Temporal Navigation in Hierarchically Structured Media

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker Moritz Wittenhagen

aus Norden, Deutschland

Berichter: Prof. Dr. Jan Borchers
Prof. Dr. Michael Rohs

Tag der mündlichen Prüfung: 26.11.2015

RWTH AACHEN UNIVERSITY

Temporal Navigation in Hierarchically Structured Media

PhD Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Moritz Wittenhagen

Contents

Abstract **xiii**

Überblick **xv**

Acknowledgements **xvii**

1 Introduction **1**

1.1 Temporal Navigation Interfaces 4

1.1.1 Content-based Temporal Navigation . 4

1.1.2 Direct Manipulation Temporal Navigation 6

1.2 Structural Navigation Interfaces 8

1.3 Thesis Structure 8

1.4 Navigating Document Histories 9

1.4.1 Selective Undo 10

1.4.2 Understanding Document Evolution . 11

2 Exploration with Semantic Navigation Interfaces **15**

2.1 Exploration 16

2.1.1 Information Foraging and Information Scent 16

2.2 Related Work 17

2.2.1 Timeline Based 18

2.2.2 Direct Manipulation 20

2.3 Semantic Navigation 23

2.3.1 Issues of Semantic Navigation Interfaces 24

2.4 Task Changes in Semantic Navigation Interfaces 26

2.5 Providing Multiple Interfaces 26

2.6 Handling Multiple Tasks 29

2.6.1 Handling Ambiguities in Different DMVN Systems 31

2.6.2 Handling Pauses 33

Loop 34

Embedded Timeline 34

2.6.3 Implications for Navigation 35

2.6.4 Experiment 36

2.7 Other Media Types 38

2.8 Requirements for Temporal Exploration . . . 40

3 Structural Navigation Interfaces **41**

3.1 Using Structure for Temporal Navigation and Exploration 41

3.2 ExamPen 42

3.2.1 Task 43

3.2.2 System Design 44

3.2.3 Data Analysis 46

3.2.4 ExamPen Summary 48

3.3 What is Structure? 49

3.4 Structural Representation 51

3.5 History Graph 55

3.6 Structural Navigation Properties 56

3.6.1 Usage in Existing Interfaces 59

3.7 Summary 62

4 Chronicler: Navigation in Source Code Histories 63

4.1 Related Work 63

4.1.1 Digests 64

4.1.2 Time-based Visualizations 66

4.1.3 Guides 69

4.2 Why Source Code? 72

4.2.1 Usecase 72

4.2.2 Source Code Structure 74

4.3 Structure Aware Timeline 78

4.4 Tree Flow 80
4.4.1 Visualization 81
4.4.2 Interaction 84
4.5 Experiment 86
4.5.1 Strategy Discussion 88
Quantitative Results 90
Qualitative Results 91
4.5.2 Navigation Tasks 92
4.5.3 Exploring Familiar Source Code . . . 94
4.5.4 Interface Enhancements 97
4.6 Summary 98
4.6.1 Limitations 98
Aside: Detailed Histories 99

5 Generalizing Tree Flow Interfaces 101

5.1 Parallel Structures 101
5.1.1 Subtrees 104
5.1.2 Visualization Properties 106
5.2 Content Linearization 106
5.3 User-created Structure 109
5.4 Summary 111

6 Summary and Future Work 113

6.1 Summary and Contributions 113

6.2 Future Work 115

6.2.1 Structural Representations 115

6.2.2 Usecases 115

6.2.3 Prescriptive Use 116

List of Publications **117**

Bibliography **119**

List of Figures

1.1 Photoshop History and Selective Undo . . . 2
1.2 Video Player . . . 3
1.3 Version Navigation . . . 4
1.4 Swifter . . . 5
1.5 Timeline Slider with Audio Waveform . . . 6
1.6 DRAGON . . . 7
1.7 Xcode Comparison View . . . 11
1.8 Visualization of "The Origin of Species" . . . 12
1.9 Workspace of a literary scholar . . . 13

2.1 Chronicle . . . 18
2.2 Semantic Timelines . . . 19
2.3 BrowseLine . . . 19
2.4 Continuum . . . 20
2.5 Slit Tears . . . 21
2.6 Schematic Storyboarding . . . 22

2.7 DimP 22
2.8 Direct Manipulation of Ranking Tables 23
2.9 DRAGON with Moving Target 25
2.10 Tasks Classification by Kathrein 27
2.11 Interface by Kathrein 28
2.12 Temporal Ambiguity 29
2.13 Comparison of Pause Handling 33
2.14 DragLocks Experiment Results 36
2.15 3D DMVN 37
2.16 GeoTime 39

3.1 Grouping of Strokes in ExamPen 45
3.2 Navigation in ExamPen 45
3.3 Neutral Stroke Chains 46
3.4 Exam with Clearly Visible 3 Pass Strategy . . 47
3.5 Task Footprints 48
3.6 Simple Structure of Words in a Sentence . . . 50
3.7 Complex Network 51
3.8 Alphaslider 52
3.9 Hierarchy of Timelines 53
3.10 Vizster 54
3.11 Hierarchical Edge Bundles 55

3.12 History Graph 56

3.13 Structural Properties 57

3.14 Motion 60

3.15 Operation Slicing 61

3.16 Code Folding in Eclipse 61

4.1 Edit Wear and Read Wear 64

4.2 Seesoft 65

4.3 Visualizing Email Conversations 67

4.4 Theme River 68

4.5 Text Flow 68

4.6 History Flow 69

4.7 Code Flows 70

4.8 Chronoviz 71

4.9 Azurite 71

4.10 Swift Grammar 75

4.11 Matching Versions 76

4.12 Code Block Tree 77

4.13 Structure Aware Timeline 79

4.14 Arctrees and Tree Stack 82

4.15 Tree Flow 83

4.16 Chronicler UI with Short History 85

4.17 Chronicler UI with Longer History 87

4.18 Box plot of Likert Scale Results 90

4.19 Cheat Sheet for Navigation Tasks 93

4.20 Task Completion Times over all Tasks 95

4.21 Aggregating Structural Changes 100

5.1 Call Graph 102

5.2 Call Path and Method Neighborhood 103

5.3 Tree Expansion of a Call Graph 105

5.4 Call Path and Method Neighborhood 108

5.5 Economic Ranking Table 111

Abstract

The process of creating large documents like a computer program, a book, or a movie can take several years. During this process the author may want to return to earlier versions of the document in order to compare different iterations or to undo an earlier change. Navigating in the evolution of a document is usually a tedious process because exiting tools typically consider the history of a document as a whole. The user's navigation goals, however, are focused around a specific change from only a part of the document.

We compare the navigation of document versions to video navigation, where we see interfaces that enable users to find a specific frame based on an object's location in the video. Unfortunately, these interfaces restrict which frames are accessible to those frames where an object moves. It is also difficult to directly translate these interfaces to non-video media media because they require a static frame of reference which does not typically exist in other media types. We discuss extensions of these video navigation interfaces that lift some, but not all, of the restrictions.

We then go on to propose an alternative approach based on a document's changing structure. Because such a structure changes slowly over time, we can base temporal navigation around this structure as a reference point. We discuss the principles of strcutrue in temporal navigation interfaces and propose an approach to visualize and interact with hierarchical structures over time. We introduce a system for source code history navigation based on the soruce code's hierarchical structure. Users perform navigation tasks twice as fast using structure aware interfaces compared to existing similar interfaces. They also strongly prefer our visualization when it comes to exploration tasks. Finally, we discuss how to extend this approach to other media types and non-hierarchical structures.

Überblick

Das Erstellen von Dokumenten, beispielsweise eines Computer Programms, eines Buchs, oder eines Films kann mehrere Jahre dauern. Während dieses Prozesses kann es relevant werden frühere Version mit der aktuellen zu vergleichen, oder eine ältere Änderung rückgängig zu machen. Das Ziele zu einer speziellen Version eines bestimmten Teils des Dokumentes zurückzukehren wird von typischen Werkzeugen nicht gut unterstützt, da häufig nur das Dokument als Ganzes betrachtet wird.

Wir vergleichen die Navigation in Dokumentversionen mit Videonavigation. In diesem Bereich existieren Interfaces, die, basierend auf der Bildposition von Objektem, einen bestimmten Zeitpunkt ansteuern. Es ist allerdings nur möglich Zeitpunkte anzusteuern, in denen sich ein Objekt bewegt. Auch die Anwendung dieser Interfaces in anderen Medientypen gestaltet sich schwierig, da von einem statischen Hintergrund ausgegangen wird. Wir stellen Erweiterungen dieser Navigationsinterfaces vor, die einige, aber nicht alle, dieser Probleme lösen.

Als Alternative stellen wir einen Ansatz vor, der auf der Struktur eines Dokuments beruht. Da die Struktur sich über Zeit nur langsam verändert, kann der Nutzer sie als Referenzpunkt für die Navigation verwenden. Wir definieren wie Struktur in einem temporalen Navigationsinterface verwendet werden kann, und wie man Veränderung hierarchischer Strukturen visualisieren kann. Wir stellen ein Navigationswerkzeugs für die Historie von Quellcode vor, und zeigen, dass Nutzer mit diesem Werkzeug schneller durch eine solche Historie navigieren können. Wir zeigen auch, dass Nutzer die vorgeschlagene Visualisierung für die Exploration stark bevorzugen. Abschließend diskutieren wir Ansätze für die Nutzung dieser Ideen für anderen Medientypen und nicht-hierarchischen Strukturen.

Acknowledgements

I have to thank numerous people who have been involved in finishing this work. I would not have gotten far without the support of everyone around me. I want to thank Prof. Dr. Jan Borchers for first introducing me to Human Computer Interaction and for giving me the opportunity to work at the media computing group. His comments on my work always provided insights from a different perspective that helped me question my own assumptions. Jan's amazing ability for words have greatly improved phrasing and argumentative structure of all scientific papers I wrote in the last few years. I owe gratitude to Prof. Dr. Karin Herrmann who provided yet another perspective on my work and familiarized me with the concept that the evolution of a document is just as relevant as the final work. Prof. Dr. Michael Rohs became the secondary advisor for this thesis and put a lot of effort in improving the editorial quality of this document.

I cannot express enough thanks to my exceptional colleagues at the media computing group. Foremost, I want to thank Christian Cherek, Florian Heller, Thorsten Karrer, Jan-Peter Krämer, Leonhard Lichtschlag, Max Möllers, and Simon Völker for their invaluable support when it came to providing feedback, writing papers, running user studies, or simply cheering for the ThoMo Code Show. Without them, I would not have learned half as much and my time would not have been half as enjoyable.

Then there are the students who trusted me enough to become their thesis supervisor. Christian Cherek, Christian Corsten, Stephan Deininghaus, Adrian Isbiceanu, Anne Kathrein, Ardi Tjandra, and Torben Schulz have contributed insights and discussions that greatly affected my own view on the subject of this work.

And finally, a huge thank you goes to my family and friends: my wife Jenn for always having my back, relentlessly improving my English, and pushing me through the final stretches of writing even when I was barely home; my parents who amazingly helped me with my crazy 'computery' projects since I was a teenager; my friends Niko, Rob, and Lukas for years of listening to my nerdy comments on any-

thing; and our cats Zorro and Hermione for being a great comfort after a long day of work and very patient listeners when practicing my talks.

Thanks a lot everyone!

Chapter 1

Introduction

"A complex system that works is invariably found to have evolved from a simple system that worked."

—John Gall

The evolution of a document is a time-based medium.

Navigation through time is usually seen as a means to select a playback time of a video or an audio file. There are a lot of tasks where users want to navigate to a certain time in such a *time-based* medium. Examples of theses tasks are editing, ethnography, or simply wanting to find a specific scene in a video. In this thesis, we also consider a less thought of time-based media type: the evolution of a document over time. Changes to a document gradually affect individual aspects of that document and can slowly transform it from something short to something long, something simple to something complex, or something disorganized to something refined. An author writing a novel, a programmer writing a piece of source code, or a designer developing a logo, create their respective artifact slowly over time in lots of individual steps. When we consider all of the versions of a document linearly arranged in time, we get something similar to a video that arranges individual video frames over time and each frame only has minor changes to the previous one.

Going back to any of the steps from a document's *history* might be interesting for several reasons.

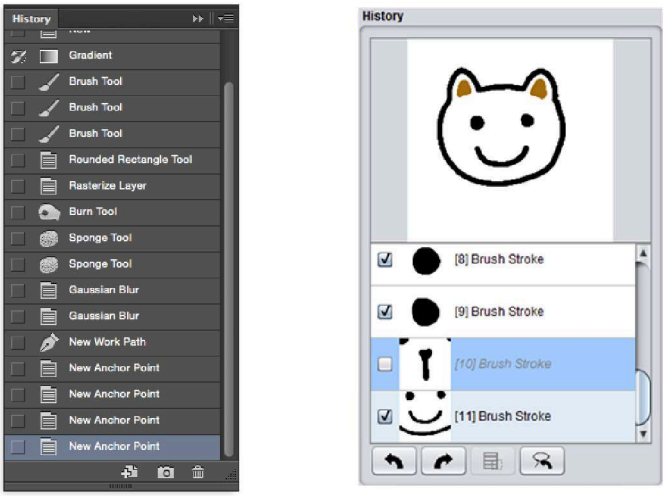


Figure 1.1: Adobe Photoshop history list (left) and selective undo for paintings [Myers et al., 2015]. The history list in Photoshop describes the past actions. When the user selects an item for undo, all items following the selected item will also be undone. Myers' selective undo for painting applications shows a similar list of changes, but individual changes can be undone without affecting subsequent changes in the list.

- The user undoes recent changes or completely reinstates an earlier version.
- The user looks at a previous version, copies a certain part of the older document that she has deleted since, and pastes the copy into the most recent version.
- The user wants to understand a document and looks at different versions from the history to understand what the document used to look like and how it changed over time.

Of course, this functionality exists in the form of undo, history lists, or version control systems (see Figure 1.1).

Undo is usually destructive.

Unfortunately, undo and history lists can easily be destructive; accidentally typing a character after hitting undo a

Figure 1.2: The Youtube playback interface with a typical timeline slider at the bottom. The interface on the right shows frames surrounding the current playhead. These provide additional context and help the user to find navigation targets.

couple of times makes it impossible to go back to the current version. They also do not allow to undo one specific change while leaving newer changes intact. Research prototypes have proposed ways to implement such a *selective undo*, e.g., [Berlage, 1994] and [Myers et al., 2015] (Figure 1.1). Here, only selected events from a history are undone by reapplying inverse changes to the end of the document.

A problem of all of these undo approaches is that they focus only around the most recent changes. Undoing a modification hundreds let alone thousands of changes ago is impossible because usually only a limited set of recent versions is available. Even with version control systems that enable this long-term undo, finding the right change to undo is still difficult. Typically we navigate the history of a file through a list of user generated change descriptions. If these descriptions are not precise enough to let the user find what she is looking for, she has to search all changes.

Our goal is to enable temporal navigation in the history of documents.

In this thesis, we aim at creating a temporal navigation interface for the history of a document. This interface helps the user understand what kind of changes occurred, which, in turn, helps to find a version of interest easily and quickly. To do so, we draw on the knowledge of temporal navigation interfaces from the domain of video navigation.

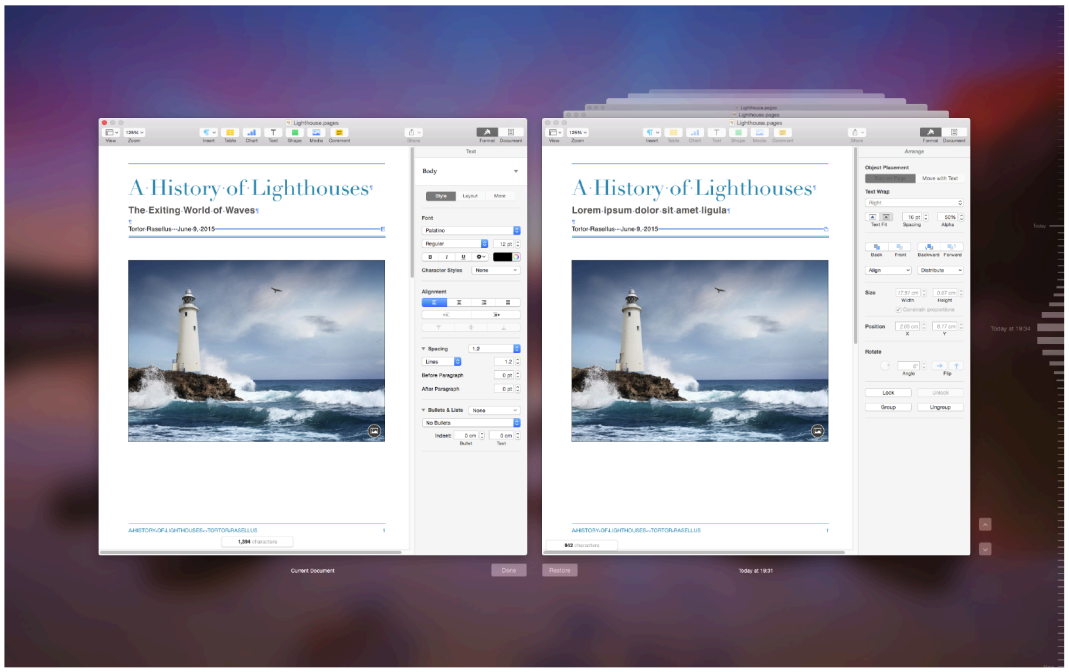


Figure 1.3: Apple's Time Machine interface for navigating versions of a text document. The timeline is stacked vertically on the right of the screen with a callout around the current location of the mouse pointer.

1.1 Temporal Navigation Interfaces

The typical navigation interface for time-based media is a *timeline*. A timeline localizes points in time uniformly on a slider. We obviously see timeline interfaces in media players such as Youtube (Figure 1.2). There are also some timeline based interfaces for version management of files or documents such as Apple's Time Machine (Figure 1.3). In these interfaces, navigation to a point in time is easy if one can estimate the target on the timeline and then drag the slider to that point; this is only the case if one can express the navigation target in terms of time.

1.1.1 Content-based Temporal Navigation

Arguably, the user's navigation goals are rarely formulated in terms of time, but rather in terms of content. In video,

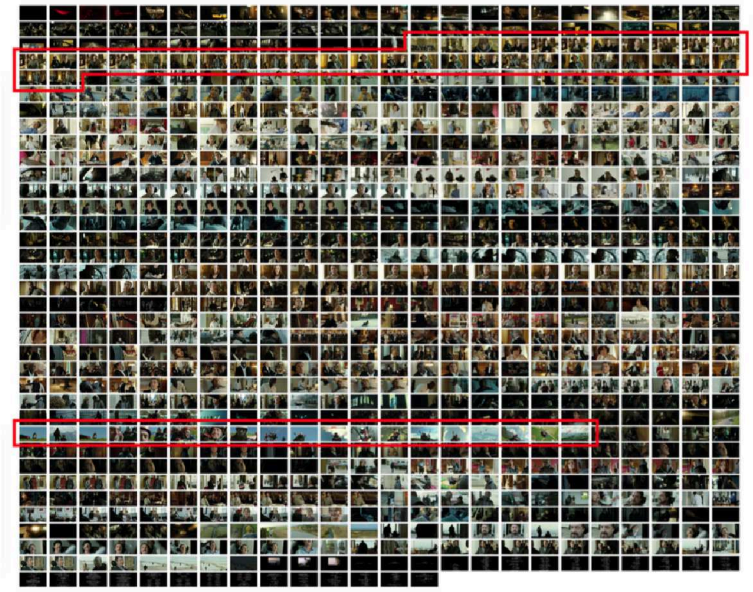


Figure 1.4: Display of thumbnail images from a movie in Swifter [Matejka et al., 2013]. As the user scrubs the timeline (not shown), Swifter shows a quick overview of the scenes in the movie. Users can also select navigation targets directly from this overview.

the user does not want to navigate to minute 20 of a soccer game, they want to navigate to the moment of the first goal. Similarly, a user does not want to find the version of a text document created at 3PM but the version that had 'the classier subtitle'.

Annotated timelines attempt to bridge the gap between content-based goals and time-based navigation.

Thus, there are a number of navigation interfaces that *annotate* the timeline with some visualization of the content. Youtube's video player shows a row of small frames around the current time as soon as the user starts dragging (Figure 1.2). Swifter [Matejka et al., 2013] uses a similar approach but uses more space which enables the user to quickly skim even long videos (Figure 1.4).

These annotations do not have to be the content itself.

The slider annotation does not need to show the contents directly but can also visualize properties of the content. Consider the waveform visualization of an audio file that shows the peak amplitude of the audio signal (Figure 1.5).

Figure 1.5: A timeline slider annotated with an audio waveform. Volume changes are easily visible and allow the user to reason about which audio events might impacting the visualization.

A user cannot directly understand the content from this visualization but she can use it to infer what event in the content caused the specific visualization. A user might want to find the part of an orchestral music piece where a soloists finishes and the whole orchestra starts again; she interprets a part of the audio with small amplitude as the soloist part and the following part with larger amplitude as the whole orchestra. Through her own interpretation, she can identify the right point in time without the visualization actually providing the answer directly. Although she may also misinterpret the visualization, she will often be able to find the right point in time by looking at multiple potential navigation targets. Pirolli [2007] calls this indirect representation of content *information scent*. People sometimes specifically design actions to create strong information scent in such a visualization, e.g., the slate clap when filming a movie scene scene creates a sound that is easily identifiable in the waveform and is used during editing for syncing video and audio.

1.1.2 Direct Manipulation Temporal Navigation

We also see interfaces that go beyond using a slider interface. Karrer et al. [2008] allow users to directly interact with the motion of the objects in a video scene (Figure 1.6). The user can select an object in the scene, the system will display its motion trail, or trajectory, and the user can then drag it to any locations on its motion trail; the video will just jump to the time where the object is at that location. Time is completely ignored in these interfaces in favor of a more goal-oriented property like an object's location. Karrer [2013] describes theses interfaces as *semantic navigation interfaces* because they try to directly represent the user's

Figure 1.6: Video navigation using the DRAGON interface [Karrer et al., 2008]. Users can select objects such as the car by clicking. The system shows the movement trajectory (yellow) and allows her to navigate in the video by dragging to a specific location. The object will snap to the time where the location on its movement trajectory is closest to the mouse cursor.

understanding of the medium's content and her task, e.g., DRAGON supports tasks that involve a moving object and a static reference point. Because of this specialization, these interfaces usually perform very well in their domain.

Direct Manipulation Temporal Navigation is too restrictive for exploration tasks.

In this work, we argue that this specialization becomes problematic when the user has to deal with a number of different tasks or badly defined tasks such as exploration of a document's history. When the task changes, the interface can quickly become unsuitable and has to be replaced by another, more suitable interface. Assume the car in Figure 1.6 stopped at some point to let another car cross the intersection. Times where an object is stationary are not accessible with a system like DRAGON. If the task is to find out why the car stopped, the user has to choose a different interface, usually the timeline, to find an answer. Depend-

ing on the task, creating semantic navigation interfaces can also be very difficult, especially when it comes to the interpretation of content. We cannot easily find an algorithm that understands and parses concepts like argumentative structure in a text.

1.2 Structural Navigation Interfaces

A document's structure is often easy to find and the user can interpret its meaning.

We propose to side-step this issue by creating an interface that builds on hierarchical structures within a medium. Such hierarchical structures are often easy to find, and can be used as information scent for higher level tasks. In written text, we can easily find structural elements such as words, clauses, sentences, paragraphs, sections, or chapters. A user who identified that two paragraphs represent two different arguments can look for changes within these paragraphs or for changes in the ordering of these paragraphs. Selecting the right representation level, like paragraph or sentence, and judging if the navigation goal has been reached are left to the user. In this work, we created an interactive prototype that visualizes the structure of a document and allows the user to follow the information scent provided by these structural changes.

1.3 Thesis Structure

The remaining part of this chapter further motivates the need for analyzing the history of documents in detail. In the subsequent thesis chapters we then discuss the following:

- Chapter 2—"Exploration with Semantic Navigation Interfaces" provides an overview of semantic navigation in the context of video and our approaches to improve direct manipulation video navigation. We attempt to create an interface that allows the user to retain the context of a situation even when an object is not currently moving. The approach informs our

understanding of the need for a different kind of interface for exploration.

- Chapter 3—"Structural Navigation Interfaces" builds on these findings and shows an interface for the exploration of the writing process of exams. Based on this simple interface we generalize the notion of structural navigation interfaces and define four navigation properties that are specifically enabled by these interfaces.

- Chapter 4—"Chronicler: Navigation in Source Code Histories" dicusses specific structural navigation interfaces for the exploration of source code. Using the inherent hierachical structure of source code gives us the opportunity to create a visualization that can then be used for navigation and exploration. A study shows that structural interfaces outperform non-structural interfaces for navigation tasks, and the visualization is strongly preferred by users in exploration tasks.

- Chapter 5—"Generalizing Tree Flow Interfaces" shows how to create similar hierarchical structures for other media types and the trade-offs we have to consider.

- Chapter 6—"Summary and Future Work" concludes our work with a review of our work and an overview of open questions.

1.4 Navigating Document Histories

We already mentioned several reasons for navigating document histories. In this section we are giving some insights into why this is a relevant topic of research. We focus on two topics, selective undo, and understanding document evolution.

1.4.1 Selective Undo

Selective undo interface present the user with a linear list to find changes.

Berlage [1994] introduced direct selective undo as a way to undo only specific commands from the document history. His work considers when and how selective undo would work. He imagines commands to undo to be selected using a history list of textual descriptions of the commands. [Myers et al., 2015] who recently presented selective undo for painting applications (see Figure 1.1), use a visual representation of the changed painting element to show the document state. [Yoon et al., 2013] propose a visualization of the document history, that supports finding the right versions for selective undo. Instead of showing the actual document content, they only highlight where in the document changes occurred.

