Chronicler: Interactive Exploration of Source Code History

Moritz Wittenhagen, Christian Cherek, Jan Borchers
RWTH Aachen University
52074 Aachen, Germany
wittenhagen, cherek, borchers@cs.rwth-aachen.de

ABSTRACT

Exploring source code history is an important task for software maintenance. Traditionally, source code history is navigated on the granularity of individual files. This is not fine-grained enough to support users in exploring the evolution of individual code elements. We suggest to consider the history of individual elements within the tree structure inherent to source code. A history graph created from these trees then enables new ways to explore events of interest defined by structural changes in the source code. We present Tree Flow, a visualization of these structural changes designed to enable users to choose the appropriate level of detail for the task at hand. In a user study, we show that both Chronicler and the history aware timeline, two prototype systems combining history graph navigation with a traditional source code view, outperform the more traditional history navigation on a file basis and users strongly prefer Chronicler for the exploration of source code.

Author Keywords

programming; source code history; navigation; tree structure; history graph; flow visualization

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g. HCI): Graphical User Interfaces (GUI)

INTRODUCTION

Accessing source code history has been shown to be a task crucial for software maintenance. It supports programmers in understanding how code changed over time [20], in finding the person responsible for a certain code snippet [8], or even suggesting sensible changes when writing new code [21].

Navigating source code history, is usually supported on the level of source code files. However, as developers are often interested in source code on a snippet level [7, 9], the focus on files makes navigation of such a snippet's history tedious. Typical 'diff views' found in todays IDEs alleviate this problem to some extent, but they fail quickly, because they do not consider many of the typical changes ocurring during development. It is not possible, for example, to properly represent moving a snippet somewhere else because this often has to be modeled as a separate delete and insert event. This may suffice when checking which recent changes occurred in a file. It does not, however, suffice when visualizing the complete history of a file with easily hundreds of versions.

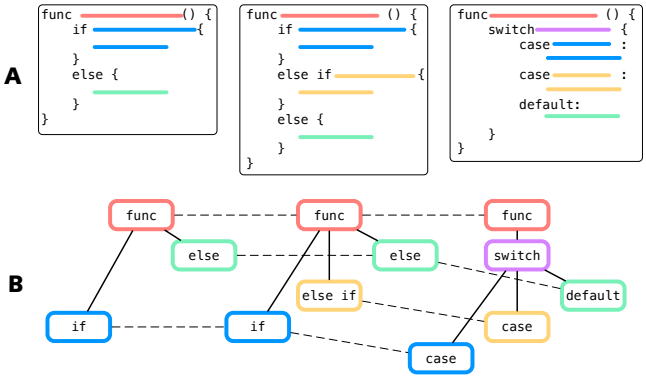


Figure 1. A short history of a source code snippet (A) and its respective history graph (B). Structural source code elements (func, if, switch, etc.) create a hierarchical tree representation (solid lines in B) for each version of the text representation. Each of these code elements has its own history and changes (dashed lines in B).

In this paper, we discuss how to use the inherent hierarchical structure of source code to visualize and interact with its history in more useful ways. Instead of only considering the history of entire files (Figure 1-A), we consider a source file to have a tree structure in which each element of the tree has its own history (Figure 1-B). Based on the *history graph* that spans this history of trees, we can then define how a user should be able to interact with individual parts of the tree, such as a method, and how she should be able to explore this individual element's history. We used this abstract definition of the interaction as a basis for creating our prototype system *Chronicler*.

Our contributions are:

- A discussion on how the graph structure facilitates interaction in source code history.
- Chronicler, a software prototype based on this discussion, that combines flow visualization of the history graph with a typical source code view.

- A user study performed with Chronicler that shows that this technique greatly benefits navigation tasks and supports the exploration of source code.

RELATED WORK

The tasks around source code have been studied by LaToza et al. [9]; they interviewed developers to understand the questions developers have when dealing with source code. Below, we loosely follow their classification of source code history tasks to discuss related work.

"When, how, by whom, and why was this code changed or inserted?" These questions are usually answered by today's version control systems (VCS) and their respective visualization tools, e.g, *commit messages* (why) or *blame* (who and when). However, they often make it tedious to understand the entire history of a piece of code, because they focus on the most recent change. Ogawa et al. [12] consider not only the latest changes but show responsibilities of different authors' over time (who). Hartmann et al. [6] automatically create annotations for code inserted from webpages showing their origin (how). Kagdi et al. [8] built a system that supports developers in finding someone who can help them understand how to make a specific change (who).

"What else changed when this code was changed or inserted? How has it changed over time?" Current VCS do not consider this beyond the grouping of changes into individual changesets, e.g., commits. Research prototypes [18, 21] use repeating change patterns to suggest related code changes for subsequent edits. Azurite [20] visualizes change events by annotating a timeline with markers showing the location of insertions, deletions, and modifications. A newer version [19] allows semantic zooming on the timeline. This allows users to quickly see in what parts of a file changes occurred. They also propose to use this to selectively undo changes from the history by applying inverse changes to the current version. Telea et al. [16] aim at providing a detailed visualization of changes over time based based on flow visualizations. Similar visualizations have also been used to visualize the history of news articles [13, 2] or changes within Wikipedia documents [17]. Among other contributions, we introduce a similar visualization that facilitates the interaction between visualization and source code.

"Has this code always been this way?" SeeSoft [3] displays information gathered from the history as a heat map on top of the source code itself. They propose a number of proposed metrics, most notably change frequency and age. Chronos [14] arranges the full text of different versions on a large plane that the user can navigate vertically to look at the content of one individual version and horizontally to explore the history. They align text that is unchanged over multiple versions by inserting whitespace, allowing the user to quickly see inserts and deletes when navigating the history.

"What recent changes have been made?" Many existing tools enable the user to compare two different versions of a file, e.g., Eclipse's Diff view. DeepDiffs [15] shows the age of text snippets inline using multi-colored highlights.

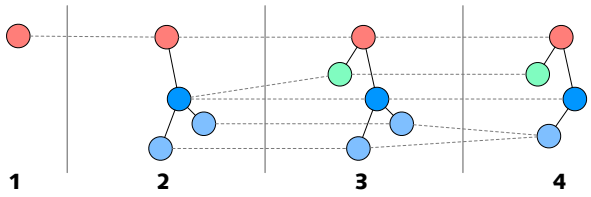


Figure 2. History Graph with four versions. Each node represents a part of a source tree, e.g., a class, method, or code block. Nodes with the same color represent the same structural element, e.g., an if block, but not necessarily the same content, e.g., a changed condition. 1: A single red node is created. 2 (insert): A new blue node and its two children are inserted as children of the red node. 3 (split): A new green node is created by duplicating the blue node but not its children, i.e., the user copied parts of the code. 4 (merge): The code represented by the two light blue child nodes is consolidated into a single node, e.g., because two special cases could be handled by a single branch.

