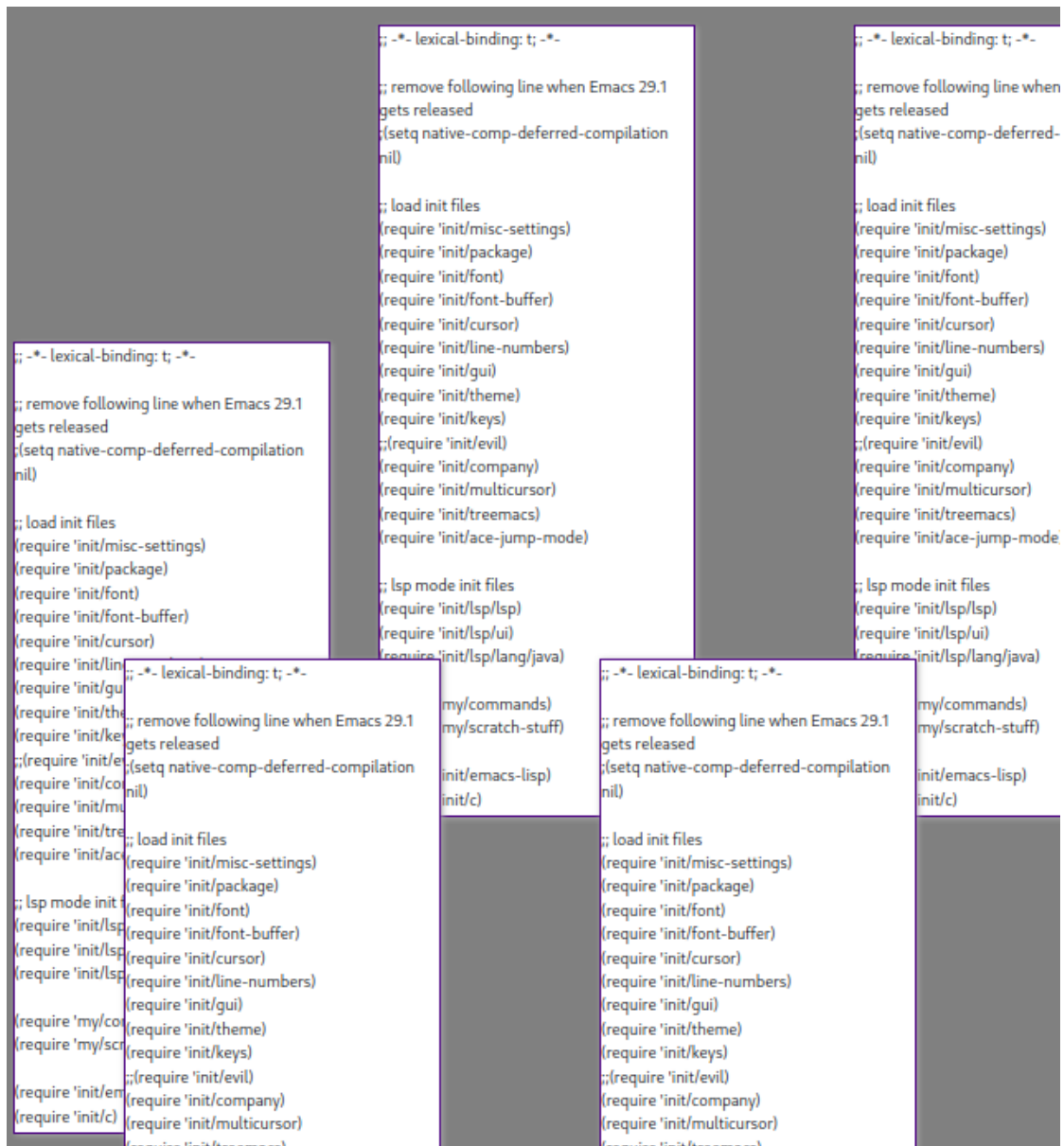


Figure 6.2: Example of commit graph layout without edges.

parent element, to make intersecting elements in the graph be displayed according to their positions and transformations in the 3D-space rather than simply one being in front of and covering the other. An example screenshot of the state described here is presented in Figure 6.2.

6.2.3 Connections of Related Commits

Phase 2 of the graph drawing phase is responsible for the edges that correspond to the parent-child relations between the commits. The edges are drawn as block-displayed HTML elements. To display these edge elements correctly and to simulate a three-dimensional space, multiple properties of the block must be set and transformations applied correctly and in concordance with each other:

- width
- two-dimensional position
- rotation
- skew
- Z -index
- Z -position

How these properties are made to work together to produce the desired appearance is discussed in the following. Part of the CSS code for positioning and transforming the diff edge blocks is reproduced in Listings 6.2 and 6.3.

There are multiple possible ways for the assignment of the two-dimensional position, depending on how the other aspects—in particular the rotation and skew transformations—are implemented. The method chosen for this prototype is to position the block in such a way that its center lies half-way between the right edge of the block of the parent commit and the left edge of the block of the child commit. When this is the case, the rotation and skew transformations can be applied relative to an axis that goes through the center point of the block, which is the default for CSS transformations.

The X and Y coordinates of the top left corner of the block were calculated according to following formulas, where w is the width of the connection block, x_p, y_p are the X and Y coordinates of the right upper corner of the parent commit and x_c, y_c are the coordinates of the left upper corner of the child commit:

$$y = \frac{y_c - y_p}{2} \quad (6.6)$$

$$x = \frac{x_c - x_p}{2} - \frac{w}{2} \quad (6.7)$$

This takes the arithmetic mean of the connected commit block coordinates, to get the center point of the line segment between them, and subtracts half the connection block width from the X coordinate, to shift the center of the block to this point.

To calculate the width and rotation of the edge block, the space between the two connected commits can be imagined as a right triangle, as shown in Figure 4.10. The short sides of this triangle correspond to the vertical and horizontal distances between the commit blocks, while the hypotenuse and the angle α represent the width and rotation, respectively, of the connection edge block.

Although using the Pythagorean theorem to calculate the hypotenuse would be close at hand, at the time of writing Web browsers did not yet support the square root function in

CSS [W70]. Therefore, the sine law was used for this purpose instead. The specifics have already been discussed in Paragraph *Calculations for the Transformations of Connections* in Section 4.3.3.

When using the rotation angle α directly—together with its opposing side, the height vertical gap between the commit blocks—to calculate the hypotenuse of the triangle, the common case, where two commits on the same plane are connected and α becomes 0, leads to a division by 0 and thus an undefined result. To circumvent this, the calculation was instead implemented using the third angle $\beta = 90^\circ - \alpha$ and its opposing side, the horizontal gap. Therefore, when w is the width of the connection block, x_p, y_p are the x and y -coordinates of the right upper corner of the parent commit and x_c, y_c are the coordinates of the left upper corner of the child commit, the calculation used to obtain the width of the edge block element is:

$$\begin{aligned} w &= \frac{x_c - x_p}{\sin \beta} & (6.8) \\ &= \frac{x_c - x_p}{\sin(90^\circ - \alpha)} \\ &= \frac{x_c - x_p}{\sin(90^\circ - \arctan \frac{y_c - y_p}{x_c - x_p})} \end{aligned}$$

Listing 6.2 shows how this calculation has been implemented in CSS.

```
--gdv-edge-rotation-angle: atan2(var(--gdv-edge-end-y) - var(--gdv-edge-start-y), var(--
  ↪ gdv-edge-end-x) - var(--gdv-edge-start-x));

/* Use sine rule with angle at the top of the triangle to obtain hypotenuse.
 * The rotation angle is not used directly, as it might be 0, in which case the
  ↪ expression value would be undefined.
 */
--gdv-edge-width: calc(
  (var(--gdv-edge-end-x) - var(--gdv-edge-start-x))
  / sin(90deg - var(--gdv-edge-rotation-angle))
);
--gdv-edge-width-half: calc(var(--gdv-edge-width) / 2);

--gdv-edge-position-top: calc(
  (var(--gdv-edge-start-y) + var(--gdv-edge-end-y)) / 2
);
```

Listing 6.2: Calculation of rotation angle and edge element width in CSS

To achieve the appearance of a rotation of the block in a three-dimensional space, two CSS transformations were applied: a rotation and a skew around the Y -axis.

The rotation is necessary to give the block the angle in space, that becomes relevant for the correct effect when two such connection blocks cross. Furthermore, it causes the apparent width to be scaled in such a way that it fits into the gap between the commit blocks. However, after the rotation, the block still has strictly horizontal upper and lower edges.

The skew provides the visual part of the rotation. After the skew has been applied, the upper and lower edges connect with the respective edges of the commit blocks and the content of the connection block is transformed in the same way. Note that although both the skew and rotation transformations logically use the same angle, in the specification of the transformations in CSS, it was necessary to invert the angle for the rotation part.

To make overlapping and crossing blocks look correct, it is necessary to also give them the correct position, relative to each other, in the Z -dimension. This requires two components: the Z -index and a translation in Z direction as part of the transformation. As with commit code blocks, the translation in the Z -direction is set equal to the Y -position. Let p_p, p_c be the plane numbers of the parent and child node respectively, the Z -index is calculated as the arithmetic mean of the parent and child planes multiplied by 1000:

$$i_z = 1000 \frac{p_c + p_p}{2} \quad (6.9)$$

The factor of 1000 is added in accordance with how commit code blocks are positioned in Section 6.2.2.

Listing 6.3 shows the CSS code for the position and three-dimensional transformation. The helper variables for the position coordinates (`--gdv-edge-start-x, ...`) are omitted; their calculation is very similar to the position of the commit nodes in Listing 6.1.

```
width: var(--gdv-edge-width);
left: calc(
  (var(--gdv-edge-start-x) + var(--gdv-edge-end-x)) / 2
  - var(--gdv-edge-width-half)
);
top: var(--gdv-edge-position-top);
z-index: calc(
  (var(--gdv-edge-parent-plane) + var(--gdv-edge-child-plane)) / 2
  * 1000
);
transform: translateZ(calc(var(--gdv-edge-position-top)))
  skewY(var(--gdv-edge-rotation-angle))
  rotateY(calc(-1 * var(--gdv-edge-rotation-angle)));
```

Listing 6.3: CSS for positioning and transforming graph edge blocks

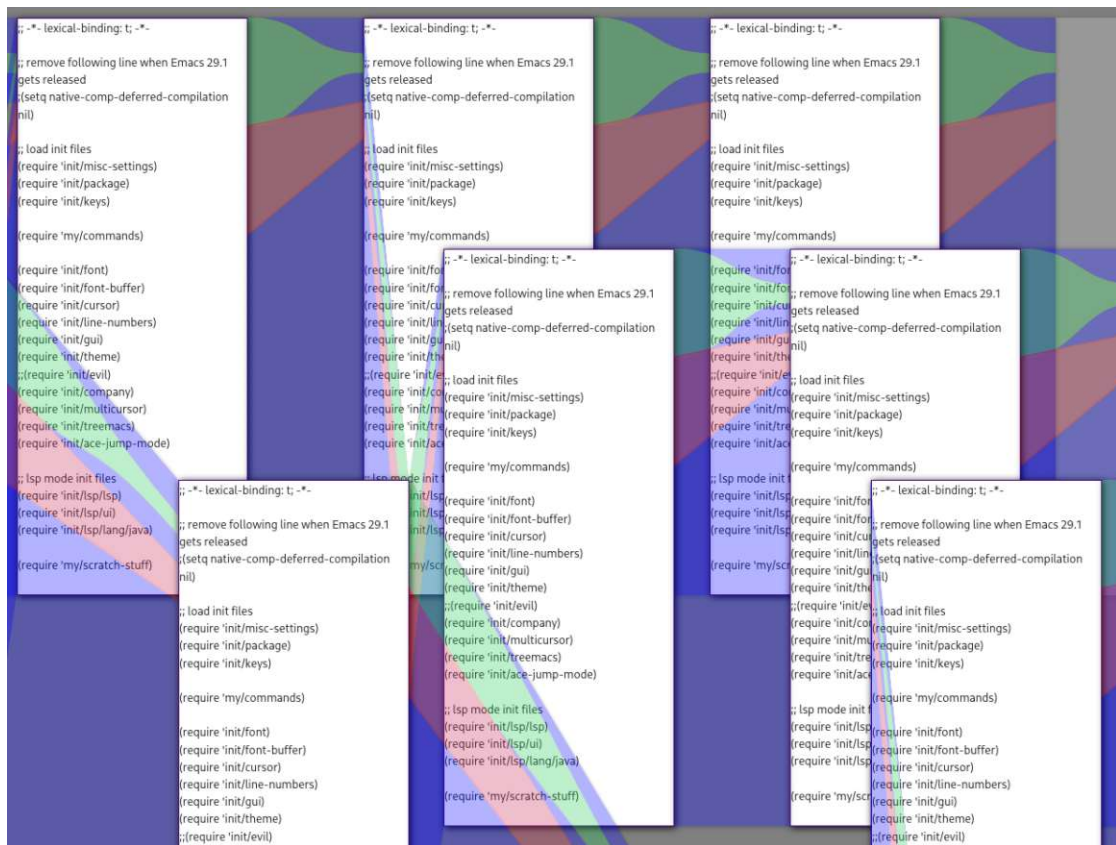


Figure 6.3: Example of commit graph layout with text blocks. Difference indications are dummies.

The edges are styled with the goal to be clearly visible but not obstructive, so objects behind them stay clearly visible and text can still be read through the edge blocks. To achieve this, they are colored light gray with an opacity of 0.3. In comparison to commit nodes the edges have a margin at the top bottom which is as wide as the border on commit nodes and a less pronounced shadow. Dummy nodes are styled the same way as edges, as they are technically not real nodes but merely exist to split long edges into segments.

6.2.4 Result

In this section, the graph layouting for the visualization prototype was developed. The dimensions and required calculations for the positioning of blocks corresponding to commit nodes have been described. Furthermore, the details of the positioning and transformation of the edges have been discussed. An example of the visualization at the end of the visualization is presented in Figure 6.3.

6.3 Iteration 3: Diff Indicators

In this iteration, the graphical indication for changes between commits was developed. The implementation is split across the frontend and the backend. The backend extracts the required data from the repository and analyses it while the frontend presents a graphical representation to the user.

Features and problems tackled in this iteration were:

- F.03 (partially): Simple difference indications as curves on single-segment edges. Correct handling of edges spanning multiple layers was added later, in Iteration 8.

6.3.1 The Backend

In order to display the difference indicators, the changes between the involved commits have to be extracted and analyzed. This could have been implemented in the frontend since the file text in the commit is sufficient information. Despite this, it was decided to do the required preprocessing in the backend for performance and simplicity of implementation.

The analysis of the changes is split into three steps:

- Diff the two versions of the file
- Split the diff into hunks
- Parse each hunk

LibGit2Sharp already provides a function to create a diff of two files in the same format that the command `git diff` would produce. A diff consists of a header and one or more hunks. The first few lines of the diff belong to the header that provides additional information about the diff itself, such as, the time it was created and which files were compared. This information is not needed for the purposes of the diff indicators and is therefore discarded. The remaining part of the diff is split into its hunks. Each hunk starts with a header line. Such hunk header lines can be identified, because they are the only lines in a diff that start with the character “@”. The following line gives an example for a hunk header line:

```
@@ -1,9 +1,10 @@
```

The “@” characters are only a signature and have no further meaning. The first two numbers after the minus are the numbers of the first and last line in the hunk before the changes in the preceding commit. The latter two numbers after the plus likewise are the numbers of the first and last line in the hunk after the changes in the successor commit.

The hunk header line is parsed by applying the following Regular Expression (Regex):

$$\text{\textasciitilde}@@ \text{\textasciitilde}-(?<start>\text{\textbackslash}d+),\text{\textbackslash}d+ \text{\textasciitilde}+(?<end>\text{\textbackslash}d+),\text{\textbackslash}d+ @@$$

The parts $(?<start>\text{\textbackslash}d+)$ and $(?<end>\text{\textbackslash}d+)$ are named capture groups in this particular Regex language, that both match a number as a string of digit characters. The other two numbers are matched for the sake of parsing the line but are ignored otherwise. The remaining lines of the hunk form the body.

Each hunk is parsed into a set of changes. A change consists of:

- **A type:** Either *insertion* or *deletion*, marked by “+” or “-”, respectively.
- **The before interval:** The interval of line numbers that the change applies in the predecessor version.
- **The after interval:** The interval of line numbers that the change applies in the successor version.

This change set is built in two steps. First, the start line numbers from the hunk header and the hunk body lines are fed into the procedure CONVERT HUNK LINES which is outlined in Algorithm 6.3. This procedure parses the hunk line by line. Lines that indicate a change are converted into single-line change objects; lines beginning with a “+” character are insertions while a “-” character denotes a deletion. Any other lines are just context and are ignored here. Two counters that keep track line number before and after the changes are continuously updated, so that the correct line numbers are known for creating the change objects.

Afterwards, the change set produced in the previous step is fed into another procedure that groups those single-line changes which form one bigger change together. Two changes A and B are merged when three conditions hold:

- A and B have the same type.
- The upper end of the after interval of A is the same as the lower end of the after interval of B .
- The upper end of the before interval of A is the same as the lower end of the before interval of B .

When two changes are merged, a new change of the same type and merged intervals is created.

This process is applied to all parent-child commit relations in the repository history graph. The resulting change sets are included in the data returned to the client as part of the corresponding edges.

Algorithm 6.3: Parsing diff hunk lines**Procedure:** CONVERT HUNK LINES**Input:** hunkLines, startLineBefore, startLineAfter**Result:** changes

```

1 let currentLineBefore ← startLineBefore
2 let currentLineAfter ← startLineAfter
3 let changes ← ∅
4 forall l of hunkLines do
5     switch l[0] do
6         case '+' do
7             /* An insertion.
8              This line only exists after the change. */
9             let i ← change (type ← '+',
10                before ← [currentLineBefore, currentLineBefore),
11                after ← [currentLineAfter, currentLineAfter + 1))
12            changes ← changes ∪ {i}
13            currentLineAfter ← currentLineAfter + 1
14         case '-' do
15             /* A deletion.
16              This line only exists before the change. */
17             let i ← change (type ← '-',
18                before ← [currentLineBefore, currentLineBefore + 1),
19                after ← [currentLineAfter, currentLineAfter))
20            changes ← changes ∪ {i}
21            currentLineBefore ← currentLineBefore + 1
22         otherwise do
23             /* Just a context line. Nothing changed. */
24             currentLineAfter ← currentLineAfter + 1
25             currentLineBefore ← currentLineBefore + 1
26
27 return changes

```

6.3.2 The Frontend

The drawing of the difference indicators was implemented as part of a dedicated component. The indicators are drawn with SVG. The height of the SVG element is set to the maximum of heights of the commit text nodes on either side. Furthermore, the viewBox of the element, that defines the coordinate system for the SVG paths, is defined so that the horizontal position 0 is on the left edge and 100 on the right edge of the element. This simplifies the process of generating the SVG paths, as the physical width of the host element does not matter.

Each change object, as received from the backend, is converted to an SVG path. How this path is constructed is explained in the following with the aid of Figure 6.4. The coordinates of the points are described both in text as well as with formal notation, using the following definitions:

- h_{line} : The height of one code line in the adjacent commit blocks
- I_{before} : The interval representing the changed lines in the parent commit
- I_{after} : The interval representing the changed lines in the child commit
- $lower(I)$: The lower bound of interval I . Represents the number of the first changed line.
- $upper(I)$: The upper bound of interval I . Represents the number of the line after the last changed line.

The path starts at point P_{tl} in the top left corner. The horizontal position is 0; the vertical position is the line height multiplied with the line number that is the lower bound of the before interval.

$$P_{tl} = (0, lower(I_{before}) \cdot h_{line}) \quad (6.10)$$

From there, it continues via a cubic Beziér curve to the right, ending at the top right corner point P_{tr} . The horizontal position is 100, the vertical position the line height multiplied by the lower bound of the after interval. The control points C_{t1} , C_{t2} for the curve are placed both at the horizontal middle position 50. The vertical position of the first control point C_{t1} is aligned with the left point P_{tl} , while the second control point C_{t2} is at the same vertical position as the right end point P_{tr} .

$$C_{t1} = (50, lower(I_{before}) \cdot h_{line}) \quad (6.11)$$

$$C_{t2} = (50, lower(I_{after}) \cdot h_{line}) \quad (6.12)$$

$$P_{tr} = (100, lower(I_{after}) \cdot h_{line}) \quad (6.13)$$

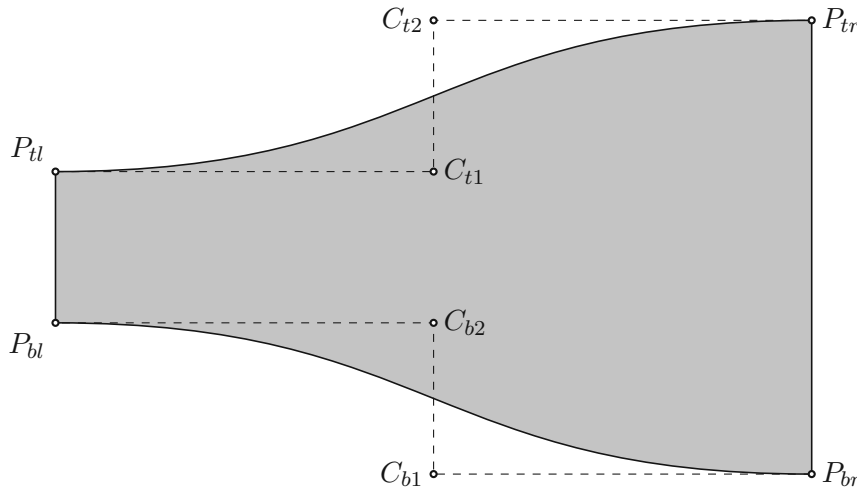


Figure 6.4: Points defining the curves used to display the visual difference indications. The dashed lines mark the control polygons of the two Beziér curves.

The next part of the path is a straight line along the right edge, from the top right point P_{tr} down to the lower right corner P_{br} , which is the upper bound of the after interval times the line height. This is followed by a second cubic Beziér curve back to the lower end of the before interval on the left side, point P_{bl} . The control points C_{b1} and C_{b2} of this curve are positioned analogously to the control points C_{t1} , C_{t2} of the upper curve described above. Finally, the path is closed with a straight line on the left edge, from the bottom left corner P_{bl} up to point P_{tl} the top left corner.

$$P_{br} = (100, \text{upper}(I_{after}) \cdot h_{line}) \quad (6.14)$$

$$C_{b1} = (50, \text{upper}(I_{before}) \cdot h_{line}) \quad (6.15)$$

$$C_{b2} = (50, \text{upper}(I_{after}) \cdot h_{line}) \quad (6.16)$$

$$P_{bl} = (0, \text{upper}(I_{before}) \cdot h_{line}) \quad (6.17)$$

The appearance of the paths is controlled via CSS. Depending on the type of the change—addition or deletion—the path is assigned a style class, which defines the appropriate colors for the difference indicator. The color used for additions is green while deletions are colored red. The fill color, which is used for the background is given an opacity smaller than 1, so that the indicators do not fully obstruct the parts of the history graph behind them. The line width for the border stroke is defined in a common style for all change paths.