Version control systems are probably the most well-known class of applications that support selective undo today. Finding the right version of the document there is usually done using a linear history of all documents in a repository. We can sometimes reduce the amount of versions to look through by only considering changes to a certain file. Most programming environments also have a special "diff view" (Figure 1.7) where the user can compare at two versions of a file side by side. Usually these views also show additional annotations based on a textual difference algorithms, e.g., [Myers, 1986].

We also see a form of selective undo in file system backup tools like Apple's Time Machine. Time Machine enables the user to look through backed up versions of the file hierarchy and then recover a modified or deleted version from the backup. While an individual file is either completely overwritten or duplicated, other files in the folder are not touched. Time Machine also works to look at individual files 1.3 and shows two editable versions side by side to selectively copy parts of the document.

All examples mentioned above do not support the user to find the right version.

All of these tools have the issue that the user must find the right version to compare or restore first. Most tools require the user to navigate the history linearly with the minor restriction of being able to focus on a certain file or

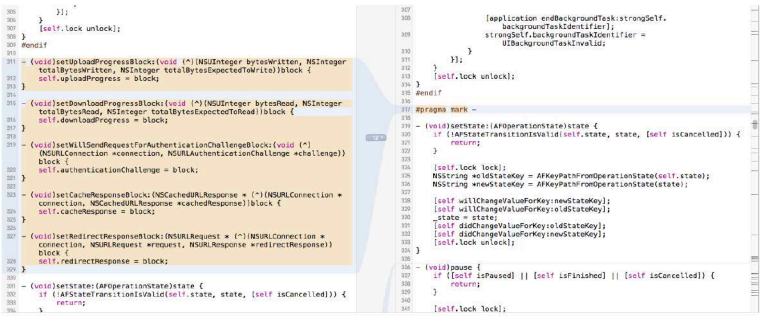


Figure 1.7: File comparison view in Apple's Xcode. The difference annotations easily show how the five methods on the left were deleted sometime between the two versions. Interestingly, this display is incorrect. The methods were actually moved somewhere else in the file, but the Diff algorithm chosen by Xcode could only model this as a delete here and an insert at the new location.

folder. Yoon et al. [2013] allow the user to consider changes by their location in the file, but the location of an entity of interest can easily change by adding text somewhere else in the document.

1.4.2 Understanding Document Evolution

It is also possible to use the history to understand the evolution of a document. We can look at this in two ways: we can use it to understand the current version better and answer questions about how old a part of a document is, what other parts it was introduced together with, or how often it has been changed. Or we can use it to understand an old version which may give us insights into the thought process of creating the document.

The visualization of Darwin's "Origin of Species" shows when certain passages where introduced.

One popular example of using the document history is an experiment by Ben Fry about Charles Darwin's "Origin of Species" [Fry, 2009b] (see Figure 1.8). His experiment visualizes changes to the book and highlights what was added or modified in each version. He reports [Fry, 2009a] that the idea came "from a discussion with a friend, who had begun to wonder whether Darwin had stolen most of his

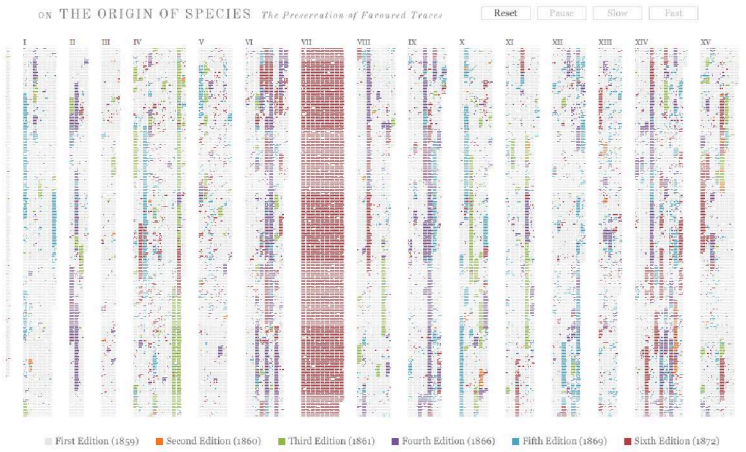


Figure 1.8: Ben Fry's visualization of "The Origin of Species". Each colors denotes a different edition and shows which text was inserted when.

better ideas from Alfred Russel Wallace." According to Fry, the understanding resulting from the experiment instead raised Fry's appreciation for Darwin's ideas.

We created an interface for literary scholars who analyze the evolution of books.

Deininghaus [2010], who wrote his diploma thesis under the author's guidance, developed an interactive tabletop application for literary scholars. His work was later published at ITS 2010 [Deininghaus, Möllers, Wittenhagen, and Borchers, 2010]. Part of Deininghaus' work was the analysis of the work-habits of literary scholars.

He found that the core work material of the interviewed scholars, the historical-critical editions, are an "extensive collection of textual variants, facsimiles of preserved original documents, commentary, historical background, or information on the text's reception and transmission". The work process of literary scholars includes understanding different passages, secondary literature, and, most relevant for this work, comparing different versions of the same text. In our paper we proposed to integrate digital representation into the typically paper-based work process (see Figure 1.9).

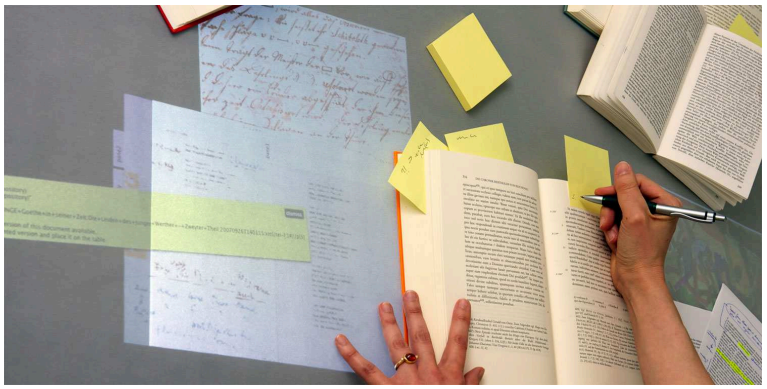


Figure 1.9: The digital table proposed in our paper. It enables literary scholars to add digital annotations to physical books on top of the table. From [Deininghaus et al., 2010]

An open issue is how to find appropriate versions to compare in the large amount of text available in such an edition. Historical-critical editions seek to support analysis from a wide number of different angles. Their contents are typically rather extensive in order to cover a large number of aspects; however, this also means that for many conceivable questions, much of that information might well be irrelevant [Deininghaus et al., 2010]. This again supports the idea that we need an interface to help find specific versions of a document. Also, when looking at an unknown document or a new question, a user often cannot know which characteristics might be most interesting to compare. Thus, an interface that is as unrestrictive as possible and supports exploration of unknown versions would also be important for this usecase.

The same concepts apply to developers.

We also strongly believe that the process of understanding and exploring histories is also relevant for other media types. Developers, for example, could use these ideas to understand a particularly complex method by navigating to an older version that might still conceptually do the same but is easier to understand. Before coming back to navigating histories of text documents, we take a look at video navigation interfaces and how well these interfaces can enable exploration tasks in the domain of video.

Chapter 2

Exploration with Semantic Navigation Interfaces

"We know what we are, but know not what we may be."

—William Shakespeare

When navigating a document's history, we have two deal with two separate groups of tasks. Navigation lets the user select the version of the document we want to see. Exploration allows the user to find out about what versions of a document actually exist.

Our original idea for navigating change histories was to use concepts found in direct manipulation video navigation interfaces. Thus, we look at existing temporal navigation interfaces in general and DRAGON, an interface for semantic navigation in video scenes in particular. While this tool works well for a certain kind of navigation tasks, it performs worse when the navigation task changes even slightly. We show how to extend the interface so that it can handle a more diverse set of navigation tasks. However, the amount of flexibility required for exploration tasks seems infeasible to achieve with the proposed interface. This informs our argument that of using an alternative represen-

tation of content over time to allow exploration based on information scent.

2.1 Exploration

In Navigation tasks, the user has a specific goal in mind.

Consider the tasks we have described so far; undo and understanding a document. Especially for undo tasks, we often have a clear goal in mind, e.g., one wants to find the document that does contain a particular sentence. These kinds of tasks can be solved with interfaces like the timeline slider, albeit without helping the user understand where they have to navigate to. This problem is tackled by content-aware navigation interfaces like DRAGON that bridge the gap between content-oriented goals and time-based navigation.

Exploration tasks require the user to first discover a series of navigation goals.

For the second example of understanding document evolution, the user may not know beforehand which versions they want to see and compare. Discovering these versions is what we consider to be exploration. During exploration the user may consider different document versions and then move on to a different version that may be better suited for their task. In our definition, exploration is the process of finding a series of before unknown navigation goals. User interfaces can support this task by providing *information scent*.

2.1.1 Information Foraging and Information Scent

Information scent represents content in a way that helps users to find the information they are looking for.

Pirolli [2007] extensively studied how user's navigate information and models users as information foragers, analogous to animals foraging in patchy environments where food is arranged into clumps. As a user gathers knowledge from an information patch, the available new information will be reduced, until she has to find a new patch. One core concept in finding these new patches is information scent. Pirolli [1997] define *information scent* as a "[...] terse representation of content [...] whose trail leads to information of interest". Pirolli and Card [1999] expand on this definition

stating that "information scent is the (imperfect) perception of the value, cost, or access path of information sources obtained from proximal cues [...]." Examples of information scent are textual or visual summaries of websites in search results. There is a strong impact of the quality of information scent on the navigation task that is to be solved.

Pirolli [2007, chapter 4] exemplifies this using a web search task. The information structure of links on the Web create a lattice graph. Possible browsing choices at each node of the graph (a link) can be idealized as a search tree that the user has to traverse to find the right web site. With perfect information scent, the amount of links to follow becomes linear, i.e., the depth of the target in the tree. As the information scent becomes worse, the user has to search more and more of the tree, requiring a full tree search in the worst case, i.e. an exponential amount of links to follow. Small improvements in the quality of information scent "can have dramatic qualitative effects on surfing large hypertext collections."

Information foragers can learn to understand what original content created a terse representation.

Pirolli models this quality as the user's ability to interpret proximal cues, e.g. the website summary, as being caused by desired information, e.g., the website discussing a specific topic. Note that the quality is not only dependent on the information scent, but also on the users' learned ability to interpret it. In summary, "the theory of information scent proposes that information foragers have mechanisms that reflect the probabilistic texture of the environment." We further use these concepts in chapter 3—"Structural Navigation Interfaces".

2.2 Related Work

We distinguish the field of temporal navigation techniques into timeline based and direct manipulation techniques. Timeline based techniques are built on top of the timeline slider and enrich the navigation in some way. Direct manipulation techniques [Shneiderman, 1982] enable the user to directly interact with the content.

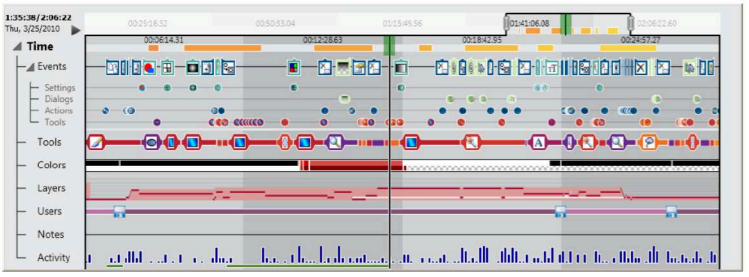


Figure 2.1: Stacked timeline visualization with editing events. The hierarchical nature of the annotations allows to summarize individual event timelines in a group and allows to hide unwanted details. From [Grossman et al., 2010]

2.2.1 Timeline Based

Multi-level timelines refine the selection of a lower-level on higher-level timelines.

Obviously, there are many extensions of timeline approaches for time-based navigation. Mills et al. [1992] created a multi-level timeline interface that samples frames from the video and equally distributes them over the timeline. A user can then select part of the timeline to create a new sub-timeline which can then sample frames from a subset. [Arman et al., 1994] automatically detect shot boundaries and find representative frames for each shot. They then present scenes with similar representatives on a second level timeline.

Stacked timelines represent different aspects of a medium at once.

Other approaches use multiple stacked timelines that represent different aspects of the medium. Mackinlay et al. [1991] developed the perspective wall, a well-known focus + context technique that uses a central 2D focus area and a perspectively distorted context area. The 2D focus area represent events on multiple stacked timelines over a spanning domain, commonly time. Grossman et al. [2010] use a timeline hierarchy for several event classes that occurred during the creation of a document (see Figure 2.1). While these events do not directly represent the state of the document at a certain time, they serve as information scent for the user. Besides events, single points in time, stacked timeline can also be used to describe time ranges. Vascon-

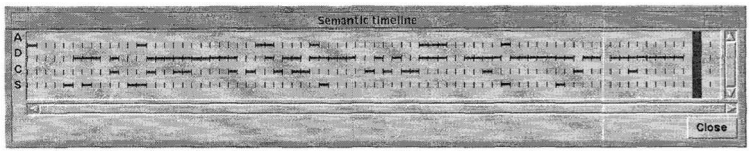


Figure 2.2: Stacked timeline visualization showing a movie trailer. Each stack level shows a different aspect of the shot (A = action, D = close-up, C = crowd, S = natural set). From [Vasconcelos and Lippman, 1998]

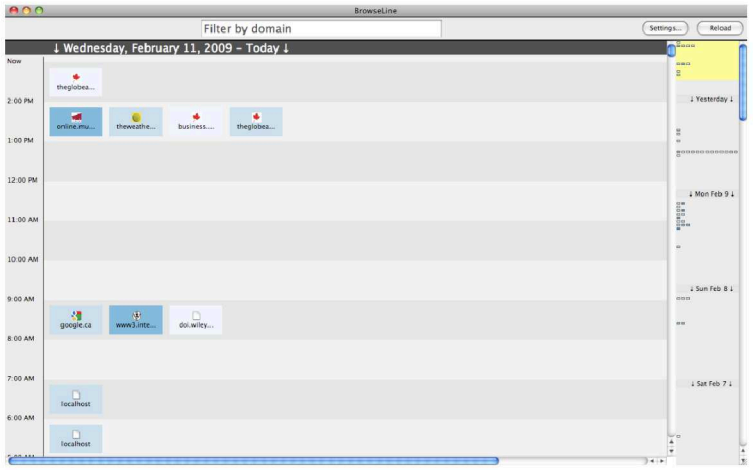


Figure 2.3: BrowseLine provides information scent based on the frequency of browsing events. The vertical timeline on the right replicates an abstraction of the main view to navigation to sparse or dense event clusters on a certain day. From [Hoeber and Gorner, 2009]

celos and Lippman [1998] propose to use stacked timelines to provide information about different aspects of shots like action or close-up (see Figure 2.2)

2D timeline visualizations often present both overviews and details.

BrowseLine [Hoeber and Gorner, 2009] uses a 2D time visualization approach for a user's browsing history. The system represents time in 2D (hours vertically, minutes horizontally) and also linearly on a scrollbar which also shows a miniature representation of the 2D view (Figure 2.3). Continuum [André et al., 2007] shows a summary timeline with an activity graph and a detail view containing information

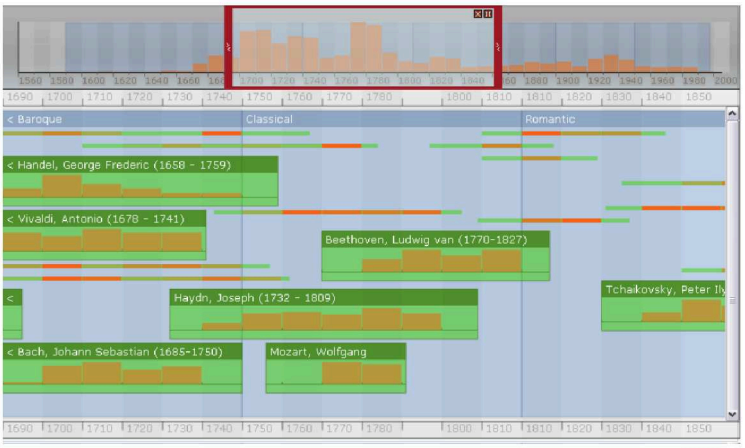


Figure 2.4: Continuum shows a hierarchical arrangement of music pieces by composer by era. From [André et al., 2007]

about activity. The summary timeline does not show the actual content but an abstraction of the data. Their stated goal is to enable navigation through a large number of events (1000 to 10000). To achieve this they use the hierarchical structure of the event data; Figure 2.4 shows their "composer example" with music pieces grouped by era and composer. Swifter [Matejka et al., 2013] shows a grid of representative video frames around the current location of the timeline slider. In this grid, the user can then easily see upcoming differences like scene changes when navigating through a longer video.

SlitTears [Tang et al., 2009, 2008] allows the user to draw lines into a video scene (Figure 2.5). At each point in time the pixels under the line are copied to the timeline. This extrudes all events at the location of the slit over time and makes it easy for the user to see changes occurring at the location of the line. By drawing lines at strategic locations, the user can then find relevant events.

2.2.2 Direct Manipulation

Direct Manipulation Temporal Navigation ignores time and allows the user to interact with the location of objects in the scene

In the last decade, researchers have created a number

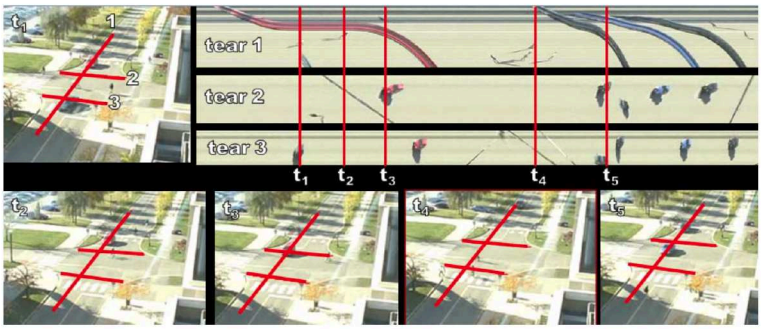


Figure 2.5: SlitTears replicate the pixels under lines drawn in a video scene on a timeline. When the pixels change this creates visible effects on the timeline that can be interpreted by the user, e.g., the long curved streak in tear 1 indicates a car coming from the top stopping at the intersection and then continuing. From [Tang et al., 2009]

of direct manipulation interfaces in the domain of video. Schematic Storyboarding [Goldman et al., 2006] is a visualization technique inspired by movie production storyboards. This visualization adds movement information about objects and the camera directly into the scene in the form of arrows. By clicking and dragging on these arrows, the user can manipulate the location of the object represented by the arrow.

Trailblazing [Kimber et al.], DRAGON [Karrer et al., 2008], and DimP [Dragicevic et al., 2008] all track selected objects and display their 2D movement trajectories within the scene. The user can then navigate through the video by dragging object to locations on that trajectory. DimP also enables background stabilization to deal with camera motion (see Figure 2.7). Trailblazing additionally localizes objects on a map spanning multiple cameras in a surveillance setting and allows the user to interact with the video by dragging the object on the map representation. [Goldman et al., 2008] allow the same interaction, but among other usecases, introduce more sophisticated object selection mechanisms through drawing on areas. We also extended DRAGON for interaction on mobile devices [Karrer, Wittenhagen, and Borchers, 2009].

Figure 2.6: Schematic storyboarding summarizes movement in a scene by 3D arrows. Clicking and dragging on these arrows scrubs through the video. From [Goldman et al., 2006]

Figure 2.7: DimP allows direct manipulation video navigation even when the camera is moving by shifting the position of the image in the video view in the opposite direction of the camera movement. From [Dragicevic et al., 2008]

Direct manipulation of table cells easily allows a visualization of all data over all points in time.

Perin et al. [2014] and Vuillemot and Perin [2015] created similar direct manipulation approaches for ranking tables. They describe two approaches. The first approach lets the user drag selected table rows to a vertical location, i.e., its rank. The second approach displays a line chart that shows the values of a selected columns for each point in time and then lets the user drag to a horizontal location to select one of these versions. Since the column data is one-

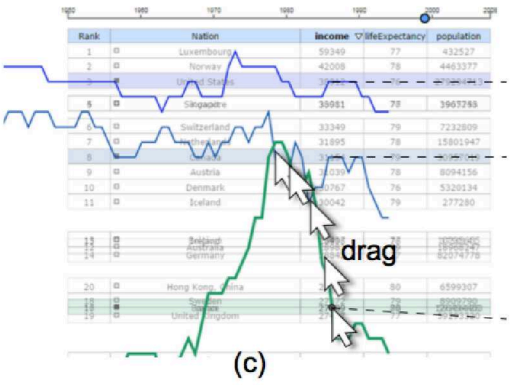


Figure 2.8: Direct Manipulation of values in ranking tables allows the user to drag a selected cell to a rank in the table. We can easily see the co-evolution of multiple selected rows at the same time. From [Vuillemot and Perin, 2015]

dimensional, the display of all values of one column at all times creates a 2D visualization (see Figure 2.8). This 2D visualization can easily show where and when several values change together or switch ranks.

For video this is not easily possible without a 3D visualization like [Eccles et al., 2008] or the reduction of 2D content to a line like SlitTears [Tang et al., 2008]. 3D Visualizations have the issue that they do not scale well in the number and size of visualized objects because of occlusion issues.

2.3 Semantic Navigation

Karrer [2013] introduced the notion of semantic interfaces. While Karrer defines these interfaces for a number of domains, we will focus on semantic navigation interfaces in the domain of video. A semantic navigation interface is designed to minimize the *semantic distance* from the users goal to a navigation action. It is the amount of effort a user has to make to translate an intention into actions performed in the interface Karrer [2013, chapter 2.2]. Karrer showed that

these navigation interfaces allow the user to solve a specific task in a highly efficient manner.

By design, they are, however, strongly linked to their specific task. We argue that this strong coupling between task and interface creates problems when the tasks are not known a priori or the amount of potential tasks becomes too large. This is typically the case during exploration tasks, because the user first has to identify her next navigation goal. Semantic navigation interfaces can also be inherently difficult to design when the task requires a very complex semantic structure.

2.3.1 Issues of Semantic Navigation Interfaces

Semantic navigation interfaces are linked to a specific task.

We look at the issues of semantic navigation interfaces in the area of video navigation. The DRAGON interface works as follows. A user selects an object in a video scene by clicking, the system computes the location of the object at each frame in the video scene until it disappears. All of these locations form a path that is visually shown on top of the video scene; thus a user can see at which locations in the scene the object will be. When scrubbing along this path, the system jumps to the point in time where the object is closest to the location of the mouse pointer. The task solved with DRAGON is effectively: "navigate to a time in the video where a specific object reaches or moves past a fixed location in the scene." A concrete navigation task could be "navigate to the time where this car leaves the intersection." Such *direct manipulation video navigation (DMVN)* interfaces allow the user to easily translate this intention into actions, i.e. click on car, drag to where the trajectory leaves the scene. The timeline does not allow this as easily because the drag target is unclear; users have to guess a target, scrub there, reevaluate the scene content, and repeat.

The object for a DRAGON task must be specific, because it has to be selected before navigation, it must also be visible so it can be selected, and it must move because otherwise we do not have a trajectory to interact with. It is also not allowed for the target location to be another moving object,

Figure 2.9: When interacting with the selected car in the scene, it is easy to navigate to static reference points, e.g., the moment the car passes the street light. It is not possible to drag the car to a moving object like the bike because the bike will likely have moved somewhere else by the time the car gets to that location.

because if it was, the visualization would not help us determine where the two objects would meet. We end up with a similar problem as the timeline slider (see Figure 2.9).

In the area of direct manipulation video navigation there are two major issues we want to discuss.

1. During exploration tasks, the user has a difficult time to define her next navigation task.

2. The problem of having restrictions around a specific tasks is difficult to solve.

The following sections discuss attempts to integrate different tasks into semantic navigation interfaces. We discuss a task selection interface by Kathrein [2011] and our Dra-

gLocks [Karrer, Wittenhagen, and Borchers, 2012] extension of DRAGON that enables interaction through objects pauses, e.g., the car stopping in the middle of the intersection. Our conclusion is that these attempts can create benefits for navigation but these techniques still only have limited use for exploration interfaces.

2.4 Task Changes in Semantic Navigation Interfaces

We analyzed how semantic navigation interfaces handle task variations using *Direct Manipulation Video Navigation (DMVN)* systems. We already discussed the problem of navigating to a point where multiple objects meet. Kathrein [2011] identified other tasks and created multiple semantic interfaces for each of these tasks.

The upcoming sections discuss two solutions for problems like this. Create an interface for each task, or create a single interface that can deal with multiple tasks.

2.5 Providing Multiple Interfaces

To specialize for a variety of tasks, we can use an interface per task.

In a scenario where more fine grained control over navigation goals is necessary, we can create interfaces for each individual task. Especially tools supporting tasks around long running video could benefit from more fine grained task distinction. Kathrein [2011], who wrote her Bachelor's thesis under the supervision of the author, identified a range of tasks that are currently not well-covered by DMVN interfaces. She classified these tasks into four distinct groups.