"Have changes in another branch been integrated into this branch?" VCS visualizations often show a graph visualization of different branches and their merge status (see e.g., github.com). For this work, we focus on navigating a linear history and exclude branching.

CODE STRUCTURE

Existing VCS allow navigation of the history only on a file basis. The user can select a version of the file from the history, she then ascertains if the version is of interest for the question at hand, e.g., how this method was changed in the last 6 months. She may have to scroll in the file to find the method again, then identify changes and their relevance before deciding if to continue to the next version. This process is very tedious and does not promote easy exploration and easy access to the history.

Our approach uses the source code's inherent hierarchical structure to visualize events of interest that the user may use to inform navigation decisions. Typically, source code changes only affect a fraction of the whole file contents. Exceptions would be a larger refactoring or the introduction of a new feature. This also means that most changes are not relevant for tasks related to an individual method. Providing the user with the means to select which parts of the code structure are relevant can thus simplify navigation and make relevant parts of the history quicker to access.

One way to structure a source code file is by means of its abstract syntax tree (AST). Every node in this tree represents a range of text in the source code file, e.g., a method node represents the whole contents of the method. A child represents a subrange of its parent node and siblings never represent intersecting subranges. For example, an if block within a method is a child of the method node, and thus represents a subrange of the method text. It will never intersect with another else block, as defined by the programming language syntax. If we know what node in this structure changed from one version to the next, we can build a *history graph* that does not only linearly connect all versions of the file but also represents the changes of individual nodes in the AST. Figure 1 shows a simplified example of such a history graph consisting of syntax trees over

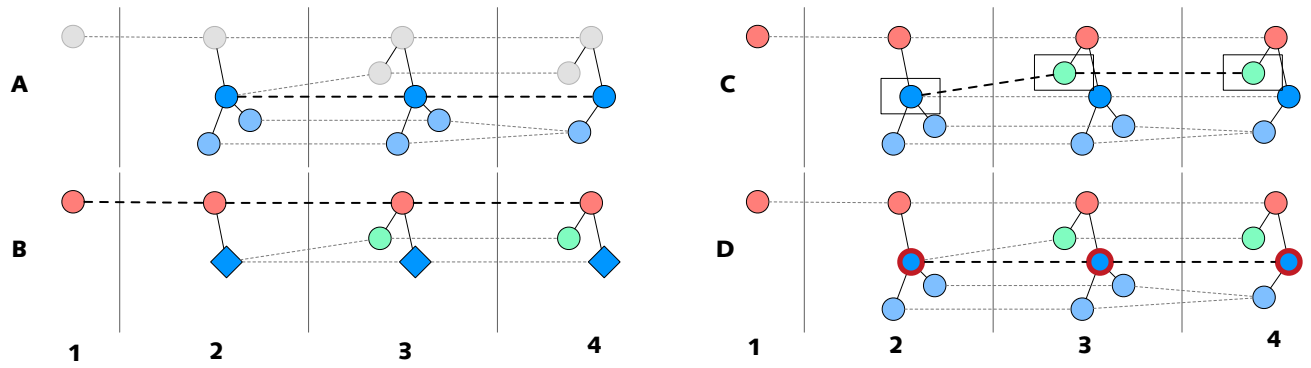


Figure 3. Structural navigation properties of a history graph. Selected history paths are shown in bold. A—Filtering: we can hide parts of the graph in order to simplify the structure to consider. If version 2 and 3 have no content changes, we can also contract them into a single version. B—Abstraction: we can find a representative for groups of structural elements and thus simplify the graph; here we use an allowed depths of 1. C—Viewport Stabilization: we can make sure that when switching between versions a node will always have the exact same position in the viewport, represented here by the boxes around the nodes. D—Propagation: we can pass on properties such as selection to other versions of a node. During navigation we can keep it the same, helping the user to orient herself.

three versions of a file. Such a history graph enables a more detailed overview of the history of a repository.

The history graph consists of trees representing each file version. Each node in such a tree can have an arbitrary number of predecessors and successors pointing to neighboring versions. Figure 2 shows an example. Similarly to [16], we look for certain *events of interest*.

Insert events (Figure 2-2) occur when the user adds a new structural element in the code. They may be interesting navigation targets to see the state of the code when the element was first introduced. Similarly, a *delete* shows the removal of a structural element. Knowing about deletes would enable the user to see code that previously existed in a method but has since been removed.

A *split* event (Figure 2-3) occurs when source code is duplicated, i.e., copied and pasted. *Merge* events (Figure 2-4) happen when two code elements are consolidated into a single element. The programmer would usually move code from one element into another, e.g., to remove overly complicated branching. Splits and merge events allow the user to reason about common ancestry and identify errors caused by oversimplification of source code such as no longer handling a necessary special case.

We will later use the term *primary successor* to denote the structural element that source code was copied from. Similarly, the *primary predecessor* is the predecessor that source code was moved into.

Interaction

We envision a source code history navigation tool to be used in conjunction with a typical source code view as found in any development environment. We explored two different kinds of interaction with a history graph. The first kind is, interaction with the history graph itself, which changes the shape of the graph. We describe one possible visualization of this graph after the interaction discussion. The second kind is, the interplay of the history graph with a source code representation, enabling users to navigate the history of the source code represented by the visualization.

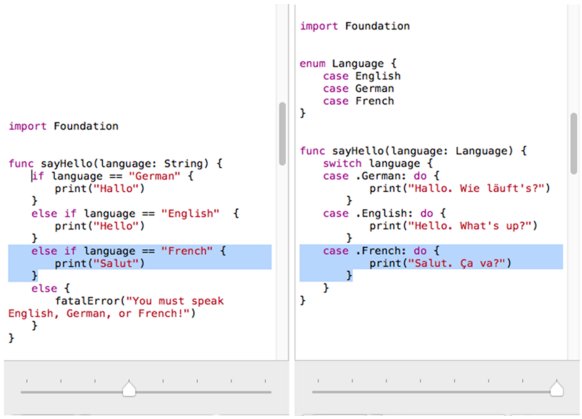
Changing the History Graph

The two techniques discussed here reduce the complexity of the history graph, and thus the visual clutter in a visualization of this graph. The user may not be interested in the history of the whole document, but instead only the history of an individual code snippet, e.g., a method. First, we can reduce the complexity by considering only one individual *history path*. A history path starting in a given node is created by adding the selected node's primary successor to the path and then repeat the process until we are at the end of the path; we then do the same with the predecessors respectively. Such a history path does not necessarily contain nodes from all trees, but at most one node from any given tree.