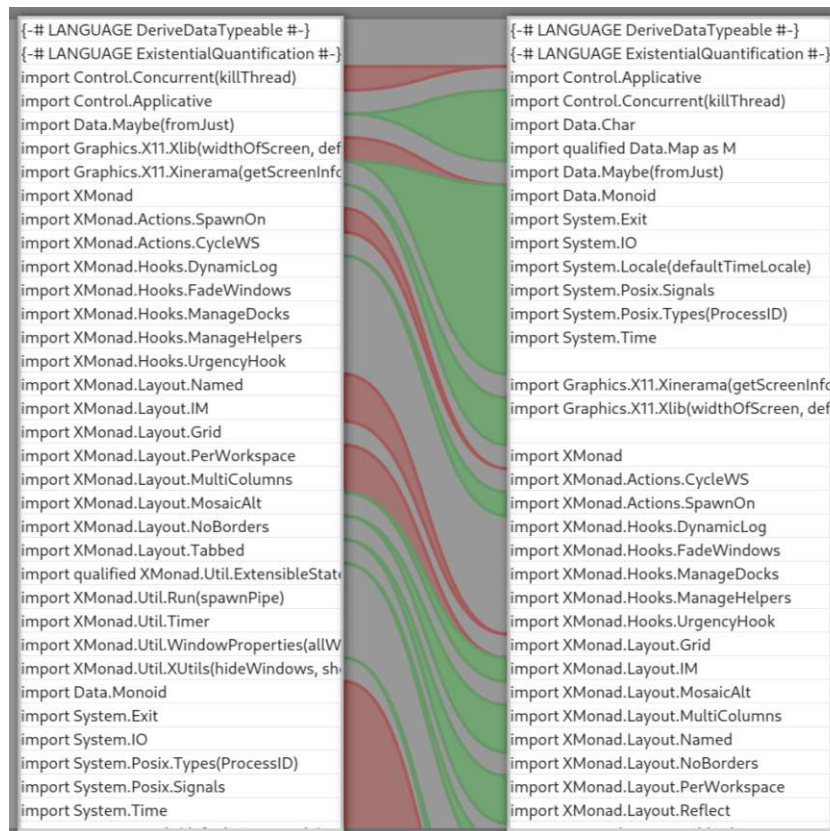


Figure 6.5: Indications of differences between two commits.

6.3.3 Result

In this iteration, the graphical indication of changes between commits was developed. The changes are shown as colored curves on the HTML block elements, that represent the graph edges. An example is given in Figure 6.5

At that point, the implementation could only handle edges between commits on adjacent layers. Long edges, which are displayed as segments of short edges and dummy node blocks, were shown incorrectly, with the diff curves identical in all edge segments, as can be seen in Figure 6.6. This shortcoming was corrected later during Iteration 8.

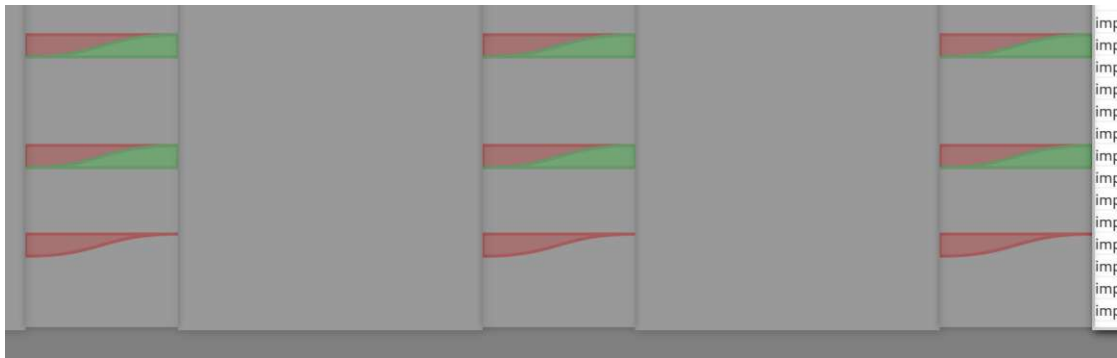


Figure 6.6: Long edge composed of multiple segments displays difference indications incorrectly.

6.4 Iteration 4: Scrolling of File Text

In this iteration, scrolling of the file text in the commit blocks was implemented.

Features and problems tackled in this iteration were:

- F.11: Scrolling
 - Synchronous scrolling of text in all commit.
 - Scrolling of text in individual commits.
 - Adjusting of difference indications to the scroll positions of the commit nodes.

This was done in two versions: scrolling in all commits for the same amount (F.11) and scrolling the text in a single commit.

6.4.1 Scrolling of All Commit Blocks

When the users scrolls up or down in one of the commit text blocks, the default behavior is to synchronously scroll all other text blocks by the same amount. This is achieved by handling the scroll event on the text block elements and adjusting the scroll position of all other commit node blocks and the connecting edges between them. Furthermore, each commit node is associated with an auxiliary object holding additional information like a reference to the corresponding HTML element.

When the scroll position of an HTML element is set programmatically, the scroll event for that element is also fired. If not addressed, this could result in an endless loop of scroll position updates and events. To prevent this, the function that sets the scroll position on a node also sets a flag `inhibitScroll` in the additional information object belonging to that node. This variable indicates, that the next scroll event received for the node should be ignored. It is initially set to `false` for every node. If a scroll event occurs with the `inhibitScroll` flag set, the event handler only resets this flag without any further processing.

Nodes are never scrolled further than the bottom of their text content. This is both intentional behavior of the prototype and a side effect of the way scrolling in HTML works. If the DOM property `scrollTop` would be set to a higher value than its maximum it is set to that maximum instead. Therefore, if one node is scrolled down more than another node has text, so that the scroll position of the scrolled node exceeds the maximum scroll position of the other node, the node scroll position of the node with the shorter text block is kept so that its last text line aligns with the block bottom.

6.4.2 Scrolling of a Single Commit Block

In contrast to scrolling all commit blocks at once, the prototype also allows the user to scroll the text of an individual commit. This function is activated by scrolling in a commit node while the *S*-key is pressed.

The propagation of the current scroll position to the other commit nodes and connecting edges is controlled by the flag `propagateScroll`. This flag is set by default. A pair of `keyup` and `keydown` event handlers on the root element of the graph view check if the *S*-key is pressed or released. When the key is held down `propagateScroll` is unset. In this case, the commit node scroll handler does not set the scroll position on the other nodes. Instead, the variable `scrollOffset` in the node information object is adjusted.

This `scrollOffset` is taken into account every time a scroll over all commits is done. When a scroll event on a commit node element is fired a “global scroll position” is calculated by subtracting the `scrollOffset` of the scrolled node from its scroll position as reported by the HTML DOM element property `scrollTop`.

$$\text{globalScroll} = \text{scrollTop} - \text{scrollOffset} \quad (6.18)$$

When the scroll update is propagated to the other nodes, their new scroll position is obtained by adding their `scrollOffset` to the global scroll position.

$$\text{scrollTop} = \text{globalScroll} + \text{scrollOffset} \quad (6.19)$$

If a node is scrolled towards the top, it might happen that the scroll position of another node with a lower `scrollOffset` becomes negative. As with scrolling a node past its maximum, this is ignored and the scroll position of the node is treated as 0 instead. If the `scrollOffset` of that node is negative, it is also increased so that the calculated and actual scroll positions of the node match. This way, a negative scroll offset can be “pushed away”. However, the value of `scrollOffset` is never changed to be bigger than 0 in that process.

6.4.3 Scrolling and Difference Indications

When commit text is scrolled the difference indications between the commits have to be adjusted to the new scroll positions of the involved text blocks. This was implemented in two parts: scrolling the diff indication block to the value of `globalScroll` and offsetting

the anchor points of the curves on either side by the `scrollOffset` of the respective commit block.

This apparently simple solution follows directly from the way scrolling is handled in the commit text blocks. However, it leads to edge cases which result in unaligned difference indication curves if not addressed. For example, when two adjacent commit nodes both have a positive `scrollOffset` and one of the two commits is scrolled towards the bottom, the calculated `globalScroll` would become negative. As HTML elements can not be scrolled to a negative position—the `scrollTop` property is automatically reset to 0—this leads to a difference in the calculated and actual positions of the curves. This particular edge case is accounted for by not letting `globalScroll` become negative and changing the `scrollOffset` variable of the scrolled commit node instead, as described in Section 6.4.2.

Some effort was put into correcting graphical glitches resulting from such edge cases, as deemed sufficient for the purposes of the prototype. Despite this, some situations remain where the difference indicator curves are not aligned correctly with the corresponding pieces of text. Comprehensive handling of all problematic cases or finding an overall simpler solution is a task for future work.

6.4.4 Results

In this iteration, scrolling (feature F.11) was implemented. All nodes can be scrolled synchronously as well as single nodes in isolation of the others. The edges between the nodes are scrolled as well and the difference indications transformed as necessary. Problems resulting from edge cases regarding the difference indications were identified and partially corrected.

6.5 Iteration 5: Simple Graphic Settings

In this iteration a UI was created for manipulating a number of aspects of the graphical presentation. This includes a zoom factor that acts as a multiplier for all dimensions in the graph visualization and thus implements both features F.07 and F.08.

Features and problems tackled in this iteration were:

- Making certain aspects of the UI user adjustable via a settings panel, such as gaps between commits and commit block size.
- F.07: Zooming in by changing the zoom factor in the settings panel.
- F.08: Zooming out by changing the zoom factor in the settings panel.

The UI was built as collapsible panel on the left side of the graph view. Contained inside this panel is a list of controls, each of which corresponds to one modifiable parameters of the presentation of the graph. The following parameters were implemented at that time:

- Line height
- Zoom factor
- Commit block width
- Commit block height
- Horizontal gap between commits
- Vertical gap between commits
- Horizontal offset between planes

Every parameter on this list is a floating point number and the same kind of control was used to make them editable. This control consists of two parts: a text box that displays the current value and can also be used to edit the value directly, and a slider. The slider was configured with a minimum value, maximum value and step size deemed to be sufficient for the parameter in question in the most common situations. Editing the value via the text box is cumbersome but allows an exact value to be set while the slider is quicker to use but less exact and limited to a certain range.

The implementation of the mechanism behind the parameter manipulation UI relies heavily on the automatic change detection that is provided as part of the UI framework. The settings panel is implemented as a separate Svelte component that is passed an object containing the parameter values. The fields of the object are bound to the UI controls and changes due to user input are automatically reflected in that object. Likewise, the graph view, that is the parent of the settings view and that passes to it this parameter object, binds the fields of the object to CSS variables in its template. The style sheet had previously already been written to accommodate this; the CSS variables had existed since the view was implemented in Iteration 2 but were fixed to default values. The formulas in style sheet of the graph view refer to these variables where appropriate, to calculate the various style properties of the view.

The Result of this iteration was a UI for changing aspects the graphical presentation of the graph in form of a collapsible settings panel. This UI enables the user to adjust the display of the graph to different requirements, depending on the situation. One aspect that can be controlled this way is a zoom factor which enables the user to zoom the graph in for more detail (F.07) and out for more overview (F.08). Figure 6.7 shows the expanded settings panel with a few of the controls.

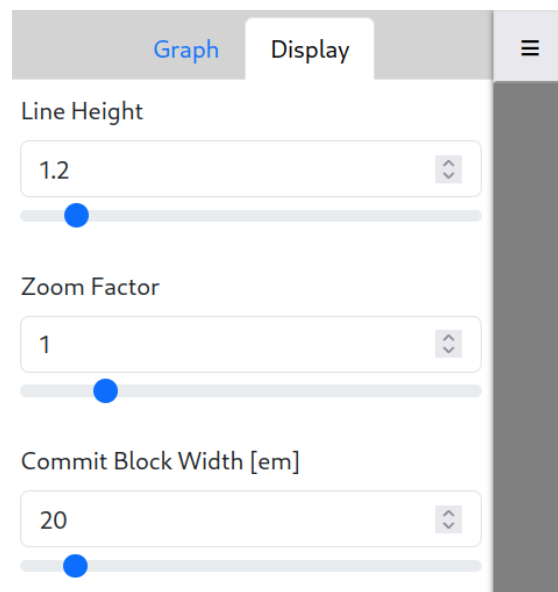


Figure 6.7: Part of the panel for graphical settings.

6.6 Iteration 6: Selecting Shown Branches

Following the implementation graphical settings UI, the panel was extended with a reference selection. This selection enables the user to choose which parts of the history graph should be shown. The graph is filtered to include only commits that are part of parent-child chains beginning from the selected references; commits that are not reachable this way are hidden.

Features and problems tackled in this iteration were:

- F.05 (partially): Showing and hiding branches via a selection from a list.

The UI for the branch selection was integrated into to the panel that was created as part of Iteration 5. The panel was split into two tabs: one for the existing graphical settings and one for the reference selection UI. The selection is built as an accordion—a container with collapsible segments—of four sections corresponding to four different kinds of references in Git: branch heads, remote tracking branch heads, tags, and stashes. For each reference, there is a check box in the appropriate section. If the check box is checked, the branch of the history identified by the reference will be included in the shown graph; if the check box is not checked, the reference will be ignored when building the graph.

The Mechanism behind the graph filtering process searches for all commit nodes that are reachable from the selected references by tracing the links from child commits to their parent nodes. The resulting graph G can be inductively described thus:

1. Every commit node that is directly referenced by a user selected reference, is in G .
2. Every commit node whose parent is in G , is also in G .

This is similar to the problem of finding all reachable objects as part of tracing Garbage Collection. McCarthy [62] first described an algorithm for automatic reclamation of Cons cells in the Lisp programming language [63]–[65]. This algorithm works in two phases: marking all reachable objects and then removing the unreachable ones. The procedure to filter the graph to the fragment that is reachable from the user selected references was based on the marking phase of such a Garbage Collection algorithm.

Algorithm 6.4 is a pseudo code outline of procedure GRAPH REACHABLE FROM in the prototype. It uses two helper data structures:

- a set *marked* that contains all nodes that have been marked, that is, which have already been determined to be reachable,
- and a set *remaining* for all nodes that have already been marked but not yet further processed.

First, the procedure takes the commits that are directly referred to by the user selected Git references, marks them—that is, adds them to the *marked* set—and also adds them to the *remaining* set. Afterwards, the algorithm goes into a loop in which it processes the found nodes one by one. In every iteration, one node n is removed from *remaining*. For every parent p of n , if p has not yet been marked, it is now marked and added to *remaining*. If instead p is already marked, nothing is done with it. This loop is repeated until *remaining* is empty.

For the *marked* set a JavaScript object was used where the key is the Git commit ID. The *remaining* set was implemented as a queue, although it should be noted that the particular order in which the nodes are removed from the set is not important; although a stack or an unordered structure would result in a different processing order, the resulting set of reachable nodes and number of loop iterations are nevertheless the same.

After the procedure terminates, the nodes in the *marked* set together with the edge set of the complete history form the filtered graph. Finally, this graph is laid out and displayed.

The Result of this iteration is the possibility to limit the history graph to a fragment reachable from a selected set of references. A UI was created for the branch selection and integrated into the settings panel from the previous iteration. In Figure 6.8 two examples are shown with the branch selection UI open: once with the whole, unfiltered history graph and once with only a few references selected for display.

Algorithm 6.4: Filtering the commit graph**Procedure:** GRAPH REACHABLE FROM**Input:** selectedRefs, commits**Result:** filteredCommits

```

/* Initialize remaining and marked to commits at selected references          */
1 let marked ← {c ∈ commits | id of(c) ∈ selectedRefs}
2 let remaining ← marked
/* Search for all nodes reachable from the currently marked ones.          */
3 while remaining ≠ ∅ do
  /* Remove a node from the “queue”                                        */
  4 let c ∈ remaining
  5 remaining ← remaining \ {c}
  /* Mark all parents of the node                                          */
  6 forall p ∈ parents of(c) | p ∉ marked do
  7   marked ← {c} ∪ marked
  8   remaining ← {c} ∪ marked
9 return marked

```

6.7 Iteration 7: Code Indentation

Up to this point the code displayed in the commit blocks had no indentation. The code was formatted with a simple HTML span element per line. Due to the rules of whitespace handling in HTML [W85], the leading whitespace in the displayed code, that is used for indentation, has no effect. Since indentation is used to make control structures in the code visible it is an integral element of code formatting and omitting it greatly affects the readability of the code. Therefore, the graphical presentation without indentation is unsuitable for displaying code. The goal of this iteration was to rectify this issue.

Features and problems tackled in this iteration were:

- Indentation of the displayed source code text.
- Adapt handling of whitespace in the displayed text as appropriate for program source code.

The solution chosen for this prototype was to change the rendering of whitespace via the CSS property `white-space`. By setting this property the way the web browser handles whitespace can be changed to one of several schemes. Per default, the browser ignores line breaks in the source text, collapses multiple consecutive space characters, and removes leading and trailing whitespace. Setting the style `white-space: pre` on an element instructs the browser to perform none of these actions and render whitespace in

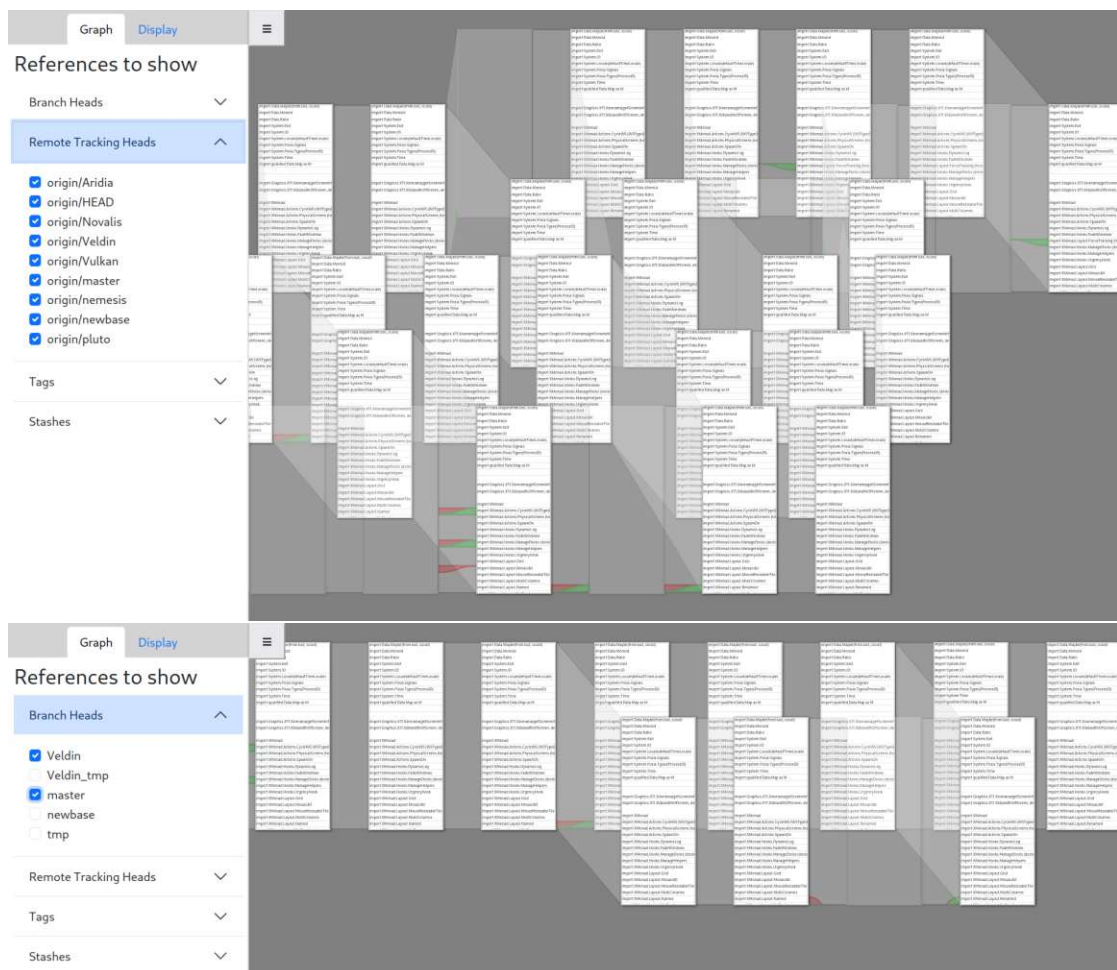
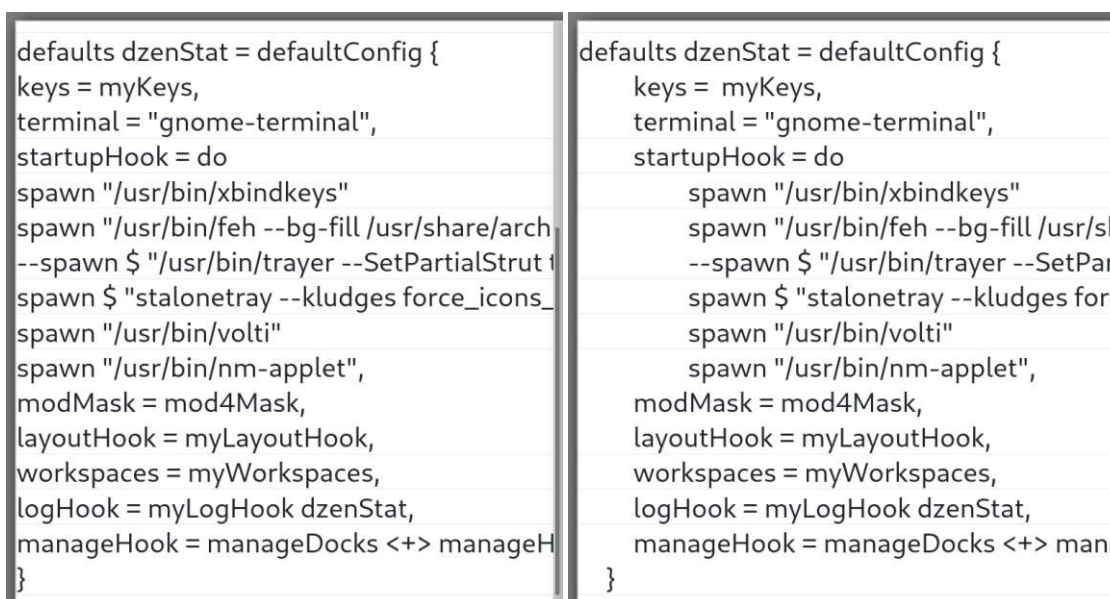


Figure 6.8: Unfiltered graph (top) and graph with only a few references selected (bottom).

the marked element exactly as present in the source text. Thus, the indentation of the presented source code is preserved.

Other possible solutions include using a `<pre>` element, which behaves like `white-space: pre` by default. However, since other default styles are associated with this tag, this would entail more style sheet code to adjust other aspects of the presentation, such as the used type face.

A third possibility would be to convert leading whitespace in a line to a number of HTML elements without content but a set width. This solution would have the advantage that the width of the indentation can be adjusted to the liking of the user. However, the chosen solution was selected over this one because of the much simpler and faster implementation.



```

defaults dzenStat = defaultConfig {
keys = myKeys,
terminal = "gnome-terminal",
startupHook = do
spawn "/usr/bin/xbindkeys"
spawn "/usr/bin/feh --bg-fill /usr/share/arch
--spawn $ "/usr/bin/trayer --SetPartialStrut t
spawn $ "stalonetray --kludges force_icons_
spawn "/usr/bin/volti"
spawn "/usr/bin/nm-applet",
modMask = mod4Mask,
layoutHook = myLayoutHook,
workspaces = myWorkspaces,
logHook = myLogHook dzenStat,
manageHook = manageDocks <+> manageH
}

```

Figure 6.9: Code without indentation (left) and with proper indentation (right).