1. Area Dependencies
2. Objects Act
3. Objects Interact

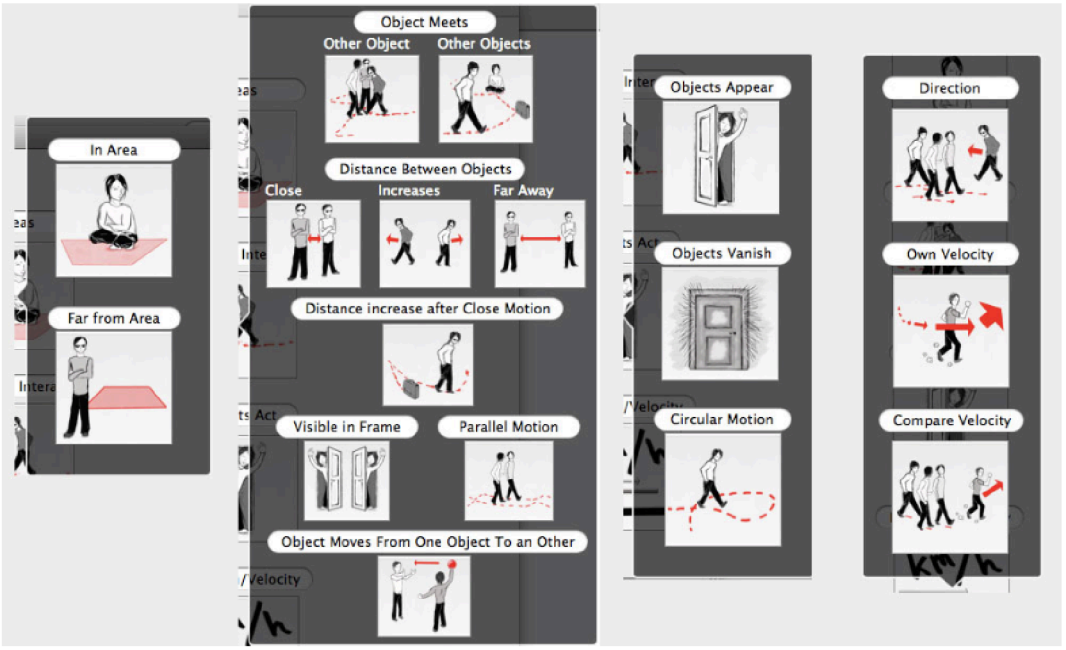


Figure 2.10: Different tasks not supported in direct manipulation video navigation

4. Direction and Velocity

Figure 2.10 shows the tasks identified for all groups.

DMVN systems have trouble with tasks involving multiple moving objects.

A common theme for these tasks is the interaction of multiple objects, e.g., are these objects in the region at the same time, do they move together, are they increasing their velocity in sync? Kathrein proposed a timeline interface that highlights time ranges or events for selected objects in a scene (Figure 2.11).

The success of such an approach heavily depends on two factors. First, the *completeness of provided interfaces* determines if the user can solve her current task. It is easy to come up with scenarios where the user requires the detection of a movement pattern that is not covered by such a system. What if we are only interested in other more specific shapes of trajectories? How does a user select objects that are not yet visible in the frame? If we created an automatic selection algorithm, how do we describe the objects that should be selected? While this could to some extent be alleviated by end-user programming, creating an event

Figure 2.11: The timeline interface to highlight time ranges of object movement patterns. Here we see that the selected objects, the red and the yellow ball, will enter and leave the orange region simultaneously.

detector for video is outside the abilities of most end-users. Second, with the growing number of interfaces, the ability of the user to *select an appropriate interface* is also problematic. This requires additional effort in the design of the interface selection which is difficult to scale to large number of tasks.

Specialized semantic interfaces are good for navigation but problematic for exploration.

It certainly makes sense to have these interfaces that are specialized towards specific semantic goals. But, when it comes to exploration of content, these interfaces put the user in a position where they have to make an uninformed choice for an interface. If a user does not know what is going to happen in the video, she can only guess at different event types with arbitrarily selected objects. We argue that these specialized interfaces should be complemented by a more general navigation interface that allows users to form hypotheses for these specific tasks.

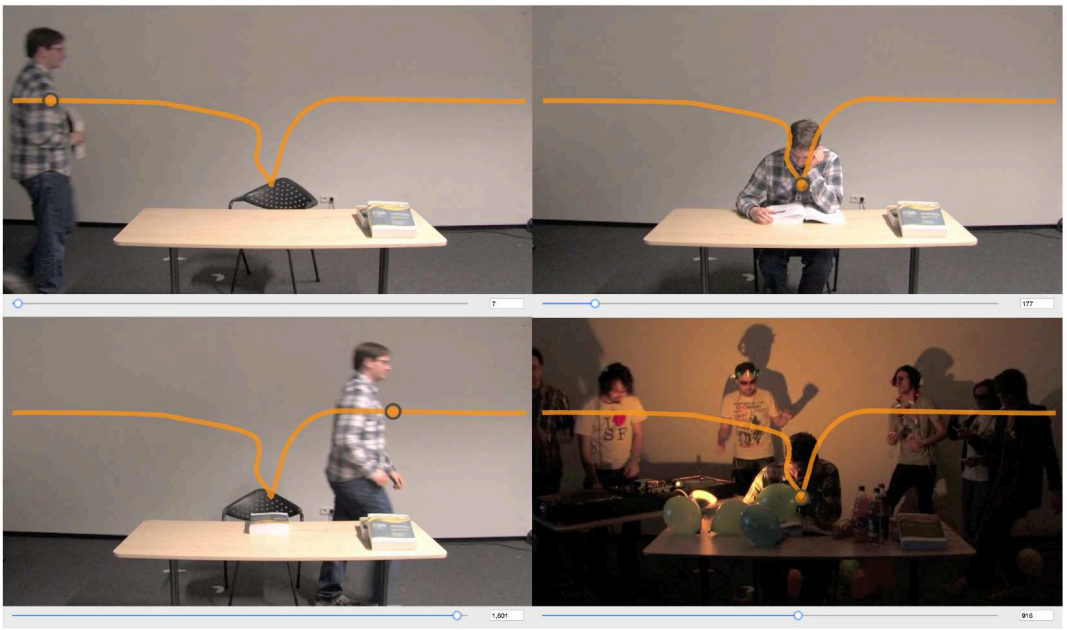


Figure 2.12: Temporal ambiguity during a pause. The top left, top right, and bottom left frame are accessible, while the bottom right frame can only be accessed using the timeline slider because the person has not moved.

2.6 Handling Multiple Tasks

Instead of providing multiple interfaces, it may also be possible to create one interface that works for multiple tasks. Such an interface would still have a small semantic distance for its designated task but also allow access to times that would usually not be covered by this interface. The work in this section has been published at CHI 2012 [Karrer, Wittenhagen, and Borchers, 2012].

Moments where objects pause in a scene are not navigable.

We analyzed how different DMVN systems deal with the ambiguities arising from certain object motion patterns. Figure 2.12 shows an example where a person moves into a scene from the left, sits down towards the center of the frame and then moves out to the right. Events while the person is sitting down are not accessible using DMVN systems. When dragging the object across that pause, typical DMVN systems skip the frames in the pause, losing the context of other events occurring during that pause. During a pause, the object stays in the same space for some

amount of time. We call this a *temporal ambiguity*. A temporal ambiguity occurs whenever an object occupies the same space at different times.

We considered three different movement patterns that lead to temporal ambiguities:

1. An object crossing its own path leads to **self-intersecting trajectories**, e.g., a roller coaster going through a loop will be at the base twice. At the intersection point, it is not clear if the user wants to navigate to the earlier or the later occurrences of this location.
2. **Recurring movement** occurs when an object repeatedly follows the same path at different points in time, e.g., a swinging pendulum or a ball bouncing up and down. This creates issues for tasks where one wants to navigate to a specific 'iteration' of the movement. When dragging an object towards a cusp in the trajectory, it is unclear whether the video is supposed to be advanced forward or backward in time.
3. **Pauses** are created by an object staying at the same location for some time, e.g., a car stopping at an intersection and then continuing on. At the location of the pause the time to choose is ambiguous.

There is no automatic method for selecting the correct time interesting to the user.

Any of the ambiguous times may be interesting to the user, so we cannot generally select one point in time over the others. This is especially problematic in pauses and recurring movement ambiguities. A user may want to navigate to where a model stops on the runway, strikes a pose, *and* the scene lighting is good. Navigate to a pass of a grandfather clock's pendulum *at* 10:09AM. These tasks are still strongly related to one object, but require a secondary condition to hold that is not modeled in the interface. In the previous section, we already argued that having an interface for a lot of these special conditions is infeasible. For the proposed tasks this would also be very difficult to implement, i.e., the computer vision algorithm to interpret the quality of scene lighting is non-trivial. However, it may be possible for the interfaces to detect and deal with the motion ambiguities.

2.6.1 Handling Ambiguities in Different DMVN Systems

Motion ambiguities have been considered in existing DVMN systems.

The DMVN literature prior to our paper describes interaction techniques intended to handle undesired jumps in the video while navigating. These same techniques also affect how self-intersecting and recurring movement ambiguities are handled. All these approaches rely on modifying the distance measure d that is used to determine the next frame f to be displayed during the interaction:

$$f(p,T) = [\underset{t \in T}{\operatorname{argmin}}(d(p,t))]_f$$

where p is the screen position the user is dragging the object to and $T \subseteq F \times P$ is the object's trajectory consisting of tuples (t_f, t_p) of a frame number and a position [Karrer, Wittenhagen, and Borchers, 2012]. When dragging the object to a position this formula determines the frame in which the object is closest to that position under a given distance measure d.

We distinguish *purely spatial* and *spatio-temporal* distance measures, where purely spatial measures only depend on t_p and spatio-temporal measures depend both on t_p and t_f.

Goldman et al. [2008] employ the trivial implementation of a purely spatial distance measure:

$$d_1(p,t) \cong \|p - t_p\|$$

Purely spatial measures may be confusing and cannot navigate pauses.

This approach always displays the frame in the video where the object is closest to p, usually the mouse cursor. It distinctly has the advantage of allowing quick access to any non-ambiguous point along the trajectory. However, should another segment of the trajectory get closer to the mouse cursor than the segment the user is currently interacting with, the playback position will jump discontinuously. This may be confusing to the user. Temporal ambiguities are not considered using d_1. Depending on the implementation of the argmin operator, a different representative frame will be selected for any ambiguous part of the trajectory. Pauses are completely skipped and are only detectable by a sudden change in the scene around the object.

Dragicevic et al. [2008] explicitly consider recurring movements at trajectory cusps by requiring their distance measure to have directional continuity and self-intersecting trajectories by requiring arc-length continuity:

$$d_2(p,t) \cong \|p - t_p\| + \|\text{arclen}(t_p, o_p)\| + k_D$$

where o_p is the previous position of the object and $k_D > 0$ a constant being added when the arc-length changes sign. d_2 is still a purely spatial distance measure and will thus skip pauses completely.

Spatio-temporal distance measures will stick to pauses but also cannot navigate them.

Karrer et al. [2008] discuss the need to disambiguate recurring movement patterns and jumps during navigation. They include the temporal component in the distance computation, so that ambiguities can be resolved through temporal distance to certain frames. Their distance function with the current frame number o_f is:

$$d_3(p,t) \cong \|(p_x - t_{p_x}, p_y - t_{p_y}, o_f - t_f)^T\|$$

Their approach can still not navigate the content of pauses. In contrast to d_1 and d_2, the consideration of the temporal component allows the detection of pauses. A dragged object will stick at the beginning of a pause because the temporal outweighs the spatial component. Dragging it further leads to the spatial component becoming stronger until the object snaps past the pause. Frames after the pause can be accessed by backtracking. This also reduces the ambiguity at cusps and self-intersecting trajectories. However, the closer the user drags to the tip of the cusp, the less impact the temporal component has.

Kimber et al. propose a spatio-temporal distance metric that handles the cusps better by also ensuring directional continuity.

$$d_4(p,t) \cong c_\theta \cdot \|p - t_p\| + \|o_f - t_f\| + k_D$$

with k_D defined as above. It otherwise behaves similar to d_3 but includes a time-dependent term c_θ that makes the 'snapping' across pauses happen more easily the longer the dragging takes.

All these distance measures emphasize slightly different tasks that may be performed during in-scene naviga-

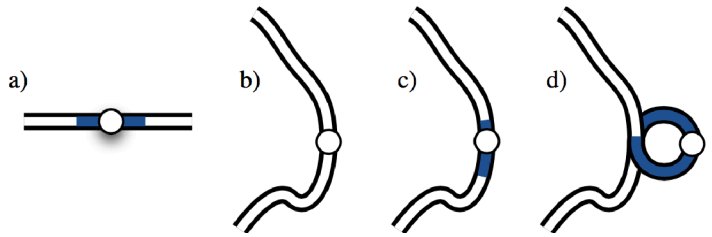


Figure 2.13: Spatial pause extent in different navigation techniques. On the timeline (a), the frames are spaced evenly, a pause in a DMVN (b) has no extent, the DMVN with embedded timeline (c) distributes the pause over a part of the trajectory, and loop (d) changes the shape of the trajectory. From [Karrer et al., 2012]

tion. Selecting an appropriate distance measure is task-dependent and could be done by the user, potentially using a *quasimode* [Raskin, 2000] like holding down the control key. While d_1 will be fastest when the user knows exactly where to drag an object and she does not care about intermediate steps, the other measures emphasize continuity and help to support keeping track of the context during navigation. Measures d_3 and d_4 enable the detection of pauses which comes at the cost of speed when accessing frames after pauses. The trade off for the spatio-temporal always boils down to the question "how many pixels is one frame worth?". This question does not have a universal answer. None of the discussed systems allow accessing content during pauses without resorting to secondary timeline slider interface.

2.6.2 Handling Pauses

We proposed two solutions that enable DMVN systems to navigate pauses: *loop*, which changes the geometry of the trajectory around the pause location, and *embedded timeline*, which changes the meaning of the trajectory around the pause location. Figure 2.13

We proposed to embed timing information into the trajectory to solve this issue.

Loop

Loop alters the shape of the trajectory.

The idea behind loop is to alter the shape of the trajectory around the pause. This techniques is inspired by the Popup-Vernier [Ayatsuka et al., 1998]. The single spatial point of the pause is expanded into the shape of a loop that can then be navigated in a single stroke. Rehoming to the timeline slider is thus no longer necessary. Besides enabling navigation, the shape of a loop provides the user with a hint about its special meaning through its recognizable shape. Frames of the pause are equally around the loop and are now accessible as if on a curved timeline slider interface. The base of the slider corresponds to beginning and end of the pause. Although this is in itself a self-intersection ambiguity, it can easily be handled by the discussed distance measures d_2–d_4.

The temporal granularity of navigation inside the loop can implicitly be adjusted through the radius of the dragging gesture. A larger gesture provides more granularity because a larger movement is necessary to spatially get closer to the next frame. This works similarly to the micrometer interface [Ahlberg and Shneiderman, 1994] or the mobile zoom slider [Hürst and Götz, 2008]. Non-pause locations on the trajectory are not affected by the loop approach, guaranteeing direct manipulation accessibility for all non-pause frames.

Embedded Timeline

Embedded timeline alters the meaning of the trajectory.

The second technique alters the meaning of the trajectory along a short extent around the pause location. Location on the trajectory does not map to the location of the object but to the time in the pause. The shape of the trajectory is not altered. This means that we embed the timeline into the shape of the trajectory. All time points on this part of the trajectory are redistributed equally. Expanding the pause into the spatial domain again guarantees that we can navigate the pause without having to resort to the timeline slider. In contrast to the loop technique, the embedded timeline causes no, potentially disrupting, visual modifica-

tion. This comes at the cost of decoupling navigation from the object in the altered part of the trajectory. There is also no way to identify a pause from the shape of the trajectory; thus, we set the color of the embedded timeline to be different from that of the rest of the trajectory. For longer pauses, this technique will show the same resolution problems as the timeline slider, making it hard to navigate to a specific frame inside the pause.

2.6.3 Implications for Navigation

We identified four different navigation targets around pauses.

A user's navigation target can be in four different areas around pauses: *before the pause*, at the *edge of the pause*, *in the pause*, and *after the pause*. We describe these cases for forward navigation. Navigation before the pause works analogous to navigation without a pause; we consider it covered by standard DMVN approaches. The edge of the pause described the frame where an object stops moving. As discussed earlier, spatio-temporal distance measures easily support navigation to the edge of a pause. Both the embeded timeline and the loop visualize the edge of the pause, but loops may make accessing the edge easier because of the ability to affect navigation granularity. Purely spatial measures make accessing the edge hardest because they are neither visualized nor do they affect interaction. Loop and embedded timeline are both designed to support navigation in a pause. We expect the loop to perform better for longer pauses because of the resolution issues mentioned earlier. Existing DMVN systems all require the use of the timeline slider to access pauses. Crossing the pause to access frames after the pause is easiest with purely spatial distance measures since dragging is not affected by the pause. Spatio-temporal measures may actually require the use of the timeline slider to cross the pause if the pause is so long that the available distance on screen is not large enough to outweigh the temporal component of the metric. Loop and embedded timeline are designed for crossing the pause easily.

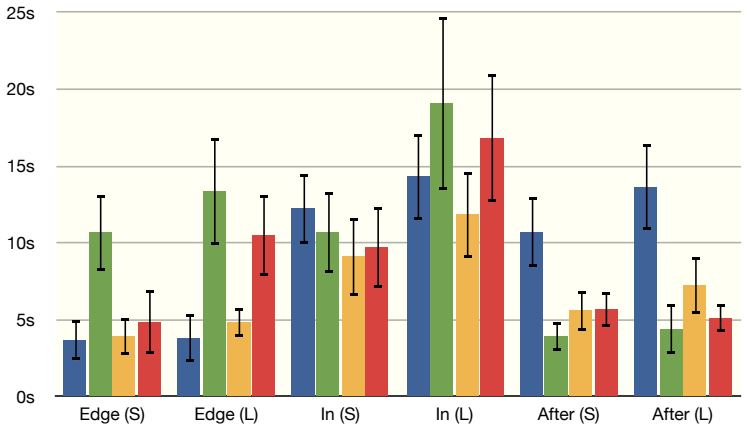


Figure 2.14: Means navigation times and 95% confidence intervals. Navigation to the edge, in, and after short (S) and long (L) pauses using spatio-temporal (blue), purely spatial (green), loop (yellow) and embedded timeline (red). Adapted from [Karrer et al., 2012]

2.6.4 Experiment

We evaluated the navigation performance of existing DMVN systems using d_1 and d_3, and the two new techniques using d_3. For details about the experiment, we refer to the paper [Karrer, Wittenhagen, and Borchers, 2012].

The choice of interface has a big impact on navigation times.

The experiment again shows the strong influence of semantic interfaces specialized for a certain task (Figure 2.14). Navigating to the edge of a long pause with spatio-temporal distance measures is on average 3.5 times faster than purely spatial measures. Navigating across a long pause is on average 3.1 times faster using purely spatial measures compared to spatio-temporal. Loop performs fastest within pauses although there is no significant evidence that it outperforms the other approaches.

These results support that even small task changes have a large impact on the interface performance for semantic navigation interfaces. Selecting the right interface for these task thus becomes crucial. In our paper, we suggested to implement this using a quasimode. The problem with introduc-

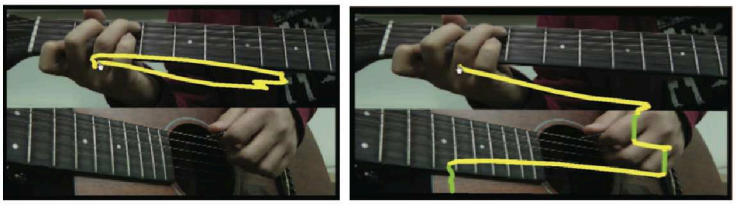


Figure 2.15: Difference between 2D and 3D trajectories. Both trajectories show the hand movement in the upper part of the video. The transformed trajectory on the right allows pauses and recurring movement to be navigated easily. It comes at the cost of losing the tight coupling between absolute position on the trajectory and absolute position in the video. From [Nguyen et al., 2013]

ing modes, even quasimodes, means we have to solve the same scaling issues mentioned for explicitly using multiple interfaces.

Using one of the two suggested techniques we could probably create an acceptable compromise. Exploration works slightly better with such an interface because we can access all frames during a pause. The visualization of the loop provides information scent for a pause and by navigating through the pause, the user can attempt to find out why an object stopped. Unfortunately, this only solves one of the discussed ambiguities and none of the tasks discussed by Kathrein [2011].

Newer solutions can handle the other ambiguities at the cost of the original strength of a DMVN interface.

Nguyen et al. [2013] presented a solution to all three ambiguity issues. They resolve ambiguities by transforming the trajectory into a 3D representation that contains time as an axis. While navigating, the trajectory transformation will be updated so that the current position on the trajectory reflects the current position of the object[1]. Figure 2.15 shows the difference between 3D and 2D trajectories. In this way, ambiguities will be distinguishable because of a distortion of the trajectory on top of the video. While this resolves temporal ambiguities, it also changes the direct manipulation interface. We stated that DMVN systems are designed

[1]Video figure: https://www.youtube.com/watch?v=YomeZfCo7P0

especially around one moving object and a static reference point. Since the user's reference points will also be transformed, they will now be moving as well. Thus, drag targets are now harder to identify. This works well for the use-case of recurring movements or pauses, because the user can predict the repetition easily. It produces issues when the movement is not repetitive; the interface thus has to be a mode. Obviously, the proposed system does not affect how we can navigate to interaction points of multiple moving objects.

3D interfaces can handles all issued we described, but easily become cluttered.

GeoTime [Kapler and Wright, 2005, Eccles et al., 2008] provides a 3D visualization that enables to see the relationship of multiple objects on a static map (Figure 2.16). It shows the map with trajectories floating above it so that they represent time. We could imagine a similar system for video where the map is replaced by the scene background, perhaps as an extension to [Nguyen et al., 2012]. GeoTime works well to see the interaction of a small number of objects on a map. Because of the difficulty of projecting the location of the floating trajectories to a point on the map, the trajectories are anchored on the map in regular intervals. This means the display easily becomes cluttered when there is lots of motion. A technique like GeoTime also relies on a static background, because the absolute position of objects becomes meaningless when the background changes.

2.7 Other Media Types

Other media types do not rely on the absolute position as much as video.

So far , we have talked about temporal navigation in the domain of video. As described in the introduction, we want to be able to also deal with other time-based media types, especially the evolution of documents over time. With other media times, we usually have the issue that we cannot take the same approach as DMVN systems, because the absolute location of objects is not meaningful. This works in individual video scenes because we can expect the background to be static, because even if there is camera movement we can reverse its effects for the purpose of navigation [Dragicevic

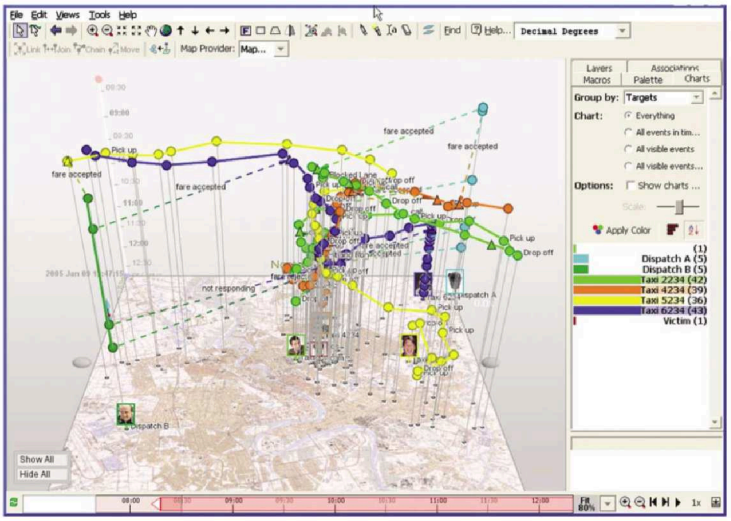


Figure 2.16: GeoTime displays 3D trajectories on top of the map with time on the y-axis. This means that intersection int the space above the map mean the same location at the same time. From [Eccles et al., 2008]

et al., 2008, Wittenhagen, 2008]. Thus, objects in the background become static reference points to navigate to.

A DMVN system could work for the history of presentation slides.

Interestingly, it could make sense for the history of some 2D media types, e.g., presentation slides or graphics programs. The objects on the slide, i.e., text, images, and video, have different locations and sizes over time. In theory, we can use the exact same interaction approaches for these. We could navigate to a version of a slide where an object was at a certain location on its slide. However, the relationship between multiple objects may actually be more interesting to the user than their location.

In text documents, movement of individual segments to fixed locations becomes meaningless. There is no such thing as a fixed background that stays constant. If the author adds something in the beginning of the text, lines in the end will move; the whole 'scene' constantly changes. Thus, we cannot use an interface such as DRAGON.

2.8 Requirements for Temporal Exploration

In summary, we want a temporal exploration interface that implements the following features:

1. *Ambiguity-free*: Any point in time is accessible.
2. *Multi-object visualization*: The change history of multiple objects is visualized.
3. *Historical object visualization*: Objects that are not visible in the current frame are visualized.

The interface should avoid ambiguities when the user selects a point in time, i.e., we want all frames to be accessible. A common timeline slider is a simple interface that implements this feature, but not the other two features that are especially important for exploration tasks. We want the user to learn the relationship of multiple objects, and see entities that are currently not visible. We discuss how to create an annotated timeline interface that enables all three features in the next two chapters.

Chapter 3

Structural Navigation Interfaces

"A scene has to have a rhythm of its own, a structure of its own."

—Michelangelo Antonioni

We discussed that it can be very difficult to extend semantic navigation interfaces to encompass a variety of navigation tasks, and they are thus not well-suited for exploration interfaces. In this chapter, we tackle the problem from another angle and show that the inherent structure of media is often easy to capture and the structure can directly inform navigation interfaces. We formalize the concept of a structural navigation interface and describe the form of structures that we find in digital media.

3.1 Using Structure for Temporal Navigation and Exploration

What we have been capturing with DMVN systems is the relationship of objects to fixed objects on a fixed background. For the exploration of the history of a document, we usually do not have such a fixed background. However, text in such a document only changes gradually over

Structure changes slowly over time and can thus be used as a slowly changing point of reference.

time and they are grouped together in chapters, sections, paragraphs, and sentences. So, we hypothesize that we can use this temporary stability with gradual changes as at least temporary reference points that help the user to orient herself. *Structural navigation* then uses this stability and the form of the structure to let the user find such anchor points. If we can communicate the structure to the user and visualize its changes over time it may provide sufficient information scent to make hypotheses about changes in the underlying content.