We can then *filter* (Figure 3-A) the history graph by reducing the visualized history graph to a subgraph based in a history path. Such a *filtered subgraph* only contains tree nodes that are in the subtrees of nodes on the history path. In this way, we reduce the breadth of displayed information and allow the user to focus on the history of individual elements. We also chose to contract the history subgraph by only including versions that have changes from their neigbors. Note that the filtered subgraph created from a history path containing all root nodes is equivalent to the full history graph.

Abstraction (Figure 3-B) of the graph allows us to reduce the depth of the information displayed. This allows users to focus on the "bigger picture", showing only the history of methods in a class instead of also considering all lines of code within these methods. Given a filtered subgraph, we reduce the depth of this subgraph by merging all subtrees with a depth larger than a given value into their parent. The more the user increases the allowed depth, the more details about changes in child structure are revealed. However, this also increases the graph's complexity.

We leave it to the user to use both filtering and abstraction in combination to select an appropriate level of detail. If a



```
import Foundation

func sayHello(language: String) {
    if language == "German" {
        print("Hallo")
    }
    else if language == "English"  {
        print("Hello")
    }
    else if language == "French" {
        print("Salut")
    }
    else {
        fatalError("You must speak
English, German, or French!")
    }
}
```

```
import Foundation

enum Language {
    case English
    case German
    case French
}

func sayHello(language: Language) {
    switch language {
    case .German: do {
            print("Hallo. Wie läuft's?")
        }
    case .English: do {
            print("Hello. What's up?")
        }
    case .French: do {
            print("Salut. Ça va?")
        }
    }
}
```

Figure 4. Two version of a short piece of source code when navigating through history. We have viewport stability, since even though the `if...else` statement was replaced with a `switch` statement, the selection appears at the same location and is highlighted correctly.

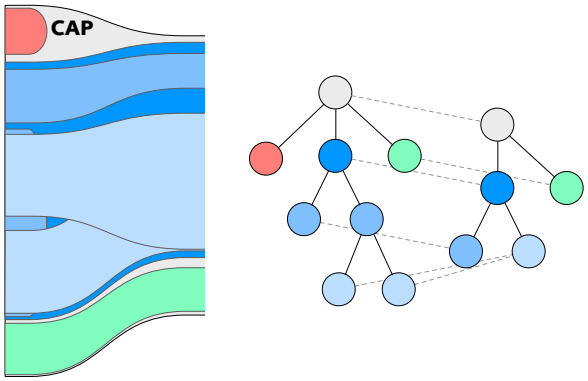


Figure 5. Tree Flow visualization of the two trees depicted on the right. The red node and a blue parent node are removed and the two light blue leaf nodes are merged into a single node. The height and position of each curve is determined by the number of represented source code lines and the starting line number.

user only wants to know about the methods being added and removed from a class, she could choose to filter the graph based on the class, and then use abstraction to hide details of changes within the methods.

So far, these techniques affect the the graph itself. In order to communicate the restricted visualization in the source code view, we suggest to gray out or hide the source code not represented in a filtered subgraph.

Interplay with Source Code

We see the history graph as a way to navigate and explore different versions of source code. The knowledge of each element's history enables new possibilities for history navigation.

Assuming the user selected a history path, we can ensure the version tree nodes on this path to always be displayed in the same location. As the user navigates through the versions of a code snippet, we can scroll each version of the code snippet to the exact same viewport location. This enables the user to know where to look for changes in the code she is navigating. We call this *viewport stabilization*. Figure 3-C shows an abstraction and Figure 4 shows the concrete effect in an example with source code.

Through a similar mechanism, we can also *propagate* other properties (Figure 3-D). We use it to make sure that the text blocks that represent the predecessors and successors on the history path are always selected. However, this concept could also be extended to propagate annotations, e.g., comments made by the user exploring an unfamiliar piece of source code, or even changes in a system like [4].

Tree Flow Visualization

Based on the interaction between graph and source code view, we need a visualization of the graph that (a) shows the history of different levels of a tree, (b) that supports easy adding and removal of abstraction levels, and that (c) enables selection of history paths by the user.

We achieve this in our visualization called Tree Flow. Tree Flow is heavily influenced by Arctrees [11], a one-dimensional tree visualization, and other flow visualizations [16, 17]. We first create an Arctree of each source code version and then connect adjacent levels of this tree based on their successors and predecessors in the history graph. Starting at the root, we draw each node as a box with a given width. Similar to [17], the height and position of the box are determined by the number of source code lines represented by a node and the starting line number in the source file. In this way, we create a relief of the source code content. We can connect two adjacent trees drawn in this way by drawing cubic bezier curves from a node to its primary successor or predecessor. *Insert* and *delete* events are drawn as caps (Figure 5 red path) in between the two Arctrees. To create an organic look, we draw the Arctrees with an effective width of zero, thus only defining the connection points for the flow connections. The individual connections of the flow visualizations are click targets, representing individual history paths the user can select. In contrast to CodeFlows [16], this visualization considers multiple tree levels at once which enables us to only remove higher tree levels, if required by the user (*abstraction*).

We specially handle a *merge* into a parent node as a *delete* and a *split* into child nodes as *insert* event. Our coloring approach works predecessor-to-successor and top-down: to color a version tree, we first propagate the colors of all nodes in the predecessor tree to their successor nodes. All nodes left uncolored are colored using a desaturated variant of their parent's color or pick a color from a color map if the parent is the tree root. This ensures that each first-level tree element and its children have similar colors, but also moved code elements will retain their original color after the move.

Since Tree Flow only visualizes structural changes, we finally mark content changes between two versions by annotating both versions with a vertical mark. This allows the user to identify content that has stayed constant, and helps to identify navigation targets not based on structural changes.