The result of this iteration is a rendering of the text in the commit that is more appropriate for source code and preserves white space. Indentation is displayed correctly and multiple consecutive space characters—which might be used for other forms of alignment—are not collapsed. A comparison for the rendering of text before and after this iteration is shown in Figure 6.9.

6.8 Iteration 8: Difference Indications on Long Edges

In Iteration 3, the base of the graphical indication of changes between commits (Feature F.03) was implemented. This version could only handle commits on directly adjacent layers properly; long edges that span multiple layers would be displayed in a meaningful way. In this iteration the code for difference indications was developed further to remove this limitation. This completes the implementation of Feature F.03 in the prototype.

Features and problems tackled in this iteration were:

- F.03 (completed): Correct display of difference indication curves on edges spanning multiple graph layers.

The Problem with difference indications on long edges lies in the way they are represented. In the process of building the layered graph, edges that connect commit nodes, that are not placed on adjacent layers, are broken up into segments, as is described in Section 6.1.2. Dummy nodes are placed on the intermediate layers and the edge segments together with these dummy nodes form a chain that connects the end nodes of the

original edge. The graph display component directly follows this structure and displays each dummy node and edge segment as a block in the DOM. Due to this, the simple implementation of showing on each block the difference indications of the associated edge results in all segment blocks of an edge looking the same while the blocks for the dummy nodes between them are blank. The lines of the commit blocks before and after the change, that the difference indication should connect, are not visually connected at all.

The Solution that was implemented in the prototype is to change the view box of the SVG picture in each of the edge and dummy node blocks. All these blocks still contain the full difference indications but only a certain part of that is shown in each block. The computation which portion of the curves to show in a particular block works is done in four steps:

1. **Traverse the edge:** Walk from the predecessor commit node via the intermediate edge segments and dummy nodes to the successor commit node. For each dummy node or edge segment, the width of the corresponding DOM element block is added to a list *widths*.
2. **Convert to intervals:** After the edge traversal the full width of the edge, as it is presented to the user, is known. Each element of the *widths* list is then converted to a subinterval of the interval $[0, 1]$ based on the width value and the position in the edge chain. These subintervals are non-overlapping and completely fill the range from 0 to 1.
3. **Associate the intervals with nodes:** The intervals produced in step 2 are associated with their corresponding graph element. This is accomplished with a dictionary that uses the edge segment or dummy node ID as the key and the computed interval as the corresponding value.

The graph view component uses the calculated intervals to set the horizontal components of the view boxes. Since the curves are drawn with horizontal coordinates in the range 0 to 100, the interval bounds are multiplied by 100. The SVG is stretched such that the portion specified by the view box fills the displayed element. This way in combination there is a smooth transition between the element blocks that make up the parts of the long edge, producing the illusion of undivided, continuous curves.

The result is the appearance of smooth curves over the length of all blocks that are part of the long edge. Figure 6.10 shows an example of a long edge that is made up of three segments. With this iteration the implementation of difference indications (Feature F.03) is complete.

Technically, Git has one additional type of reference: the *note*. However, since note references always point to special note objects, that are not commits, they could not be handled meaningfully in the prototype as it is presented here. They are therefore ignored.

The frontend is extended to show a handle over each commit, above which a stack of references appears. Both the handles and the labels are optional; the user can disable them in the graphical settings panel. Two checkboxes were added at the top of the panel to enable this.

The handle is an SVG. It consists of a circle with a gray border and a green fill. It is visually connected to the commit block with a fine line that is the same color as the edge of the circle. When the mouse cursor hovers over the handle the commit hash is shown as a tooltip. The handle could be used for more features that require interaction from the user, such as selecting commits for direct comparison or for pulling up a menu of actions to apply to the commit. However, the prototype has no such features.

Above the handle is the stack of references pointing to that commit. Each element in the stack is displayed as a label with a rounded rectangle background. The background color depends on the type of reference: green for branch heads, red for remote branch heads, yellow for tags and light blue for stashes.

The Result of this iteration was a handle over each commit that shows the commit hash in a tooltip, and a list of references to the commit, displayed as a stack of labels. An example, including the tooltip, is shown in Figure 6.11. The handle could be used for more types of interaction with the user. However, no such feature has been implemented in the prototype yet. Adding features that make further use of the handle, such as selection of commits for direct comparison, is part of future work (see Section 9.1).

6.10 Summary of the Prototype

At this point, the prototype was deemed complete enough to act as a proof of concept for the evaluation. This section summarizes the development of the prototype, lists the features that were not implemented and the reasons for their omission, and concludes with final remarks on the implementation.

6.10.1 Implemented Features

Of the thirteen features listed in the concept (see Table 4.1) the prototype implements six completely (F.01, F.02, F.03, F.07, F.08, F.11) and one in part (F.05).

Graph view with parallel branches (F.01, F.02) To display the history graph with the contents of a certain file at each commit node (F.01) with branches displayed as parallel timelines (F.02), which form the core of the visualization, were the features that took the most effort to implement. The graph layouting, implemented in Iteration 1 and

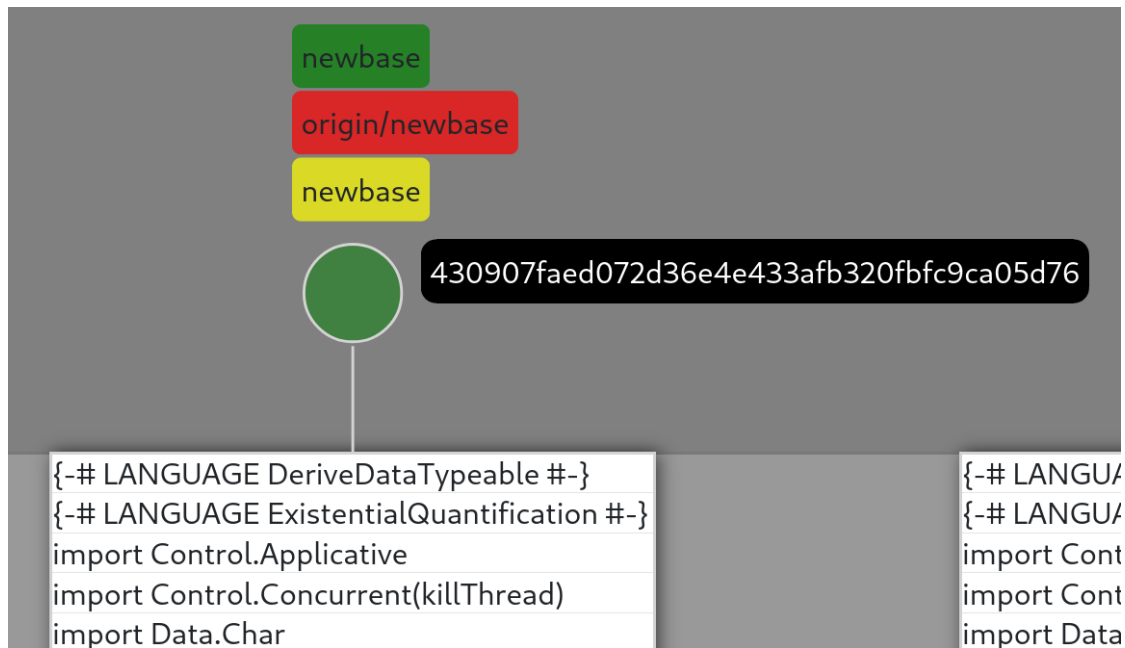


Figure 6.11: Handle over commit block with commit ID and stack of references.

Iteration 2, was based on the algorithm described by Sugiyama, Tagawa, and Toda [54]. Feature F.02 was completed in those first iterations, while some aspects of F.01 were developed later, namely: proper display of the code text in the commit block with indentation in Iteration 7, and the reference labels in Iteration 9. During the latter iteration, small handles above each commit were also added, which are intended for further interactive functionality in the future.

The simple difference indications between commits (F.03) were built using cubic Beziér curves in SVG. Most of this was implemented in Iteration 3, while the handling of long edges, which span multiple graph layers, was added in Iteration 8. The long edge handling was postponed to the later iteration because at the earlier point, when the rest of the feature was done, it was not yet clear how long edges would be rendered in the final version of the prototype.

Text scrolling (F.11) was added during Iteration 4. Initially, this feature seemed to be simple to implement with comparatively low effort. However, due to the way the commit text blocks and the difference indications on the edge blocks are implemented, a rather large number of edge cases relating overscrolling arose, which greatly increased the complexity of the code.

Graphical settings A settings panel, with which aspects of the graphical presentation can be manipulated—such as the width of the commit blocks and text line height—was

added to the prototype in Iteration 5. Zooming the view of the history graph in and out (F.07, F.08) was also done as part of this iteration. Since most of the editable properties were already represented by variables—particularly CSS variables for the calculations in the style sheet—the corresponding UI elements in the settings panel just had to be wired up to those variables and not much of the existing implementation had to be changed.

Focusing on selected branches (F.05) was partially realized in Iteration 6. A reference selection in the form of a list of check boxes was added to the settings panel from the previous iteration. Based on the selection, the graph was reduced to the commit nodes reachable from the selected commits using a Garbage Collection algorithm [62]–[65], then the reduced graph was displayed.

6.10.2 Features Not Implemented in the Prototype

Six of the thirteen proposed features have not been implemented (F.04, F.06, F.09, F.10, F.12, F.13) and one only partially (F.05).

Change indications based on code structure (F.04) The alternative way to indicate changes between commits in a more structural fashion—rather than just line changes—would require a comparatively high amount of effort to implement, both for processing the data and for the graphical presentation. As this feature was also rated less important than others in the semi-structured interviews on the concept (see Chapter 5), it was dropped from the prototype.

Collapsing of linear history parts (F.06) was also left out because it would have required greater changes to the rendering logic. This seemed unnecessary implementation overhead for a proof of the visualization concept.

Direct comparison of selected commits (F.09, F.10) was thought to not belong to the core of the visualization—a “tacked on” feature that can easily be added as part of a bigger, integrated tool, but not essential to the visualization itself. The implementation would also require a whole new UI layer in the prototype. For these reasons, features F.09 and F.10 were omitted from the prototype and also not further discussed in the evaluation (see Chapter 7).

Highlighting of selected blocks in other commits (F.12, F.13) is quite a complex feature. On the one hand it would have required adaptations in the code that displays the commit blocks and further processing of the changes between commits. On the other hand it is not obvious how to answer the question if a change that is adjacent to the selected block relates to that block or not. Are lines added directly after the last (or before the first) lines of the block in a later commit considered part of that block? Conversely, should lines deleted from an earlier commit be considered part of the block? It was decided to tackle those questions later and postpone the feature to future work.

6.10.3 Notes on the Implementation And Limitations

Part of the purpose of the prototype was to evaluate if an implementation of the visualization with ordinary web technologies—HTML, SVG, CSS and JavaScript without dedicated 3D graphics—is feasible. Overall this worked well. A combination of Z -index for the branch layers and 3D CSS transformations for the difference connection blocks was used to provide an illusion of a three dimensional space. Many of the calculations required for the 3D transformations—such as the rotation angle and length of a connection block—were done purely in CSS; this also involved using trigonometric functions in CSS, which were a recent addition to Web browsers at the time of writing [W70]–[W72].

The biggest limitation of the prototype is that it does not scale well with history size; when applied to a repository with around 100 commits it takes a few seconds until the graph is rendered and with bigger histories this time seems to grow worse than linearly. With growing history size, the browser also becomes increasingly unresponsive during this time. This does not appear to be caused by the prototype itself as runtime measurements of the graph processing code reveal that it finishes within a fraction of a second. It seems that the number of DOM nodes puts considerable processing load on the rendering engine of the Web browser. The prototype was tested mainly with Firefox in versions 108 to 120, as Firefox was the only major browser at that time, which supported all required CSS features. Although Safari versions 15 to 17 as well as contemporary versions of Chromium were not able to properly display the history graph, they nonetheless also exhibited the observed delay in rendering and unresponsiveness. Showing only a part of the history graph could be a possible solution in the future.

The synchronized scrolling feature also appears progressively more choppy for a growing number of history nodes. Further investigation in future work (see Section 9.1) is required to find the cause of this performance issue. Possible candidates are again the rendering engine of the Web browser and the automatic change detection of the UI framework.

Occasionally, when the user hovers over a commit block and scrolls, the scroll position snaps back to the top. The cause of this bug is currently unknown and further investigation is required for future improvements.

This concludes the description of the prototype development phase; in the next phase the prototype was used in the evaluation interviews.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Semi-Structured Expert Evaluation with the Prototype

After the development of the prototype had reached a stage where it was considered complete enough to illustrate the concept, experts were invited to an evaluation interview regarding the concept with the aid of the prototype. The evaluation was done in the form of semi-structured interviews. This chapter describes this evaluation, including the planning that was done before the interviews, the obtained results, a summary and interpretation of the results, and threats to the validity of the evaluation that were identified.

7.1 Plan

The initial plan was to evaluate the visualization with scenario-based interviews, where the participants would be presented with a number of problem statements and asked to show how they would solve these problems with the proposed visualization on the example of the prototype. However, since the prototype turned out to be rather limited—it can only handle rather small repositories¹, only half of the proposed features are implemented and it has limited interactivity—such an evaluation procedure was deemed to be not meaningful. Instead, the evaluation was done using semi-structured interviews, like the evaluation of the concept (see Chapter 5 Semi-Structured Expert Interviews on the Concept).

¹Technically, the prototype itself can handle bigger repositories quite well, but it displays the whole history of the file in the repository at once, which quickly gets too much for the web browser to render.

The interview was structured into five segments:

- **Demography:** Some data about the interviewee's person and experience in the field was raised for comparison.
- **Demonstration:** The prototype and its individual features were demonstrated to the interviewee.
- **Purpose of Individual Aspects of the Prototype:** This segment focused on the general aspects of the prototype and the implemented features.
- **Proposed Features Not Implemented in the Prototype:** Features that were proposed earlier (see Chapter 4 Conceptual Design and Chapter 5 Semi-Structured Expert Interviews on the Concept), but had not been implemented in the prototype, were discussed in this segment.
- **Purpose of the Visualization As Presented in the Prototype:** The interviewee's general impression of the visualization as presented and closing remarks were gathered.

The individual questions are listed together with the results in Section 7.2; for the questionnaire as used during the interviews, see Appendix B.

7.1.1 Demonstration

The demonstration of the visualization with the prototype to the interview participants was planned ahead of the interviews to reduce differences between the individual demonstrations and avoid omissions of features and details in the explanation, which could influence the results. A description of the plan follows. Minor differences between the plan and individual demonstrations during the interviews still occurred, such as the order in which some features were presented and the part of the VCS's history with which they were demonstrated and explained.

First, the way the graph itself is displayed was explained. The layout of the history graph as a timeline of parallel branches from left to right was described, with the commit blocks all showing the same file at the respective commit. The connections between the commit block with the colored difference indicator curves were explained and compared to a typical side-by-side diff view in a merge tool or IDE, with Kompare [W68] and IntelliJ IDEA [W54] as examples. It was shown that the handle above a commit block shows the respective commit ID. Furthermore, it was explained that the handles are intended to be used for more interactive functions in the future.

Next, the way Git references are displayed was shown, with the labels above the commit handle and which color signifies which type of reference. A brief pan over the history graph was done to show an example of each kind of reference and at least one commit with more than one reference, to show how the reference labels stack.

The third part of the presentation was a short explanation how scrolling the text in the commit blocks works. The default behavior, where scrolling one block results in simultaneous scrolling of all commit blocks, was demonstrated first. After that, scrolling of an individual code block was shown too. To give an example where this feature might be useful, a pair of commits with a large number of changes in between was used, resulting in big offsets in position of the same code block between those commits, which could be reduced by scrolling only one of those two commits.

In the fourth part of the presentation the function of the display settings panel was described. It was shown which aspects of the graphical presentation can be changed. The zoom factor was increased to show more detail and reduced to get a rough overview of the whole history graph. It was shown that the height of individual text lines can be adapted to personal preference. The width of the commit blocks was changed; once made narrower to fit more commits along the timeline on the screen and once increased to be able to see longer lines of text without resorting to horizontal text scrolling. Changing the height of the commit blocks was also briefly demonstrated. Finally, it was shown how all the distances can be changed: the gap between commits on the same timeline, the gap between two parallel branch timelines and the offset between commits on different timelines.

The last part of the demonstration explained the selection of the references to display. The panel listing the references grouped by type was shown. The effect of enabling and disabling individual references was explained and demonstrated, in particular that disabling a reference which is on a branch included by another references just hides the label and that commits which are not reachable from any selected reference are hidden so that the displayed graph becomes smaller. An extreme case where only a single reference is selected and thus a large part of the graph is not rendered anymore was shown. This concluded the demonstration.

7.1.2 Pilot Interview

Before the evaluation interviews were conducted, a pilot interview was done to test the plan and eliminate problems with the interview questions and the demonstration. The data from the pilot interview is not included in the evaluation results.

The pilot revealed that a question in the demography section, about the repository mining tools the interviewee had used, was missing; it was added between the pilot and the first evaluation interview. Furthermore, the plan for the demonstration of the prototype was worked out more explicitly afterwards, as the presentation during the pilot was somewhat unstructured and hard to follow due to the initial plan being too loose.

7.1.3 Further Notes

After the interviews regarding the concept (see Chapter 5), the features to directly compare selected commits (F.09 and F.10) were deemed to be just “tacked on”; something that might be added to, and called from, the visualization when integrated into a larger tool

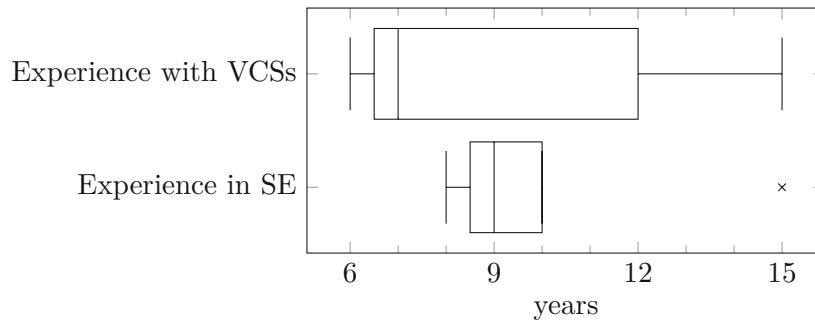


Figure 7.1: Participants' experience in the field

or IDE, but that is not essential to the concept. Therefore, these features were dropped from the evaluation.

Furthermore, to contrast synchronized scrolling feature that the prototype implements (F.11) with “ordinary” scrolling of the text in just a single commit block, a question asking about the purposefulness of the latter was added to the questionnaire.

7.2 Results

A presentation of the results of the interviews follows. The answers to the questions are listed and visualized. For questions on a scale, a brief statistical analysis is added.

7.2.1 Demography

As in the interviews on the concept (see Chapter 5 Semi-Structured Expert Interviews on the Concept) the participants were asked to provide some information about themselves, in particular their experience in the field of SE in general and more specifically working with VCSs. Furthermore, they were asked about the tools they use in their work with VCSs.

Years of experience First, the participants were asked about their years of experience in the field of SE and working with VCSs. The results are visualized as a box plot in Figure 7.1. Most participants reported 9 to 10 years of experience in SE, with singular answers of 8 and 15 years as the lowest and highest respectively. For working with VCSs, two interviewees gave an experience of 15 years, while the rest answered in the range of 6 to 9 years. Of particular note, one participant gave several years more experience with VCSs than with SE.

Used VCSs For the questions regarding the VCSs they have used at least once in the past, have used the most, and are currently using (Figure 7.2), all seven participants answered with Git. Five had also used Subversion in the past, one further named Mercurial. CVS, though explicitly listed, was not selected by any of the interviewees.

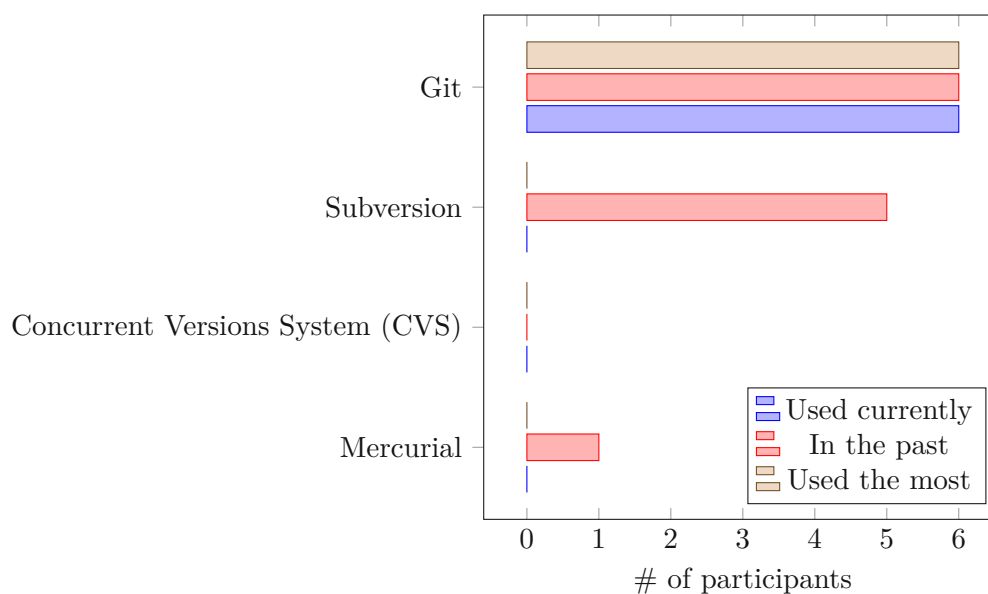


Figure 7.2: VCSs used by participants.

Alternative frontends When asked if the participants had used alternative frontends when working with VCSs—that is, other than the default command line interface—they listed the following tools:

- IntelliJ-based IDEs [W54] (*5 times*)
- Eclipse [W57]
- GitKraken [W64] (*3 times*)
- SourceTree [W74] (*2 times*)
- Fork [W76] (*2 times*)
- GitLens [W87], a VS Code [W67] extension
- GitHub Desktop [W60]
- GitHub [W15]
- GitLab [W17]
- TortoiseGit [W75]
- TortoiseSVN [W86]

Repository mining tools To the question, which mining tools to extract data about the history of on VCS repository the participants had used, the interviewees listed the following:

- IntelliJ [W54]
- Eclipse [W57]
- Binocular [66][W88] (*3 times*)
- GitHub project statistics [W15] (*2 times*)
- GitLab project statistics [W17] (*3 times*)
- GitInspector [W77] (*3 times*)
- A custom script [30]

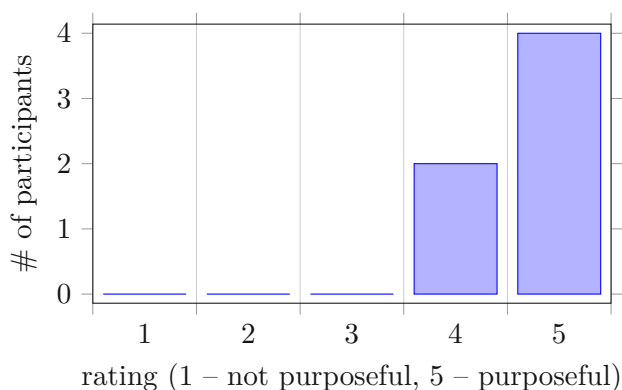


Figure 7.3: Purposefulness of limiting the displayed history to selected references (F.05)

Other tools Finally, the interview participants were asked for any other tools they had used when working with VCSs. No further tools—besides the ones already listed above for visualization and repository mining—were added.