What structure is used is not necessarily as important. The syntactical structure of chapter, sections, etc. as described above is very much based on the medium of written text. But we could also imagine this structure to be more meaningful like relationships between characters in a book, or the connection of methods through the call graph in software development.

Structure is an abstraction of the content of a document. It can reflect only the presence of some content object, or more detailed information about the relationship of different objects. Examples of such relationships are be neighborhood, distance, or containment.

In the next section, we describe an example of how we can capture content changes in a given structure and how this helps us to navigate that content. We then continue to define structure and finish this chapter with how structure can enable navigation before we continue to discuss visualization in the next chapter.

3.2 ExamPen

We use ExamPen as an example of how structure can serve as information scent.

We now show how, by capturing content changes in a very simple structure, a user can understand a lot higher-level semantic information. This can then be also be augmented with user generated semantic information.

At the CHI 2010 Workshop on Next Generation of HCI and Education, we presented *ExamPen* [Karrer, Wittenhagen,

Lichtschlag, and Borchers, 2010], a digital pen based tool for analysis and visualization of student performance during written exams. The focus of the paper lies on supporting teachers to understand their students' perceptions of the exam and strategies employed to solve the exam. We enable support for this task through very simple annotation of content with structural information and then recording changes within this structure over time. Even though this approach enables a rich understanding of the exam-taking process, it is virtually free with modern technologies and does not require extra efforts from the student or the teacher.

3.2.1 Task

It is crucial but difficult to create fair exams.

Exams are the prevalent way to evaluate students' performance at schools and universities. Devising fair exams is difficult and time consuming. A teacher has to design questions that are are easily understandable and unambiguous. The questions must be able to differentiate different levels of skills. An exam can be highly stressful for the students, so a fair question also clearly communicates the required knowledge and effort to the students. Each task should also yield an appropriate amount of marks that reflects this required effort and knowledge.

ExamPen supports teachers to evaluate exam questions before reuse.

Our system is designed to support teachers to refine exam questions over multiple iterations. These iterations can span from test-drives with teaching assistants to repeatedly using questions over multiple years. Past answers can indicate difficulties the students may have had with a question. This helps to uncover misleading question texts or tune the marks a question awards. By including the timing of questions, i.e., how long and when students work on a task, we aim at enabling even more understanding of students' strategies and the relationship between exam questions. Understanding how students perceive and approach the exam helps the teacher to identify potential flaws in their exam design or grading policies. With ExamPen, teachers can not only see the final version of the exam papers, but explore the paper's evolution over the course of the exam.

Including timing makes new quality indicators accessible, like the time a student spent on a specific task or the number of times she revisited a task to modify her answer. Individual strategies on how a student approaches an exam can enable teachers to provide better personalized tutoring on areas students have difficulties with or on test-taking strategies as a whole.

3.2.2 System Design

The structure of our example exam is a linear list of fixed task regions.

ExamPen structures the exam into task regions. Each task has a dedicated space where the students write their answers. We explicitly chose to use digital pen technology and not keyboard or tablet based systems (e.g, [de Silva et al., 2007]) to leave the experience of writing the exam unchanged. This has the advantage that we do not increase the cognitive load through an unfamiliar system, which is linked to worse test performances [Oviatt et al., 2006]. We captured the students' work over time by printing the exam on Anoto[1] micro-dot pattern; students then wrote the exam using Anoto digital pens. The pens not only record the location of where a stroke was drawn but also the timestamp of when the stroke was drawn. Together, the timestamps and the location within the task structure allow us to visualize strategies and dynamic behavior of the test-takers. This structure is very simple and does not yet have the property that it dynamically changes.

Stroke chains are logical groups of pen strokes occurring together.

Our ExamPen software groups strokes into *stroke chains* based on both, temporal and spatial proximity. Temporal to group individual letter strokes into words and sentences. Spatial to avoid grouping words across the end of a line. This secondary structure on top of the completely unstructured pen input (x, y, time) approximates the perception of words or lines without actually understanding them. We can use this to then create colored bounding boxes around the written text where the color represents the time of writing (see Figure 3.1). If the order of certain words seems important, the teacher can quickly navigate there using a color annotated timeline (Figure 3.2).

[1] http://www.anoto.com

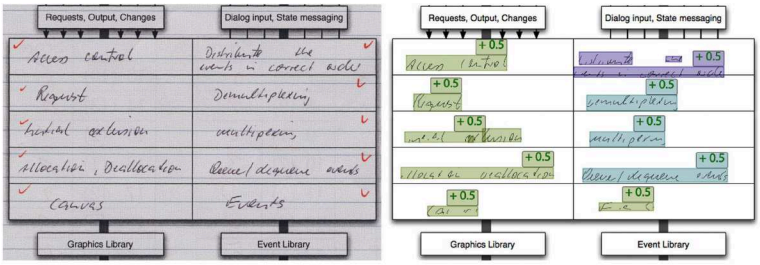


Figure 3.1: Grouping of handwritten text in ExamPen. The digital version of the exam can be annotated with timing information and awarded points. From this we can easily see that the student was unsure about what to put down and only gave the correct answers in later passes of the exam (olive ≈ 25 min, blue ≈ 55 min, purple ≈ 75 min). From [Karrer et al., 2010]

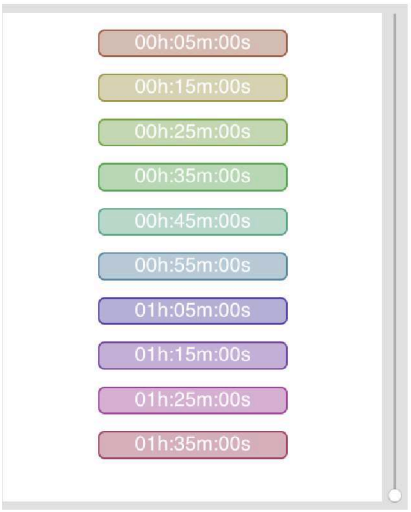


Figure 3.2: Time-based navigation in ExamPen. Since we annotated stroke chains with a color representing time, a teacher can quickly navigate to their creation using the annotated timeline.

From just looking at the grouping and their colors, we can already gather a quick overview on how long a student took for a task and how often she revisited it for annotations or corrections. We can also use these colored boxes for navigation to the moment something was written by us-

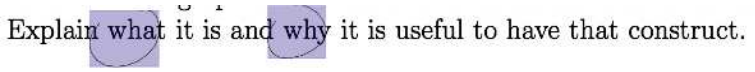


Figure 3.3: Neutral stroke chains around keywords made towards the end of the exam. The student checks if he has finished all important parts of the task. From [Karrer et al., 2010]

ing a timeline slider that is annotated with the same time-derived colors. The teacher can add annotations for correct, wrong, and neutral, as well as the amount of awarded or deducted points to the stroke chains. Examples for neutral chains are margin notes or emphasis marks like underlining of important aspects in an exam question. These annotations are integrated into the structure as another level of information that could not be trivially automated. Based on this mixture of automatically and manually collected data our software creates visualizations for each individual student and the group as a whole.

3.2.3 Data Analysis

We visualize the time a student spends in each task in a visualization based on a floating bar chart. Bars represent a stroke chain, the position of the bar is determined the task, the width of the bar is determined by the stroke chain duration, and the height of the bar is determined by the teachers estimate of the time required to be spent on the task 3.4.

Our visualization of activity within a structure over time indicates exam solving strategies.

This visualization highlights several characteristics about the student's exam solving strategy. In Figure 3.4, the student went through the whole of the exam in three front to back passes. We see how often each question is being revisited by only focusing by one horizontal slice of the graph. The teacher can gauge the ratio of time spent to time expected to be spend when accumulating all strokes for one tasks. Especially tasks 13b also took a lot longer than expected. The fact that there are some strokes in the earlier iterations may indicate that the student skipped the task knowing that they are not as clear on the answer. Questions

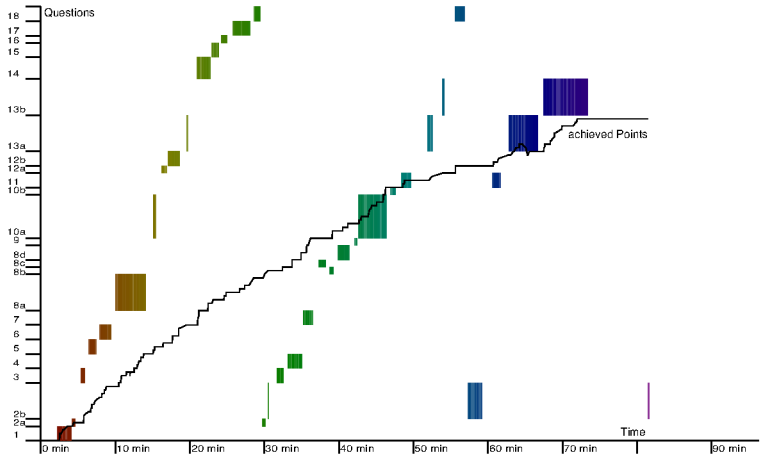


Figure 3.4: The diagram shows how the students solves the exam in three clearly visible passes. The impact diagram shows where points were added and deducted. It is easy to identify that task 13b took longer than expected by the teacher. From [Karrer et al., 2010]

8a is only worked on once, indicating that the student did not feel they needed to add to their answer in later iterations. The shape is also roughly square, suggesting that the teacher's estimate worked well for this particular student.

Besides visualizing the exam progress of individual students we can look at accumulated data for the whole class. We project the ratio r_t of average time spent and time expected to be spent on to rectangles by giving them a fixed width c for $r_t \leq 1$ and determine the height as $r_t \cdot c$. For $r_t > 1$ we use height c and width $\frac{c}{r_t}$. This helps to determine if the group as a whole spent a different amount of time on a task than the teacher expected and can indicate unfairly designed questions (see Figure 3.5). A wide *footprint* means more time spent than alloted, narrow less time than alloted, and a square is a desirable result. This gives a quick overview of which tasks could benefit from additional analysis.

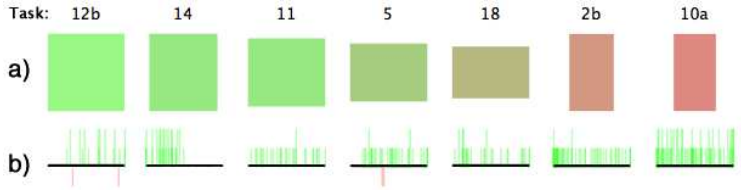


Figure 3.5: The task *footprints* at the top indicate the quality of the task duration estimation by the teacher. Tasks 18 took longer than expected on average and tasks 2b and 10a shorter. The point distributions underneath shows that task 14 suggests to the students they need to write more than actually expected. From [Karrer et al., 2010]

Semantic information on top of the given structure can reveal additional details.

Up to this point, we only used automatically generated information based on stroke chains in the task structure over time. When incorporating the mark annotations we can also visualize at what time, i.e., the aggregate of time spent on writing for a specific question, marks were made or lost. If the students feel they needed to write more than actually expected, we see a skewed distribution of point awards to the start or the end of the time spent (see Figure 3.5).

3.2.4 ExamPen Summary

The example indicates that simply having information of structure over time can help the user to interpret what happened.

ExamPen shows that looking at the creation process within a fixed structure can support the understanding of that creation process. This structure was intentionally very simple, which made it easy to track over time. Teachers can already use a simple structure like this to recognize exam solving strategies. Custom annotations can then also add semantic information identified on top of this structure.

Interestingly, this this is possible without changing the process of writing and exam for the students and it is easy to imagine that mark annotations could also be made using a pen. So, simply by adding structure and timing information, we can create an interface that supports understanding and visualizing documents over time.

Our goal is to find a structural visualization that works for other media types, like creative writing, source code, or presentations. A lot of documents for any of these purpose already have a concept of changing over time. There are version control systems for source code, e.g., git or subversion, and design documents, e.g. Pixelapse[2]. Authors sometimes physically keep drafts of their books before release [Barrows, 2015]. Other documents like presentation often have multiple versions stored for backup purposes.

However, all these document histories are missing structural information. With some effort, we can add this structural information later, as we show in chapter 4—"Chronicler: Navigation in Source Code Histories". The rest of this chapter deals with how having structure supports temporal navigation.

3.3 What is Structure?

Dix et al. [2004, chapter 20] states "[...] the data sets that arise in information systems typically have many discrete attributes and structure: hierarchies, networks, and most complex of all, free text.". We follow the same notion of modeling structure as a graph. However, we seek to avoid the complexity of free text and other intricate media types by not actually fully modeling such structure. Instead we attempt to only employ simple rules common to a medium to represent it. In written text words are separated by spaces, sentences by periods, paragraphs by empty whitespace. These rules are in place to make a text human readable and are rarely broken. However, we only aim at providing a frame of reference which the user can employ for a higher-level task. Figure 3.6 shows such a simple structure for the example of a sentence.

Structure represents the relationships of content entities in a document.

We consider structure to be an abstraction of content of one version of a document. It describes some relationship of entities in the medium. Entities can be *content*, such as characters in text or pixels in an image. They can also be

[2] https://www.pixelapse.com

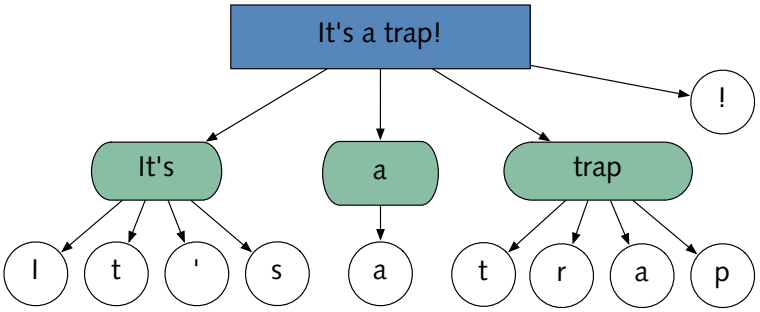


Figure 3.6: A simplified structure of a sentence. In this example, a sentence would be delimited by special punctuation character and is made up of words that are themselves made up of characters. Note that this means we do not even consider special cases like the contraction and we also get bad results when someone adds an ! in the middle of sentence. However, such a structure is very independent of content, e.g., the language used.

structural elements, such as words formed by characters, or a group of pixels. The relationship could be spatial colocation, containment, or other arbitrary adjacency measures such as a call graph connection in source code. It typically makes sense to consider a mixture of relationships, e.g., we never consider a structure without a containment relationship.

We can describe these structures in terms of a graph. A *structure graph* is a directed graph with vertices V representing entities, and edges E representing the relationship between these entities. Note that a graph, in this general definition, may have cycles or multiple connections between between two nodes. We speak of an ordered structure if all edges between two vertices have an ordering. For our work, we consider three different shapes of such a structure graph.

1. *Linear* structures only represent an ordering of entities. The task structure in ExamPen is such a linear structure because we only considered the content created in the solution space ordered by the tasks number they belonged to.

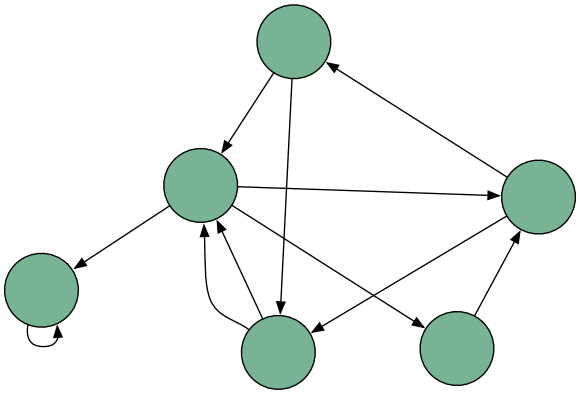


Figure 3.7: A complex network structure has no restrictions about the layout or connection of individual nodes.

2. *Hierarchical structures*, i.e. trees, are graphs that have a special root node, are acyclic, and every node can be reached from the root by exactly one path. Such a tree is *ordered* when each node's children have an ordering. The latter is often easy to find when content itself is arranged linearly, e.g. sentences in a paragraph and words in these sentences have a trivial ordering (Figure 3.6). Two dimensional content usually does not have this inherent ordering, but we may be able to create one (see chapter 5—"Generalizing Tree Flow Interfaces").

3. For now, we consider all other graphs to be *complex networks* 3.7. An example of this would be the structure spanned by the call graph, or the emotional relationship between characters in a book. While other special graph types might have interesting properties for history navigation, they are not further discussed in this thesis.

3.4 Structural Representation

Before we generally describe how to use structure for temporal navigation, we look at other examples of interfaces explicitly using structural information. The goal of this is

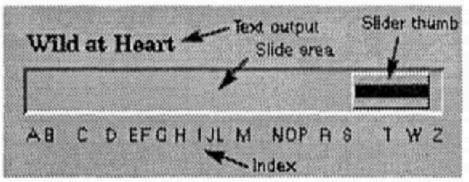


Figure 3.8: The Alphaslider allows people to select entries from a list. The spacing of the letters below the slider indicates how many entries there are for each letter. From [Ahlberg and Shneiderman, 1994]

to get an overview of how different kinds of structures are used for interface design.

The Alphaslider [Ahlberg and Shneiderman, 1994] abstracts from a list of words by index the list by first letter. Letters are then presented underneath the slider interface where the spacing after each letter proportionally represents the amount of represented words. This is, of course, a simple linear structure on top of the list. Ishak and Feiner [2006] create a content-aware scroll bar by using structural knowledge of the document to create a linear scrolling experience through important content that is non-linearly arranged. Examples are the typical two column layout in scientific papers and faces in an image. This approach does not affect the visual appearance of a standard scroll bar.

As Johnson and Shneiderman [1991] put it, "a large quantity of the world's information is hierarchically structured: manuals, outlines, corporate organizations, family trees, directory structures, internet addressing, library cataloging, computer programs ...and the list goes on." It is no surprise that we see a lot research for how to use this hierarchical structure for different purposes.

Some of the approaches we already discussed in the previous chapter also have a strong hierarchical component to them. Continuum [André et al., 2007] uses the hierarchical nature of the data to group items together. The detail view still displays all items but uses boxes and colors to enable the user to visually focus on the relevant level first, i.e., first find the right era, then the right composer, then the piece.

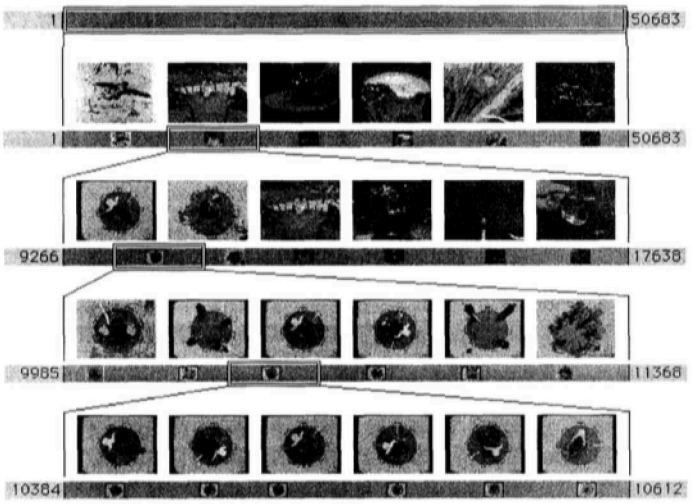


Figure 6. Applying the magnifier two more times.

Figure 3.9: User-created hierarchy of timeline sliders. Each level refines which range of frames from the level above is shown in more detail. From [Mills et al., 1992]

[Mills et al., 1992] rely on the user to create a hierarchy (see Figure 3.9). While the medium of video does have a hierarchical structure, consider a movie with chapters, scenes, and shots, this structure is difficult to find automatically. [Arman et al., 1994] used shots as their structure but do not consider the higher levels.

In all of these examples we see the trend to use hierarchies in the data that is being navigated. But even approaches that deal with complex networks are often doing this by using the specific form of the network or a secondary structure. Heer and Boyd [2005] designed a system to explore different aspects of a social network that is based on clustering the people in the network into 'communities'. These communities do not overlap, and thus, create a linear structure on top of the graph (see Figure 3.10). Repeatedly applying this approach within the sub-clusters could also create hierarchies. [Holten, 2006] visualize a complex network by showing a secondary hierarchy as a radial tree on the outside of a circle. The graph is then drawn on the inside of the circle connecting the individual nodes placed with in the secondary hierarchy (see Figure 3.11). Hive-

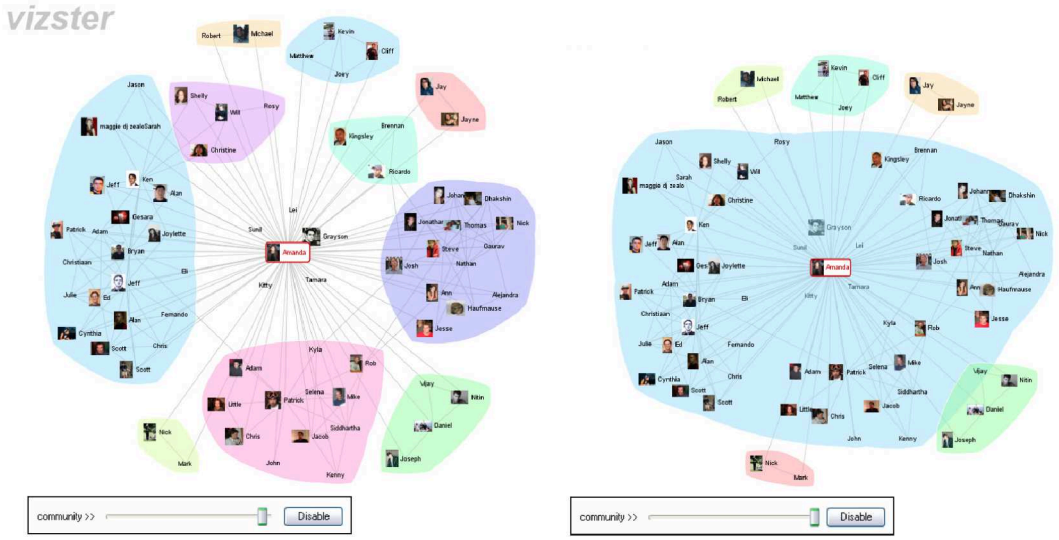


Figure 3.10: Vizster showing different clusterings of a social network centered around one person. The clusters do not overlap and thus create an unordered tree of the data; a social network is made up of clusters made up of people. From [Heer and Boyd, 2005]

Plots [Krzywinski et al., 2012] use certain node properties to place nodes on three distinct axes arranged in a circle. These properties can be determined by the user or automatically, based on graph properties such as a nodes rank or clustering coefficient. Again, the assignment of a node to an axis and then to a location on the axis creates a hierarchy.

While we cannot completely ignore complex network structures, we see that there is a strong tendency to use hierarchical information for visualizations. We create a specialized interface for tree structures in chapter 4—"Chronicler: Navigation in Source Code Histories". For the rest of this chapter, we continue with complex networks because the navigation properties we derive from structure are independent of its shape.

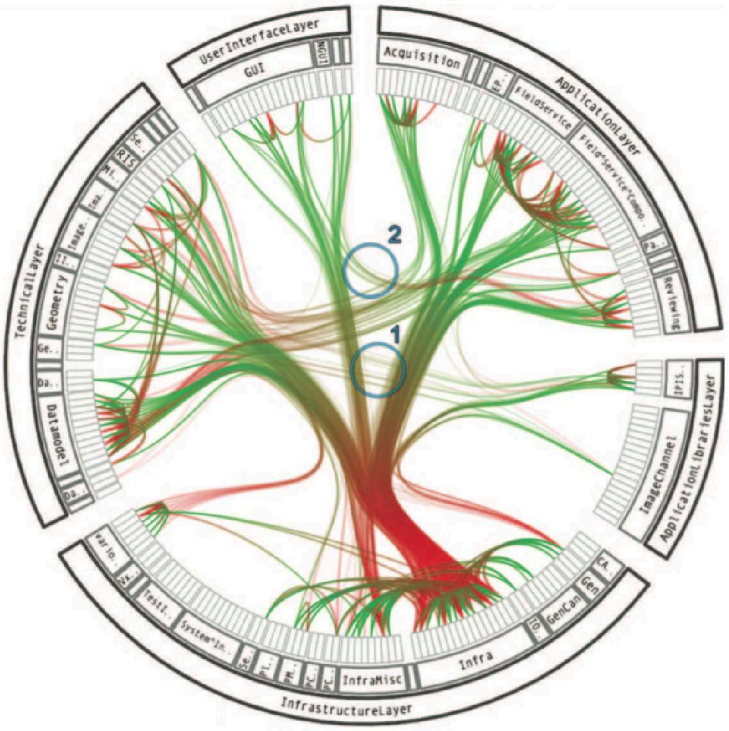


Figure 3.11: Visualization of a large call graph using Hierarchical Edge Bundles. A radial tree visualization encodes the location of a caller, while the graph in the middle encodes call graph connections. Direction is encoded by color (caller = red, callee = green). From [Holten, 2006]

3.5 History Graph

We defined the history of a document to be a sequence of individual versions of this document where most versions had small changes. Adding the concept of structure to this means that we now have structure graph for each version of the history. A *history graph* connects all these individual structure graphs.