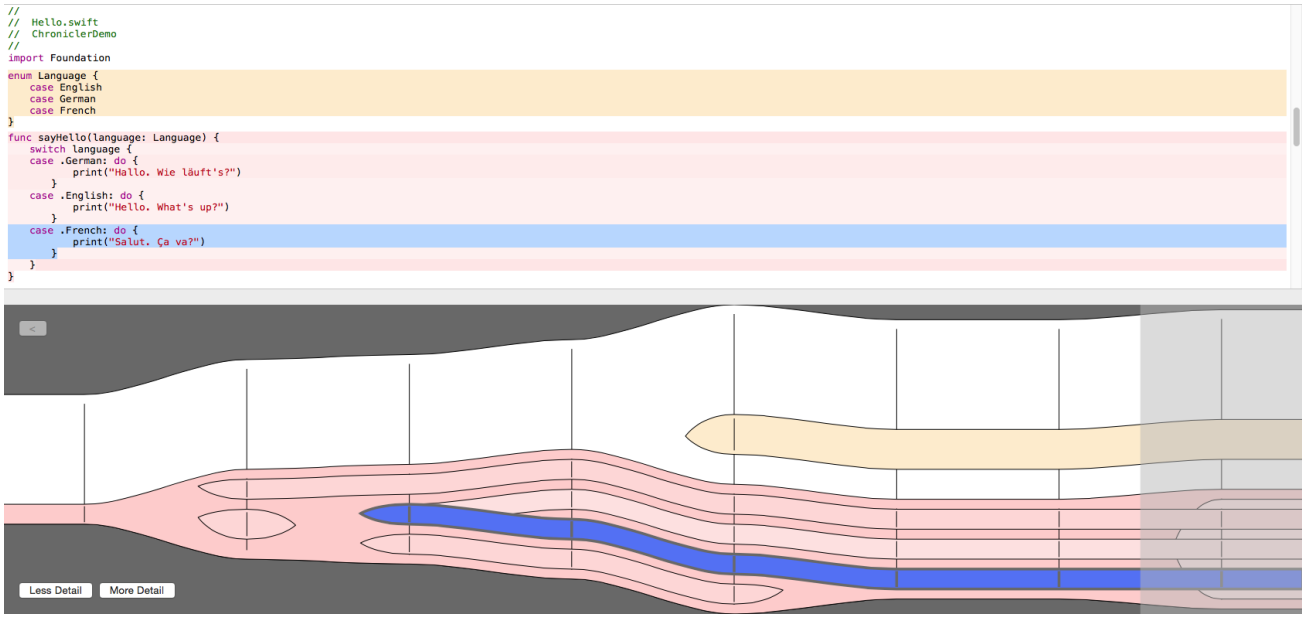


Figure 6. User interface of Chronicler for a short file with a short history. We see the source code view at the top and the Tree Flow visualization at the bottom. The gray bar in the bottom right indicates the current version. We can easily see how long the individual blocks have been around, e.g, the yellow `switch` statement was only recently added, and the initial version contained only a short version of the red `sayHello` function. The currently selected code snippet is highlighted in dark blue in the visualization and in the source code. Vertical lines indicate a change at a commit, the yellow `enum` statement e.g. was not changed since it was introduced.

PROTOTYPE

We built three prototypes that fulfill different subsets of our structural navigation properties (Figure 3). A Normal Timeline that allows to navigate in the history of a file and does not provide any structural navigation properties. A History-aware timeline that provides viewport stability. And Chronicler, that uses Tree Flow visualization to create an annotated timeline. On the Tree flow visualization Chronicler also provides Propagation of the selected area, Filtering, and Abstraction. We also created a fast parsing and matching algorithm for source code that enabled us to run a user study with long code histories provided by our study participants.

Building a History Graph

In contrast to [1], we do not match the complete AST to create a history graph. We implemented a tree parser based on code blocks which fits the notion that developers are often interested in code on the "snippet level" [9, 7]. Most programming languages group code into blocks through statements like begin and end or brackets. Usually, the same grouping mechanism is used for namespaces, classes, methods, loops, etc. Thus, we can approximate a rough syntax tree by only parsing code based on this grouping mechanism. This removes the complexity and performance requirements of parsing the whole AST. This also allows us to parse many different programming languages (e.g. JavaScript, Java, C, Objective-C, or Swift), since we can quickly adapt our matching algorithm. Parsing and matching only to the code block allows us to easily test and verify the concepts discussed in this paper.

Our matching is based on a simple bag-of-words model. We define a *word* to be any group of alphanumeric characters, i.e., variable names, language keywords, words in strings, words in comments, and numbers. Any element of the syntax tree is annotated with all *words* from the text it was parsed from. We can then match two elements by comparing the contents of the two multi-sets (bags) B_1 and B_2. The match score between two nodes is defined as:

$$\frac{1}{2} \cdot |B_1 \cap B_2| \cdot \left(\frac{1}{|B_1|} + \frac{1}{|B_2|} \right)$$

This results in values between 0 for bags with no common elements, ans 1 for two bags that are the same. When matching two trees we use the fact that changes are iterative to avoid a lot of exhaustive tree searches. We first check for trivial matches, i.e., completely unchanged nodes and nodes with large scores (0.9 has worked well for our prototype). We only do full tree searches for the remaining nodes. This allows us to match even large trees quickly. In our experience, we could reliably match nodes with 3 or more words.

Our final structure is a tree, similar to the AST, in which each node represents a range in the original source code. Each subtree represents a subrange of the text represented by its parent. Children are easily linearly ordered by their occurrence in the parent's text. An abstract example of such a code block tree can be seen in Figure 1.

Chronicler

The Chronicler prototype consists of a source code view and the Tree Flow visualization. The Tree Flow visualization also serves as timeline to navigate through source code history. Whenever the user starts to drag on the visualization, we switch