7.2.2 Purpose of Individual Aspects of the Prototype

After being given a demonstration of the concept by means of the prototype the interview participants were asked about their opinion on the purposefulness of the implemented prototype. All questions in this section were in a survey format with a scale from 1 to 5, where 1 meant *not purposeful* and 5 *purposeful*.

Limiting shown history The possibility to limit the displayed history to a set of selected references (part of F.05) was rated generally positive (Figure 7.3). Out of the 6 interviewees, four gave a rating of 5, with the remaining two answering with 4, giving an average rating of $4.\bar{6}$ and a median of 5.

Zooming in more detail and zooming out for more overview were both rated very similar in purposefulness (Figure 7.4). Zooming in to focus on a small part of history received ratings in the range from 3 (neutral) to 5 (purposeful). Three of the six interviewees gave the highest rating of 5, while one gave the lowest rating 3, with an average of $4.\bar{3}$ and the median at 4.5. Zooming out for the purpose of getting more overview of the history graph was rated in the range of 4 to 5 by all participants, with the median at 5 and an average of $4.\bar{6}$.

In direct comparison, zooming out was rated somewhat better than zooming in; while the latter feature received one neutral rating, the former feature was rated uniformly on the *purposeful* side, also getting the better median and average rating. Also, the difference in rating for the two features is no bigger than 1 on the scale for all interview participants. Of these, three participants thought zooming out to be more purposeful, one interviewee thought the opposite, and two interviewees gave both features the best rating.

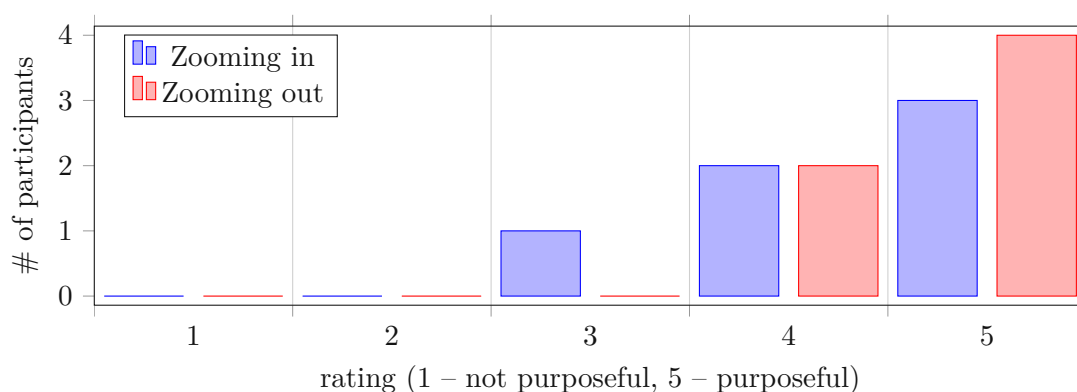


Figure 7.4: Purposefulness of zooming in and out for more detail or overview (F.07 and F.08)

Scrolling of text, synchronized in all commits and isolated in a single commit, were both rated with a clear tendency towards *purposeful* (see Figure 7.5). Synchronized scrolling (F.11) was rated as 5 by one half of the interviewees and 4 by the other half, resulting in an average rating of 4.5 with a median of 4.5.

Isolated scrolling of a single commit was rated by most participants as either 4 or 5, with one participant rating it as 2. The latter participant explicitly said they see no value in scrolling only one commit in the context of this visualization. If all answers are considered, both average and median are 4. If the single rating of 2 were deemed to be an outlier, the average would become 4.4 though the median would still be 4.

In direct comparison, most interviewees regarded one form of scrolling as more purposeful and vital to the use case than the other; only one participant gave both features the same rating at 4. In particular, the participant who considered isolated scrolling as not very purposeful (rating 2), thought that synchronized scrolling is very important and purposeful for the visualization.

The same participant gave additional feedback that they would prefer a more intelligent implementation for the scroll synchronization. While the presented visualization prototype scrolls all commits by the same amount, they would prefer a scrolling scheme where the other commit blocks are scrolled with respect to the changes relative to the user-scrolled base version, so that all commit blocks show conceptually the same text block, even if they are at vastly different positions in the file due to changes in the rest of the file. This works well for two versions shown side-by-side, however, implementing such a scrolling scheme in the presented visualization, where the whole commit graph is shown, is not trivial; when branches that are later merged exist, a naive implementation might lead to loops in scrolling adjustments, thus never coming to rest.

7. SEMI-STRUCTURED EXPERT EVALUATION WITH THE PROTOTYPE

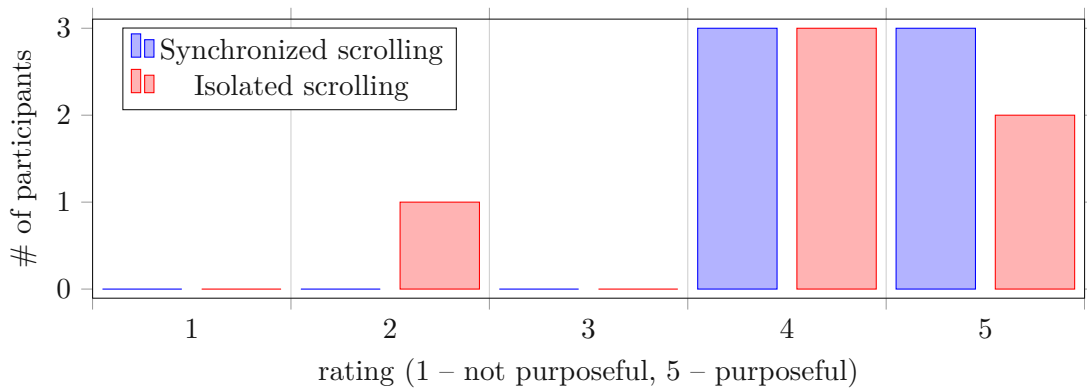


Figure 7.5: Purposefulness of two different variations of scrolling the text in commit blocks (F.11).

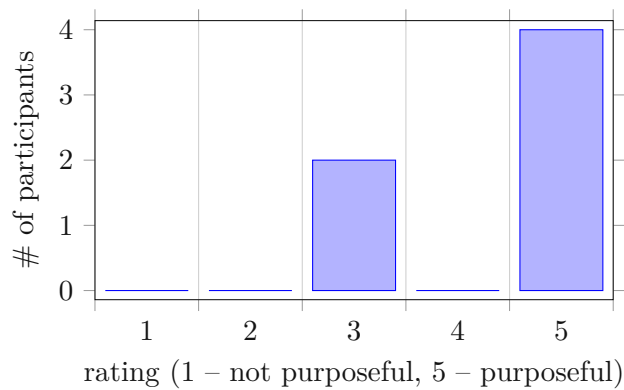


Figure 7.6: Rating of purposefulness of the ability to manipulate aspects of the graphical presentation.

The ability to manipulate aspects of the presentation so that the user can adjust the size of the commit blocks, the gaps between them, and so on, was rated in the range from 3 to 5, with an average of $4.\bar{3}$. The distribution of the ratings is visualized in Figure 7.6. Two thirds of the participants rated the feature on the *purposeful* side of the scale, the remaining third gave a neutral rating. Interestingly, no interviewee gave a rating of 4 for this feature; participants regarded it as either very purposeful or gave a neutral opinion.

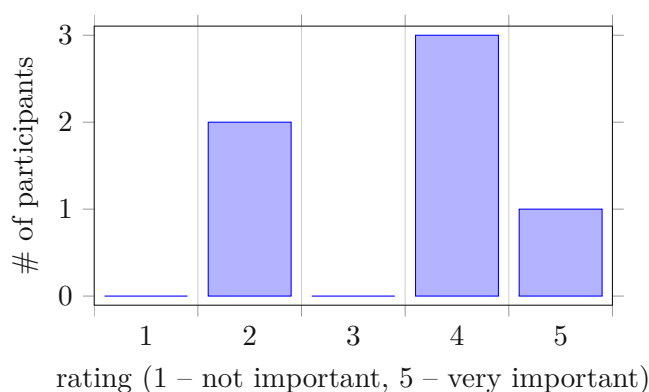


Figure 7.7: Importance of focusing on individual branches by fading others (F.05)

7.2.3 Proposed Features Not Implemented in the Prototype

In this part of the interview, features that were proposed in the concept but had not been implemented in the prototype, were presented in this part of the interview. The features were described briefly with the aid of a sketch. The interviewees were then asked to rate the importance of this feature for a practical implementation of the visualization on a scale from 1 (not important) to 5 (very important).

Focusing on individual branches and hiding others As F.05 was only partially implemented in the prototype it was presented in its initially intended form in this part of the interview. A histogram of the results is shown in Figure 7.7. The opinions of the participants were rather mixed with a slight hinge towards *important*. The responses ranged from 2 to 5 on the importance scale, with half of the interviewees giving a 4, which is also the median. The average rating was 3.5.

Folding of linear parts of the history that contain no branching points (F.06) was generally rated towards the *important* end of the scale (Figure 7.8). All answers were either 4 or 5, with a median of 4 and an average 4.3.

Highlighting selected blocks The features *Selecting a code block and highlighting its corresponding block in a single selected other commit and over the whole history* (F.12 and F.13) were rated quite differently by the interviewees Figure 7.9. There was no consensus on the importance of *Selecting a block of code in one commit and highlighting the corresponding block in another commit*, with only a slight hinge toward *important*. It received ratings from 2 to 5, with the answers distributed slightly more to the edges to that range. Both median and average are 3.5.

Selecting a block of code in one commit and highlighting it over the whole history received less diverse ratings but the overall result was neutral. Two thirds of the participants had a neutral opinion on the importance of this feature, while the remaining two participants

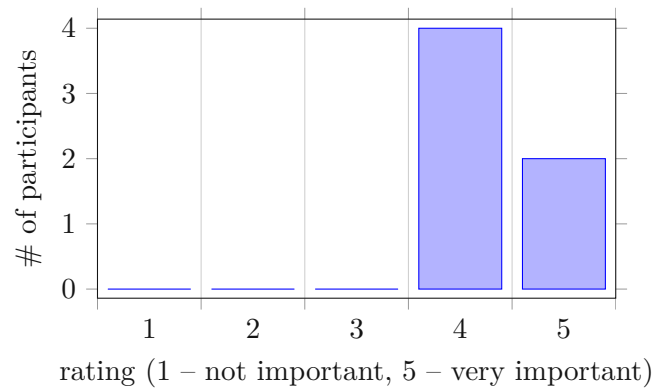


Figure 7.8: Importance of folding linear parts of the history without branching points (F.06)

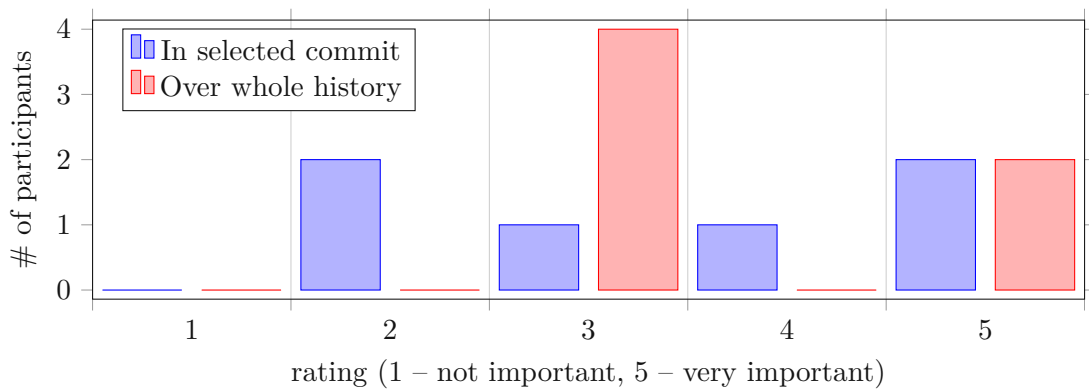


Figure 7.9: Importance of highlighting a selected block (F.12 and F.13).

thought it to be very important for the visualization. The average of the ratings is $3.\bar{6}$ and the median 3.

When comparing the responses to both questions it can be observed that all participants had a similar opinion on both versions of the feature; the importance rating of both features does not differ by more than one point on the scale for any interviewee. Both participants that rated the *single commit* variant with 5 gave the same rating for the *whole history* variant; all other interviewees, who rated the importance of the *single commit* variant in the range from 2 to 4, gave the neutral rating of 3 for the *whole history* version.

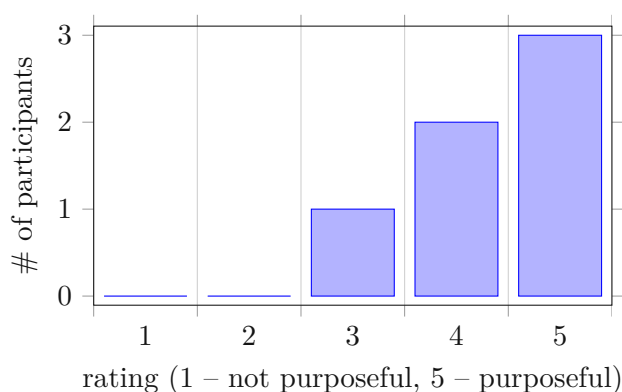


Figure 7.10: Purposefulness of the visualization as presented in the prototype

7.2.4 Purpose of the Visualization As Presented in the Prototype

This final section of the interview evaluation interview consisted of questions regarding the purpose of the visualization as a whole as it was presented. The question format was again in the form of ratings on a scale from 1 to 5, as well as open questions for closing thoughts.

How purposeful is the proposed visualization? The answer to this question was on a scale from 1 (not purposeful) to 5 (purposeful). The responses ranged from 3 (neutral) to 5 (purposeful) with a hinge towards *purposeful* (Figure 7.10). The average rating was $4.\bar{3}$ and the median 4.5.

Is the visualization helpful in getting an overview over the evolution of code? The helpfulness of the visualization was to be rated on a scale from 1 (not helpful) to 5 (very helpful). The responses on this question were clearly towards the *helpful* side of the scale; four out of six participants gave a rating of 5 while the lowest rating was 3 (Figure 7.11). Accordingly, the median is 5, while the average of ratings is 4.5.

Merge conflicts The helpfulness of the visualization, *for identifying possible merge conflicts in a file, that has been changed in multiple branches*, was to be rated on a scale from 1 (not helpful) to 5 (very helpful). The responses range were mixed but tended to be more towards the *helpful* side (see Figure 7.12). The ratings ranged from 2 to 5 on the scale. The median was 4, as was the average.

How helpful is the proposed visualization in teaching Software Engineering practice? The helpfulness of the visualization was to be rated on a scale from 1 (*not helpful*) to 5 (*very helpful*). The responses ranged from *neutral* to *very helpful*; they were evenly distributed between 3 and 5 inclusive, with the average and median both at 4 (see Figure 7.13).

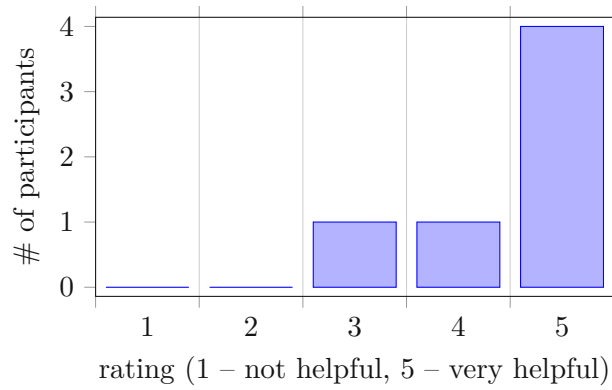


Figure 7.11: Is the visualization helpful in getting an overview over the evolution of code?

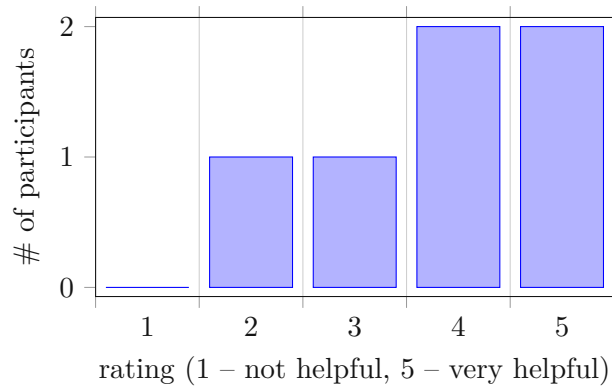


Figure 7.12: Helpfulness of the visualization for identifying potential merge conflicts.

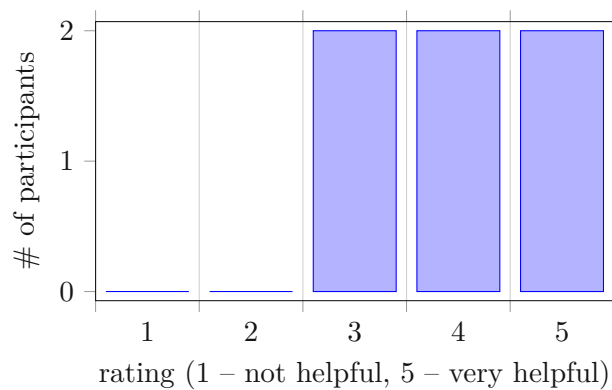


Figure 7.13: Helpfulness of the visualization for teaching.



Figure 7.14: Probability for the participants to use the visualization in their workflow

Would you use the proposed visualization as part of your workflow? For the final structured survey question the participants were asked estimate how likely they would integrate the presented visualization into their version control workflow on a scale from 1 (*unlikely*) to 5 (*definitely*).

The answers were very diverse; one third of the interviewees said they thought it *likely*, one third tended more to the *unlikely* side, while the remaining third was unsure (see Figure 7.14). Every option on the 5 tick scale was given once, with 3 being answered twice. Consequentially, both median and average are 3.

Would you like to see a feature in the visualization that was not presented?

The participants were asked for features they would want to have in a tool implementing the visualization, in order to increase its fitness for the purpose or adapt it to more use cases. The following feature suggestions were given:

- Highlighting of commits and branches in an interactive fashion
- Also highlighting the change curves when highlighting a changed block
- To interactively change the order in which the branches are displayed
- Change the orientation of the time axis
- Change the default starting point: at the earliest commit or the latest ²
- Change the colors, as people with color blindness might have problems discerning the predefined ones.
- Indicators for diverging changes of the same text piece on different branches, which could indicate possible merge conflicts

²The prototype always starts at the beginning of the time line, that is, at the time of earliest commit.

- Use handles on commit nodes to move commits and change presented branch structure.

Do you have any further remarks? When asked for final remarks and thoughts about the visualization and the presented prototype, the following responses were given:

- That a file selection UI is required. ³
- That exposure to and practice with the tool is required in order to understand its full potential.
- That it is unfortunate, that the visualization only shows the history of a single file.

7.3 Summary and Interpretation

Overall, the visualization was received rather positively by the interview participants. Regarding both the visualization as a whole, as well as the individual implemented features that were presented, none received the lowest rating on the scale of 1 to 5 for *purposefulness* and only one feature—scrolling of commit text in one isolated commit (part of F.11)—was rated with 2 by one participant. The median and average of the ratings were all at least 4, clearly placing the opinion of the participants towards *purposeful*.

Similarly, for the features that had not been implemented, the lowest rating for *importance* was never given, while two features—highlighting a selected text block in one selected other commit (F.12) and the full implementation of focusing on individual branches (F.05)—received ratings of 2. The most clearly *important* rated feature was the folding of linear code parts (F.06), for which the lowest rating was 4, with the average at 4.3. The other three non-implemented features were rated more neutrally but still with a hinge towards *important*, with the medians in the range of 3 to 4 and average ratings around 3.5.

The visualization was also rated more towards the *helpful* end of the rating scale for all three listed usage scenarios—getting an overview over the code evolution, identifying possible merge conflicts, and for teaching of SE practice. The lowest rating was again a 2, which was only given once to *identification of potential merge conflicts*. The participants opinion for the helpfulness of the visualization was the highest for *getting an overview over the code evolution*, which was rated as 4.5 on average, while more than half of the ratings, and consequently the median, was 5. While not as high, the rating of the visualization was still clearly *helpful* for both other scenarios.

Additional features that the participants listed as what they would like to see in an implementation of the visualization can be divided into three groups: more interactivity;

³Currently, the path to the version control repository and the file within it is hard coded; implementing a proper file selection was regarded as not vital for a proof-of-concept prototype and deferred to future work.

more control over the graphical presentation, such as colors and perspective; and more explicitly highlighting the changes to a selected code block. The latter one is interesting, as the proposed—but not yet implement—features it relates to from the concept (F.12 and F.13) were rated as relatively unimportant compared to the rest. Adding the suggestions from the interview participants in the future might make the highlighting of changes to a selected text block more interesting and increase its perceived importance. Therefore, features F.12 and F.13 should not be disregarded in the future before this has been explored.

Despite the visualization being generally regarded as purposeful and helpful by the interview participants, if they would integrate such a tool into their workflow, the responses were mixed, with a completely neutral average and median. It thus seems uncertain if the visualization would see much use in the practical field of SE. However, the presented prototype was crude and limited; a more polished and feature complete version might attract more interest. Therefore, future work should look into creating a more practical implementation.

7.4 Threats to Validity

Small Sample Size Six developer participated in the evaluation interviews. This small number of data points is far from a statistically relevant size. It is possible that the small number of interview participants biased the interview results. However, the purpose of the interview was to find out if the concept is worth exploring further, but was not developed enough yet for a large scale evaluation.

Influence due to the Interviewer The interviewer might have some influence on the perception and decisions of the interviewees. This could be in different forms, such as suggestive phrasing or a bad demonstration of the prototype. While it was attempted to avoid this by minimizing the interaction between interviewer and interviewee during the filling in of the questionnaire, this factor cannot be fully eliminated, especially during the demonstration phase of the interview.

Bias due to Phrasing The phrasing of the questions in the questionnaire and the explanations of the prototype could be ambiguous, leading to misunderstandings which could have possibly biased the answers of the participants. Furthermore, the phrasing could be otherwise suggestive and influence the responses more directly. A pilot interview was performed to find such problems in the wording, however in cannot be fully ruled out that such influences were still present in the evaluation interviews.

Issues With Performance and User Friendliness of the Prototype The prototype as implemented and presented is very crude and still has problems with user friendliness. The performance issues discussed earlier also mean that even with smaller repositories some functions only work slowly. While the interview participants were informed that

these problems are due to the fact that the prototype is very minimal to act as a proof of concept, it might still cause a subconscious subjective bias in the participants.