Evolution of a document implies small punctual changes. This means that not only the whole document, but also individual content elements change from one version to the next. Whenever the content represented by a structural el-

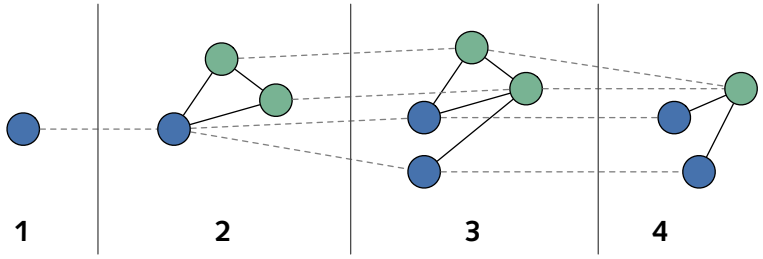


Figure 3.12: History Graph with four versions. 1: A single blue node is created. 2: The original blue node stays the same and two new green nodes are inserted and connected to the blue node. 3: The green nodes stay the same and the blue node is duplicated. 4: The two green nodes merge into a single green node.

ement changes, we want to connect the two versions of the structural element in either version. The graph resulting from this is what we call a *history graph*. These changes are also represented in the history meaning that each structural node has predecessors and successors in the (temporally) adjacent structures. Figure 3.12 shows a history graph for a small network.

Nodes do not necessarily have only one predecessor and successor. Duplicating content from one element to another creates a *split* (Figure 3.12-3), i.e. two or more successors, because both successors evolved from one node. We can also take content from multiple nodes and move it into a single node which creates a *merge* (Figure 3.12-4), i.e, two or more predecessors. These can also occur at the same time, e.g., if part of the content is copied to an existing node.

3.6 Structural Navigation Properties

Interfaces that are based on a history can use the knowledge about elements and their relationship to provide the user with context when navigating that history. We define the following features that are enabled by employing structure for temporal navigation.

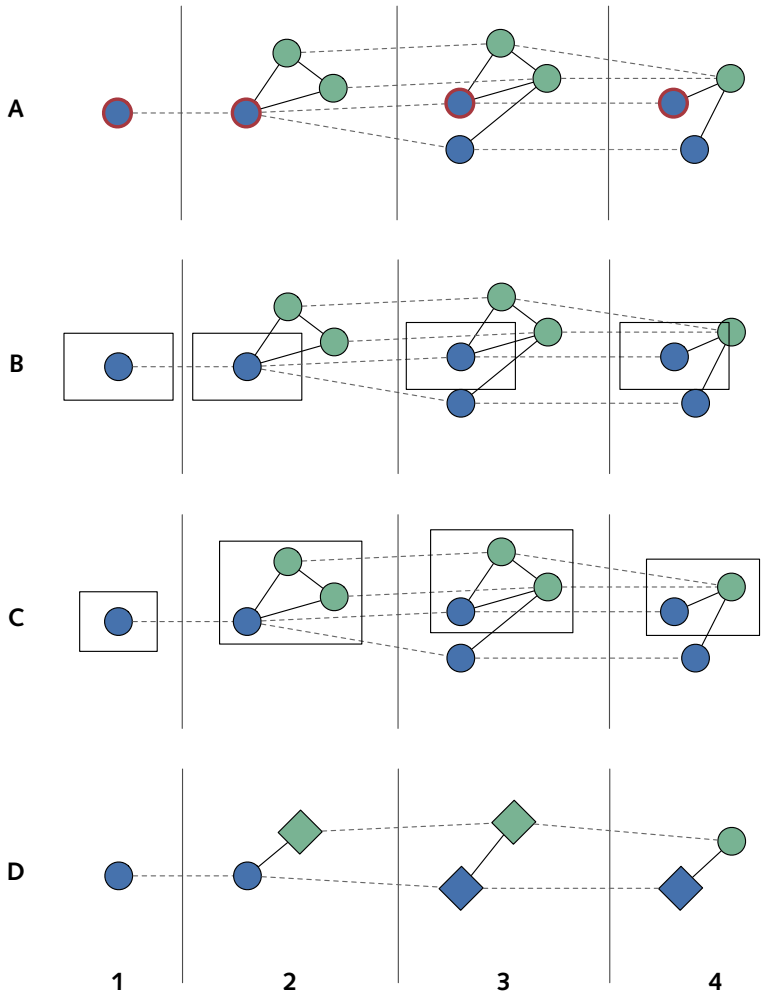


Figure 3.13: Structural navigation properties of a history graph. A—Structural Propagation: we can pass on properties such as selection to other versions of a node, here selection of the top blue node. During navigation we can keep it the same, helping the user to orient herself. B—Viewport Stability: we can make sure that when switching between versions a node will always have the exact same position in the viewport. Again, we chose the top blue node. C—Context Awareness: we can focus on a defined environment, here direct neighbors of the blue node. D—Structural Abstraction: we can find a representative for groups of structural elements and thus simplify the graph, here multiple elements of the same color are abstracted.

1. *Structural propagation* (Figure 3.13-A) allows us to forward interaction from one version to another. A simple example of this can be a selection made in one version that occurs when we navigate to the next one. More complex examples are annotations occurring at all versions of a node.

2. Knowledge about the history of sub structures directly enables us to provide *viewport stability* during navigation (Figure 3.13-B). If we know what structural element a user is currently focusing on, either implicitly through the current viewport, or explicitly through selection, we can keep that structural element visible at all times.

3. Often, not only the object of attention but also its context changes. Sometimes, we may want to focus on a certain part of a structure and ignore changes in surrounding elements. An interface aiming at providing *structural filtering* (Figure 3.13-C) can show only a selected object and a well defined context. This can be achieved through viewport manipulations like zooming and hiding other parts of the structure.

4. We can use *structural abstraction* (Figure 3.13-D) to reduce the amount of structural elements that the user needs to consider. This works through finding or creating a representative for a group of structural elements. In our figure, we use a content property, i.e., the node color, but we will later also use structural properties, i.e., representing subtrees through their root elements. Structural Abstraction helps when we cannot or do not want to visualize the evolution of individual structural elements, e.g., because we want to avoid clutter. We can independently also hide the respective content elements if required.

Choosing between different history paths is dependent on the implementation.

Note that the paths through the history graph chosen in Figure 3.13 always follow the top blue node. Which blue node is chosen is dependent on the construction of such a *history path*. One option is to build the path outward, from the starting node in both directions by following the predecessor and successor relationships. If we select any blue

node from version 3, the resulting path is unique. Choosing the blue node from version 1 or 2, requires us to make a decision at the split. How to construct the right path depends on the particular usecase and implementation. If we want to always follow the evolution of the structural element linearly through its history we must decide for one path at some point.

3.6.1 Usage in Existing Interfaces

There are application as well as research projects that already use one ore more of these features. In the following we describe some of the examples we found in multi-track media editors, version control systems, or research papers. Multi-track media editors like Maya[3], Adobe Premiere[4], or Motion[5] arrange content in tracks on a timeline; tracks are often arranged in a tree.

Structural propagation helps to keep track of object locations.

Structural propagation is available in all of the multi-track editors mentioned above. Figure 3.14 shows Motion. When navigating to a different time, the selection is kept, and when a group of objects is selected, the visible selection is dynamically extended when a new group object appears on the timeline. Ginosar et al. [2013] created an editor for authoring multi-stage source code examples. Changes in one line of one version can be propagate to the other stages of the source code example.

Structural filtering is used to hide unwanted content.

Structural filtering is also common in multi-track media editors. We usually can hide any element or group in the structure using checkboxes (Figure 3.14). Because of the static nature of the structure, i.e., the same tree for all versions, this is essentially the same as structural filtering without considering time, e.g., hiding layers and layer groups in Photoshop. Maruyama et al. [2012] filter events from the change history based on their relation to a structural element, e.g. only changes within a certain method. All other elements in the history and the code view are

[3] http://www.autodesk.com/maya
[4] http://www.adobe.com/premiere
[5] http://www.apple.com/motion

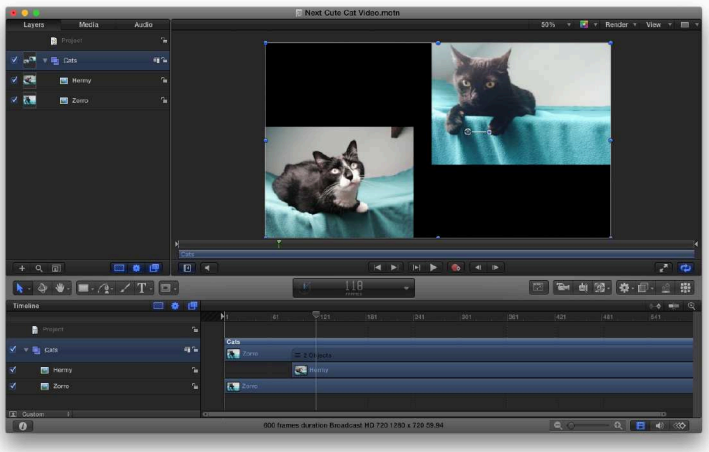


Figure 3.14: Motion. The selection around both images is based on the selection of the structured timeline at the bottom. When navigating to a time where one of the two images is not visible, e.g, frame 60, the selection updates to only the visible image. The arrow in the bottom left allow structural abstraction by hiding individual tracks in a group and and the check boxes enable structural filtering by hiding content of unchecked tracks at all times.

grayed out (see Figure 3.15). Some commercial version control tools, e.g., SourceTree[6], can filter the history of changes based on changes affecting a specific file. In SourceTree, all other changes are hidden, while GitX-dev[7] highlights the changes affecting a file. SourceTree also employs a filtering interface where the user selects a file or folder from any version, while GitX-dev uses search.

There are not many examples of viewport stability or structural abstraction

We could only find a few navigation examples of viewport stability and structural abstraction. Time Machine in OS X Yosemite uses it when looking at old versions of directories and always scrolls the current version of a selected file into the viewport. We also see it used as a means to show tracked regions in video trackers, e.g., [Jepson et al., 2003]. Most other tools we found use it only as a feature for outputs, e.g., the virtual camera in Blender can follow an ob-

[6] http://atlassian.com/sourcetree
[7] http://rowanj.github.io/gitx/

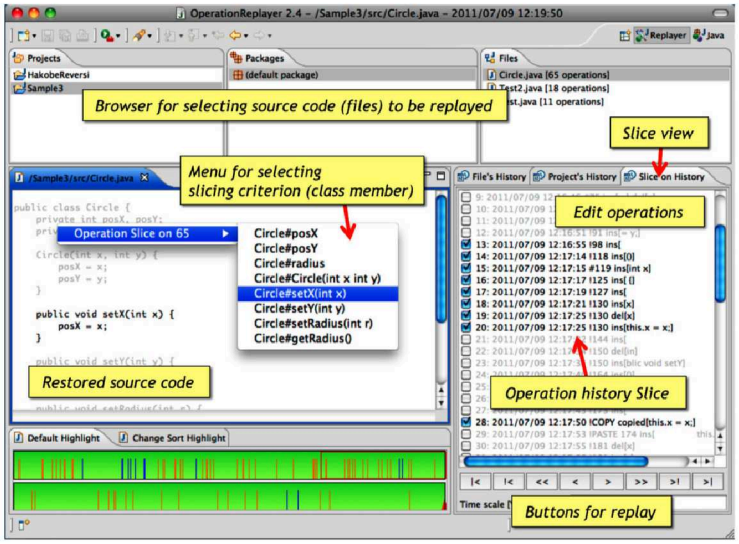


Figure 3.15: Operation slice of the method setX, all code and history not belonging to the method is grayed out. From [Maruyama et al., 2012]

```
public static String getString(String key) {
    try {
        return fgResourceBundle.getString(key);
    } catch (MissingResourceException e) {
        return "!" + key + "!";//$NON-NLS-2$ //$NON-NLS-1$
    }
}
```

```
public static String getString(String key) {
```

Figure 3.16: Code Folding in Eclipse. Clicking the – button hides code that a user is currently not interested in and represents it as the method name.

ject, keeping it in the viewport of the rendered output as it moves. We see structural abstraction in multi-track media editors where the user can represent all group members by the group itself (Figure 3.14). This has no impact on how the content is displayed, but only simplifies the timeline.

We see structural abstraction for content without the consideration of time in programming environments like Eclipse, where we can fold code blocks in on themselves and only represent them by a summary line, e.g. the method (Figure 3.16).

All tools described here circumvent the problem of splitting and merging in some way. The multi-track editors

Splitting and merging is not handled in the tools described here.

do not allow splitting and merging in their structure and all other systems described here select all possible path. They can do so because they either only highlight paths (GitX-dev) or accept jumps between the alternative versions, sorted by change date, when navigating the history (SourceTree).

3.7 Summary

We discussed ExamPen, an introduction of how structure can create information scent for time-based navigation. Afterwards, we introduced the notion of a structure as a graph representing relationships between content entities. Finally, we discussed how a navigation interface may benefit from knowledge of this structure over time, and we defined four navigation properties which are enabled by considering structure in temporal navigation.

Now that we described the existing structures in media, we can create interfaces based on these structures. We first introduce a tree based interface for structural navigation in source code histories. In chapter 5—"Generalizing Tree Flow Interfaces", we then show how that approach can be generalized for other media and structure types.

Chapter 4

Chronicler: Navigation in Source Code Histories

"History doesn't repeat itself, but it does rhyme."

—Mark Twain

The previous chapter defined the notion of a structural navigation interface and their advantages. In this chapter, we show that we can use this to implement Chronicler, a history navigation interface for source code. We start with an overview of related work in the area of how document histories have been used in the past. We then go on to explain why we picked source code as an example domain and finally introduce our prototype system Chronicler.

4.1 Related Work

History digests summarize history as a whole. History guides help to find a specific version.

We look at research around the use of history in two ways. First, tools that provide a *digest* of the history as a whole and does not allow the developer to find to specific version. Second, tools that *guide* the user to find a specific version. There are few pure digest systems because many visualizations for histories are designed over time and can thus easily be used for navigation.

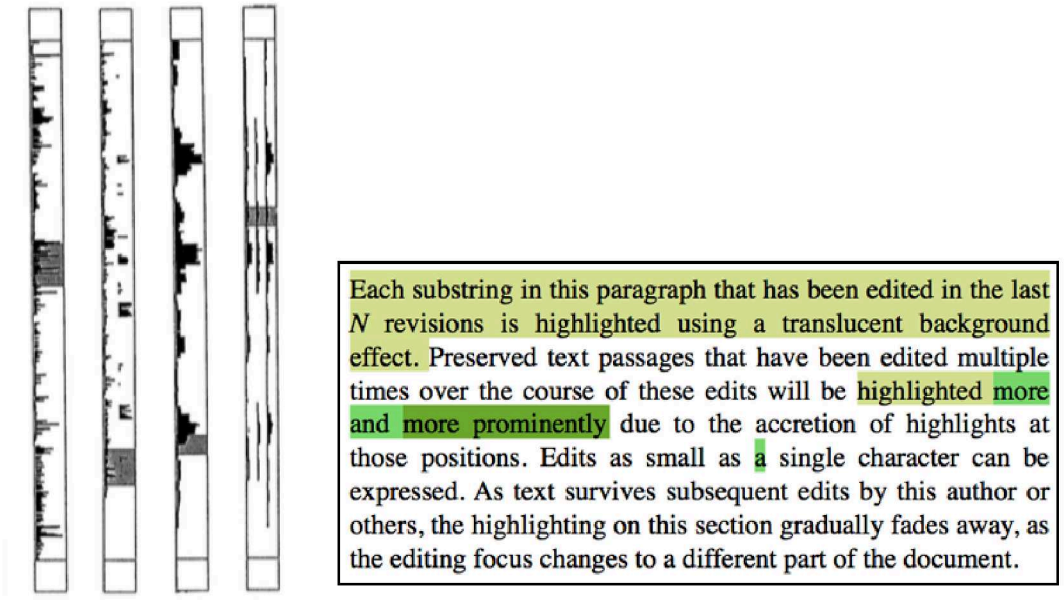


Figure 4.1: Examples of Edit Wear and Read Wear. Left (from [Hill et al., 1992]): the first two scrollbars are annotated with edit wear showing the frequency of changes, while the two scrollbars on the right show read wear. Right (from [Shannon et al., 2010]): in-line edit wear showing the age of edits as a colored highlight of the text.

4.1.1 Digests

Computational wear changes the appearance of a document over time.

The most common form of digests provides the user with a summary of certain aspects of the document, e.g., the quantity of changes of a certain line of code. One highly influential work in the area of history digests is Edit Wear and Read Wear [Hill et al., 1992]. They introduce the idea of "computational wear" where repeatedly editing or displaying sections of a document slowly make them more visually pronounced. Figure 4.1 shows their example of a scrollbar that displays how often a line in a text document was edited or read. Edit Wear can also be integrated directly into the document. Shannon et al. [2010] highlight recently changed parts of a text in-line.

Another seminal work describing a history digest for source code is Seesoft [Eick et al., 1992] (see Figure 4.2). Seesoft is a larger system for adding visual information to source code. It displays the content of multiple files

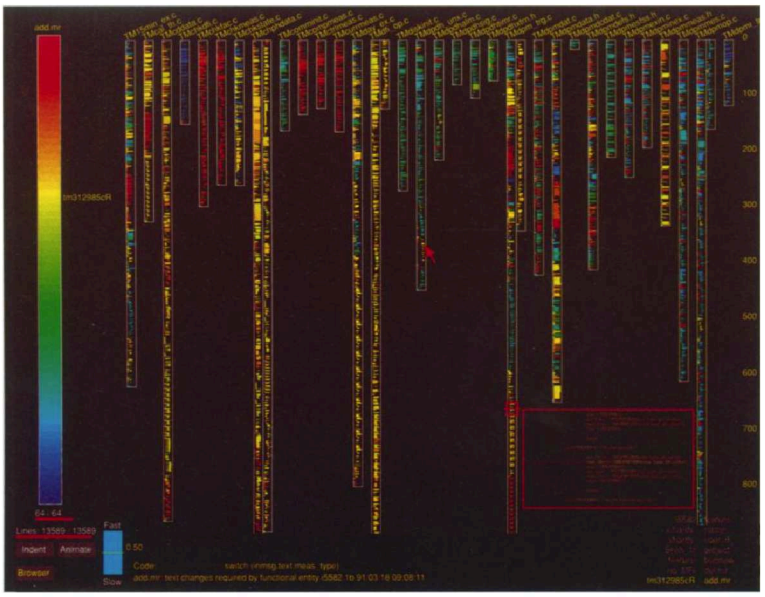


Figure 4.2: Seesoft visualization of source code age. The color indicates the age of a piece of source code with the bar on the left acting as a legend. From [Eick, 1994].

shrunken down to fit on to the screen. The system can then add information to the lines of the displayed files through a number of proposed metrics, most notably change frequency and age. Eick [1994] later extended his work with other text document types and new functionality like the ability to highlight files and changes that occurred together with a selected change.

Prescriptive Digests inform a developer's next steps.

Digests are also used prescriptively where information from the history informs the user's next development task. Zimmermann et al. [2003] use history to analyze the decisions that have been made in regards to system architecture to inform necessary restructurings. Ying et al. [2004] collect information about what files were changed together in the past. In turn, they can use this to identify other files that may have to be considered for an upcoming change. Kagdi et al. [2008] built a similar system that supports developers in finding someone who can help them understand how to make a specific change. According to [LaToza et al., 2006], this role of answering questions about unknown code and

pointing out responsibilities is traditionally filled by the *team historian*.

4.1.2 Time-based Visualizations

Visualizations over time can trivially be used to annotate a timeline.

There are a lot of tools that are designed to visualize changes over time. While they are often meant to be digests, they could also easily be used as a guide by using the visualization to annotate a timeline.

Viégas et al. [2006] extract keywords from email conversations and use these as a summary of a conversation history. The chosen keywords are arranged over time, which means that it is difficult to follow a keyword's age because the user has to look for the same keyword repeatedly. Highlighting keywords by search allows users to see keywords occurring at multiple times (see Figure 4.3).

Several researchers built storyline or flow graphs based on Theme River [Havre et al., 2002] (see Figure 4.4) to show the development of certain aspects of code. Flow graphs are stacked bar graphs where similar bars of two adjacent stacks are bridged using smooth curves. This smooth visualization of a bar for all stacks is called a *stream*. Flow layouts are particularly suited for history visualization because we often see gradual changes in individual stacks over time. Software evolution storylines [Ogawa and Ma, 2010] visualize which developers actively took part in development over time and who is responsible for what part of the code. They created a force-directed graph layout designed to look like a well-known movie storyline visualization[1].

Rose et al. [2009] introduce a technique to show the evolution of topics in news. They explicitly consider that these topics impact each other and visualize splitting and merging of topics. Cui et al. [2011] built on Rose's work, and, besides creating a more organic visualization (see Figure 4.5), they also introduce interaction techniques that help a user to filter and highlight topics of interest. An initial dense

[1] http://xkcd.com/657/

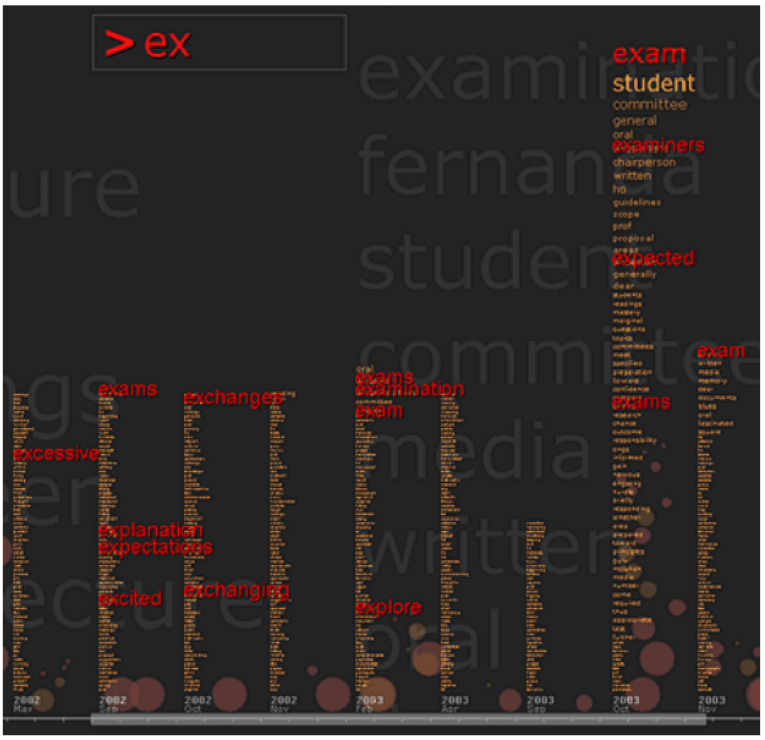


Figure 4.3: History of email conversations visualized as stacks of keywords. The content of the search field at the top is highlighted to identify recurring keywords. From Viégas et al. [2006].

stacked graph can be pulled apart to highlight a certain topic and its connections to neighboring topics. They also allows the user to add threads, lines on top of the flow visualization, emphasizing selected keywords within a topic.

Streams may also be ordered.

So far, the order of streams has been arbitrary; the data these visualizations were based on, i.e., topics in the news, have no real order. We also see flow visualization that use structural information to inform the order of streams. Viégas et al. [2004] created History Flow, a visualization that reveals patterns of collaborative and antagonistic behavior in wiki changes (see Figure 4.6. History Flow represents the absolute position and extent of paragraphs and lines on a Wikipedia page; at each point in time, the line that is at the top of the document, will also be at the top of

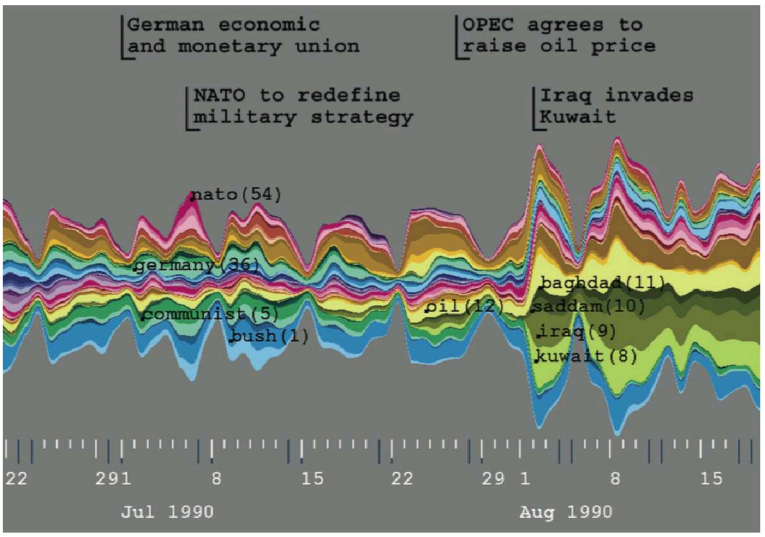


Figure 4.4: An example of the Theme River visualization. Here, we see news events from July 1990. It is easy to identify the age of a topic and the amount of news activity at a certain point in time. From [Havre et al., 2002]

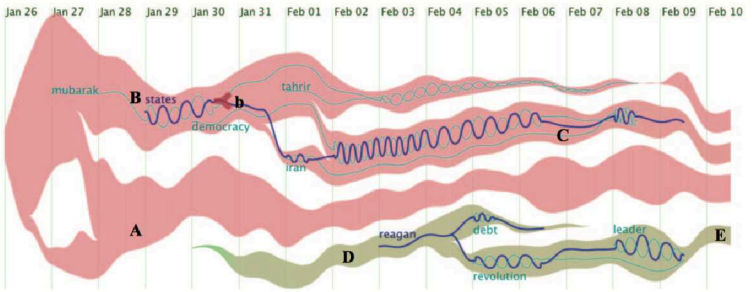


Figure 4.5: Text Flow visualization of news data from early 2011 with topics "budget / spending" (A), "protest in Egypt" (B and C), "republican" (D), and "campaign" (E). Events have been separated to easily see the relationship of topics through splits and merges. The threads on top of the flow graph, highlight user-selected keywords and their relationships. From [Cui et al., 2011]

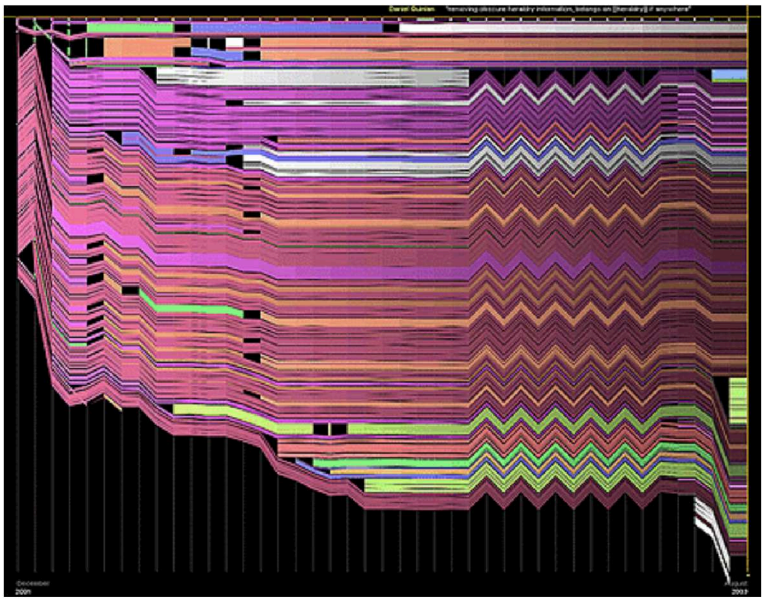


Figure 4.6: History Flow of the history of the *Chocolate* Wikipedia page. It is easy to see the quick growth of the document at the beginning and an *edit war* where different authors switched between two variants of the page. From [Viégas et al., 2004]

the visualization. Figure 4.7 shows CodeFlows [Chevalier et al., 2007, Telea and Auber, 2008], a visualization designed to reveal structural source code changes. It uses the tree structure of the syntax tree source to create a flow visualization of changes between several versions. They match the syntax trees of individual versions based on structural similarity, and, for each line, show the evolution of the largest matched tree node.