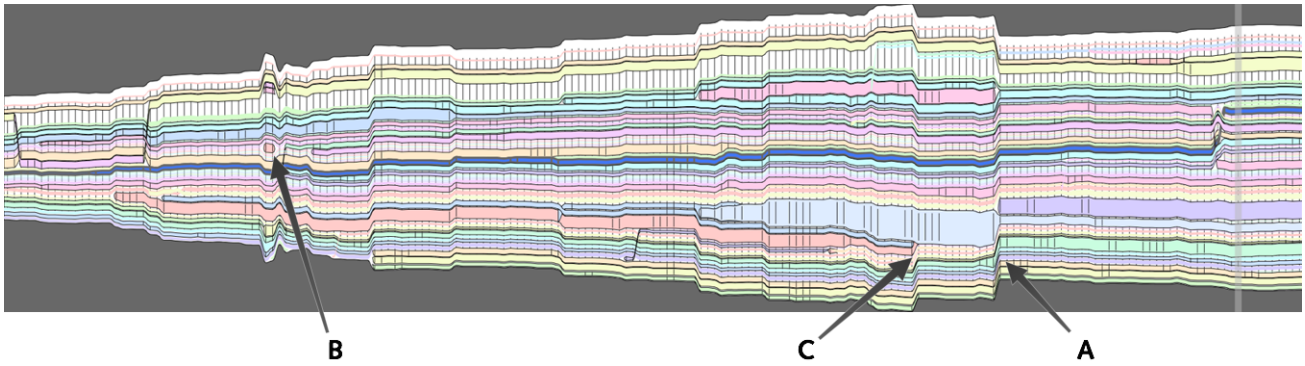


Figure 7. Chronicler visualization for a longer file (about 1000 lines) with 200 versions. We used Abstraction to reduce the depth of the tree to the method level. The visualization reveals details about the selected method (dark blue) and shows that it has been around for the lifetime of the code, and was moved to another location in the file recently. We can also identify large refactorings (A), short lived experiments (B), and methods that were merged into another one (C).

the code to the version corresponding to the horizontal location of the mouse pointer.

Users can select a code block by clicking on the corresponding history paths in the visualization. This selection scrolls the code block into the viewport, highlights it, and highlights the history path in the visualization. Selecting a code block in the source code view will also highlight the corresponding history path. As described in Figure 3-C and -D the selected code block is highlighted and kept in the viewport during history navigation.

To dynamically change the depth of the *history tree*, as described in Figure 3-B, we added two buttons at the bottom left of the visualization. Decreasing the depth of the *history tree* in Figure 6 would let the red `func`-block to be the only representation for this part of the tree. This would also hide the lighter red lines in the visualization. To filter based on a given history path (Figure 3-A), the user can double-click a history path. This moves the selected code block into view and changes the visualization to only display the history of this subgraph. We gray out the remaining source code to make sure the user knows on which part the visualization is currently focused. In this way, the user can descend into the tree to reveal more details about classes, methods, or individual code blocks.

Figure 6 shows the Chronicler interface for a short source code file. The currently selected code block is highlighted in the source code, and the corresponding *history path* is highlighted in the visualization. The buttons to change the abstraction level are in the bottom left of the visualization.

To show also small changes, that do not affect the size of a code snippet, we added vertical lines at commits where something was changed. In Figure 6 the first `case`-statement within the red block was touched in nearly every version, while the yellow `enum`-statement was not changed after it had been introduced.

Figure 7 shows a longer history from a file in a public GitHub repository[1]. Currently the visualization is not filtered, providing an overview on the evolution of the file to highlight events like a major refactoring. A filtered version showing only an individual method would look similar to the short file in Figure 6.

[1] http://github.com/AFNetworking/AFNetworking

History-Aware Timeline

For some tasks it might be unnecessary to provide the user with a complete visualization of the *history graph*. Therefore, we developed a history-aware timeline that only provides viewport stability and selection propagation (Figure 3-C and -D).

We created an interface that consists of a source code view and a timeline slider. When clicking into the source code, the source code view highlights the code block around the line containing the text cursor and selects the history path starting at the respective history graph node.

After selecting a code block, the user can navigate the history of the code block. The first line of each version will always be scrolled to the same viewport position as the first line of the previously displayed code block. In this way, the user can easily identify the past versions of the original code block during navigation. Figure 4 shows two states of the history-aware timeline UI during navigation.

Of course, timeline navigation has its disadvantages; For example, the user cannot tell what version to navigate to in order to find a particular change. It may however be sufficient to get back to an earlier version that the user still remembers clearly. The history-aware timeline is thus more suitable for navigating known source code than exploring an unknown code history.

EXPERIMENT

We conducted a study, comparing Chronicler, the history-aware timeline, and a normal timeline. The normal timeline only considers file versions and no structural knowledge. It only uses the heuristic that source code changes are typically small and scrolls back to the same absolute viewport location.

Our goal was to find out how users approach history navigation with each of these tools as well as which tool performs best in terms of task time. Therefore, we divided our experiment

in three parts: In the first part, all users were asked to discuss problem solving strategies for four different exploratory tasks. Secondly, users had to perform a set of eight concrete navigation tasks using one of the tools in a between groups experiment. Third, we let again all users explore their own repository with familiar source code; here they could choose all tools again.

Before the first part users we explained all three tools on an example file. We showed them how the structural navigation properties are implemented in the different tools and let them play around before the first part started. The normal timeline did not provide any structural navigation properties, the history-aware timeline has viewport stability, and Chronicler has all four structural navigation properties.

After the first part, users were asked to fill out a questionnaire regarding their opinion on how suitable each tool was for the given task. In the second part, we measured the users' task completion times for the navigation tasks. The exploration of their own code was used to collect additional qualitative data, which gave us insights into how developers could use history navigation tools. Throughout the study, we took notes to gather qualitative data about users' navigation behavior and their impressions on the tools and tasks.

We recruited 21 people (2 female) between the age of 21 and 35 from a local developer mailing list at our computer science department. They reported a median coding experience of 4 on a five-point Likert scale. The study took about 45 minutes. Free snacks and drinks were offered during the study; participants were not compensated in any other way.

Part 1: Strategy Discussion

We identified four source code understanding tasks that can be solved using the code's history.

Task 1: How do you find out where a method originates from?
This is relevant in tasks where the programmer wants to understand a code snippet that may seem to be out of place or a duplication of code. The snippet's history may reveal that the out of place code has been moved from a different method without being properly adapted.

Task 2: How do you understand this version of a method?
Quick access to the history of a method allows the programmer to look at older versions of a method. Older versions of a method often are less complex since they handle fewer special cases, making the core functionality easier to extract.

Task 3: There used to be an interesting line in this code snippet. How do you find it?
A developer may have run into a problem and was told that there used to be a special solution for this issue somewhere else, maybe pointed out by the team historian [10].

Task 4: How do you identify a big refactoring in the code?
Identifying refactoring helps to understand reference points in the history itself. People may have to treat leftovers from old code differently than things that have been introduced more recently. Also, the changes during the refactoring may clarify the purpose of the code.

We call all of these tasks exploration tasks; the user does not know in advance which version holds the answer to the questions. After each task, we asked each participant to evaluate how well each tool was suited for the tasks on a five-point Likert scale. Our hypotheses were as follows:

Hyp 1: The history-aware timeline and Chronicler will be preferred over the normal timeline.