Limited Functionality of the Prototype Similarly to the above, the limited functionality of the prototype as it was presented might bias the interview participants. Though features that were planned in the concept but not implemented in the prototype were discussed and so the participants were informed of the fact that this prototype does not fully represent the visualization as a whole, it might still influence their opinion and responses.

Results

In this chapter, the attained results of this thesis are summarized with respect to the research questions.

RQ1: How to visualize a graph-based source multi-diff? During the conception phase (see Chapter 4), a visualization of a single file's history was proposed, where the history graph of the VCS repository is shown, and the state of a selected file is displayed at each commit node. The history graph is laid out as a layered graph, where the nodes are represented by a block of text, which contains the contents of the file at the corresponding commit. The edges connect commit nodes with a parent-child relation and graphically indicate the changes between the two versions. The full set of proposed features is listed in Table 4.1.

Before the implementation began, several possibilities for the implementation of the graph view were considered and compared (see Section 4.3 Implementation Considerations). The question if technologies specifically for three dimensional graphics should be employed, or UI technologies employing two dimensional are sufficient for the task. Due to the layered nature of the proposed graph visualization, and the fact that text rendering is required, it was decided to explore the 2D route further in this thesis.

For the layout of the history graph nodes and edges, a literature research into the topic of graph rendering was done. Based on this research, an algorithm for laying out layered graphs due to Sugiyama, Tagawa, and Toda [54] was chosen as the foundation of the history graph view; the concrete implementation used in the prototype is described in detail in the chapter on the implementation (see Section 1 and Section 2).

RQ2a: How do experts assess the proposed solution? This research question was tackled in two parts: in the expert interviews regarding the concept (see Chapter 5) and the evaluation of the implemented prototype (see Chapter 7).

As part of the interviews on the concept before the implementation of the prototype, the proposed graph visualization was explained and the individual features (see Figure 4.1) were presented to the participants. The concept as a whole was well received by the interviewees; all of them thought the proposed visualization to be *purposeful* overall. However, the responses regarding the helpfulness for teaching SE practice were neutral. Among the listed features, most were rated as *important*. Features rated more neutrally were the structural difference indications (F.04), zooming (F.07, F.08), and the highlighting of a selected code block in other versions (F.12, F.13).

During the evaluation interviews, the developed prototype was used to give the participants a demonstration of the implemented features (F.01, F.02, F.03, F.07, F.08, F.11). The assessment of the participants was generally towards *purposeful*, reaffirming the positive reception of the interviews on the concept; zooming in and out (F.07, F.08) were also rated more towards *purposeful* this time. In contrast to the earlier interviews, the participants of the evaluation thought the proposed visualization could be helpful for teaching.

In summary, the involved experts assess the proposed visualization of the file history with indications of the differences to be helpful and the individual features to be purposeful for the task.

RQ2b: What expectations and requirements do experts have for the proposed solution? During both sets of interviews (see Chapter 5, Chapter 7), the participants were asked if they would like an implementation of the concept to provide features which were not presented as part of the concept or present in the prototype.

The interviewed experts answered that they would expect more interactivity than the developed prototype provided. Furthermore, more control over the graphical presentation was desired; a particular example was to change the color palette, as the used colors for the difference indications—green and red—are hard to discern for color blind users. More options for the view onto the graph, such as orientation of the time axis or the starting point of the visualization in the history, were also listed.

RQ3: How purposeful is the proposed visualization method for comprehending the evolution of a source file? The section about information needs in the semi-structured interviews on the concept (see Section 5.2.2) revealed that the participants deem it important to get an overview over potentially non-linear history of source code in a project, and that they further think that visualizations can help with that. This establishes that there is a need among developers for tools to help them understand how the code evolved. During both the interviews on the concept (Chapter 5) and the evaluation of the prototype (Chapter 7), the question, whether the proposed history graph visualization would be helpful for this task, was posed. The responses clearly tended towards *helpful*.

Conclusion

In this thesis, a visualization presenting the evolution of a file in a VCS, including branches, was explored. A literature research was conducted on the state of the art in MSR in general, and visualizations of file history in particular. Both academic research as well as tools established in SE practice have been examined and compared.

From the literature research it was concluded, that branches are underrepresented in current research, despite being a widely used aspect of VCSs in modern SE practice. Visualizations that incorporate branches often only show the commit graph itself or focus on visualizing statistics.

Based on these findings, a concept was designed for a visualization which combines the history graph of the VCS with all versions of a file and diffs between related versions (RQ1). A list of features that are part of, or can be accessed from, this visualization was devised. The concept is described in Chapter 4.

To assess this concept (RQ2a), it was presented to working software engineers in questionnaire-driven interviews. The concept was explained to the interviewed experts with the help of mockups. Afterwards, the interviewees were asked to assess the presented features as well as the visualization concept as a whole. Most proposed features were rated clearly *purposeful*. The remaining features were assessed more neutrally but still towards the *purposeful* end of the scale. The visualization was also thought to be *purposeful* as a whole and for getting an overview of code evolution (RQ3). Finally, the interviewees were asked what further requirements they have towards an implementation of the proposed concept (RQ2b). Only one feature was listed: changing the colors used by the visualization. The results of the interviews are presented in Section 5.2.

Due to the overall positive results of these interviews, a prototype of the concept was developed. After considering several ways to implement the different aspects of the visualization, it was decided to base the prototype only on standard web technologies—HTML, CSS, JavaScript and SVG—instead of employing facilities specifically for 3D

graphics. Only a subset of the proposed feature list was implemented in the prototype; features considered non-essential were left out.

Then, experts were invited to a second evaluation of the concept, this time with the aid of the prototype. This was initially planned to be a scenario-based evaluation where the participants would have used the prototype themselves. Due to the limited interactivity, this plan was discarded and the interviewees were given a demonstration instead. Afterwards, the interviewees were asked to assess the visualization and its individual features again (RQ2a). The results—described in Section 7.2—were again positive: Both the features and the visualization as a whole were rated as *purposeful* again, reaffirming the positive results of the earlier interviews of the concept. Furthermore, the interview participants were asked for their expectations towards future developments with regards to the concept (RQ2b) given their experience with the presented prototype. The most notable response was a requirement for more interactivity, such as the ability to reorder the shown branches of the graph or to focus on a part of the history by selecting the branches or paths along the commit graph directly, rather than just a list of Git references to toggle.

In the final section of the interview, the participants were asked if they think that the visualization is useful for getting an overview over the code history (RQ3). The interviewees thought that such a tool would indeed be helpful in the exploration of, and getting insights about, a file's history. They also thought it would be helpful for the use case of identifying potential merge conflicts. Furthermore, the interview participants thought the proposed visualization would be helpful in the context of teaching SE practice to students who are not yet well versed in the use of VCSs.

Overall, the visualization was well received by the interviewed experts and the concept is worth investigating further. However, for practical application, more work is needed. On the one hand, the tool itself needs to be developed further, implementing more of the proposed features and a selection of the analyzed file. On the other hand, some of the interviewed experts said that they would only incorporate such a tool into their workflows if it could be integrated with their IDE or the project management tool. More candidates for future work are listed in Section 9.1.

The prototype also proved that an implementation of the concept is reasonable. The implementation as an HTML frontend with ordinary web technologies worked well. Despite the visualization being three dimensional in nature, The three dimensional nature of the visualization The simulation of the three dimensional presentation with mainly The two dimensional layout with calculated element positions, *Z*-ordering and CSS 3D transformations for the commit connections produced a reasonable impression of a three-dimensional view with orthographic projection. This can also be extended in the future with a transformation of the view container itself, to support different viewing angles onto history graph.

9.1 Future Work

Because the primary purpose of the prototype was the evaluation of the viability of the concept, it was developed as a standalone tool with a preference for speed of development rather than practicality. In the following, potential targets for future work are listed.

Not all of the proposed features in the concept have been implemented in the prototype in the context of this thesis. Limiting of the history graph to selected parts was only implemented in part, by selecting a list of Git references, while the more interactive selection of individual graph branches was not. Furthermore, highlighting the history of a user-selected block of text was not implemented, but was named as an important missing feature in the evaluation interviews. Future work should implement and explore these features.

The performance of the prototype would be unacceptable for practical application. While histories on a scale of less than one hundred commits could be handled, repositories bigger than that quickly bring the tool to grind to a halt; the prototype as it is would be unable to handle realistic repositories of projects, on which multiple developers have worked for several years. Brief attempts to find the issue seem to indicate, that the algorithms of the tool themselves are quick enough and the required analysis of the history and production of DOM objects for display in the web browser happens in less than one second; it seems, that for bigger graphs, the rendering engine of the browser is overwhelmed by the amount of objects to display. Future developments could explore this further and attempt to mitigate the issue by rendering only part of the history graph at any given time.

During the development of the prototype, small handles were added over the text blocks at each commit node. These handles were not present in the description and sketches of the prototype. They were intended to be used for interaction with commit nodes, such as selection for diffs. However, the function those handles have in the prototype are to display the commit hash on hover. Besides the features which require selection of individual commits, these handles could be extended to display a menu of actions. Furthermore, uses not considered in this thesis could be explored.

When the participants of the evaluation interview were asked, if they would incorporate a tool like the proposed visualization into their daily workflow, the responses were mixed. Possible reasons could be unfamiliarity or the overhead of installing and preparing an additional tool. Lowering the barrier of entrance could be explored as a measure to increase developer interest. In the future, the visualization could be made available from the central project management tool, or provided as a plugin to an IDE such as IntelliJ IDEA [W54].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Questionnaire of the Interviews on the Concept

The questionnaire which was used for the evaluation of the prototype (see Chapter 5 Semi-Structured Expert Interviews on the Concept) is included here. The questionnaire was created with Google Forms [W89]. For the interviews, the web form was used. It was exported as PDF for inclusion in this thesis. The included questionnaire begins on the next page.

During the interviews, the elaboration for question for Feature F.11 included an animated image to demonstrate synchronized scrolling. Since this can obviously not be recreated in a printed document, the first frame of the animation is shown as a static image instead.

Semi-Structured Expert Interview

Demography

A few short questions about your experience

1. How many years of experience do you have in Software Engineering (SE)?

2. How many years of experience do you have with working with Version Control Software (VCS)?

3. What VCSs do you currently use?

Wählen Sie alle zutreffenden Antworten aus.

Git

Subversion (SVN)

CVS

Mercurial (Hg)

Sonstiges: _____

4. What VCSs have you used in the past?

Wählen Sie alle zutreffenden Antworten aus.

- Git
- Subversion (SVN)
- CVS
- Mercurial (Hg)
- Sonstiges: _____

5. Which VCS do you use the most?

Wählen Sie alle zutreffenden Antworten aus.

- Git
- Subversion (SVN)
- CVS
- Mercurial (Hg)
- Sonstiges: _____

6. Have you used alternative frontends when working with VCSs in the past? Which ones?

A. QUESTIONNAIRE OF THE INTERVIEWS ON THE CONCEPT

7. Have you used repository mining tools to extract more data about a VCS history in the past? Which ones?

8. Have you used visualization tools to aid your work with VCSs? Which ones?

9. Have you used other tools to aid your work with VCSs? Which ones?

Information Needs

-
10. Is it important to get an overview over possibly non-linear history of a project's source code?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

11. Do visualizations help in getting an overview over the history?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

12. How important is it to be able to see changes to the same piece of code in different versions?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

13. How important is it to be able to see changes to the same piece of code across different branches?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

14. How important is it to have simple, direct navigation through the whole history?

Examples:

- Panning to move along the history
- Move commits directly with the mouse
- Scrolling for zooming in and out

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

Importance of Features

What follows is a brief description of a proposed visualization, followed by a list of features, that are to be graded for importance. The features are each first described briefly in text and many further illustrated with images, followed with a rating scale for the feature, before continuing with the description of the next feature. The images are in part sketches of the envisioned visualization and in part screenshots of other tools with a similar feature.

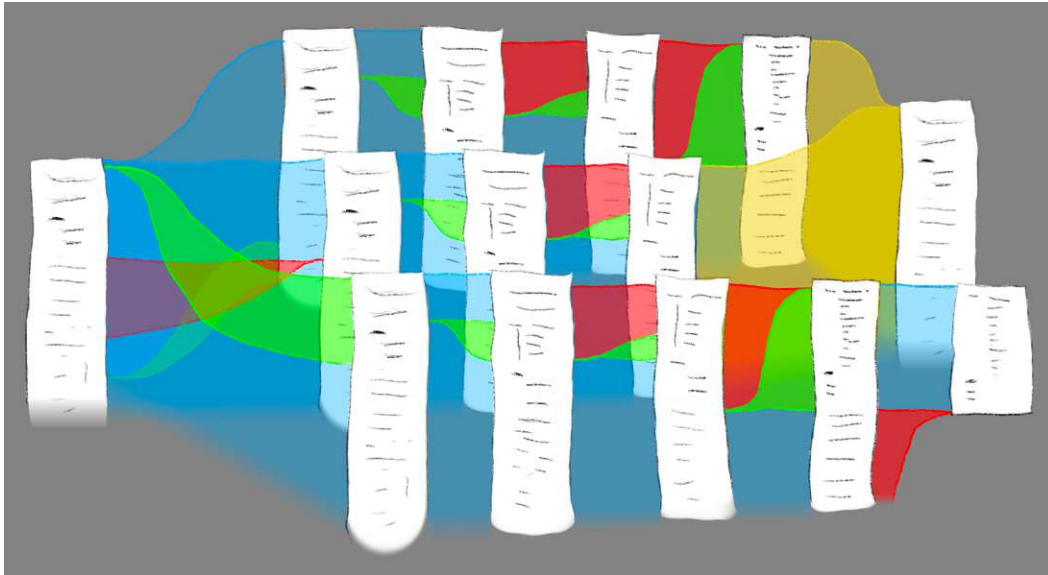
The Proposed Visualization

The visualization centers around the commit graph of the possibly non-linear history of a VCS repository. The visualization shows the evolution of one file. At each node representing a commit, the contents of the file in that commit is displayed in form of a white strip with text.

Between related commits – where one commit is the historical parent of the other commit – translucent strips connect the commit strips, indicating changes between the two commits with colored ribbons. The color indicates the kind of change: red for deletions, green for additions, and blue for unchanged parts. Yellow indicates merges, which are deemed to be complicated to visualize in more detail.

Because the connecting strips between the commits are translucent, they do not completely hide part of the history shown behind them

Sketch of the Proposed Visualization



15. 1. View of the whole commit graph in the context of a file

Refer to the graphic above for an example

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

16. 2. View of multiple branches (more than one, but not necessarily all of them) side by side

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

3. Simple indication of changed code pieces between related commits

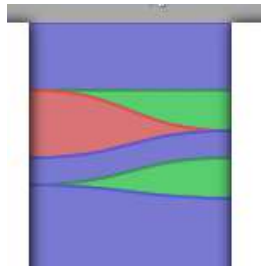
In a connecting strip between two commits, colored ribbons indicate changes between the two states of the file.

- Red indicates a block, that is deleted from the older to the newer version
- Green indicates a newly added block
- Blue indicates unchanged blocks

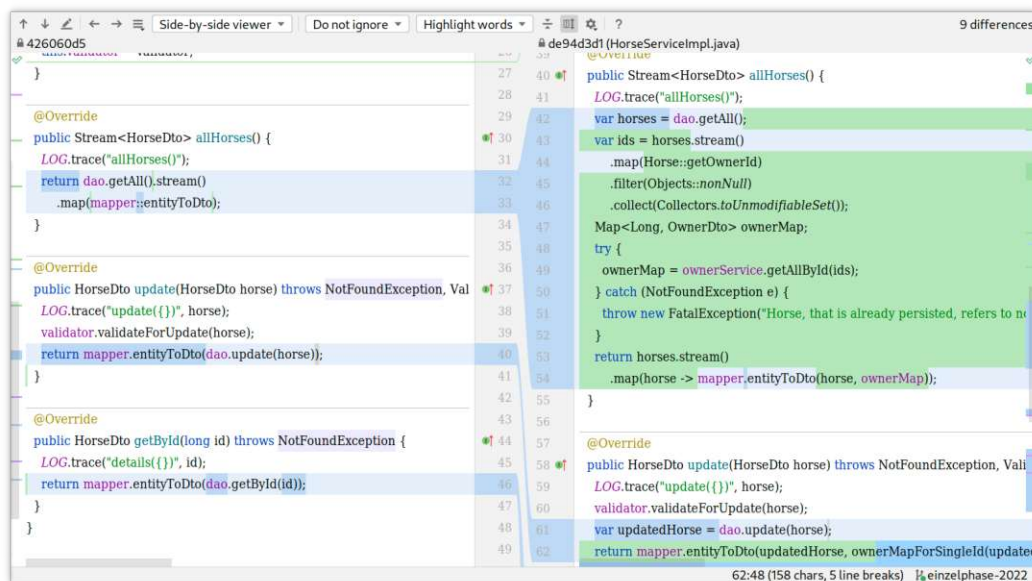
Examples are provided with screenshots from a mockup prototype and the diff views of Kompare and IntelliJ

Simple difference indicators

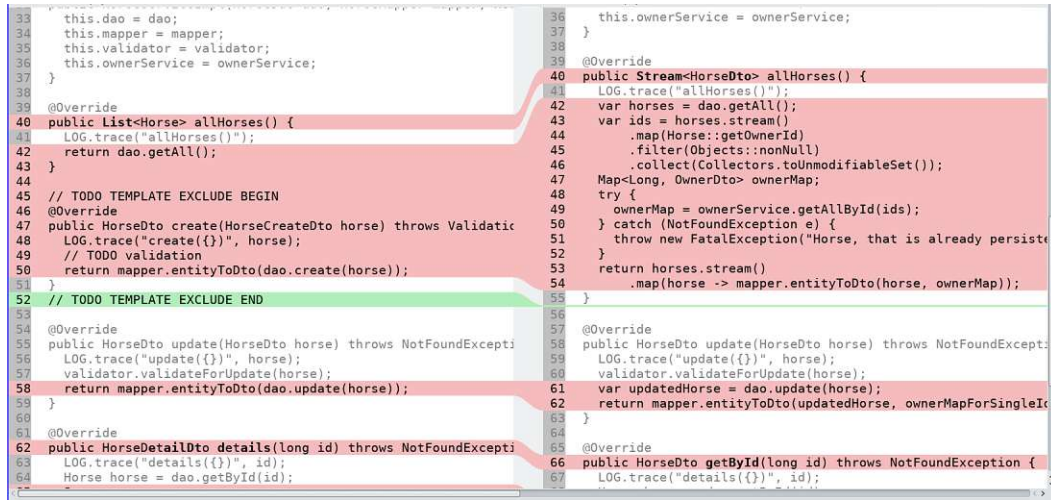
Red parts indicate deleted lines, green parts added lines, and blue unchanged parts



Diff view in IntelliJ



Diff view of Kompare



```

33  this.dao = dao;
34  this.mapper = mapper;
35  this.validator = validator;
36  this.ownerService = ownerService;
37  }
38
39  @Override
40  public List<Horse> allHorses() {
41    LOG.trace("allHorses()");
42    return dao.getAll();
43  }
44
45  // TODO TEMPLATE EXCLUDE BEGIN
46  @Override
47  public HorseDto create(HorseCreateDto horse) throws Validati
48    LOG.trace("create({})", horse);
49    // TODO validation
50    return mapper.entityToDto(dao.create(horse));
51  }
52  // TODO TEMPLATE EXCLUDE END
53
54  @Override
55  public HorseDto update(HorseDto horse) throws NotFoundExcepti
56    LOG.trace("update({})", horse);
57    validator.validateForUpdate(horse);
58    return mapper.entityToDto(dao.update(horse));
59  }
60
61  @Override
62  public HorseDetailDto details(long id) throws NotFoundExcepti
63    LOG.trace("details({})", id);
64    Horse horse = dao.getById(id);
36  this.ownerService = ownerService;
37  }
38
39  @Override
40  public Stream<HorseDto> allHorses() {
41    LOG.trace("allHorses()");
42    var horses = dao.getAll();
43    var ids = horses.stream()
44      .map(Horse::getOwnerId)
45      .filter(Objects::nonNull)
46      .collect(Collectors.toUnmodifiableSet());
47    Map<Long, OwnerDto> ownerMap;
48    try {
49      ownerMap = ownerService.getAllById(ids);
50    } catch (NotFoundException e) {
51      throw new FatalException("Horse, that is already persiste
52    }
53    return horses.stream()
54      .map(horse -> mapper.entityToDto(horse, ownerMap));
55  }
56
57  @Override
58  public HorseDto update(HorseDto horse) throws NotFoundExcepti
59    LOG.trace("update({})", horse);
60    validator.validateForUpdate(horse);
61    var updatedHorse = dao.update(horse);
62    return mapper.entityToDto(updatedHorse, ownerMapForSingleId
63  }
64
65  @Override
66  public HorseDto getById(long id) throws NotFoundException {
67    LOG.trace("details({})", id);
  
```

17. 3. Simple indication of changed code pieces between related commits

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

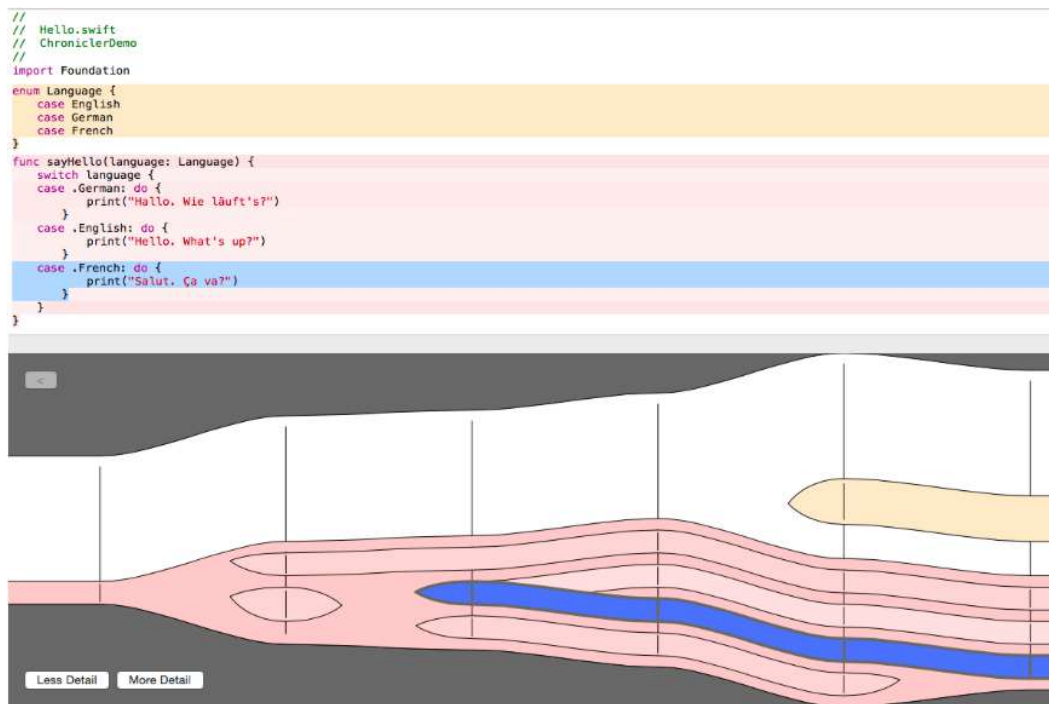
4. Indicate long term structural code change between commits (Code Flows)

As opposed to the simple scheme described above, Code Flows is an alternative, more elaborate scheme for indicating code changes that involves identifying code blocks and their changes and movement continuously over longer sections of the history. See below for an illustrated example using an existing tool that uses Code Flows

Structural difference overview (Code Flows, pictured: Chronicer).