4.1.3 Guides

Guides often rely on annotated timelines as an overview.

We also see systems that are specifically designed to guide the user to find a specific version. They typically also have an annotated timeline component that serves as an overview. A system focused on guiding the user to find a specific time is Chronoviz [Weibel et al., 2011, 2012]. They use the co-evolution of multiple information sources to al-

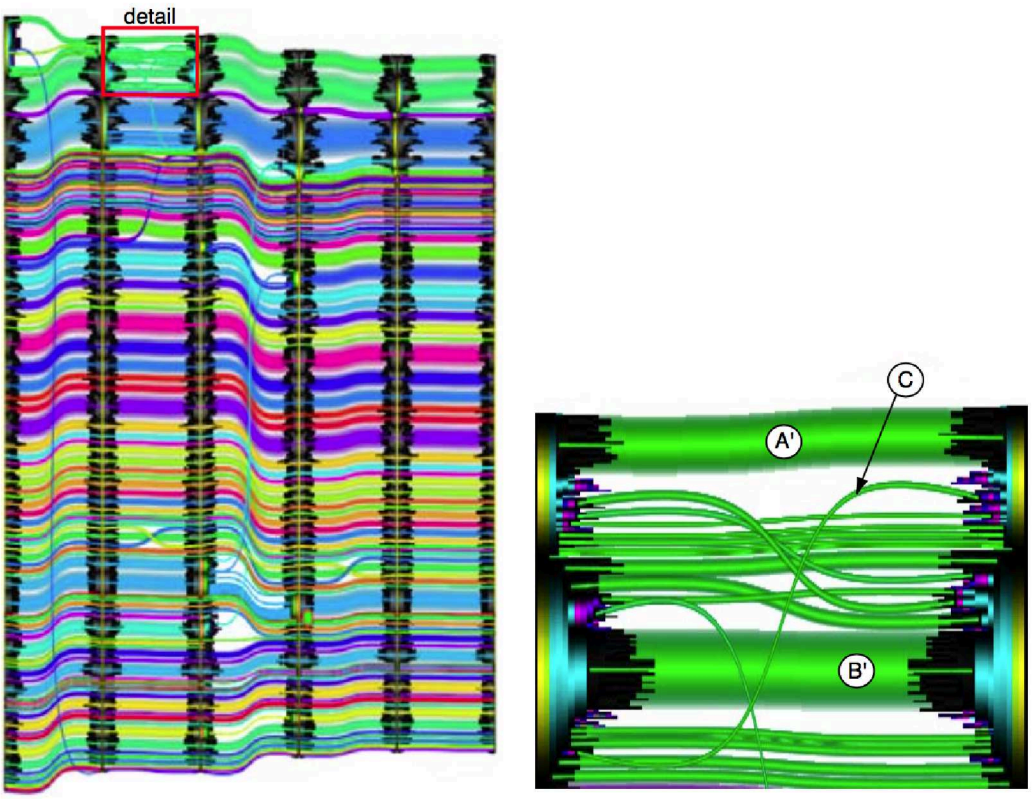


Figure 4.7: The Code Flows visualization of six similar versions of a C++ file. It is easy to see how large parts of the code stay stable over all versions, and which part of the code changes in each version. From [Telea and Auber, 2008]

low the user to rediscover a relationship between events from different sources. Figure 4.8 shows their interface collecting data from a flight simulator; the interface allows the user to understand what was going on in the other domains while she wrote a note on paper. Interestingly, notetakers experienced with this system start to rely on the connection and spend less effort on writing annotations like times or verbose explanations.

Servant and Jones [2013] designed Chronos, a system that enables to quickly navigate the history of a text document, especially source code. They arranged all versions of the documents on a plane and align multiple versions over time so that horizontal scrolling always shows the same part of the text, e.g., a method. Azurite [Yoon et al., 2013] visualizes change events by annotating a timeline with mark-

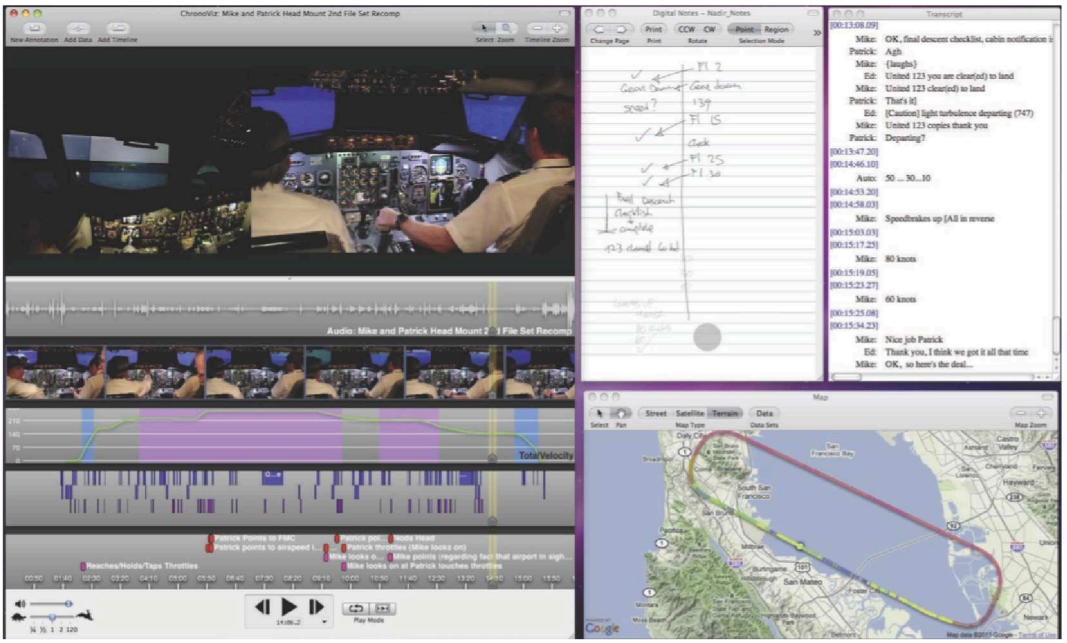


Figure 4.8: The Chronoviz interface for a simulated flight session. On the left, we see multiple timelines with various annotations like audio or velocity. The views on the right show two-dimensional data, like text or a map location. Semi transparent text in the notes view indicates that it had not been written at the selected time. From [Weibel et al., 2012]

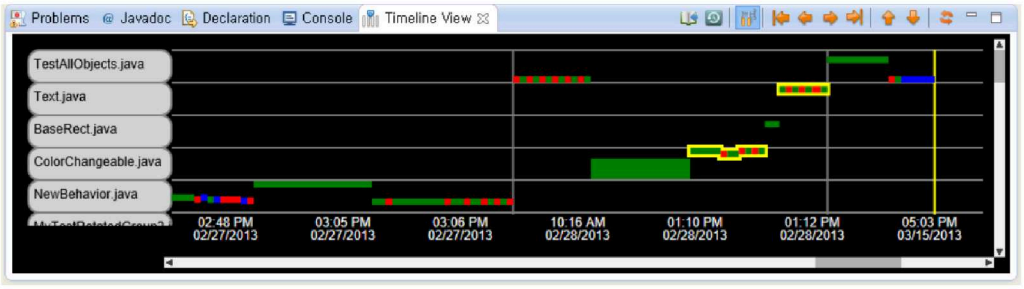


Figure 4.9: Azurite's timeline navigation displays which files were recently changed and where in the file the change occurred relative to the length of the file. From [Yoon et al., 2013]

ers for insertion, deletion, and modifications (Figure 4.9). They also propose to use this to selectively undo changes from the history by applying inverse changes to the current version.

DMVN systems are guides that do not consider time.

DMVN systems as discussed earlier are guides without a temporal visualization. They instead simply help to arrange a certain state of the video content. We see this as a disadvantage of such systems because it limits what times in a video are accessible. Even DragLocks does not consider time a primary navigation concern, although it is not as restrictive as the other systems. However, the duration of movements or pauses in DragLocks are not considered in these systems.

Our goal is to create a guide that uses structure as information scent.

Instead, we aim at creating a system that annotates a timeline with structural changes that can serve as information scent. If that visualization does not express the users goals, it is still possible to use it as a normal timeline; the worst case is a fallback to an unannotated system. This is a parallel approach to adding more specialized navigation methods on top that the user can choose for specific tasks.

4.2 Why Source Code?

We use source code as an example for two reasons. It is a medium where history exploration is a well-known usecase and it has a rich syntactic structure.

4.2.1 Usecase

The tasks around source code have been studied by LaToza and Myers [2010]; they interviewed developers to understand the questions developers have when dealing with source code. We reiterate the questions they describe in the area of history, how different systems support solving these questions, and if and how we want to handle these questions.

- "When, how, by whom, and why was this code changed or inserted?" These questions are often tackled in today's history navigation interfaces of version control systems. Many interfaces cover only the last

changes affecting a line of code, e.g. Xcode's[2] blame functionality. Thus, developers have to manually search the version history of a file for these changes. Ogawa and Ma [2010] propose to answer the questions of who was responsible for a change by visualizing the responsibilities of different authors' over time. Our goal is to easily allow developers to navigate back through the history of code. We want to to so by providing structural hints as to the nature of those changes. For now, we leave out the question of how and by whom the code was changed.

- "What else changed when this code was changed or inserted? How has it changed over time?" We see this tackled in research prototypes that attempt to automatically predict what other code to change [Ying et al., 2004, Zimmermann et al., 2004]. In his master's thesis, under the guidance of the author, Schulz [2014] proposes CodeShape, a guide that integrates sketches of the state of software into the timeline. These sketches can be used to understand architecture changes over time. A change that also affected the sketch may also indicate it had a bigger impact on the code than a simple one-line bugfix. We do not want to tackle this question directly, but allow the user to identify hints that correspond to these questions, e.g, by structural changes occurring together.

- "Has this code always been this way?" Static visualizations of the amount of changes in source code [Eick et al., 1992] don't help to navigate to a certain change but they do help to understand how stable a part of the code has been. Chronos [Servant and Jones, 2013] allows navigation of the full text of a file over multiple versions. This detailed representation makes it difficult to get an overview of what changed over time. Instead we want to use a flow visualization similar to [Telea and Auber, 2008] and adapt it to interactively switch between overviews and detailed visualizations.

- "What recent changes have been made?" This is to a small extent visible in existing version control sys-

[2] https://developer.apple.com/xcode/

tems. SourceTree[3], e.g., show the most recent list of changes on all branches in a table. The user can then look at the change summaries or the individual changes. It is also possible to show only changes related to a specific branch or changes to individual files. We also see this in DeepDiffs [Shannon et al., 2010], where recently changed lines get multicolored highlights to show their age. This again fits our core goal of highlighting changes in the code structure.

- "Have changes in another branch been integrated into this branch?" We also see this in the graph visualization of version control system, e.g. in Tower[4]. Versions of the two branches are connected when the changes are integrated. For now, we will leave out branching of into multiple versions of the same file. Our focus will be on duplicating (branching) and (integration) merging of content within a file. If a file is changed in multiple parallel branches, we expect the user would have to select the correct branch for their question using some other means. So for now we assume a history to be linear.

Another benefit of source code is that it has a very rigid structure because it has to be machine readable.

4.2.2 Source Code Structure

The syntax tree of source code is a rich hierarchical structure.

The obvious choice of structure is the syntax tree which is very easy to parse. Programming languages are generally based on context-free grammars; this makes it easy for the parser to understand the language but the rules can also be given to programmers as a recipe of how to create syntactically correct code. Source code files are easily parsed into abstract syntax trees that are a hierarchical representation of text based on that grammar; Figure 4.10 shows an example.

[3] https://www.sourcetreeapp.com
[4] http://www.git-tower.com

GRAMMAR OF A SWITCH STATEMENT

switch-statement → **switch** *expression* **{** *switch-cases*$_{opt}$ **}**
switch-cases → *switch-case* *switch-cases*$_{opt}$
switch-case → *case-label* *statements* | *default-label* *statements*
switch-case → *case-label* **;** | *default-label* **;**

case-label → **case** *case-item-list* **:**
case-item-list → *pattern* *guard-clause*$_{opt}$ | *pattern* *guard-clause*$_{opt}$ **,** *case-item-list*
default-label → **default** **:**

guard-clause → **where** *guard-expression*
guard-expression → *expression*

Figure 4.10: Parts of the grammar required for the switch statement in Swift. Following these *production rules* creates a valid statement. The completed statement forms a tree that has one expression subtree and an arbitrary number of switch-case subtrees. From [Apple, 2014]

We cannot consider our structure independent of the history aspect. It must be easy to match individual syntax trees representing a method to another syntax tree representing a changed version of this method. Chevalier et al. [2007] discovered that you can use the shape of the syntax tree to match two versions of a C++ file. Based on this they created the visualization shown in Figure 4.7. We want to create a similar visualization and then enable it with the structural navigation properties described earlier.

There are however problems with using the structure as the means to matching. Simply reorganizing a method, e.g., for error handling, can quickly create a structural difference that we would be unable to match. Adding new code also affects the structure, which leads to the same problem. While this may have been intended by Telea and Auber [2008] to focus on detailed changes in between two versions (see detail in Figure 4.7), it does not work for our purposes. To properly use any of our navigation properties, we need such additions and reorganizations to match well. E.g., when the methods in a class where refactored for a new error handling model, we still want the two versions of the class to match.

We decided to use a top-down approach for matching based on a bag-of-words model. For us, a word is any group of alphanumeric characters, i.e. variable names, lan-

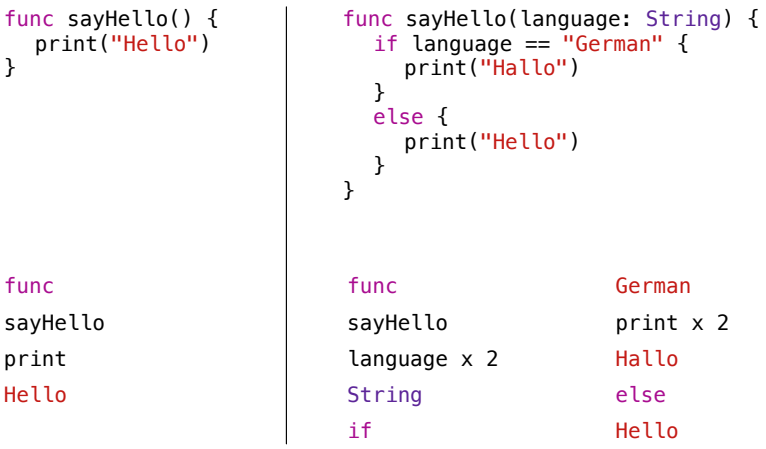


Figure 4.11: Two versions of a method in Swift and their respective word bags. The matching score between the two methods would be $\frac{2}{3}$

guage keywords, words in strings, or words in comments, and numbers. Any element of the syntax tree is annotated with all keywords 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. Then, the match score between two nodes is defined as

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

Figure 4.11 shows an example.

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.

This matching approach works well until we get to the level of individual lines. The amount of keywords in a line is often too small and nodes cannot be matched or are matched incorrectly. Thus, we decided to stop parsing at the level of code blocks, because they are easily understood by a developer and contain enough keywords to be reliably matched (2–3 already works well in our experience).

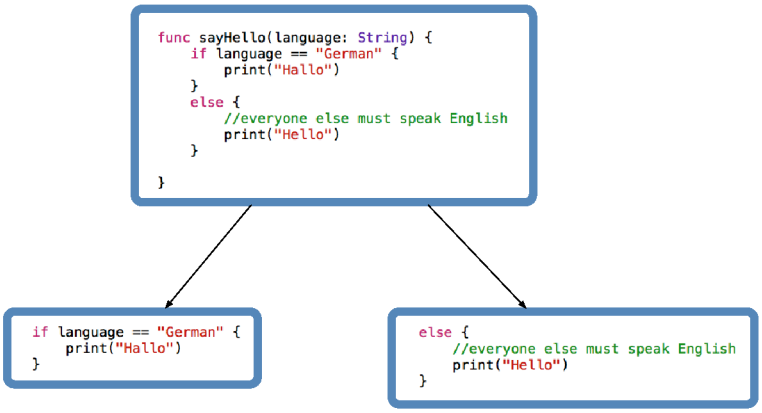


Figure 4.12: Simple code block tree of a short method. Including text before opening brackets makes sense because it represents the control flow responsible for a particular code block.

Since this work focuses on how to use the tree structure for interaction, we found this trade-off acceptable. In languages like Java or Swift, this structure entails, classes, methods, closures, control flow.

This structure is also easier to parse than the full syntax tree, which means we can greatly improve parsing speed by replacing the language parser, e.g., clang[5], with a trivial code block parser. The latter also allows us to become language agnostic and parse any language that uses curly braces to denote code blocks, e.g., C, C++, Objective-C, Swift, Java, or JavaScript. We discuss approaches to use a more detailed structure in the limitations section of this chapter.

So, our final structure is a tree, where each node represents some 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 example of such a code block tree can be seen in Figure 4.12

[5]http://clang.llvm.org

We can now describe how this structure can be employed for improving interaction techniques around source code history. We envision such a structural navigation technique to be used in conjunction with a view of the content, i.e., the source code.

4.3 Structure Aware Timeline

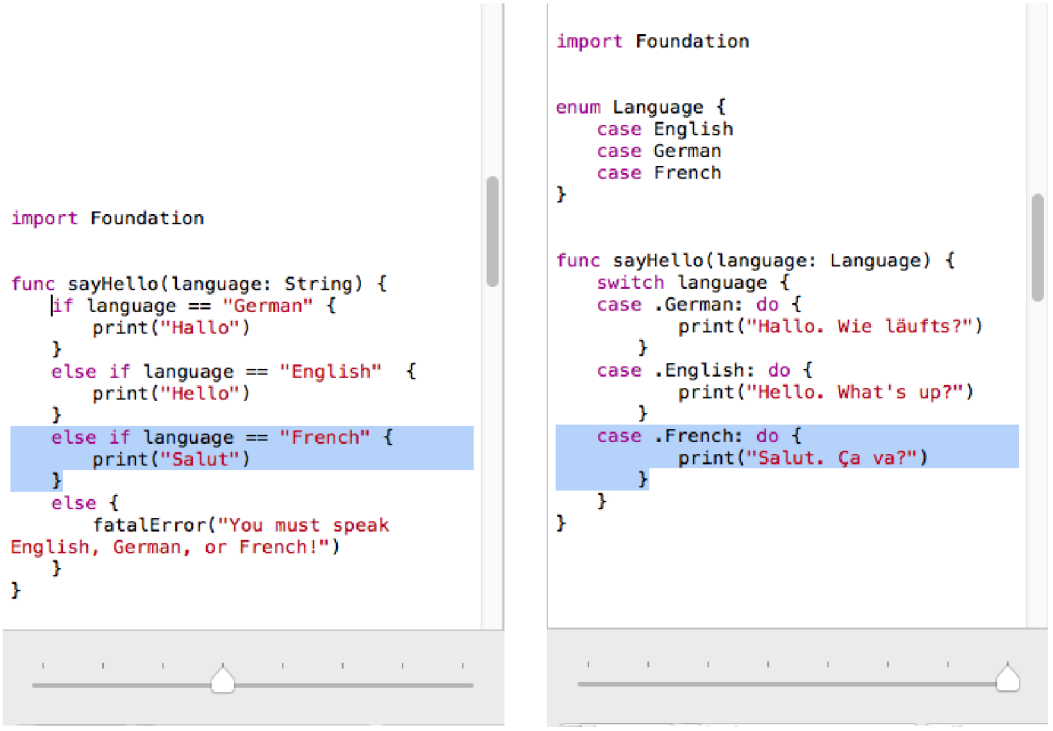
For some problems it might be unnecessary to provide the user with a complete visualization. Thus, we describe how to create a structure aware timeline using the features described in chapter 3.6. Such a structure aware slide could work for local navigation in a defined scope.

We created an interface that consists of a source code view and a timeline slider. The source code code view highlights the structure around the line containing the text cursor, i.e., the code block containing that line. The timeline, a simple slider, represents the versions of the file.

We *propagate* this highlight to every version of that code block when navigating through history. In this way, the user can easily identify the current version of the original code block.

When the user navigates to a different version by dragging the handle on the timeline, the source code view displays that version of the file and automatically scrolls the version of the originally selected code block to the same position. Enabling *viewport stability* like this enables the user to focus on the changes within her particular selection. It does not matter if the code block was moved from a different method or where it is in the file. Figure 4.13 shows this behavior. In conjunction with the highlight, the user can easily see if the length of the code block changes during navigation.

Using the timeline as an interface has the advantage that it is completely agnostic to the type of structure. Even if we had a underlying structure other than a tree, we could still employ the same method. However, since history paths



```
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 läufts?")
        }
    case .English: do {
            print("Hello. What's up?")
        }
    case .French: do {
            print("Salut. Ça va?")
        }
    }
}
```

Figure 4.13: Two version of a short piece of source code when navigating using the Structure Aware Timeline. Even though the if…else statement was replaced with a switch statement and a new enum declaration was added at the beginning of the file, the selected statement appears at the same location and is correctly highlighted.

can split and merge often, we have to decide where to continue without explicit user interaction. When we encounter a split, we compare the next node on all potential paths to follow and select the one with the best match score. Of course, the timeline has all usual disadvantages of timeline navigation. The user cannot tell what version to navigate to in order to see a particular change. It does not fulfill our desired goals from chapter 2.8. To deal with this problem, we now describe our *Tree Flow* visualization of structural changes.

4.4 Tree Flow

Our goal for tree flow was to create an interface that enables navigation or exploration without restricting which versions are accessible. We also wanted the user to be able to identify the relationship of multiple objects at once and even see objects from the history that are not visible in the current version. This is exactly what we can visualize using the hierarchical structure of source code. The relationships visualized right now are colocation and containment in a source code file.

Tree flow abstracts from the content and only displays structure.

We chose to visualize an abstraction of the content changes on the timeline because the amount of source code in a typical file does not fit into the code viewport. Any visualization on top would have the issue that it cannot be understood as a whole. One such inline visualization is Chronos [Servant and Jones, 2013]. Chronos arranges code on a large surface, that allows horizontal navigation through time and vertical navigation through the code version. Different versions of code arranged horizontally are aligned to each other to facilitate comparison. This alingnment can leave vertical gaps that users may overlook when navigating through horizontally. Schulz [2014], who did his Master's thesis under the guidance of the author, observed this issue during a user study comparing, Chronos, Azurite, and his own tool Code Shape.

The idea behind tree flow is to extend the code flows visualization in such a way that it can easily deal with hundreds of versions of a source file. We want to be able to provide an overview of the history of a whole file but also be able to focus on the history of individual classes, methods, or control structures to enable a developer to focus on code on the lower levels. LaToza and Myers [2010] and Holmes and Begel [2008] describe that developers are often interested in code on the "snippet level".

4.4.1 Visualization

We cannot simply copy the Code Flows visualization approach for several reasons.

- Since we want to display hundreds of versions, we do not have room for the icicle plot visualization of code lines in the middle of the visualization.
- For the same reason, we need a different curve rendering approach. This can easily be seen in the detail part of Figure 4.7: the bridges connecting the different versions sometimes overshoot vertically and attach to the connection points diagonally. If we want continuity of this curve between multiple versions, these overshoots will become even larger. We solve this by imposing a derivative of zero at the start and end of each of these bridges; this guarantees continuity between multiple curves in a row without creating vertical overshoots.
- A less trivial reason is that Code Flows generates an essentially flat visualization of the tree and assigns colors to tree elements based on common flow patterns and the type of syntax tree node that created it. This produces similar colors for similar elements, but it also means that we always see a mixture of different levels of the tree. We want to instead retain the tree's hierarchical nature to allow the user to select which level of detail is interesting to her at a given time.

We used Arctrees as a one-dimensional visualization of a tree.