Hyp 2: For Task 1, Chronicler will be preferred over the history-aware timeline, because the structure tree visualization actually shows a merge, or lack thereof, when a code snippet used to be part of a different context.

Hyp 3: For Task 2, the answer to the question is not provided by the visualization. However, since structural changes, e.g., the length of a method or the existence of certain code blocks, could be information scent for "easier to understand", we expected Chronicler to outperform the timelines.

Hyp 4: In Task 3, we specifically asked for a non-structural element that cannot be visualized in Chronicler. Still the visual information may suggest the existence of rich information that is not there, therefore we expected the history-aware timeline to perform better than Chronicler in Task 3.

Hyp 5: For Task 4, we expected Chronicler to outperform the timelines, since when only looking at a small part of the code, refactorings are very difficult to identify in these conditions.

We identified examples of these tasks in existing files from our own development projects and a public GitHub repository[2]. We urged participants to think about how they would solve the task with each tool, instead of just completing it quickly. They could switch back and forth between the three conditions at their own leisure and amend their previous strategies.

Results

Quantitative Analysis

To analyze the nonparametric data from our questionnaire we used a Kruskal-Wallis test and a Wilcoxon test for pairwise comparison. Overall, we found a significant preference ($Z = 10.34$, $Z = 6.90$, $p < 0.0001$ both) for Chronicler over both the normal timeline, as well as the history aware timeline. We also found a significant preference ($Z = -7.05$, $p < 0.0001$) for the content aware timeline over the normal timeline, thus we accepted Hypothesis 1. For Hyp. 2, we found Chronicler to perform better than both timelines ($Z = 5.66$, $Z = 4.90$, $p > 0.0001$ both). For, Task 2, there is no significant difference between Chronicler and the history-aware timeline (Hyp. 3). This did not match our expectation; participants did not consider the visualization to have an additional benefit over a history-aware timeline. For Task 3, we found both Chronicler and the history-aware timeline to perform significantly better than the normal timeline. However, there was no significant difference between Chronicler and the history-aware timeline, thus we have to reject Hyp. 4. Users probably were not as easily distracted by the visualization as we expected, and just used Chronicler in similar ways as the history-aware timeline to fulfill this task. In Task 4, there was a significant

[2] http://github.com/AFNetworking/AFNetworking

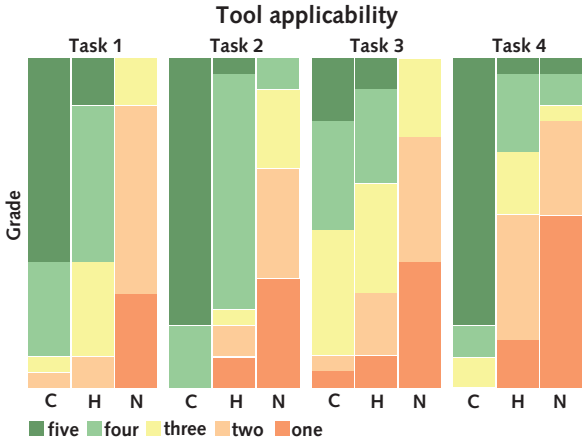


Figure 8. The results from the Likert-Scale questionnaire asking how well the tool was suited for each of the discussed strategies. Generally Chronicler was rated highest with an exception for Task 3. H: History-Aware Timeline, N: Normal Timeline, C: Chronicler

difference between Chronicler and the timelines ($Z = 5.66$, $Z = 4.90$, $p < 0.0001$ both), but no significant difference between the two timeline conditions, thus we can accept Hyp. 5. As we expected, both slider conditions did not help to solve this task, since they barely visualize a refactoring. Figure 8 shows a stack bar diagram of the questionnaire results for each individual strategy.

Our results clearly indicate that Chronicler is generally better suited for these kinds of exploration tasks, with the exception of understanding a method where people could not see a particular benefit of the visualization. As we expected, people always preferred a navigation with viewport stability over a normal timeline.

Qualitative Results

When being explained the Chronicler prototype, many users directly started making assumptions about the code history. They started playing around with the visualization, and many discovered features like the viewport stability without it being directly explained . Users were highly interested in looking into their own repositories and many stated they'd like to have this available for their daily routine.

Regarding the comments and observations we made during the first part of our experiment, we identified two key differences between the timeline conditions and the Chronicler conditions. First, Chronicler encourages lateral exploration of other code blocks. Even when given a task about a specific method, participants often also looked at other methods or code elements for comparison. In the timeline conditions we usually observed users to be focused on one code block. Second, the visualization of structural change was often used as an indicator of relevant versions. During exploration, participants started at versions emphasized by a structural change occurring. The timeline conditions do not have any such indicator, therefore we observed a lot more random or even exhaustive searches of versions.

As expected, the normal timeline was difficult to use over multiple versions. 13 of the participants worked around this using the 'Find' and 'Find Next' commands. The rest manually searched for the method of interest in each version they looked at.

With the visualization of Chronicler, people explicitly looked for key versions with structural changes (17) and/or content changes (5) when solving Tasks 1 and 2. Most participants (18) used the visualization to identify valid navigation ranges before attempting any navigation. Only four participants looked at each version individually.

In the timeline conditions however, only four people attempted to identify key versions first. Nine participants randomly selected older versions to find a version to compare and 12 proposed an exhaustive search of all versions.

18 participants also explicitly noted the helpfulness of the overview provided by Chronicler for all tasks. For tasks 1 and 2, these participants not only tested their proposed strategies with the indicated code, but also looked at other methods. Only one participant did so for the timeline conditions. 14 also explicitly mentioned the ability to focus on a method as a means to understand detailed changes (Tasks 1 – 3).

In Task 4, refactorings were mainly spotted in two ways. With the timelines, one group (10) of participants used the length of the file as a global indicator; they either looked at the end of the file without using structural features or at the size of the scrollbar. Large changes in scrollbar height or a large jump of the end of the file were then interpreted as a potential refactoring. The rest (11) looked at a specific method and tried to identify versions with large changes in such a local view. Two participants used a number of variable declarations at the beginning of the file as a static reference point. They used changes in these variables as indicator for a refactoring.

Both groups usually verified their hypotheses in the Chronicler condition however. There, the refactorings were reported to be easily identifiable through structural reorganization or length changes.