The black lines mark the places of commits on a timeline and the colored areas show the evolution of the code parts marked in the same color in the code view above.

In the proposed visualization, instead of having a split between timeline and code and the black commit marker lines, the state of the code in the respective commits would be directly interspersed in the Code Flow view.



18. 4. Indicate long term structural code change between commits (Code Flows)

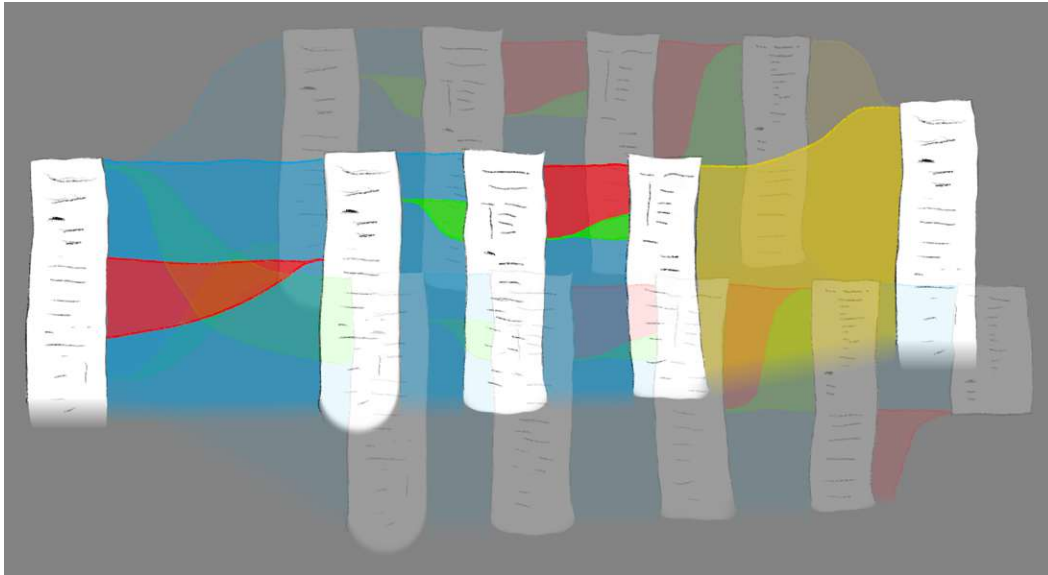
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

5. Focusing individual select branches by fading or hiding others

130 To get a better view at an individual branch, others can be faded or completely hidden.

A single branch is focused and others are faded



19. 5. Focusing individual select branches by fading or hiding others

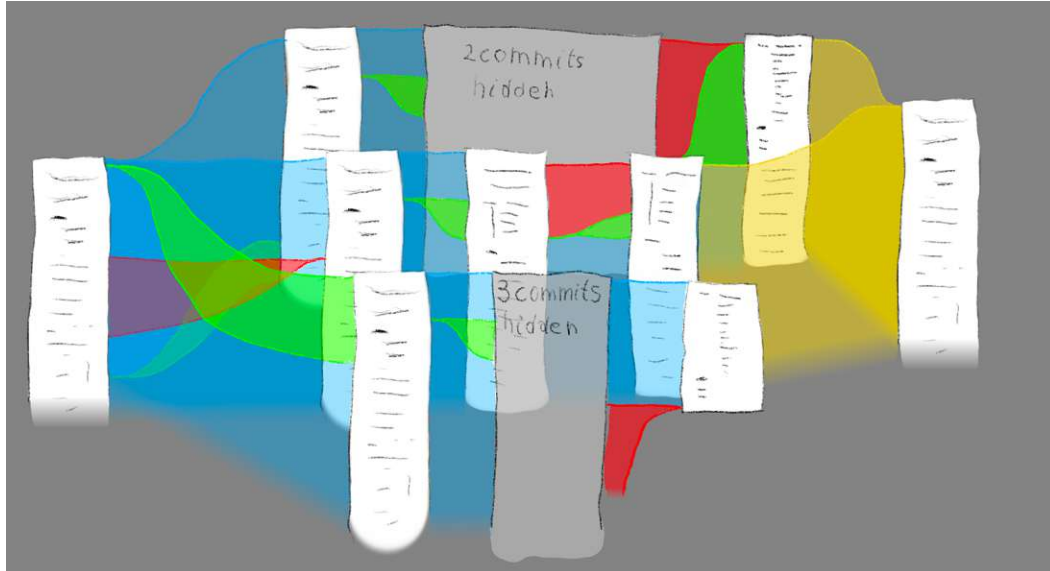
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

6. Folding of linear parts of the history, that contain no branching points

Only the first and last commit of a linear run of commits – where no commit has more than one predecessor or successor – is shown, with a marker in between that displays the fact that commits have been hidden.

On the branch in the front, 3 commits are hidden; in the branch in the back, 2 commits are hidden



20. 6. Folding of linear parts of the history, that contain no branching points

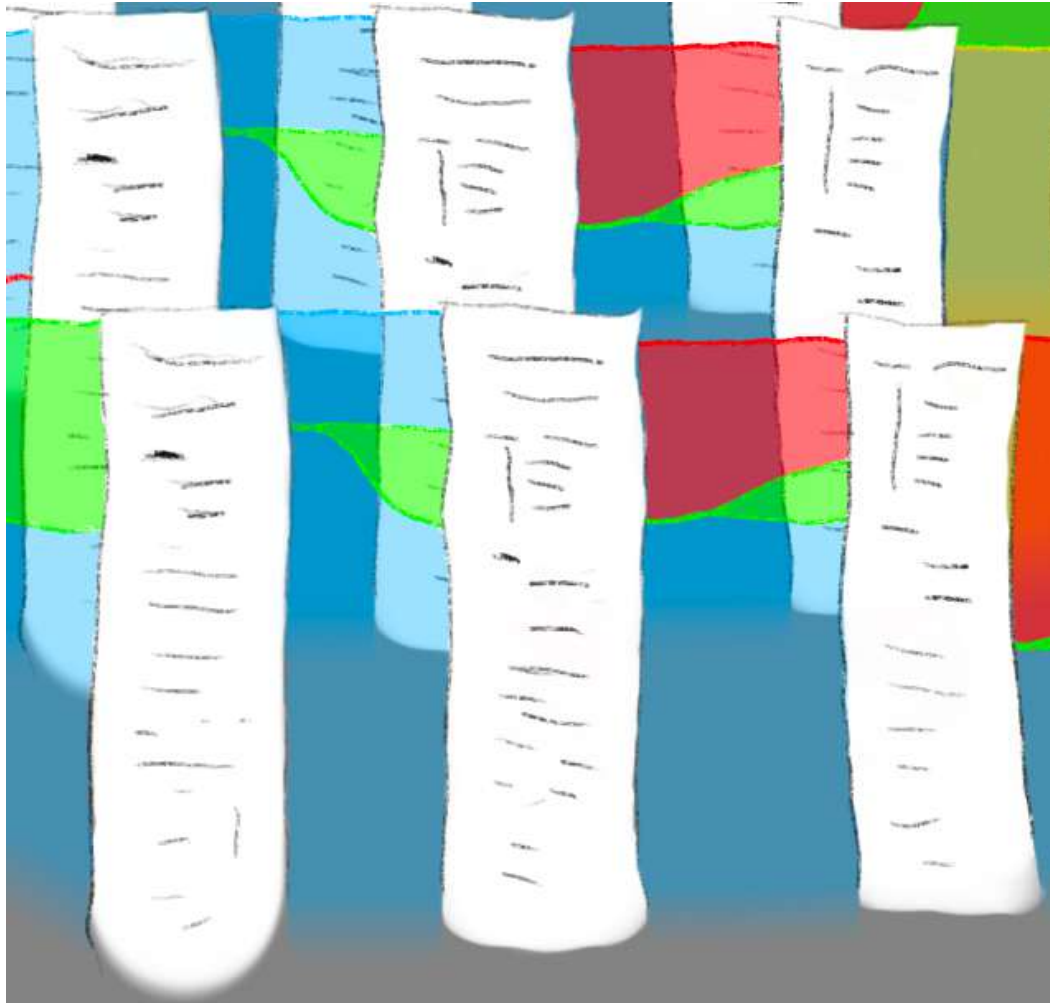
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

7. Zooming in, to see a few related commits in more detail

Similar to zooming text in a text document, zooming into a part of the history to be able to inspect it in more detail, in exchange for less commits visible at the same time.

Zooming in for more detail



21. 7. Zooming in, to see a few related commits in more detail

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

A. QUESTIONNAIRE OF THE INTERVIEWS ON THE CONCEPT

22. 8. Zooming out, to get more overview of the historic development with less detail

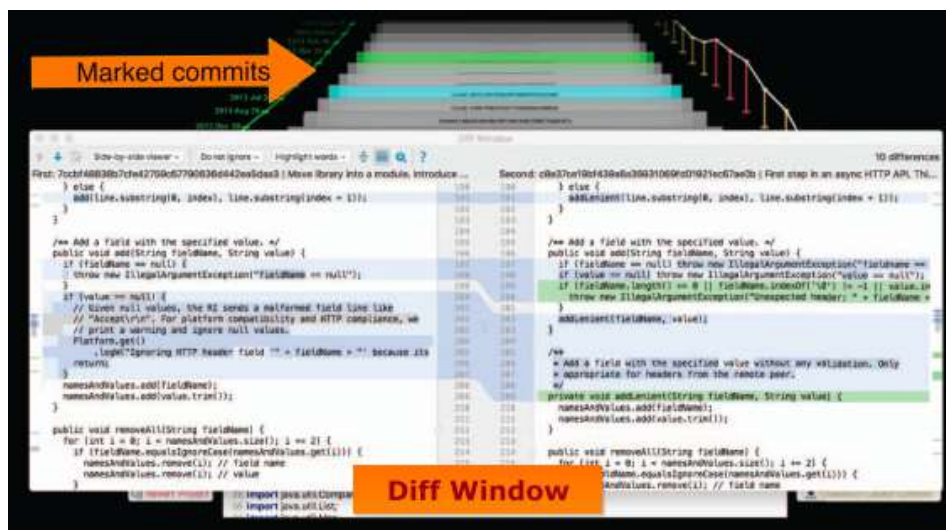
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

9. Direct comparison of two commits by selecting them

The selected commits are highlighted in the history timeline and the code, together with difference indicators, is shown in the foreground.

Direct Diff of two selected commits (pictured: Code Time Machine)



23. 9. Direct comparison of two commits by selecting them

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

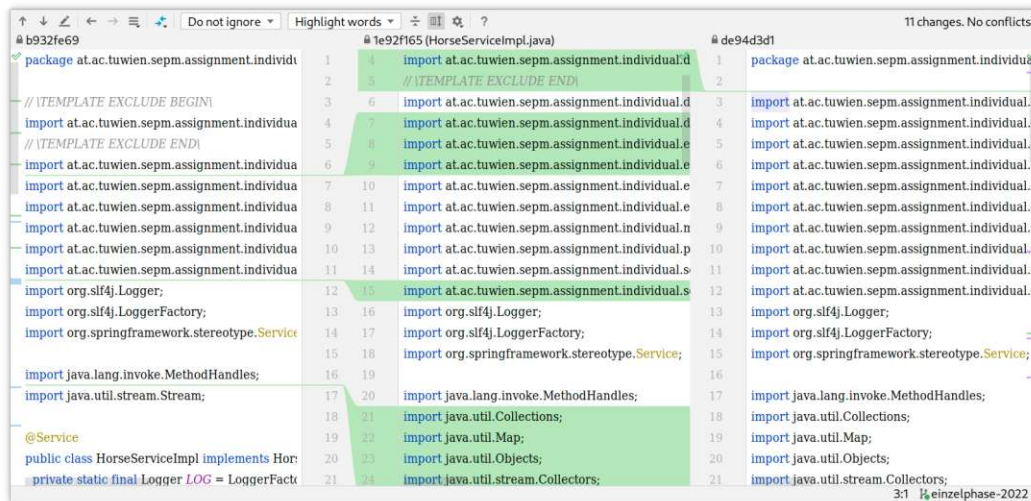
10. Direct comparison of two commits with regards to a common base commit (3 way diff) by selecting them

Both selected commits are compared with common base commit, showing the diverged evolution from there.

Alternatively, this can also be used to compare two commits and the commit that merges them

3-way diff view (pictured: IntelliJ)

Such a view can be used to compare two diverged commits to their common base, or to compare a merge commit with both of its ancestors.



24. 10. Direct comparison of two commits with regards to a common base commit (3 way diff) by selecting them

Markieren Sie nur ein Oval.

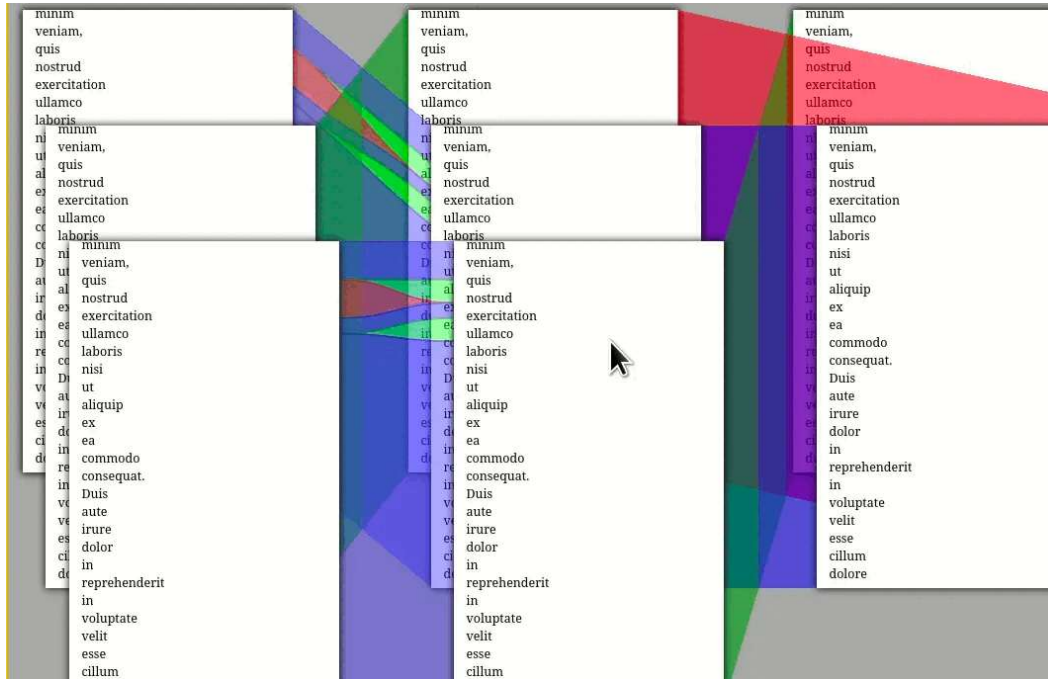
	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

11. Synchronized scrolling of text in all commits

Scrolling the text in one commit view to scroll all others with it at the same time.

Synchronized scrolling example

Scrolling one code strip, scrolls all others simultaneously



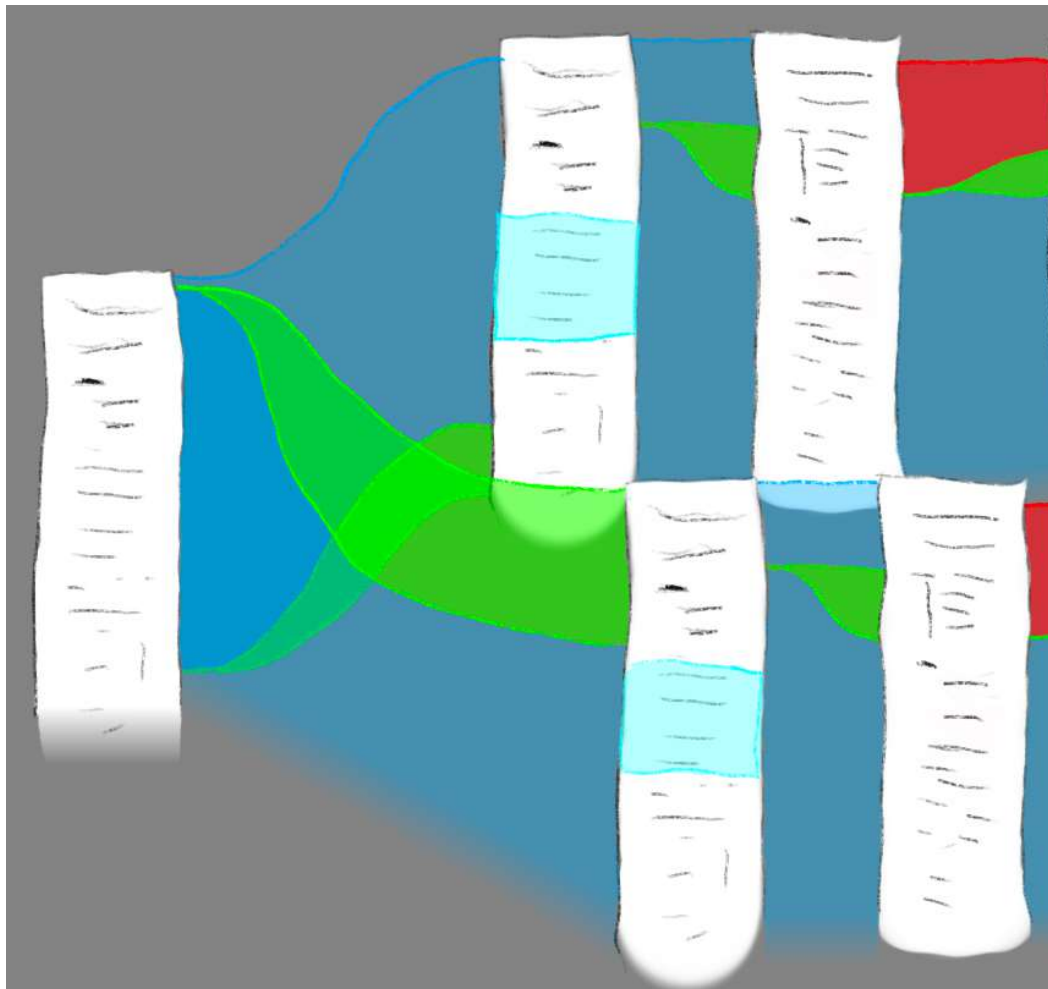
25. 11. Synchronized scrolling of text in all commits

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

12. Selecting a block of code in one commit and highlighting the corresponding block in another commit

Block highlighted in two commits: In the one, where the block was selected and in the commit, that was selected for comparison.



26. 12. Selecting a block of code in one commit and highlighting the corresponding block in another commit

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

13. Selecting a block of code in one commit and highlighting it over the whole history

Code block highlighted in every commit of the history, where there is a corresponding block.



27. 13. Selecting a block of code in one commit and highlighting it over the whole history

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

Purpose of the Visualization as a Whole

28. How purposeful is the proposed visualization?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very purposeful

29. Is the visualization helpful in getting an overview over the evolution of code?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not helpful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very helpful

30. How helpful is the proposed visualization in teaching Software Engineering practice?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not helpful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very helpful

31. Would you like to see a feature in the visualization that was not presented?

List of presented features:

1. View of the whole commit graph in the context of a file
2. View of multiple branches (more than one, but not necessarily all of them) side by side
3. Simple indication of changed code pieces between related commits
4. Indicate long term structural code change between commits (Code Flows)
5. Focusing individual select branches by fading or hiding others
6. Folding of linear parts of the history, that contain no branching points
7. Zooming in, to see a few related commits in more detail
8. Zooming out, to get more overview of the historic development with less detail
9. Direct comparison of two commits by selecting them
10. Direct comparison of two commits with regards to a common base commit (3 way diff) by selecting them
11. Synchronized scrolling of text in all commits
12. Selecting a block of code in one commit and highlighting the corresponding block in another commit
13. Selecting a block of code in one commit and highlighting it over the whole history

32. Do you have any further remarks?

APPENDIX **B**

Questionnaire of the Prototype Evaluation

The questionnaire which was used for the evaluation of the prototype (see Chapter 7 Semi-Structured Expert Evaluation with the Prototype) is included here. The questionnaire was created with Google Forms [W89]. For the interviews, the web form was used. It was exported as PDF for inclusion in this thesis. The included questionnaire begins on the next page.

Evaluation of the Prototype

Demography

1. How many years of experience do you have in Software Engineering (SE)?

2. How many years of experience do you have with working with Version Control Software (VCS)?

3. What VCSs do you currently use?

Wählen Sie alle zutreffenden Antworten aus.

- Git
 Subversion (SVN)
 Concurrent Versions System (CVS)
 Mercurial (Hg)
 Sonstiges: _____

4. What VCSs have you used in the past?

Wählen Sie alle zutreffenden Antworten aus.

- Git
 Subversion (SVN)
 Concurrent Versions System (CVS)
 Mercurial (Hg)
 Sonstiges: _____

5. Which VCS do you use the most?

Wählen Sie alle zutreffenden Antworten aus.

- Git
- Subversion (SVN)
- Concurrent Versions System (CVS)
- Mercurial
- Sonstiges: _____

6. Have you used alternative frontends when working with VCSs in the past?
Which ones?

7. Have you used repository mining tools to extract more data about a VCS
history in the past? Which ones?

8. Have you used visualization tools to aid your work with VCSs? Which ones?

B. QUESTIONNAIRE OF THE PROTOTYPE EVALUATION

9. Have you used other tools to aid your work with VCSs? Which ones?

Demonstration of the Prototype

Purpose of Individual Aspects of the Prototype

10. Limiting the shown graph to a selected set of branches

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

11. Zooming in, to see a few related commits in more detail

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

12. Zooming out, to get more overview of the historic development with less detail

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

13. Synchronized scrolling of text in all commits

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

14. Isolated scrolling of text in an individual, selected commit

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

15. Manipulation of aspects of the presentation (gaps between commits, size of commit blocks, ...)

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

Proposed Features Not Implemented in the Prototype

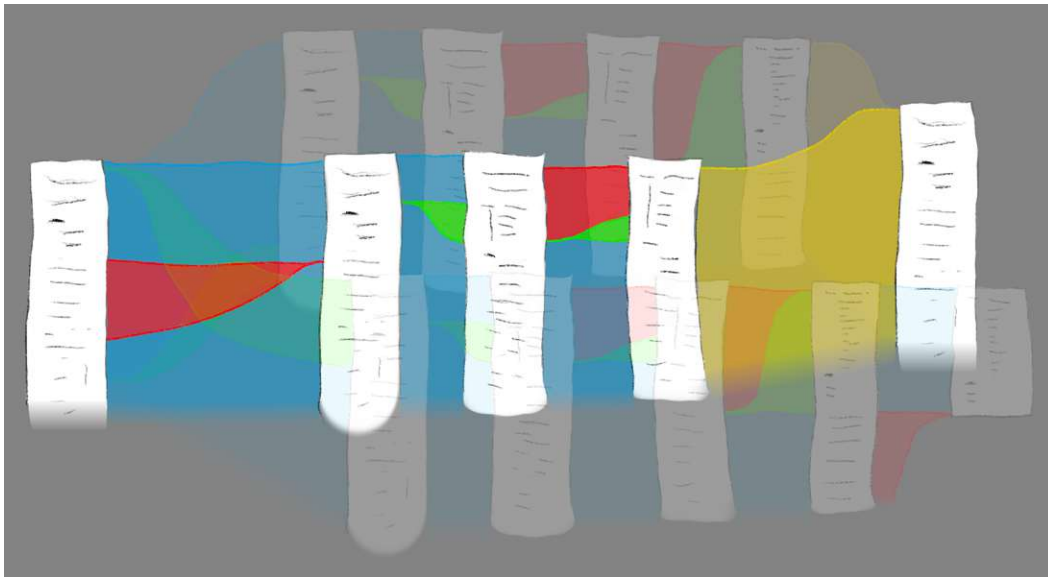
This section discusses features, that are considered for inclusion in future versions.