For the latter point we looked at different tree visualizations; Schulz et al. [2011] present a comprehensive survey of such visualizations and Jürgensmann and Schulz [2010] give a visual overview. Tree visualizations are usually based on two-dimensional representations of the tree. However, Arctrees [Neumann et al., 2005] are a one-dimensional visualization of trees that also consider non-ordered tree graphs and also visualize the relationship of child nodes. For our purposes, we only consider a simplified version of Neumann's visualization that we call a *tree stack*. We can easily draw such a tree stack at any width

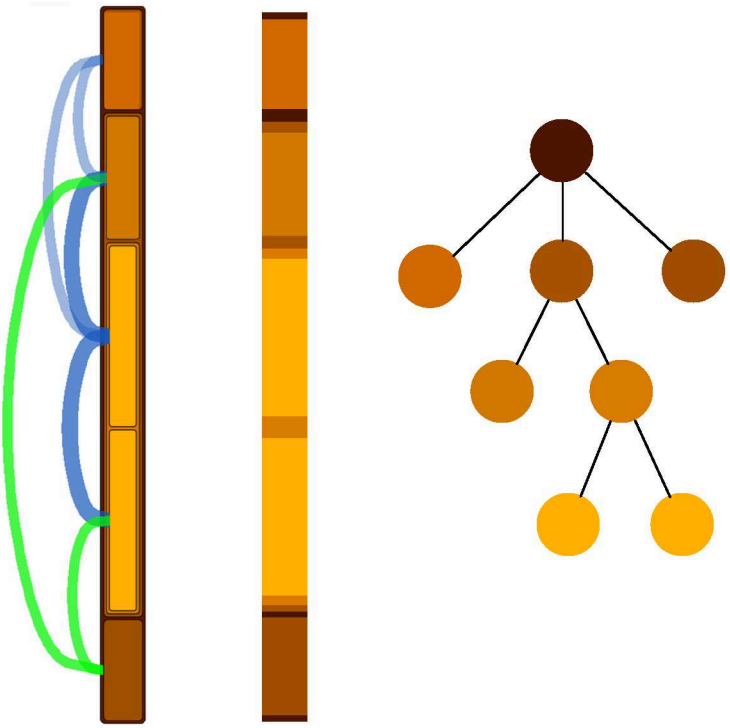


Figure 4.14: Arctrees (left). From [Neumann et al., 2005]) and Tree Stack (center) and the represented tree (right). The details of the Arctrees more clearly show tree levels and relationship between sibling nodes. Tree Stacks can easily be drawn at any width.

and, using predecessor and successor information from our matching, we can easily connect the two using the aforementioned bezier curves at the top and bottom of each tree level.

The code block tree can easily be represented as a tree stack.

Each version of the source code is parsed into code blocks as described above visualized as tree stack with zero width, and placed on the timeline; versions are distributed uniformly. The vertical extent of each tree node is determined by the number of lines represented by the node. We scale each stack based on the extent of the longest source code version, meaning that the longest represented version fills the whole vertical space available while shorter versions only fill proportionally less space. Every node in each

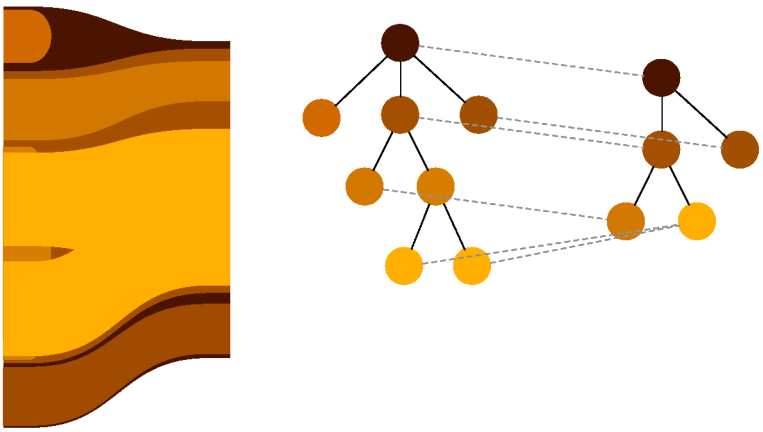


Figure 4.15: Tree flow connection of the two trees depicted on the right.

tree stack is connected to its neighboring versions by cubic bezier curves connecting start and end of each version. If a node does not have a neighboring version (insert or delete of the node), we create a rounded end of the visualization. The visualization in this way easily handles splits, merges, inserts and deletes. We specially handle merges and splits into or from parents. A split from a parent into a child is drawn as an insert. A merge from a child into its direct parent is drawn as a delete.

We then color the visualized history graph starting at the first version by assigning colors from a predefined color set to each first level node, i.e. the children of the root node. Each child is subsequently colored by taking its parent color and reducing the saturation by a constant factor. This approach provides a clear distinction between first level elements, retains visual similarity of child and parent elements, and still shows the evolution of individual child elements.

All versions of a code block will have the same color.

We follow each colored node's successor mapping to color the nodes best matching successor using the same color. Any uncolored node will receive a new color based on the rules above. This approach is different from Code Flows coloring scheme which removes structural information in the depth of the tree in favor of more clearly visualizing

merges and splits of child elements. The colors chosen for the visualization are also visible in the code view to enable the user to quickly understand the relationship between the two.

Finally we annotate content changes within the individual code block histories. If there is a content change between two versions, both versions of a particular code block are annotated with a vertical mark. This makes it easy to see if the content of a code block has stayed constant.

Chronicler is an implementation of tree flow connected with a source code view.

We implemented this tree flow visualization in a prototype system called Chronicler. Figure 4.16 shows the history of the examples discussed in Figures 4.11 (versions 1 and 2) and **??** (versions 4 and 8). Figure 4.17 show a longer history from a file in a public GitHub repository[6]. We now continue to describe how the user can interact with the system.

4.4.2 Interaction

We use the tree flow visualization as an annotated timeline. Thus, when the user starts drags, we switch the code view to the version under the mouse pointer.

During navigation, we retain the current selction and position of the selected code block.

We use *structural propagation* and *viewport stability* as described for the structure aware timeline. The code block selected in the source code view will be highlighted and kept stable during navigation. It will also be highlighted in the tree flow visualization. The selection can also be made in the visualization; when clicking on the history path of a code block, the current version of code block, if it exists, will be scrolled into view. This makes it easy to explore the different aspects of the visualization.

The user can change the depth of the visualized tree.

We implement *structural abstraction* by allowing the user to dynamically change the depth of the tree that is visualized. This means that they can focus on the depth of the tree that is currently of interest. For example, consider a file containing multiple classes. It may be interesting to see when these classes were inserted or how they grew over time, the

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

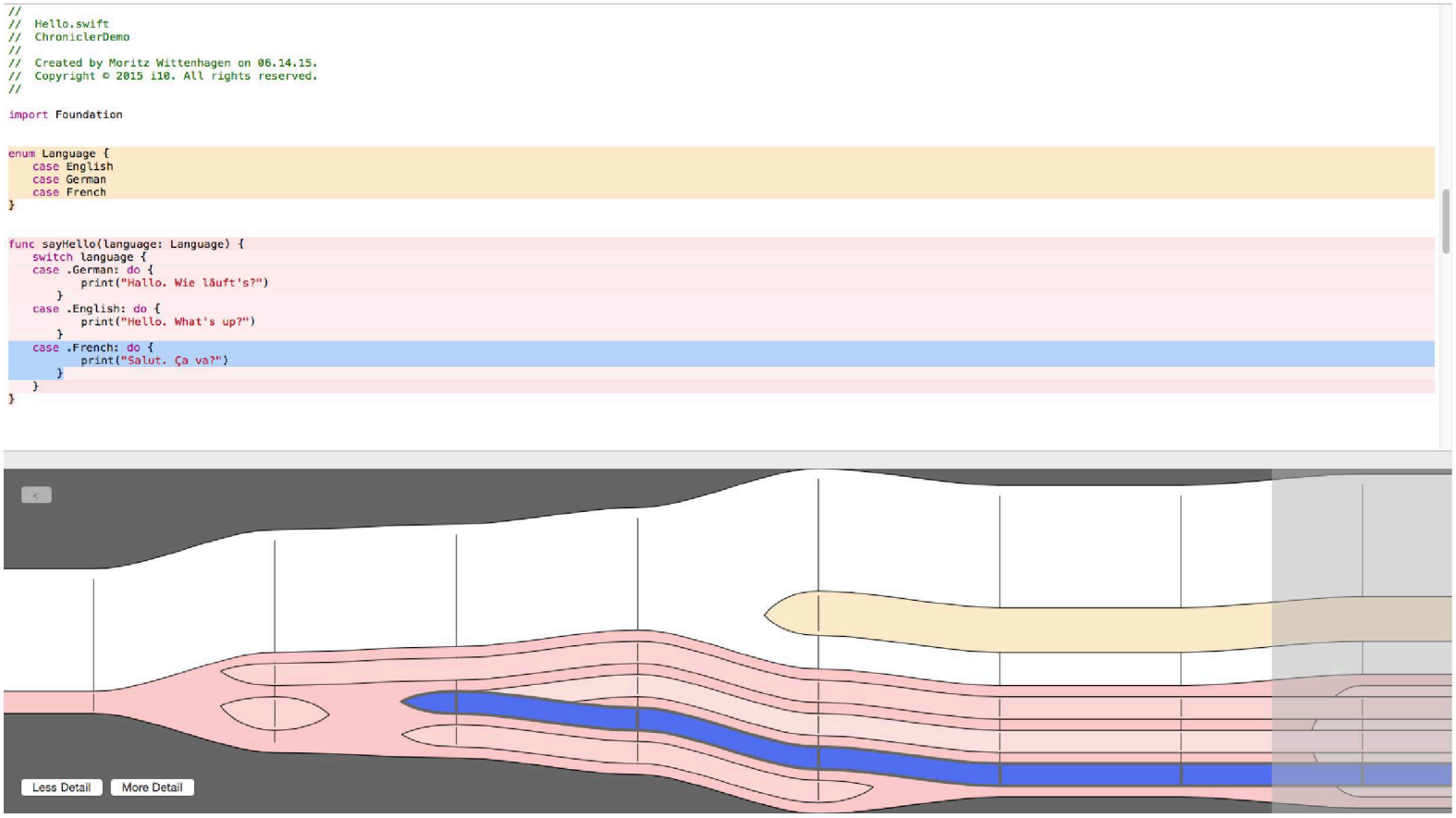


Figure 4.16: User interface of Chronicler for a short file with a short history. We see the source code view at the top and the tree flow visualization at the bottom. The gray bar in the bottom right indicates the current version. We can easily see how long the individual blocks have been around, e.g, the switch statement was only recently added, and the initial version contained only a short version of the sayHello function (the white block are the comments at the beginning of the file)

user in this case does not care about individual methods and does not show these details. Later, it may interesting to see why a class grew, showing more details about methods, enabling the user to see if methods were inserted or maybe moved from another class. This does not impact structural propagation, which means that the selection in the code view might be overlaid on top of the visualization.

The user can exclusively focus on the history of selected history path.

We use *structural filtering* to focus the visualization on a particular sub-history. By double clicking on a history path, the user can select a history path to visualize in more detail. Starting from a code block in Figure 4.17, we would end up with something similar to Figure 4.16. The available space is then only filled by the history of the selected subtree as defined by its history path. We gray out all other content to make sure the user knows what the visualization shows at a point in time. In this way a user can descend into the tree to reveal more details about classes, methods, or individual code blocks.

4.5 Experiment

We compared Chronicler and two timeline variants.

We conducted a study, comparing Chronicler, the structure aware timeline, and a normal timeline. The normal timeline represents existing user interfaces such as Xcode's version editor, or Tower's file history view. The study was separated in three parts. First, after showing participants the three tools, we let user discuss problem solving strategies for four different problems. This provided insights into how users approached history navigation with each of these tools. The qualitative results from this part indicate that users devise very different strategies when using the slider-based tools than when using Chronicler. The first part also familiarized our participants with each of the conditions. Second, participants were given eight concrete navigation tasks with one of the three conditions. Third, users were asked to bring a repository with familiar source code; they could then explore it with all of the three tools. This part of the study was optional, but was designed to provide insights into how history navigation tools could be used.

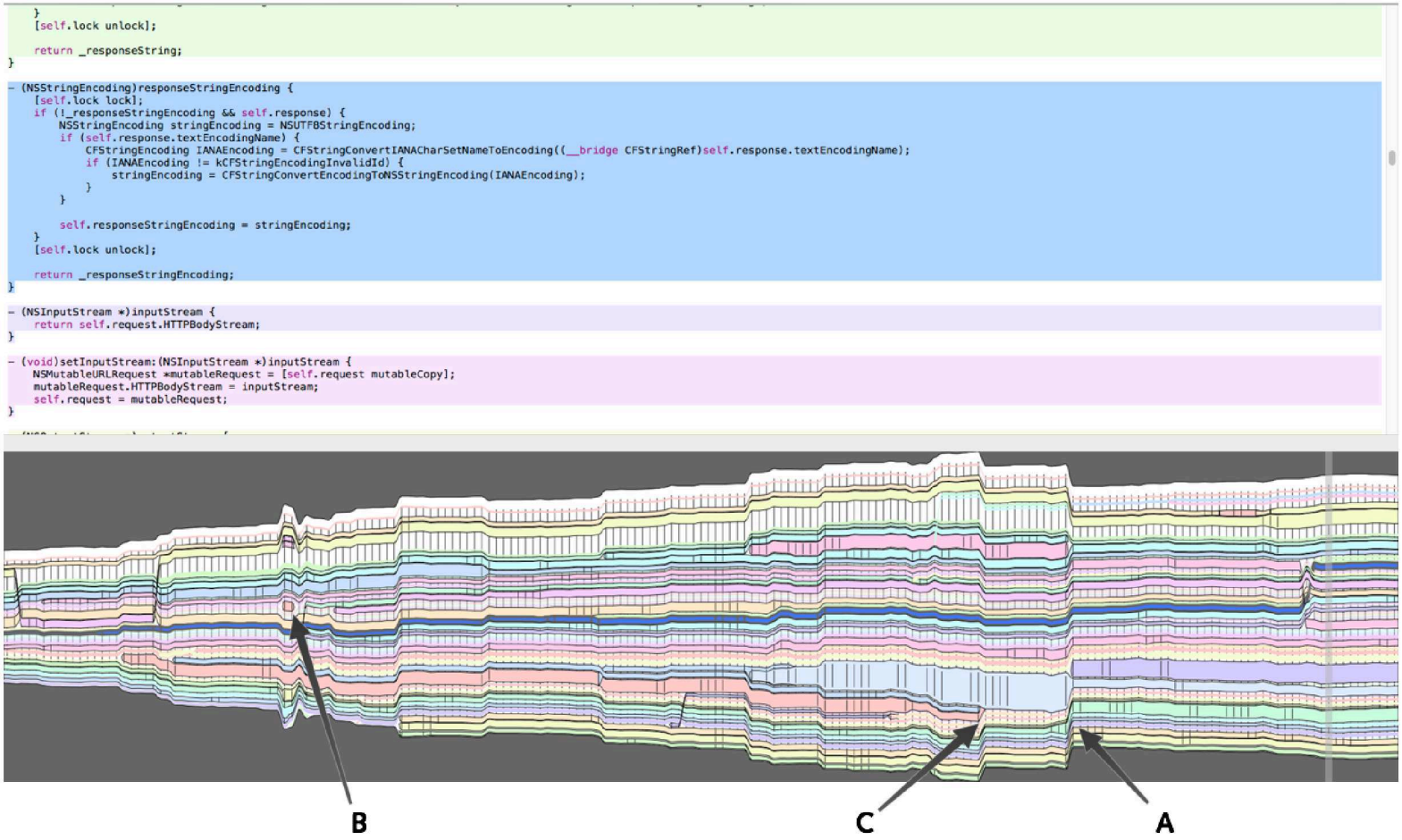


Figure 4.17: User interface of Chronicler for a longer file (∼ 1000 lines) with 200 versions. We used structural abstraction to reduce the depth of the tree to the method level. The visualization reveals details about the selected method (dark blue) and shows that it has been around for the lifetime 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).

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

4.5.1 Strategy Discussion

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

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 history may reveal that the out of place code has been moved from a different method without being properly adapted.

2. How do you understand this version of a method? Quick access to the history opens up the option to look at different versions of a method. This may be particularly interesting when the version of interest handles a lot of special cases that did not use to be there. By understanding the simpler version, the user already knows what the core functionality is, and has an easier time to identify and particularly focus on the special cases in the version of interest.

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 [LaToza et al., 2006].

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.

Following our definition, all of these tasks are exploration tasks; the user does not know in advance which version holds the answer to their questions.

We hypothesized that Chronicler would outperform the timelines for 3 of 4 conditions.

After each task, we asked each participant to evaluate how well each tool was suited for the tasks on a five point Likert scale. For obvious reasons, we hypothesized the other conditions to always be preferred over the normal timeline. For Task 1, we hypothesized Chronicler to be preferred over the structure 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. In Task 2, Chronicler does not provide the answer in the visualization. However, since structural changes, e.g., length of the method or existence of certain code blocks, could be information scent for "easier to understand", we expected it to outperform the timelines. Task 3 is specifically asking for a non-structural element that cannot be visualized in Chronicler. The visualization still provides some information, i.e., the age of the surrounding code block and when it changed. We expected the structure aware slider to be preferred, because the visualization may suggest the existence of rich information that is not there. For task 4, a refactoring is very difficult to identify in the timeline conditions because the user can only see a small part of the code at a time. The visualization provides a global view and was thus expected to outperform the structure aware timeline.

We identified examples of these tasks in existing files from our own development projects and a public GitHub repository[7]. Participants spent roughly ten minutes for each task. We urged them to think about how they would solve the task with each tool instead of doing it quickly. They could switch back and forth between the three conditions at their own leisure and amend their previous strategies.

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

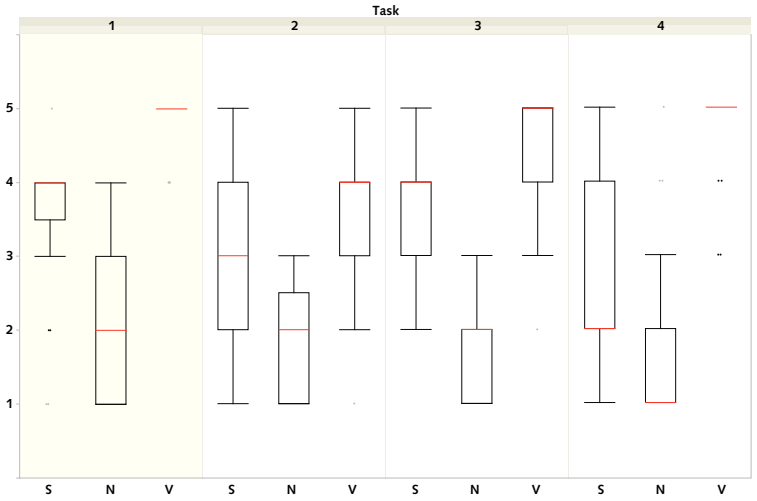


Figure 4.18: The results from the questionnaire asking how well the tool was suited for each of the discussed strategies with median. S: Structure Aware Timeline, N: Normal Timeline, C: Chronicler

Quantitative Results

Chronicler outperformed the timeline for all conditions.

To analyze nonparametric data we used a Kruskal-Wallis test and Wilcoxon tests for pairwise comparison. Overall, we see a significant preference ($p < 0.01$) for the visualization over both the normal timeline, as well as the structure aware timeline. For, Task 2 there is no significant difference between Chronicler and the structure aware timeline. This did not match our expectation; participants did not consider the visualization to have an additional benefit over a structure aware timeline. And in Task 4 there is no significant difference between the timeline and the structure aware timeline. This is unsurprising because both slider conditions barely visualize a refactoring. All other tasks show a significant preference ($p < 0.01$) for Chronicler over the structure aware slider and a significant preference of the structure aware slider over the normal timeline ($p < 0.05$). Figure 4.18 shows a box plot of the results for each individual strategy.

The results clearly indicate that Chronicler is generally better suited for these kinds of exploration tasks, with the ex-

ception of understanding a method where people could not see a particular benefit of the visualization. Unsurprisingly, people would always prefer a structure aware navigation method over a normal timeline.

Qualitative Results

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.

Chronicler encourages lateral exploration.

Second, the visualization of structural change is used as strong indicator for relevant versions. During exploration, participants start at versions emphasized by a structural change occurring. The timeline conditions do not have any such indicator and we see a lot more random or even exhaustive searches of versions.

Structural changes are used as a strong indicator for relevant versions.

We now describe the detailed observations. 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. The strategies in the slider conditions otherwise were the same for most participants and we report them as one. A notable exception are two participants who, in the normal timeline condition, used a number of variables at the beginning of the file as a static reference point. They used changes in these variables as an indicator for a refactoring in task 4. Otherwise the normal timeline was treated as a less helpful version of the structure aware timeline, where the user had to manually adjust the viewport after navigation.

Using the timelines, refactorings were spotted in two ways. One group (10) of participants used the length of the file as a global indicator; they either looked at the end of the

Some participants used suddent changes in the length of a file as an indicator for change.

file without using structural features or at the size of the scroll bar. 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. Both groups usually verified their hypotheses in the Chronicler condition however. There, the refactorings were reported to be easily spottable through structural reorganization or length changes.

In the timeline condition, only four people attempted to identify key versions first (Tasks 1 and 2). Key versions were the beginning and end of the method, as well as "large" changes. Nine participants randomly selected older versions to find a version to compare and 12 proposed an exhaustive search of all versions.

Chronicler encourages people to look for versions with structural change.

With the visualization of Chronicler, people instead explicitly looked for versions with structural changes (17) and/or content changes (5). Here, only four proposed to look at each version. Most participants (18) used the visualization to identify valid navigation ranges before attempting any navigation.

Both, overview and detail visualizations can be helpful.

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).

All participants tried to select small, easily identifiable code snippets when looking at the code in accordance with [LaToza and Myers, 2010]. When possible they selected a snippet that was completely visible on screen.

4.5.2 Navigation Tasks

The second part of our study tested navigation speed.

So far we have only dealt with exploration tasks. The sec-

Find the last version

```
- (void)operationDidStart {
   …
   …
   if ([self isCancelled]) {
        [self finish];
    }
}
```

```
- (void)operationDidStart {
   if ([self isCancelled]) {
        [self finish];
    }
   …
   …
}
```

Figure 4.19: Cheat sheet for a navigation tasks showing the initial version on the left and the final version on the right.

ond part of our study was designed to consider navigation. Navigation requires the participant to know what the target version looks like. This is not the case for the exploration tasks used in the strategy discussions. Local undo often requires such a navigation task. Consider a the user wanting to go back to an earlier version after trying an experimental change. She likely still knows what the earlier version looked like and thus has a clear idea what version she wants to go back to. Improved history navigation interfaces could support that use case. Another example of navigation tasks are variants of task 3 mentioned above 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., more or less than 10% of the versions on the timeline. We looked for two examples for each combination of characteristics.

Participants were given a cheat sheet (Figure 4.19) containing 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. We gave people as much time as they wanted with these cheat sheets to understand the tasks before starting the experiment. We measured the task completion time from starting the experiment to the last navigation event.

Our hypotheses were as follows:

1. Overall, the structure aware conditions are faster than the normal timeline.
2. For each trial, the structure aware conditions are faster than the normal timeline.
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.
4. Not represented events are faster to find using the structure aware timeline because there is no distracting visualization.

The structure aware interfaces are roughly 2 times faster than the normal timeline.

We performed a between-groups study, where each participant was randomly assigned to a condition. We analyzed the measurements with a multi-factor ANOVA and post-hoc analysis using paired t-tests. The analysis shows a significant decrease in task completion times when using the structure aware tools (effect of condition: $p < 0.01$, normal timeline vs. chronicler: $p < 0.01$, normal timeline vs. structure aware timeline: $p < 0.01$). Figure 4.20 shows a bar graph of the task completion times over all tasks.

No significant differences between the two structure aware conditions were found. We could also not find a significant effect of any individual task on performance. While this does not mean that Chronicler cannot be faster than the structure 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 navigation.

4.5.3 Exploring Familiar Source Code

The final part of the study let participants explore familiar source code.

For the last part of the study, we let participant exploring familiar code. Participants were asked to bring their own repositories in any of the following programming languages: JavaScript, CoffeeScript, Java, C, Objective-C, or

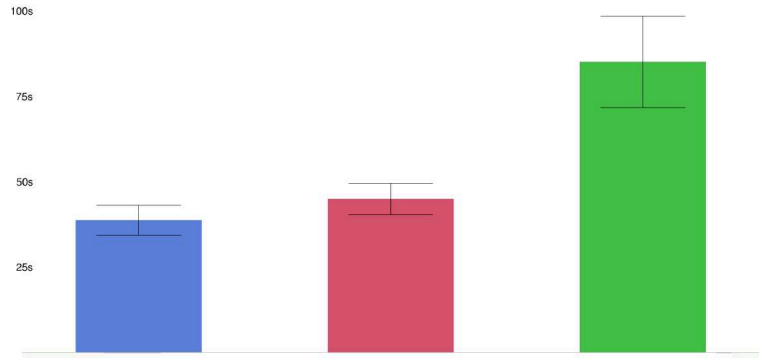


Figure 4.20: Task completion times over all tasks with standard error bars. Blue: Chronicler (39.47s), red: structure aware timeline (45.05s), green: normal timeline (85.33s). On average structure aware navigation is roughly twice as fast as the normal timeline.

Swift. Before the study we explicitly told participants that we would not keep the code and that they could delete it from the study machine when the study was over. As mentioned earlier, we assume history to be linear for now, so when loading the participant's repositories we always used those branches from the version control system that contained the most changes on a loaded file.

Four participants did not bring any familiar source code. Unsurprisingly, 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 usecases.

- *General use*: There were some comments indicating that people would use the tool.
  - "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, which they may replace with a tool like Chronicler.

- *Structural soundness and stability*: Some people focused on the structural soundness of their code and 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."
  - "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 usecase.
- *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 customer.

  - "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."