All participants tried to select small, easily identifiable code snippets when looking at the code in accordance with [9]. When possible, they selected a snippet that was completely visible on screen.

Part 2: Navigation Tasks

The second part of our study was designed to consider navigation that requires the participant to know what the target version looks like. We consider local undo to be a history navigation task, e.g., when the user wants to go back to an earlier version after trying an experimental change. The user likely still knows what the earlier version looked like. Another example are variants of strategy Task 3 where the user is explicitly told what to look for.

We selected eight navigation tasks based on two characteristics: the representation in the structure, i.e., if the code that was to be found is visualized in the structure, and the amount of versions having the particular change, i.e., fewer or more

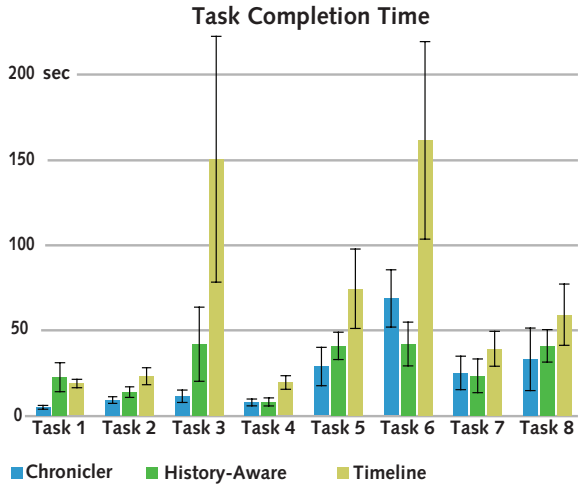


Figure 9. Task completion times for individual tasks with standard error bars.

than 10% of the versions on the timeline. We looked for two examples for each combination of characteristics.

We performed a between-groups study; each participant was randomly assigned a condition. Since users did not know the repository we created sheets that contained a unique description of the lines of code they were supposed to look for. In this way, they had an easy way to check if they had reached the right version or not. All targets contained a small code snippet that occurred for only one time period within the history. We gave people as much time as they wanted with these sheets to understand the tasks before starting the experiment. We measured the task completion time from starting the experiment to the last navigation event. When performing the navigation, users were asked to find the correct version as quickly as possible.

Our hypotheses were as follows:
Hyp 1: Overall, conditions with viewport stability are faster than the normal timeline.
Hyp 2: For each trial, conditions with viewport stability are faster than the normal timeline.
Hyp 3: Represented events are faster to find using Chronicler because the user can see where they have to navigate to, especially when less than 10% of versions have the indicated change.
Hyp 4: Events not represented are faster to find using the history aware timeline because there is no distracting visualization.

Results

Overall, Chronicler had the lowest task completion times with a mean of 39.47s. Users with the history-aware timeline needed 45.05s, and users with the normal timeline slider needed 85.33s on average.

We analyzed our measurements with a multi-factor ANOVA and post-hoc analysis using paired t-tests. The analysis showed a significant effect of condition on task completion time ($F = 8.20 p < 0.01$). Pairwise comparison showed that both Chronicler and the history-aware timeline were significantly faster than the normal timeline ($MD = 45.86$, $StdErr = 12.47$, $p < 0.01$ for Chronicler and $MD = 40.28$, $StdErr = 12.27$, $p < 0.01$ for history aware timeline).

Figure 9 shows timing results for the individual tasks. The exceptional high standard error bars for the timeline condition in Task 3 and 6 can be explained by the different search strategies. As mentioned in Part 1, some people used a random search strategy to solve the task, which resulted in longer search times for tasks with a small amount of target versions.

There were no significant differences between Chronicler and the history-aware timeline conditions. There also was no significant effect of any individual task on performance, and no interaction effects between task and condition. While this does not mean that Chronicler cannot be faster than the history-aware timeline in terms of navigation time, it indicates that it may not be worth it to spend the screen space on the larger visualization for quick navigation tasks.

Part 3: Exploring Familiar Source Code

For the last part of the study, we let participants explore familiar code. They were asked to bring their own repositories in a programming language of their choice.

Since not all participants could bring their own code, this part of the study was optional. Nevertheless, all but four participants brought familiar source code and participated in this last part. Everyone exclusively used the Chronicler visualization, although all conditions were available to them. The reception was positive throughout and people described a number of interesting use cases.

General use: There were some comments indicating that people would use the tool regularly.

- "With this, I actually would look into my old code more often, right now I don't do this at all."
- "The lack of good version control made me store old versions of my methods within the comments a lot."

Especially the second comment highlights that people may have kept history versions around manually, a habit that they may replace with a tool like Chronicler.

Structural soundness and stability: Some people focused on the structural soundness of their code an identified regions that were meant to be structured differently. Others said they could use the structural stability of certain code parts to identify edge case handling.

- "This tool would make me create better code structure, just by showing what the originally planned structure was."
- "This tool can show me, If my code should be refactored soon."
- "With this tool I can see my code structure is messed up, I think I will fix that for my next version."
- "When I look at this, I ask myself why I decided to move this piece of code."

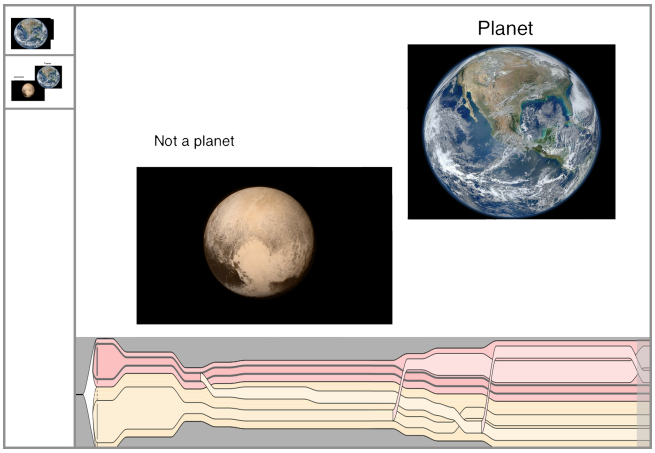


Figure 10. The UI prototype for presentation slides, the Tree Flow visualization shows multiple objects that are copied into another slide, reorderd, and resized.

- "This could be used to identify edge cases for a method. Typically the edge case handling gets introduced later than the main functionality, with this tool I can find it easily."

Collaboration: We saw some examples where people used the tool to identify what a colleague did.