Focusing individual select branches by fading or hiding others

To get a better view at an individual branch, others can be faded or completely hidden.

B. QUESTIONNAIRE OF THE PROTOTYPE EVALUATION

A single series of commits is focused, other branches are faded or hidden



16. Focusing individual select branches by fading or hiding others

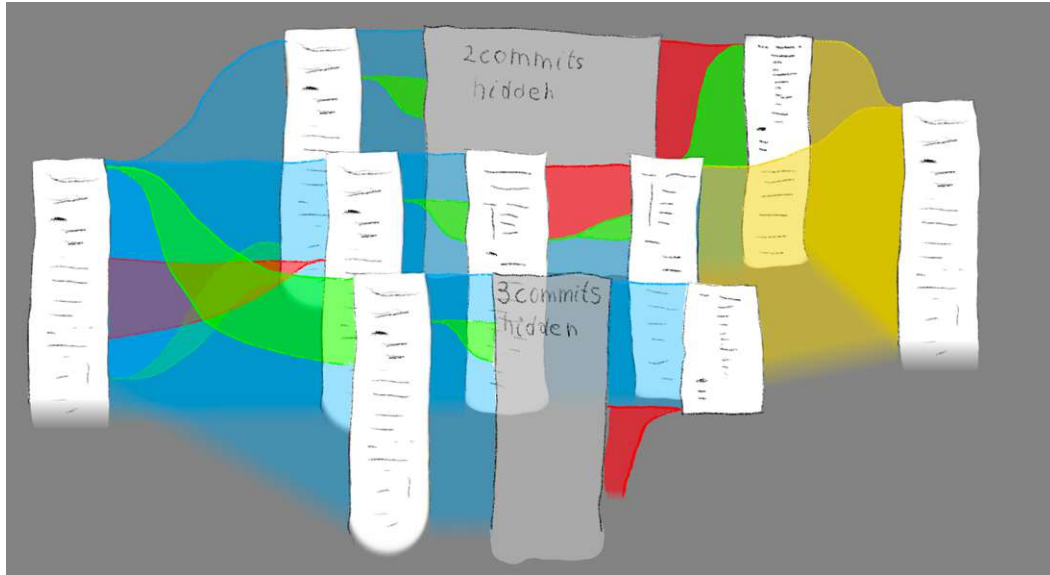
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

Folding of linear parts of the history, that contain no branching points

Only the first and last commit of a linear run of commits – where no commit has more than one predecessor or successor – is shown, with a marker in between that displays the fact that commits have been hidden.

On the branch in the front, 3 commits are hidden; in the branch in the back, 2 commits are hidden



17. Folding of linear parts of the history, that contain no branching points

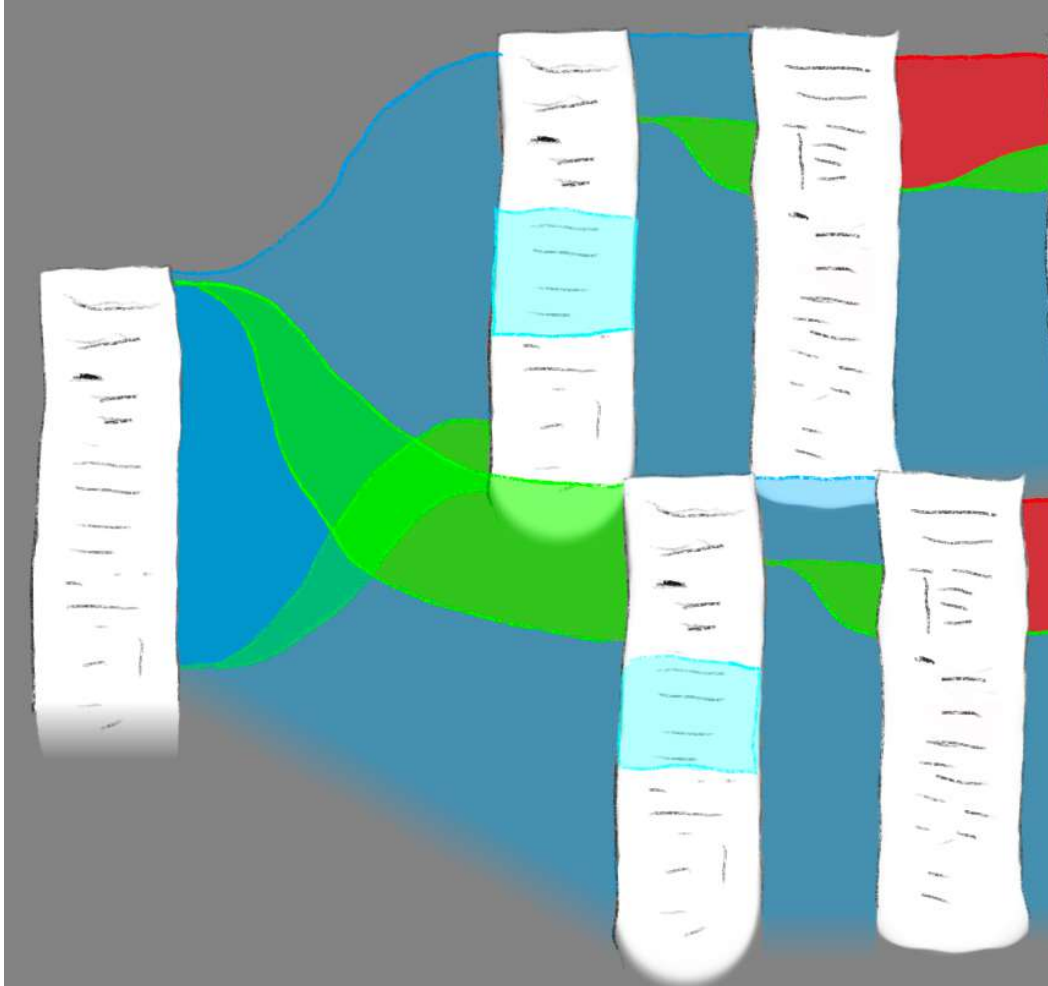
Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

Selecting a block of code in one commit and highlighting the corresponding block in another commit

B. QUESTIONNAIRE OF THE PROTOTYPE EVALUATION

Block highlighted in two commits: In the one, where the block was selected and in the commit, that was selected for comparison.



18. Selecting a block of code in one commit and highlighting the corresponding block in another commit

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

-
19. Selecting a block of code in one commit and highlighting it over the whole history

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not important	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very important

Purpose of the Visualization Presented in the Prototype

20. How purposeful is the proposed visualization?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not purposeful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	purposeful

21. Is the visualization helpful in getting an overview over the evolution of code?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not helpful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very helpful

22. How helpful is the visualization for identifying possible merge conflicts in a file, that has been changed in multiple branches?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not helpful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very helpful

B. QUESTIONNAIRE OF THE PROTOTYPE EVALUATION

23. How helpful is the proposed visualization in teaching Software Engineering practice?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not helpful	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	very helpful

24. Would you use the proposed visualization as part of your workflow?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
unlikely	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	definitely

25. Would you like to see a feature in the visualization that was not presented?

26. Do you have any further remarks?

List of Figures

1.1	Situation in <i>5D Chess</i> [W2]	3
2.1	Example history before (left) and after (right) rebasing of <i>B</i> to <i>A</i>	14
2.2	IntelliJ [W54] graphically presents a diff. Changed code blocks are marked blue.	15
3.1	Example of SplitStreams [41] visualization	19
3.2	Partial sample screenshots of Code Time Machine (top left) [43], Chronicler (top right) [44] and Chronos (bottom) [42].	20
3.3	Main view of TypeV [45] with a long term change overview.	21
3.4	History graph of VisGi [4] with sunburst diagrams of the file and directory structure at the given commits.	21
3.5	Git history graph view, as provided by Gitk [W62].	22
3.6	History graph with patches as shown by <code>git log --graph --patch</code>	24
4.1	Overview of the state of a file in several commits across multiple branches	27
4.2	Overview with one branch highlighted. The other branches are faded. . .	29
4.3	Linear sequences of commits are collapsed.	31
4.4	Diff view with additions and deletions in Kompare	32
4.5	How a Code Flows difference visualization represents the difference between two versions based on their AST [46].	33
4.6	Tree Flows in the bottom view of Chronicler [44]	34
4.7	Example of SplitStreams [41]	35
4.8	Example of a three-way diff view in IntelliJ [W54] of two versions against a merge.	36
4.9	Wrong appearance of crossing connections in simple, 2D layout	38
4.10	Relationships between the corner points of two commit blocks (gray) and the connection block (blue) width h and rotation angle δ . The side d corresponds to the tangent of δ in the circle c of radius w with its center at S	40
4.11	Difference indicators produced with Beziér curves in SVG.	41
5.1	Experience of the interview participants. Note that, due to the small sample size of 3, the quartiles are not meaningful.	45
5.2	VCSs used by the interview participants.	45
		151

5.3	Ratings on the importance of overview over non-linear history and the helpfulness of visualizations for that purpose.	47
5.4	Ratings about the importance to be able to see changes to a piece of code over the history.	47
5.5	Importance of simple and direct navigational control.	48
5.6	Importance of the overview of the whole graph (F.01) and parallel branches (F.02).	49
5.7	Importance ratings of difference indications (F.03, F.04).	50
5.8	Ratings for <i>Focusing on individual branches</i> (F.05) and <i>Folding of linear history parts</i> F.06	50
5.9	Importance ratings for zooming the view of history graph (F.07, F.08) . .	51
5.10	Importance ratings for direct comparison of selected commits (F.09, F.10)	51
5.11	Importance ratings for synchronized scrolling of commits (F.11)	52
5.12	Importance ratings for highlighting a selected block in other commits (F.12, F.13)	52
5.13	Ratings of the purposefulness of the proposed visualization	53
5.14	Ratings for the helpfulness for getting an overview of code evolution . . .	54
5.15	Ratings of the helpfulness in the teaching of SE practice	54
6.1	Simple visualization of two results of the graph layering. The blue circles represent commits, the gray dots are dummy node, the red lines are the segmented edges. The graphs are shown rotated from vertical to horizontal.	69
6.2	Example of commit graph layout without edges.	72
6.3	Example of commit graph layout with text blocks. Difference indications are dummies.	76
6.4	Points defining the curves used to display the visual difference indications. The dashed lines mark the control polygons of the two Beziér curves. . . .	81
6.5	Indications of differences between two commits.	82
6.6	Long edge composed of multiple segments displays difference indications incorrectly.	83
6.7	Part of the panel for graphical settings.	87
6.8	Unfiltered graph (top) and graph with only a few references selected (bottom).	90
6.9	Code without indentation (left) and with proper indentation (right). . . .	91
6.10	Edge with difference indication spanning multiple layers.	93
6.11	Handle over commit block with commit ID and stack of references.	95
7.1	Participants' experience in the field	102
7.2	VCSs used by participants.	103
7.3	Purposefulness of limiting the displayed history to selected references (F.05)	104
7.4	Purposefulness of zooming in and out for more detail or overview (F.07 and F.08)	105
7.5	Purposefulness of two different variations of scrolling the text in commit blocks (F.11).	106

7.6	Rating of purposefulness of the ability to manipulate aspects of the graphical presentation.	106
7.7	Importance of focusing on individual branches by fading others (F.05) . .	107
7.8	Importance of folding linear parts of the history without branching points (F.06)	108
7.9	Importance of highlighting a selected block (F.12 and F.13).	108
7.10	Purposefulness of the visualization as presented in the prototype	109
7.11	Is the visualization helpful in getting an overview over the evolution of code?	110
7.12	Helpfulness of the visualization for identifying potential merge conflicts. .	110
7.13	Helpfulness of the visualization for teaching.	110
7.14	Probability for the participants to use the visualization in their workflow	111



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1	Proposed features for the visualization	26
-----	---	----



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Listings

6.1	CSS for positioning commit graph node blocks	71
6.2	Calculation of rotation angle and edge element width in CSS	74
6.3	CSS for positioning and transforming graph edge blocks	75



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

6.1	Building the next layer	62
6.2	Improving node positions of one layer during a down sweep	67
6.3	Parsing diff hunk lines	79
6.4	Filtering the commit graph	89



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Glossary

- .NET** A platform for cross-platform applications [W81]. 58, 161
- 3-way diff** The differences of two versions of some text with respect to a common ancestor version. See diff. 15, 16
- ASP.NET** A framework for Web services based on .NET [W82]. 58
- Bezier curve** A parametric continuous curve defined by a mathematical formula. Typically described in terms of two end points and a number of control points. 39–41, 80, 81, 95, 151, 152
- Bootstrap** A CSS style framework [W79]. 57
- Branch** Separate development lines in a VCS, typically contrasted with trunk. See Paragraph *Branches* in Section 2.3.3. 1–3, 7, 11–14, 17, 20–23, 26–30, 34, 36, 37, 47–50, 55, 58, 66, 70, 87, 88, 93, 94, 96, 97, 100, 101, 105, 107, 109, 111, 112, 117–119, 151–153
- Clamping** Fitting a number into a given interval. For some number n and an interval $[l, u]$, clamping ensures that for the result n_r the inequality $l \leq n_r \leq u$ holds. If $n < l$ then $n_r = l$; if $n > u$ then $n_r = u$; otherwise $n_r = n$. 67
- Commit** The representation of a version in a VCS. See Section 2.3.1 Versions. 1, 3, 10–15, 17–23, 25–31, 33–38, 40, 46, 48, 51, 52, 54, 57–59, 61, 68–78, 80, 82–89, 91–97, 100–102, 105–108, 111, 112, 115, 117–119, 151, 152, 157, 159
- Darcs** An open source DVCS. 11
- Diff** The differences between two versions of some text, usually human readable. 15, 151, 161
- Directed acyclic graph** A graph where the edges have a direction and that contains no cycles. Often shorted to DAG. 11, 14, 26, 161, 163
- F#** A programming language focused on the functional and object oriented paradigms, targeting .NET [W80]. 58

- Garbage Collection** Automatic memory management technique. Objects which are not referenced anymore are discarded and their memory reclaimed, without requiring the programmer to free them explicitly. 88, 96
- Git** A widespread open source DVCS [W1]. 8, 9, 11–16, 22, 23, 26, 29–31, 45, 46, 58, 59, 61, 66, 87, 88, 93, 94, 100, 102, 103, 118, 119, 151, 162
- JavaScript** Dynamically typed, prototype based, object oriented programming language supported by Web Browsers. 18, 88, 97, 117
- Mercurial** An open source DVCS similar to Git [W7]. Sometimes also known by its command name `hg`. 8, 9, 11–14, 45, 102, 103, 162
- SHA-1** A cryptographic hash function. 11
- Subversion** A centralized VCS, that aims to be the successor of CVS. Also known by its initials and command name `svn` [W28]. 7–14, 18, 45, 102, 103
- Svelte** A Web UI framework for building SPA. 57, 86
- Tag** Designated version in a VCS. Used for example for marking releases. See Paragraph *Tags* in Section 2.3.3. 12, 13, 58, 87, 90, 93, 94
- Trunk** Main line of development in a VCS. See Paragraph *Trunk* in Section 2.3.3. 12–14, 29, 161

Acronyms

- API** Application Programming Interface. 36, 39, 57
- AST** Abstract Syntax Tree. 20, 31, 33, 151
- CLI** Command Line Interface. 46
- CSS** Cascading Style Sheets. 18, 36–39, 70, 71, 73–75, 81, 86, 89, 96, 97, 117, 118, 157, 161
- CVS** Concurrent Versions System. 7, 9–16, 45, 102, 103, 162
- DAG** directed acyclic graph. 11, 14, 26, 161
- DOM** Document Object Model. 36, 37, 57, 84, 92, 97, 119
- DSL** Domain Specific Language. 18
- DVCS** Distributed Version Control System. xiii, xv, 1, 8–12, 14, 21, 25, 26, 161, 162
- FOSS** Free and Open Source. 8, 17
- HTML** Hypertext Markup Language. 18, 36, 37, 40, 71, 82–85, 89, 90, 97, 117, 118
- HTTP** Hypertext Transfer Protocol. 57, 59
- IDE** Integrated Development Environment. xv, 15, 19, 22, 34, 100, 102, 103, 118, 119
- ILP** Integer Linear Programming. 63
- MSR** Mining Software Repositories. xiii, xv, 5, 17, 18, 25, 117
- RCS** Revision Control System. 7–14, 16
- Regex** Regular Expression. 78
- REST** Representational State Transfer. 18, 58, 59

- SCCS** Source Code Control System. 7–11
- SE** Software Engineering. 4, 43–45, 53–55, 102, 112, 113, 116–118, 152
- SPA** Single Page Application. 57, 162
- SVG** Scalable Vector Graphics. 40, 60, 68, 80, 92, 94, 95, 97, 117
- UI** User Interface. 57, 85–88, 96, 97, 115, 162
- VCS** Version Control System. xiii, xv, 1–5, 7–14, 16–19, 21, 26, 31, 34, 43–46, 100, 102–104, 115, 117, 118, 151, 152, 161, 162

Bibliography

- [1] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, “Cohesive and Isolated Development with Branches”, in *Fundamental Approaches to Software Engineering*, J. de Lara and A. Zisman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 316–331, ISBN: 978-3-642-28872-2.
- [2] V. Kovalenko, F. Palomba, and A. Bacchelli, “Mining File Histories: Should We Consider Branches?”, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, New York, NY, USA: Association for Computing Machinery, 2018, pp. 202–213, ISBN: 9781450359375. DOI: 10.1145/3238147.3238169. [Online]. Available: <https://doi.org/10.1145/3238147.3238169>.
- [3] H. Lee, B.-K. Seo, and E. Seo, “A Git Source Repository Analysis Tool Based on a Novel Branch-Oriented Approach”, in *2013 International Conference on Information Science and Applications (ICISA)*, 2013, pp. 1–4. DOI: 10.1109/ICISA.2013.6579457.
- [4] S. Elsen, “VisGi: Visualizing Git branches”, in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4. DOI: 10.1109/VISSOFT.2013.6650522.
- [5] N. M. Tiwari, G. Upadhyaya, H. A. Nguyen, and H. Rajan, “Candoia: A Platform for Building and Sharing Mining Software Repositories Tools as Apps”, in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 53–63. DOI: 10.1109/MSR.2017.56.
- [6] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, “A History Querying Tool and Its Application to Detect Multi-version Refactorings”, in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 2013, pp. 335–338. DOI: 10.1109/CSMR.2013.44.
- [7] M. H. Bianchi and J. L. Wood, “A user’s viewpoint on the Programmer’s Workbench”, in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE ’76, Washington, DC, USA: IEEE Computer Society Press, 1976, pp. 193–199.

- [8] T. A. Dolotta and J. R. Mashey, “An introduction to the programmer’s workbench”, in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 164–168.
- [9] F. W. Allen, M. E. S. Loomis, and M. V. Mannino, “The Integrated Dictionary/Directory System”, *ACM Comput. Surv.*, vol. 14, no. 2, pp. 245–286, 1982, ISSN: 0360-0300. DOI: 10.1145/356876.356882. [Online]. Available: <https://doi.org/10.1145/356876.356882>.
- [10] S. S. Weston, “Panvalet Security and Control”, *EDPACS*, vol. 6, no. 9, pp. 1–3, 1979. DOI: 10.1080/07366987909449449. [Online]. Available: <https://doi.org/10.1080/07366987909449449>.
- [11] L Manderson, “Fill-in-the-blank JCL: why we did it and how”, in *Proceedings of the 17th Annual ACM SIGUCCS Conference on User Services*, ser. SIGUCCS ’89, New York, NY, USA: Association for Computing Machinery, 1989, pp. 247–252, ISBN: 0897913302. DOI: 10.1145/73760.73801. [Online]. Available: <https://doi.org/10.1145/73760.73801>.
- [12] M. J. Rochkind, “The source code control system”, *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364–370, 1975. DOI: 10.1109/TSE.1975.6312866.
- [13] A. L. Glasser, “The evolution of a Source Code Control System”, *SIGSOFT Softw. Eng. Notes*, vol. 3, no. 5, pp. 122–125, 1978, ISSN: 0163-5948. DOI: 10.1145/953579.811111. [Online]. Available: <https://doi.org/10.1145/953579.811111>.
- [14] N. B. Ruparelia, “The history of version control”, *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 5–9, 2010, ISSN: 0163-5948. DOI: 10.1145/1668862.1668876. [Online]. Available: <https://doi.org/10.1145/1668862.1668876>.
- [15] P. Baudiš, *Current Concepts in Version Control Systems*, 2014. arXiv: 1405.3496 [cs.SE].
- [16] W. F. Tichy, “RCS — a system for version control”, *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985. DOI: <https://doi.org/10.1002/spe.4380150703>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380150703>.
- [17] D. Grune and Others, *Concurrent versions systems, a method for independent cooperation*. VU Amsterdam. Subfaculteit Wiskunde en Informatica, 1986.
- [18] B. Berliner, “CVS II: Parallelizing software development”, in *Proceedings of the USENIX Winter 1990 Technical Conference*, vol. 341, 1990, p. 352.
- [19] M. Shaikh and T. Cornford, “Version management tools: CVS to BK in the Linux kernel”, in *3rd Workshop on Open Source Software Engineering*, 2003, pp. 127–132.

- [20] C. F. Malmsten, “Evolution of Version Control Systems - Comparing Centralized against Distributed Version Control models”, 2010. [Online]. Available: <https://api.semanticscholar.org/CorpusID:54786231>.
- [21] B. de Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?”, in *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009, pp. 36–39. DOI: 10.1109/CHASE.2009.5071408.
- [22] D. Roundy, “Darcs: distributed version management in haskell”, in *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '05, New York, NY, USA: Association for Computing Machinery, 2005, pp. 1–4, ISBN: 159593071X. DOI: 10.1145/1088348.1088349. [Online]. Available: <https://doi.org/10.1145/1088348.1088349>.
- [23] N. Bertino, “Modern version control: creating an efficient development ecosystem”, in *Proceedings of the 40th Annual ACM SIGUCCS Conference on User Services*, ser. SIGUCCS '12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 219–222, ISBN: 9781450314947. DOI: 10.1145/2382456.2382510. [Online]. Available: <https://doi.org/10.1145/2382456.2382510>.
- [24] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, “How do centralized and distributed version control systems impact software changes?”, in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: Association for Computing Machinery, 2014, pp. 322–333, ISBN: 9781450327565. DOI: 10.1145/2568225.2568322. [Online]. Available: <https://doi.org/10.1145/2568225.2568322>.
- [25] N. N. Zolkifli, A. Ngah, and A. Deraman, “Version Control System: A Review”, *Procedia Computer Science*, vol. 135, pp. 408–415, 2018, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2018.08.191>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050918314819>.
- [26] S. P. De Rosso and D. Jackson, “Purposes, concepts, misfits, and a redesign of git”, *SIGPLAN Not.*, vol. 51, no. 10, pp. 292–310, 2016, ISSN: 0362-1340. DOI: 10.1145/3022671.2984018. [Online]. Available: <https://doi.org/10.1145/3022671.2984018>.
- [27] J. W. Hunt and M. D. McIlroy, “An algorithm for differential file comparison”, in *Computing Science Technical Report No. 41*, Murray Hill, N.J.: Bell Telephone Laboratories, 1976.
- [28] P. Heckel, “A technique for isolating differences between files”, *Commun. ACM*, vol. 21, no. 4, pp. 264–268, 1978, ISSN: 0001-0782. DOI: 10.1145/359460.359467. [Online]. Available: <https://doi.org/10.1145/359460.359467>.

- [29] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories”, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, New York, NY, USA: Association for Computing Machinery, 2018, pp. 908–911, ISBN: 9781450355735. DOI: 10.1145/3236024.3264598. [Online]. Available: <https://doi.org/10.1145/3236024.3264598>.
- [30] M. Stöger, *Continuous analysis, monitoring, and comparison of student project portfolios in software engineering courses*, eng. Wien, 2023. DOI: 10.34726/hss.2023.109702. [Online]. Available: <https://doi.org/10.34726/hss.2023.109702>.
- [31] G. Ghezzi and H. Gall, “A framework for semi-automated software evolution analysis composition”, *Automated Software Engineering*, vol. 20, no. 3, pp. 463–496, 2013, ISSN: 0928-8910. DOI: 10.1007/s10515-013-0125-z. [Online]. Available: <http://dx.doi.org/10.1007/s10515-013-0125-z>.
- [32] G. Ghezzi and H. C. Gall, “Towards software analysis as a service”, in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, 2008, pp. 1–10. DOI: 10.1109/ASEW.2008.4686315.
- [33] G. Ghezzi and H. C. Gall, “SOFAS: A Lightweight Architecture for Software Analysis as a Service”, in *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, 2011, pp. 93–102. DOI: 10.1109/WICSA.2011.21.
- [34] G. Ghezzi and H. C. Gall, “Replicating mining studies with SOFAS”, in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR ’13, Piscataway, NJ, USA: IEEE Press, 2013, pp. 363–372, ISBN: 978-1-4673-2936-1. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2487085.2487152>.
- [35] R. Müller, D. Mahler, M. Hunger, J. Nerche, and M. Harrer, “Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization”, in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 107–111. DOI: 10.1109/VISSOFT.2018.00019.
- [36] R. Dyer, H. Nguyen, H. Rajan, and T. Nguyen, “Boa: analyzing ultra-large-scale code corpus”, in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH ’12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 87–88, ISBN: 9781450315630. DOI: 10.1145/2384716.2384752. [Online]. Available: <https://doi.org/10.1145/2384716.2384752>.
- [37] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories”, in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 422–431. DOI: 10.1109/ICSE.2013.6606588.

- [38] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: Ultra-Large-Scale Software Repository and Source-Code Mining”, *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, 2015, ISSN: 1049-331X. DOI: 10.1145/2803171. [Online]. Available: <https://doi.org/10.1145/2803171>.
- [39] F. Servant and J. A. Jones, “History slicing”, in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 452–455. DOI: 10.1109/ASE.2011.6100097.
- [40] F. Servant and J. A. Jones, “History slicing: assisting code-evolution tasks”, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12, New York, NY, USA: ACM, 2012, 43:1–43:11, ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393646. [Online]. Available: <http://doi.acm.org/10.1145/2393596.2393646>.
- [41] F. Bolte, M. Nourani, E. D. Ragan, and S. Bruckner, “SplitStreams: A Visual Metaphor for Evolving Hierarchies”, *IEEE Transactions on Visualization and Computer Graphics*, p. 1, 2020, ISSN: 1941-0506. DOI: 10.1109/TVCG.2020.2973564.
- [42] F. Servant and J. A. Jones, “Chronos: Visualizing slices of source-code history”, in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4. DOI: 10.1109/VISSOFT.2013.6650547.
- [43] E. Aghajani, A. Mocci, G. Bavota, and M. Lanza, “The Code Time Machine”, in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 356–359. DOI: 10.1109/ICPC.2017.6.
- [44] M. Wittenhagen, C. Cherek, and J. Borchers, “Chronicler: Interactive Exploration of Source Code History”, in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’16, New York, NY, USA: ACM, 2016, pp. 3522–3532, ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858442. [Online]. Available: <http://doi.acm.org/10.1145/2858036.2858442>.
- [45] M. D. Feist, E. A. Santos, I. Watts, and A. Hindle, “Visualizing Project Evolution through Abstract Syntax Tree Analysis”, in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 2016, pp. 11–20. DOI: 10.1109/VISSOFT.2016.6.
- [46] A. Telea and D. Auber, “Code Flows: Visualizing Structural Evolution of Source Code”, *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008. DOI: <https://doi.org/10.1111/j.1467-8659.2008.01214.x>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01214.x>.
- [47] A. Marcus, L. Feng, and J. I. Maletic, “3D Representations for Software Visualization”, in *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis ’03, New York, NY, USA: Association for Computing Machinery, 2003, 27–ff, ISBN: 1581136420. DOI: 10.1145/774833.774837. [Online]. Available: <https://doi.org/10.1145/774833.774837>.

- [48] C. Greil, *Conflict awareness by visualisation in multi-branch and multi-project software development*, eng. Wien, 2021. DOI: 10.34726/hss.2022.96883. [Online]. Available: <https://doi.org/10.34726/hss.2022.96883>.
- [49] M. Slade, “A layout algorithm for hierarchical graphs with constraints”, 1994.
- [50] S. Autexier, “Similarity-Based Diff, Three-Way Diff and Merge.”, *Int. J. Softw. Informatics*, vol. 9, no. 2, pp. 259–277, 2015.
- [51] 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*, ser. IWPSE '07, New York, NY, USA: Association for Computing Machinery, 2007, pp. 90–97, ISBN: 9781595937223. DOI: 10.1145/1294948.1294971. [Online]. Available: <https://doi.org/10.1145/1294948.1294971>.
- [52] P. Eades, X. Lin, and R. Tamassia, “An algorithm for drawing a hierarchical graph”, *International Journal of Computational Geometry & Applications*, vol. 6, no. 02, pp. 145–155, 1996, ISSN: 02181959. DOI: 10.1142/S0218195996000101.
- [53] O. Bastert and C. Matuszewski, “Layered drawings of digraphs”, in *Drawing Graphs: Methods and Models*, M. Kaufmann and D. Wagner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–120, ISBN: 978-3-540-44969-0. DOI: 10.1007/3-540-44969-8_5. [Online]. Available: https://doi.org/10.1007/3-540-44969-8_5.
- [54] K. Sugiyama, S. Tagawa, and M. Toda, “Methods for Visual Understanding of Hierarchical System Structures”, *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981, ISSN: 21682909. DOI: 10.1109/TSMC.1981.4308636. [Online]. Available: http://media.wix.com/ugd/6cbb0c_289d09ca219c4c9a8df5bf05c16214dc.pdf.
- [55] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, “A technique for drawing directed graphs”, *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.
- [56] S. Pupyrev, L. Nachmanson, and M. Kaufmann, “Improving layered graph layouts with edge bundling”, in *Graph Drawing: 18th International Symposium, GD 2010, Konstanz, Germany, September 21-24, 2010. Revised Selected Papers 18*, Springer, 2011, pp. 329–340.
- [57] D. Eppstein, M. T. Goodrich, and J. Y. Meng, “Confluent Layered Drawings”, *Algorithmica*, vol. 47, no. 4, pp. 439–452, 2007, ISSN: 1432-0541. DOI: 10.1007/s00453-006-0159-8. [Online]. Available: <https://doi.org/10.1007/s00453-006-0159-8>.

- [58] M. R. Garey and D. S. Johnson, “Crossing Number is NP-Complete”, *SIAM Journal on Algebraic Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983, ISSN: 0196-5212. DOI: 10.1137/0604033. [Online]. Available: https://learn.fmi.uni-sofia.bg/pluginfile.php/160153/mod_resource/content/4/Crossing-Number-Is-NP-Complete_Garey_Johnson.pdf.
- [59] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, “Algorithms for drawing graphs: an annotated bibliography”, *Computational Geometry: Theory and Applications*, vol. 4, no. 5, pp. 235–282, 1994, ISSN: 09257721. DOI: 10.1016/0925-7721(94)00014-X.
- [60] S. Dresbach, “A new heuristic layout algorithm for directed acyclic graphs”, *Operations Research Proceedings*, no. June 1994, pp. 121–126, 1994. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4906&rep=rep1&type=pdf>.
- [61] P. Healy and N. S. Nikolov, “How to layer a directed acyclic graph”, in *Graph Drawing*, Springer, vol. 2265, 2001, pp. 16–30, ISBN: 3540433090. DOI: 10.1007/3-540-45848-4_2.
- [62] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part I”, *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.
- [63] J. McCarthy, “History of LISP”, in *History of programming languages*, 1978, pp. 173–185.
- [64] R. R. Fenichel and J. C. Yochelson, “A LISP garbage-collector for virtual-memory computer systems”, *Communications of the ACM*, vol. 12, no. 11, pp. 611–612, 1969.
- [65] D. E. Knuth, “Lists and Garbage Collection”, in *The Art of Computer Programming*, third edition. Addison-Wesley, 1997, vol. 1, ch. 2.3.5, pp. 408–423, ISBN: 978-0-201-89683-1.
- [66] 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)*, 2018, pp. 76–86. DOI: 10.1109/VISSOFT.2018.00016.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Web References

- [W1] “Git”, [Online]. Available: <https://git-scm.com> (visited on 05/31/2022).
- [W2] “5D Chess with Multiverse Time Travel”, [Online]. Available: https://store.steampowered.com/app/1349230/5D_Chess_With_Multiverse_Time_Travel/ (visited on 12/27/2021).
- [W3] “Subversion Design, Goals”, [Online]. Available: <https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#goals> (visited on 04/03/2024).
- [W4] “BitKeeper”, [Online]. Available: <http://www.bitkeeper.org/> (visited on 06/05/2024).
- [W5] “Git – A Short History of Git”, [Online]. Available: <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git> (visited on 04/03/2024).
- [W6] “LKML: Linux Torvalds: Kernel SCM saga..”, [Online]. Available: <https://lkml.org/lkml/2005/4/6/121> (visited on 04/03/2024).
- [W7] “Mercurial SCM”, [Online]. Available: <https://www.mercurial-scm.org/> (visited on 11/13/2022).
- [W8] “Mercurial v0.1 - a minimal scalable distributed SCM”, [Online]. Available: <https://lore.kernel.org/lkml/42692470.9050605@tmr.com/T/> (visited on 04/03/2024).
- [W9] “monotone”, [Online]. Available: <https://www.monotone.ca/> (visited on 04/03/2024).
- [W10] “Darcs”, [Online]. Available: <https://darcs.net/> (visited on 04/03/2024).
- [W11] “Eclipse Foundation”, [Online]. Available: <https://www.eclipse.org/> (visited on 05/21/2024).
- [W12] “Stack Overflow”, [Online]. Available: <https://stackoverflow.com/> (visited on 05/21/2024).
- [W13] “Stack Overflow Developer Survey 2015”, [Online]. Available: <https://survey.stackoverflow.co/2015#tech-sourcecontrol> (visited on 04/03/2024).

- [W14] “Stack Overflow Developer Survey 2022”, [Online]. Available: <https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems> (visited on 04/03/2024).
- [W15] “GitHub: Let’s build from here · GitHub”, [Online]. Available: <https://github.com/> (visited on 01/20/2024).
- [W16] “BitBucket”, [Online]. Available: <https://https://bitbucket.org/> (visited on 05/21/2024).
- [W17] “The DevSecOps Platform | GitLab”, [Online]. Available: <https://about.gitlab.com/> (visited on 01/20/2024).
- [W18] “The Perl Programming Language”, [Online]. Available: <https://www.perl.org/> (visited on 05/21/2024).
- [W19] “GNU Emacs”, [Online]. Available: <https://www.gnu.org/software/emacs/> (visited on 05/21/2024).
- [W20] “Emacs’s Switch to Git - Mastering Emacs”, [Online]. Available: <https://www.masteringemacs.org/article/emacss-switch-to-git> (visited on 04/03/2024).
- [W21] “Ugliest... repository... conversion... ever”, [Online]. Available: <http://esr.ibiblio.org/?p=5634> (visited on 04/03/2024).
- [W22] “EmacsWiki: Git for Emacs Devs”, [Online]. Available: <https://www.emacswiki.org/emacs/GitForEmacsDevs> (visited on 04/03/2024).
- [W23] “Welcome to Python.org”, [Online]. Available: <https://www.python.org/> (visited on 05/21/2024).
- [W24] “PEP 507 – Migrate CPython to Git and GitLab”, [Online]. Available: <https://peps.python.org/pep-0507/> (visited on 04/03/2024).
- [W25] “The FreeBSD Project”, [Online]. Available: <https://www.freebsd.org/> (visited on 05/21/2024).
- [W26] “Using Git for FreeBSD Development”, [Online]. Available: <https://wiki.freebsd.org/Git> (visited on 04/03/2024).
- [W27] “FreeBSD Completes Its Transition From Subversion To Git For Development”, [Online]. Available: <https://www.phoronix.com/news/FreeBSD-Developing-On-Git> (visited on 04/03/2024).
- [W28] “Apache Subversion”, [Online]. Available: <https://subversion.apache.org/> (visited on 11/13/2022).
- [W29] “Monotone: FAQ”, [Online]. Available: <https://wiki.monotone.ca/FAQ/> (visited on 04/08/2024).
- [W30] “Subversion Design, Transaction and Revision Numbers”, [Online]. Available: <https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#model.txns-and-revnums> (visited on 04/04/2024).

- [W31] “Git – What is Git?”, [Online]. Available: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>.
- [W32] “Git – Basic Branching and Merging”, [Online]. Available: <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging> (visited on 05/14/2024).
- [W33] “Mercurial: the definitive guide: Trends in the field”, [Online]. Available: <https://book.mercurial-scm.org/read/intro.html#trends-in-the-field> (visited on 07/01/2024).
- [W34] “A tour of Mercurial: merging work”, [Online]. Available: <https://hgbook.red-bean.com/read/a-tour-of-mercurial-merging-work.html> (visited on 05/14/2024).
- [W35] “CVS–Recursive Behaviour”, [Online]. Available: https://www.gnu.org/software/trans-coord/manual/cvs/html_node/Recursive-behavior.html#Recursive-behavior (visited on 04/04/2024).
- [W36] “CVS: The normal way to rename”, [Online]. Available: https://www.gnu.org/software/trans-coord/manual/cvs/html_node/Outside.html#Outside (visited on 06/15/2024).
- [W37] “CVS: Moving the history file”, [Online]. Available: https://www.gnu.org/software/trans-coord/manual/cvs/html_node/Inside.html#Inside (visited on 06/15/2024).
- [W38] “CVS: Rename by copying”, [Online]. Available: https://www.gnu.org/software/trans-coord/manual/cvs/html_node/Rename-by-copying.html#Rename-by-copying (visited on 06/15/2024).
- [W39] “Subversion Design, Locking vs. Merging - Two Paradigms of Co-operative Developments”, [Online]. Available: <https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#model.lock-merge> (visited on 04/05/2024).
- [W40] “Basic Merging, Merge Tracking”, [Online]. Available: <https://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html#svn.branchmerge.basicmerging.mergetracking> (visited on 04/05/2024).
- [W41] “The Darcs Book: Getting started, Terminology”, [Online]. Available: <https://darcsbook.acmelabs.space/chapter02.html#terminology> (visited on 05/13/2024).
- [W42] “The Darcs Book: Working locally”, [Online]. Available: <https://darcsbook.acmelabs.space/chapter03.html> (visited on 05/13/2024).
- [W43] “Fossil: The History And Purpose Of Fossil”, [Online]. Available: <https://www2.fossil-scm.org/home/doc/trunk/www/history.md> (visited on 04/08/2024).

- [W44] “Mercurial: the definitive guide: A short history of revision control”, [Online]. Available: <https://book.mercurial-scm.org/read/intro.html#a-short-history-of-revision-control> (visited on 07/01/2024).
- [W45] “What Mother never told you about SVN Branching and Merging”, [Online]. Available: <https://designbygravity.wordpress.com/2009/10/19/what-mother-never-told-you-about-svn-branching-and-merging/> (visited on 04/24/2024).
- [W46] “Branching and Merging – CVS GUI”, [Online]. Available: <http://cvsgui.org/cvs-concurrent-versions-system/branching-and-merging/> (visited on 05/14/2024).
- [W47] “Subversion Best Practices”, [Online]. Available: <https://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html> (visited on 07/13/2024).
- [W48] “Planning Your Repository Organization”, [Online]. Available: <https://svnbook.red-bean.com/nightly/en/svn.reposadmin.planning.html#svn.reposadmin.projects.chooselayout> (visited on 07/13/2024).
- [W49] “First Time Git Setup: Your default branch name”, [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#_new_default_branch.
- [W50] “Standard Branching: The default branch”, [Online]. Available: <https://wiki.mercurial-scm.org/StandardBranching> (visited on 07/13/2024).
- [W51] “Subversion Design: Merging and Ancestry”, [Online]. Available: <https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#model.merging-and-ancestry> (visited on 05/14/2024).
- [W52] “Git – merge strategies”, [Online]. Available: <https://git-scm.com/docs/merge-strategies> (visited on 05/14/2024).
- [W53] “Git - git diff Documentation”, [Online]. Available: <https://git-scm.com/docs/git-diff> (visited on 05/20/2024).
- [W54] “IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains”, [Online]. Available: <https://www.jetbrains.com/idea/> (visited on 12/20/2021).
- [W55] “Patch version 1.3”, [Online]. Available: <https://groups.google.com/g/mod.sources/c/xSQM63e39YY/m/apNNJSkji0gJ> (visited on 07/12/2024).
- [W56] “Git - git-apply Documentation”, [Online]. Available: <https://git-scm.com/docs/git-apply> (visited on 05/21/2024).
- [W57] “Eclipse IDE | The Eclipse Foundation”, [Online]. Available: <https://eclipseide.org/> (visited on 01/20/2024).
- [W58] “JUnit”, [Online]. Available: <https://junit.org/> (visited on 06/01/2024).
- [W59] “Apache Software Foundation”, [Online]. Available: <https://apache.org/> (visited on 06/01/2024).

- [W60] “GitHub Desktop”, [Online]. Available: <https://desktop.github.com/> (visited on 12/20/2021).
- [W61] “BitKeeper Documentation, revtool”, [Online]. Available: <http://www.bitkeeper.org/man/revtool.html> (visited on 06/05/2024).
- [W62] “Git - gitk Documentation”, [Online]. Available: <https://git-scm.com/docs/gitk> (visited on 06/05/2024).
- [W63] “Git - git-log Documentation”, [Online]. Available: <https://git-scm.com/docs/git-log> (visited on 06/05/2024).
- [W64] “GitKraken Legendary Git Tools | GitKraken”, [Online]. Available: <https://www.gitkraken.com/> (visited on 01/20/2024).
- [W65] “HgKExtension - Mercurial”, [Online]. Available: <https://wiki.mercurial-scm.org/HgkExtension> (visited on 06/05/2024).
- [W66] “February 2017 (version 1.10)”, [Online]. Available: https://code.visualstudio.com/updates/v1_10#_preview-minimap (visited on 07/13/2024).
- [W67] “Visual Studio Code - Code Editing. Redefined”, [Online]. Available: <https://code.visualstudio.com/> (visited on 01/20/2024).
- [W68] “Kompare”, [Online]. Available: <https://apps.kde.org/de/kompare/> (visited on 11/25/2022).
- [W69] “WebGL: 2D and 3D graphics for the web”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (visited on 07/12/2024).
- [W70] “types: `sqrt()` | Can I use... Support tables for HTML5, CSS3, etc”, [Online]. Available: https://caniuse.com/mdn-css_types_sqrt (visited on 09/07/2023).
- [W71] “CSS Values and Units Module Level 4”, [Online]. Available: <https://drafts.csswg.org/css-values/#trig-funcs> (visited on 11/24/2023).
- [W72] “types: `sin()` | Can I use... Support tables for HTML5, CSS3, etc”, [Online]. Available: https://caniuse.com/mdn-css_types_sin (visited on 11/24/2023).
- [W73] “Canvas API - Web APIs | MDN”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (visited on 11/20/2023).
- [W74] “Sourcetree | Free Git GUI for Mac and Windows”, [Online]. Available: <https://www.sourcetreeapp.com/> (visited on 01/20/2024).
- [W75] “TortoiseGit – Windows Shell Interface to Git”, [Online]. Available: <https://tortoisegit.org/> (visited on 01/20/2024).
- [W76] “Fork - a fast and friendly git client for Mac and Windows”, [Online]. Available: <https://git-fork.com/> (visited on 01/20/2024).

- [W77] “GitHub - ejwa/gitinspector: The statistical analysis tool for git repositories”, [Online]. Available: <https://github.com/ejwa/gitinspector> (visited on 01/20/2024).
- [W78] “Superset”, [Online]. Available: <https://superset.apache.org/> (visited on 07/13/2024).
- [W79] “Bootstrap v5.0”, [Online]. Available: <https://getbootstrap.com/docs/5.0/getting-started/introduction/> (visited on 07/11/2024).
- [W80] “F# Software Foundation”, [Online]. Available: <https://fsharp.org> (visited on 06/29/2024).
- [W81] “What is .NET?”, [Online]. Available: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> (visited on 06/30/2024).
- [W82] “What is ASP.NET?”, [Online]. Available: <https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet> (visited on 06/29/2024).
- [W83] “GitHub – libgit2/libgit2sharp: Git + .NET = ”, [Online]. Available: {<https://github.com/libgit2/libgit2sharp>} (visited on 06/03/2023).
- [W84] “Stack Overflow Developer Survey 2021”, [Online]. Available: <https://insights.stackoverflow.com/survey/2021> (visited on 06/03/2023).
- [W85] “How whitespace is handled by HTML, CSS, and in the DOM”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace (visited on 09/08/2023).
- [W86] “Home · TortoiseSVN”, [Online]. Available: <https://tortoisesvn.net/> (visited on 01/20/2024).
- [W87] “GitLens | Free Git Extension for Visual Studio Code”, [Online]. Available: <https://www.gitkraken.com/gitlens> (visited on 01/20/2024).
- [W88] “BUSY - Building Systems | Project Binocular”, [Online]. Available: <https://busy.inso.tuwien.ac.at/projects/binocular/> (visited on 01/20/2024).
- [W89] “Google Forms”, [Online]. Available: <https://docs.google.com/forms> (visited on 06/27/2024).