- "A colleague added something to our repository and I was not able to find it quickly. This tool would have helped a lot in this situation."
- "I think I am able to see which commits were done by me, and which were done by my colleague."

Especially the first example seems to be an interesting use case.

Customer communication: Lastly, a developer mentioned a very interesting problem that regularly came up in their company, since they create special versions of their software for each customers.

- "This would help a lot to identify the age of a bug in a project. If the bug is discovered in a newer version of the project, all affected branches can be easily notified."

LIMITATIONS AND FUTURE WORK

We also received user comments how to further improve the Chronicler interface. Users specifically requested zooming on both axes of the visualization and a search functionality through all history versions. We noticed that it may be beneficial to select multiple history paths at once. This could, for example, enable users to compare the evolution of two methods side by side.

One limitation we accepted for the study is the code parser that only considers code blocks. Some users noted that they would be interested in the evolution of individual lines. This limitation is mostly produced by only using heuristical matching. We would like to explore how a structure aware IDE could track changes over time. Such an IDE could include a VCS that offers a more detailed history to help with undo or provide support to discuss changes for code reviews. Including the structure awareness into the VCS would also omit the need for a matching algorithm, since the system can save the required information during the development.

Until now, our matching algorithm only parses single file histories, we would to add functionality for complete repositories as well. To display this in Tree Flow visualization we would add another layer of Abstraction that displays file level. This would allow to see when parts of the code are moved into other files, or which files have a longer lifetime.

Application to Other Types of Media

History graph visualizations are not only useful for source code history. Chronicler nicely demonstrates the application to a medium that is essentially one-dimensional, namely lines of code. However, we can also use a system like this for other media types.

Figure 10 shows a UI prototype that implements history graph navigation for presentation slides. The presentation is represented as the root node of the history graph, the slides are the first child nodes and objects on the slides are leaf nodes. Instead of representing the position and number of lines in a file in Tree Flow, as we did with source code, we use the vertical position to denote the order of slides as well as z-position of objects on the slide and size to denote the area covered by the object. We can still easily see when objects are copied to different slides, their area changes, or they are reordered.

Similar approaches may also be interesting for 2-dimensional graphics tools such as Photoshop. Capturing workflow histories has been shown to be relevant, and information about tool use can support navigation in such a history [5]. An approach using the structure of the file, e.g., through layer groups and layers, may also help to identify relevant content changes.

SUMMARY

We introduced the notion that the history of individual code elements can create new navigation techniques around source code histories. We defined four navigation properties that allow users to interact with a history graph and a corresponding content view. Tree Flow, our visualization of a history graph, is designed to support these properties by allowing dynamic filtering and abstraction of the visualized graph.

We showed that our prototypes, Chronicler and the history-aware timeline, outperform timeline-based navigation through source code. While we could not observe a significant difference in navigation times between Chronicler and the history-aware timeline, users significantly preferred Chronicler to a user interface without the visualization. This is confirmed by our qualitative observations which indicate that Chronicler is better suited for exploration tasks around source code.

REFERENCES

1. Fanny Chevalier, David Auber, and Alexandru Telea. 2007. Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees. In *IWPSE '07: International Workshop on Principles of Software Evolution*. 90–97.

2. Weiwei Cui, Shixia Liu, Li Tan, Conglei Shi, Yangqiu Song, Zekai Gao, Huamin Qu, and Xin Tong. 2011. TextFlow: Towards Better Understanding of Evolving Topics in Text. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2412–2421.
3. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. 1992. Seesoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering* 18, 11 (1992), 957–968.
4. Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Björn Hartmann. 2013. Authoring Multi-Stage Code Examples with Editable Code Histories. In *UIST '13: Proceedings of the 26th annual ACM symposium on User Interface Software and Technology*. 485–494.
5. Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *UIST '10: Proceedings of the annual ACM symposium on User Interface Software and Technology*. 143–152.
6. Björn Hartmann, Mark Dhillon, and Matthew K. Chan. 2011. HyperSource: Bridging the Gap Between Source and Code-Related Web Sites. In *CHI '11: Proceedings of the annual conference on Human Factors in Computing Systems*. 2207–2210.
7. Reid Holmes and Andrew Begel. 2008. Deep Intellisense: A Tool for Rehydrating Evaporated Information. In *MSR '08: Proceedings of the International Working Conference on Mining Software Repositories*.
8. Huzefa Kagdi, Maen Hammad, and Jonathan I. Maletic. 2008. Who Can Help Me with This Source Code Change?. In *ICSM '08: IEEE International Conference on Software Maintenance*. 157–166.
9. Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions About Code. *PLATEAU '10: Evaluation and Usability of Programming Languages and Tools* (2010).
10. Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *ICSE '06: Proceedings of the 28th international conference on Software Engineering*. 492–501.
11. Petra Neumann, Stefan Schlechtweg, and Sheelagh Carpendale. 2005. ArcTrees: Visualizing Relations in Hierarchical Data. In *EUROVIS'05: Proceedings of the joint Eurographics / IEEE VGTC conference on Visualization*. 53–60.
12. Michael Ogawa and Kwan-Liu Ma. 2010. Software Evolution Storylines. In *SOFTVIS '10 Proceedings of the 5th international symposium on Software Visualization*. 35–42.
13. Stuart Rose, Scott Butner, Wendy Cowley, Michelle Gregory, and Julia Walker. 2009. *Describing Story Evolution From Dynamic Information Streams*.
14. Francisco Servant and James A. Jones. 2013. Chronos: Visualizing Slices of Source-Code History. In *VISSOFT '13: Proceedings of IEEE Working Conference on Software Visualization*. 1–4.
15. Ross Shannon, Aaron Quigley, and Paddy Nixon. 2010. Deep Diffs: Visually Exploring the History of a Document. In *AVI '10: Proceedings of the international Conference on Advanced Visual Interfaces*.
16. Alexandru Telea and David Auber. 2008. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum* 27, 3 (2008), 831–838.
17. Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. 2004. Studying Cooperation and Conflict Between Authors with History Flow Visualizations. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 575–582.
18. A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. 2004. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering* 30, 9 (2004), 574–586.
19. YoungSeok Yoon and Brad A. Myers. 2015. Semantic Zooming of Code Change History. In *VL/HCC '15: IEEE Symposium on Visual Languages and Human-Centric Computing*.
20. YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of Fine-grained Code Change History. In *VL/HCC '13: IEEE Symposium on Visual Languages and Human-Centric Computing*. 119–126.
21. T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 563–572.