4.5.4 Interface Enhancements

Our participants requested some interface enhancements for the Chronicler interface, no specific requests for the slider interfaces was mentioned.

- *Diff*: Almost everyone wanted to see a typical Diff to compare between two versions. This of course makes sense. We would suggest to implement such a view in way that it displays the actual structural differences as described here. In this way, a user could then see that the code in Figure 1.7 was not actually deleted but moved as a whole.

- *Zoom on both axes*: Similarly, most people requested to be able to zoom in on a certain time range as well as the depth of the tree. They mentioned that it could sometimes be less interesting to look at the history as a whole, but maybe only at a part of the history to a certain event, e.g, the last refactoring.

- *Searching in time*: Some people mentioned they would like to find all versions containing a certain string, type, or other element by searching. This would be easy to add by highlighting the structural element containing such a string. It might be interesting to filter this search by structural elements to avoid clutter from other methods.

- *Manual grouping*: A few people also suggested to allow manual grouping of elements that they identified to be interesting as a group. This would create a new structure on top of the automatically parsed structure that would contain more meaning. It is, however, not clear how to deal with splits and changes within such a user defined structure.

We agree that these changes would be beneficial to a production level system.

4.6 Summary

Depending on the tasks the user should be able to switch between a structure aware timeline and our tree flow visualization.

We showed that the existing history navigation techniques are outperformed easily by structural navigation techniques with roughly a 2 fold speedup. This is likely to be even more when comparing it to other commonly used interfaces, e.g., combo boxes to select versions. While we could not show a difference in navigation times between the two structural navigation techniques for navigation tasks, we see significant differences in preferences for exploration. This indicates that, when designing for navigation tasks, we could use a structure aware timeline, which takes up less screenspace and can thus be visible in the interface permanently. If also designing for exploration tasks, this timeline can easily unfold into the Chronicler timeline. With the comments from developers in our study, we can also confirm developer's interest in this area.

4.6.1 Limitations

The visualization should not completely exclude other code blocks when looking at the detail view.

The design currently has some limitations that should be addressed in the future. Besides the enhancements mentioned by our study participants, we also identified other limitations of the system. Currently, stepping into a method history is a discrete step where surrounding blocks are not visible or accessible anymore. This produces ambiguous visualizations between structure elements ending and moving into other methods. We imagine this to be replaced by a focus + context or fish-eye visualization where the selected code block takes up most the available vertical space and all other elements taking up the remainder, distorting the proportional height but leaving the order intact. A user could then still see code blocks splitting and merging from the selection and could even follow them easily, switching the focus to the new containing block. It is also currently not

possible to compare the behavior of several methods in parallel. Providing a structural filtering method on multiple structural elements could provide a more situation specific arrangement of methods.

It would be interesting to see the behavior of more detailed elements within the code blocks.

The fact that we are currently using code blocks as the finest structural detail limits the usefulness of the tool for individual lines. Interestingly, we heard no requests for showing a sub-history of a structural representation lower than the level of code blocks during our study. This indicates that we do not necessarily need to visualize these structures in the same way. Instead of filtering the history based on these smaller elements, we can only use structural propagation to highlight the path of such smaller structures, e.g., the history of an assignment on top of the code block structure similar to [Cui et al., 2011]. This would better support tasks focused on individual elements.

Aside: Detailed Histories

Related work asks for a more finer grained recording of histories.

One way to get to the history of individual lines and statements, would be to record finer grained code histories as argued by [Yoon et al., 2013] and others. Hill et al. [1992] already stated "it is in keeping with the generalization of Edit Wear and Read Wear that all interaction histories should be recorded permanently, event by event, and be made accessible for later redisplay by interface objects such as menu-items, [...]." We could then use these individual changes to built the correct history over time; changes occur in an existing structure, in a line, or create new structure, a new line. This makes it trivial to know in what structure changes happened and where things were copied from. We could even easily bridge the code to other sources [Hartmann et al., 2011] and these sources' histories.

Recoding and sharing everything has strong privacy implications.

Besides storage space issues, this approach also has strong privacy implications for the user creating a document. Especially in non-source code applications, having access to everything the user may have tried and then deleted afterwards will not sound appealing to a lot of users. Also, having the history cluttered with every experimental ver-

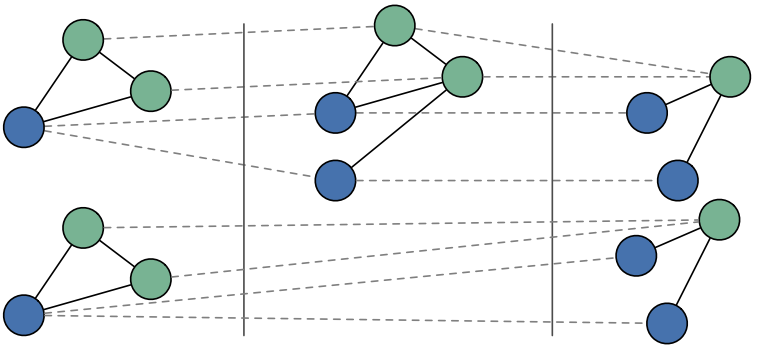


Figure 4.21: Aggregate changes within a history graph with original (top) and aggregated graph (bottom). The successors of a node in the aggregated graph are all successors of the successors in the original graph.

sion may not be practical. It is not necessarily interesting to undo individual character changes after a certain time has passed.

An editor can aggregate small fine grained details into larger chunks to easily match.

To avoid these concerns, we propose to only build the history ephemerally. Assume we have a persisted version from something like a version control system and a structure aware editor. As the document changes, the editor records the editing history of the document for undo. It is easy to know which fine structures are affected or created for these individual edits, and where they mave been moved or copied from. This enables the user to use a system like Chronicler for fine-grained undo during an editing session, similar to how undo works now. We can then aggregate all these individual changes into a large change from one persisted version to the next without the requirement for heuristic matching later on (see Figure 4.21). The user would still be in control about what is persisted, but without losing the benefit of the correct structural evolution of fine grained elements.

In the next chapter, we will introduce other uses for the structure tree visualization of Chronicler. We will also discuss how to find appropriate structures in other media types.

Chapter 5

Generalizing Tree Flow Interfaces

"There is no abstract art. You must always start with something. Afterward you can remove all traces of reality."

—Pablo Picasso

With Chronicler, we introduced a tool that handles the specific structure of source code arrangement in a file. We can easily extend Chronicler for written text like books using the separation of text into sentences, paragraphs, chapters, etc.. This chapter deals with the challenges of generalizing the idea of tree flow to other media types. We look at structures that exist in parallel, e.g, the syntax tree and the call graph, then we discuss structures that are not ordered, e.g., the elements in a 2D image, and finally we look at unstructured data where the structure can be user-generated.

5.1 Parallel Structures

Depending on the the medium, other structures might be interesting.

Our idea was to find structures that represent as many tasks as possible by using structure as information scent. However, as [Pirolli, 2007] stated, quality of information scent "can have dramatic qualitative effects on surfing

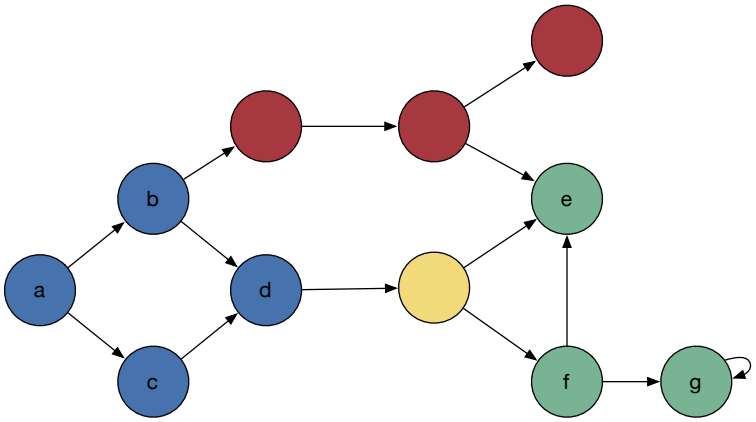


Figure 5.1: A small example call graph. Blue nodes have a call path to the yellow node, red nodes have no call path to yellow, and green nodes are callees.

large hypertext collections." No matter what structure we choose, there will always be tasks that are not well-represented by a given interface. A structure like the code hierarchy used for Chronicler expresses spatial and containment relationship of lines in a file. But, we could also easily be interested in other relationships of source code, e.g., the function call relationship described by the call graph. Note that we can still access and navigate to any version of the call graph within the syntax tree, but information scent is limited to the displayed methods. With the proposed extension of Chronicler that enables users to select multiple methods, she could then choose to select the methods called by a method of interest.

Sometimes, multiple structures deserve special consideration.

But, the history of the call graph, representative for a similar second structure in a medium, may be important enough to warrant special consideration. This is one step back into semantic interfaces, but instead of creating a specialized interface for each specific task, we consider classes of tasks that occur in different structures. Supporting code navigation within a single version by use of the call graph has been shown to be a useful tool [Karrer et al., 2011a]. In our CHI 2013 paper, we showed that a developer's navigation behavior is affected by the chosen call graph navigation tool [Krämer, Karrer, Kurz, Wittenhagen, and Borchers,

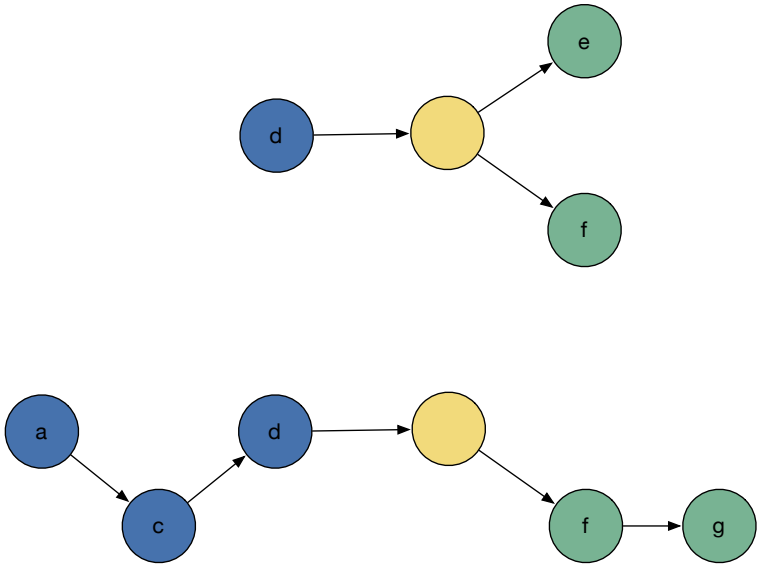


Figure 5.2: Top: the immediate callers (green) and callees (blue) of the yellow method as used by Stacksplorer. Bottom: an arbitrary call path through the yellow method node as used by Blaze.

2013]. Stacksplorer [Krämer, 2011] and Blaze [Kurz, 2011] are tools that show a specific linear graph subset part of the call graph. While Stacksplorer displays the method adjacent to current methods, i.e., the callers and callees, Blaze displays one of the possible call paths that affect the current method (Figure 5.2 shows both).

The history of the call graph could support developers to find a recently introduced bug. Consider a developer looking at a code snippet that stopped working recently, e.g., according to a test case. When she investigates, she finds the method called by the test case has not changed, which, to her, indicates a side effect in one of the callees. A common approach to find the issue would be to step through the method and look at unexpected return values, and then repeat the process in the callee. Depending on how deep the issue is hidden in the call graph, this is a tedious process. The idea is that if we can visualize changes within in the call graph structure over time, we can use this to draw conclusions about the bug.

Parallel structures can represent different relationships on the same entities.

Call graph structure and syntax tree structure are *parallel*, because even though they represent some of the same vertices, i.e., methods, they represent very different relationships; just because two methods are implemented at different ends of a file, does not say anything about if they call each other. The two structures have a different shape, the syntax tree was an ordered tree structure, and the call graph is an ordered network. In the call graph, two nodes are connected when one node, a method, calls another method, i.e. the caller and callee. There are obvioulsy not many restrictions to this relationship, a caller can call a method several times or not at all, it can even call itself. We do consider calls to be ordered by their appearance in the caller's source code.

If we want to use a visualization like Chronicler for the history of the call graph, we have to find a visualization for this graph. To our knowledge, there is no approach that can easily visualize a lot of versions of such a complex graph structure over time. Complex networks are difficult to visualize slicable, i.e., in one dimension, because of their inherent multi-dimensionality; Gibson et al. [2013], Parker et al. [1998], and Erten et al. [2004] are some example approaches for 2D and 3D visualization. We can attempt to reduce such a complex graph into a tree structure, which enables us to then use the tree flow visualization approach. The literature shows a number of ways to reduce a network into a tree structure, which in turn could be visualized as a slice using the tree stack approach. Heer and Boyd [2005], Archambault et al. [2006] find non-overlapping clusters in the graph which, when applying this approach recursively, form a tree. We discuss two ways to do this, create a subtree of the graph and use a secondary structure.

5.1.1 Subtrees

The call graph may be interesting as a whole or a local subset.

The goal of understanding the call graph over time can be twofold. We can consider the user to be focused on an individual method, similar to Stacksplorer, or we can attempt to understand the call graph as a whole, e.g., to see connected components in the code [Zimmermann et al., 2003].

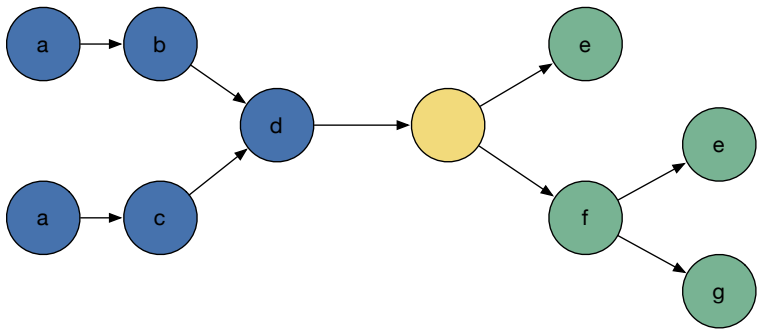


Figure 5.3: Tree expansion based on the focus method (yellow) for both callers (blue) and callees (green) of the call graph in Figure 5.1. Nodes that can be reached over multiple paths in the call graph are repeated in the tree.

At first, we consider the former case; we will deal with the latter in the next section.

Using a focus element we can build subtrees of the graph.

With a focus element, a method selected by the user, we can easily build a subtree from the call graph. For some structures, it may be interesting to find a spanning tree of the strongly connected component, rooted in the focus method. This does not work for our example usecase, because it may be interesting that a change in method a coincides with changes in b, but not c. Instead, we create a tree expansion of the the graph that represents all possible call paths originating in the focus method. For each call we append a new child node representing the called method. The traversal stops when it encounters a method that is already represented on the path from the origin. We can do an analogous traversal for callers. We can invert the process and represent callers the same way, we could even do it at the same time representing callers and callees in parallel. Figure 5.1 shows a call graph and Figure 5.3 shows its tree expansion.

The tree shows changes in a call connected subset of methods.

Visualizing the history of this tree using the tree flow visualization now gives us information about changes in the call structure. In the tree flow visualization, we can see reordering of calls, as well as changes happening in the depth of the tree. More importantly, when looking for changes that may have caused a bug, the user can directly hone in

on changed parts, so she can reason about how they may have caused the problem. This has the potential to shorten the search time for these bugs substantially. If the change is structural, we may even be able to see it in the visualization itself. Examples of this would be removing an important call in a previous version.

5.1.2 Visualization Properties

We can visualize the history of visualization properties.

If we are interested in the call graph as a whole, we can not directly use tree flow. However, since the call graph is a complex network, we will need a specific visualization for it, otherwise, we cannot see the graph as a whole, which would defeat the purpose. Holten [2006] represent callers and callees in a ring visualization, where the locations on the ring are aligned based on the location of nodes in a source hierarchy, e.g., packages, components, and methods. This is nothing else but an extended syntax tree that includes the file structure and stops at the method level.

One possible way to show the history of the graph is to not exchange the tree flow visualization in this case, but only the content visualization. The content view could show something like hierarchical edge bundles Holten [2006] or HivePlots [Krzywinski et al., 2012], and the history could show the secondary, hierarchical structure of the visualization.

5.2 Content Linearization

Similar to textual media, we want to represent the history of other media types. The evolution of an image can be interesting for understanding its origin; one famous example from Art History is Picasso's "The Bull", e.g. [Lavin, 1993].

Consider presentations as a highly structured visual medium. Navigating through the history of a presentation can help the user to go back to the version of this slide that

had the picture of that car. Similar to source code, navigating back to an older version of images is tedious.

Like text, visual media are often hierarchical.

Some visual media types have a rich syntactical structure which is later lost in the presentation version. A typical presentation created with a slideware presentation tool contains slides, slide groups, objects, and object groups. All of these can be used to group slides and slide elements. We could easily use this information to create a tree structure.

The structures are often not ordered.

Consider the case of visualizing the location history of the elements on the slide. We can easily build a tree structure using the above elements, but with two-dimensional media the elements in the tree do not have an ordering. It needs to be linearized in some way.

So far, we have not explicitly talked about this linearization even though it also occurs in source code, i.e., text, where we abstracted the 2-dimensionality of lines, containing words, by simply representing all content of the line by the lines height. We call this *projection*. The simplest form of projection, axis projection, takes data and reduces it to one axis. This works for lines in code because the content in a line forms a logical unit; typically, everything in a line belongs to a single statement. With images, axis projection does not work because we have to consider side by side objects individually. In most cases, there is no trivial axis over which we can summarize content. We may be able to visualize each axis separately. Figure 5.4 shows projection approaches.

This process of taking multi-dimensional data and finding a linear representation is what we call *content linearization*. We suggest three approaches to deal with this problem:

1. It is possible to create an automatic content ordering. For example, Ishak and Feiner [2006] used a Hamiltonian path through faces in an image to allow linear scrolling from one face to another at any zoom level. Let us assume we do this on a video frame and we have perfect face detection. As people move, we always know where they are in the video frame. We could then try to find another Hamiltonian path, that

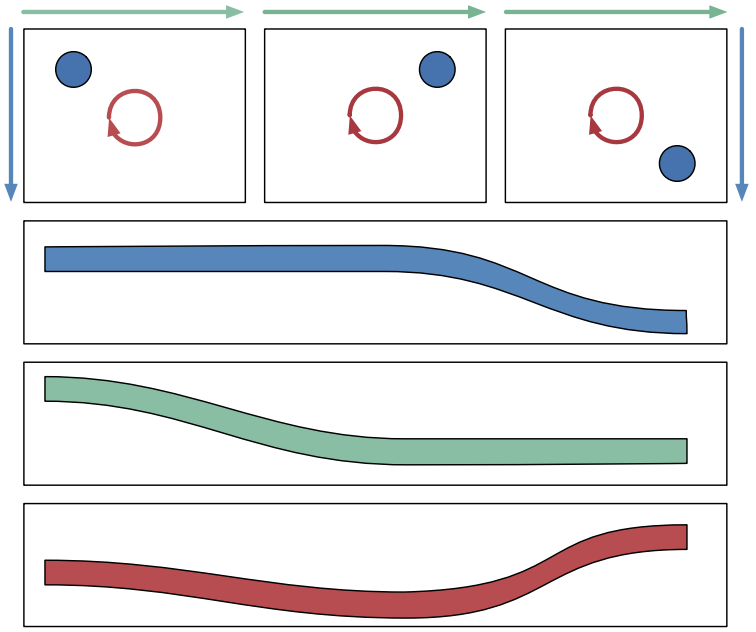


Figure 5.4: Three versions of a 2D canvas and tree flow visualizations with different projection approaches. Top: projection on y-axis. Movement from version 1 to version 2 is not visible. Middle: projection on x-axis. Movement from version 2 to version 3 is not visible. Bottom: projection on radial axis. Shown movements are visible but the visualization is harder to interpret.

is similar to our original path, e.g., in length. This approach is easy to implement, but it comes with the issue that the structure may change drastically as people move, which can become an issue when the user does understand the reason for the drastic change.

2. Sometimes, we can find other, non-trivial axes for projection that provide enough information to create a tree graph. This mostly works for special cases with some additional content restrictions. Presentations where the amount of items on a slide is limited by readability for the audience enable us to create a radial axis (Figure 5.2). We can then project the center location of items and item groups onto that axis. The user can then draw conclusions about the area a tree element is in at a given time, and which other el-

ements it is close to. There are some trade-offs that should be considered with this approach. Objects next to each other but across the axis origin will be far away from each other. Two objects on the same projection axis, will have to share the available space, which makes it infeasible for large amounts of objects. There is also a jump from $360°$ at the bottom to $0°$ at the top of the graph and a jump when moving through the origin. This approach is not so well suited for video or animation, because it also transforms motion paths through the scene. A straight line in the video would result in a curved representation in the annotation. Worse, movement towards or away from the center will not be represented at all.

3. As with the structure of a visualization, we can more generally use *secondary structure* of the same data. Assume we want to create a tree flow visualization showing where a user has taken the photos in their photo library. If we visualize the locations on a map, we can easily build a tree structure over continents, countries, regions, cities, etc. We can also map the locations through history by grouping them over a time and distance threshold, similar to stroke chains in ExamPen, e.g., if two photos were taken at the same location within a certain time, the location "continues" in the history. To linearize the locations on each level of the tree structure, we can use the secondary structure of the location's name to order them.

5.3 User-created Structure

Users often structures documents during creation.

Some applications, especially graphics application actually may have a rich user-created structure that is independent of object locations. An example is Adobe Illustrator where the user creates layers, layer groups, or smart objects for their drawing. These groups can create a very detailed and complex hierarchy. A user generated hierarchy with user-generated labels may be the better solution to visualize changes of a document than the objects' location. And we will still see splits, merges, and movements as the user

moves content between layers, groups layers, or rasterizes layer groups. The visualization would now represent an unknown user-defined relationship, which may still be easily understood by another user.

We can use structural propagation to let the user create structures after the fact.

We can use similar approaches to build structures after the fact. As described earlier, some media types like video have a very sparse syntactical structure. Navigating through a video using a technique like tree flow requires us to create a new structure from the provided syntactic information. As with DRAGON, where the user selects objects to show their trajectory, we may be able to select objects to insert them as an element of the structure. Goldman et al. [2008] showed that objects can also be selected in regions by using painting over objects as a selection technique. When the user selects an object in one of these ways, we can use the movement information of every individual pixel, e.g., through flow fields [Brox et al., 2004], as a match through time. Then, using structural propagation, we can label all reachable pixels in all frames as belonging to the object, which we can visualize. By selecting objects one after another and grouping them explicitly, the user can build a tree structure over time. It may even be feasible to automatically register splits and merges. We see an example of this grouping in Trailblazing [Kimber et al.], where objects under a certain distance threshold are grouped together.

Users already know how to add complex structures to spreadsheets.

Another example of a media type with a sparse syntactic structure are spreadsheets. Spreadsheets have no inherent information about their content and content elements usually have lots of different relationships, i.e., each table column provides another ordering of the data. Spreadsheet applications let the user sort data by these individual columns and combination of the columns. We can use a similar approach to group information over time. Perin et al. [2014] and Vuillemot and Perin [2015] argued that navigating through the evolution of tables can be interesting to understand the development of soccer ranking tables or macroeconomic data. Vuillemot and Perin [2015] also introduced an approach to navigate through table elements by showing a ranking trajectory for data under a given ordering. While this works for sparse datasets like a soccer table, larger tables quickly become cluttered (see

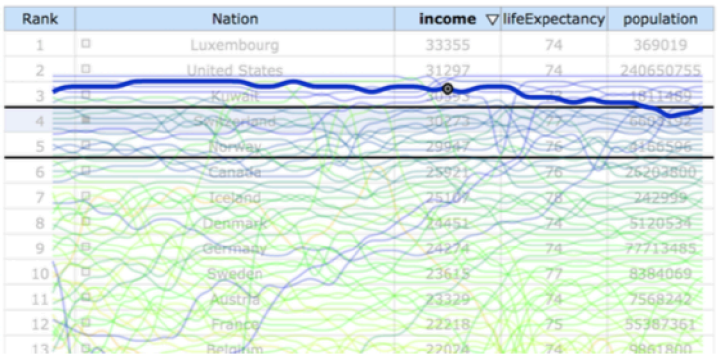


Figure 5.5: Excerpt of a ranking table with macroeconomic data. The overlay easily gets cluttered for large tables. The colors provide a form of visual grouping. We propose to allow the user to make these groupings more explicit. From [Vuillemot and Perin, 2015]

Figure 5.5). We suggest to let the user define the structure with ordering and grouping techniques together with structural propagation to create a tree history of such a table. If the same column exists in all versions of the data, we can propagate sorting and grouping to all versions of the table, For macroeconomic data, a user could create a grouping of countries into geographic regions, free trade zones, economic growth factors, etc.; the best grouping completely depends on the user's questions.

5.4 Summary

We have provided some concepts of how we can find and use hierarchical structure to create tree flow like visualizations of the history of different media types. The final chapter summarizes this work and describes potential interesting questions for future work.

Chapter 6

Summary and Future Work

"Never make predictions, especially about the future."

—Casey Stengel

In this chapter, we briefly summarize the contents of our work and highlight contributions. Finally, we give a brief outlook on future work.

6.1 Summary and Contributions

This work was motivated by the difficulty to navigate a document's history and find appropriate changes to compare, undo, or understand. We likened this to temporal navigation and exploration in the context of navigation tasks of more classic temporal media, in particular video.

Semantic navigation interfaces restrict accessible frames.

While we see content driven navigation interfaces in that domain, we recognized that those interfaces run into problems when the user tries to explore a media file. The main reason for this are that we cannot easily display all content from a whole video over time. Thus, we have to look at miniature versions of the content, which can quickly

become confusing, and are thus unsuited to get a quick overview of a whole video. Another option is to abstract from the content, as we have seen with DMVN systems, where we represent the movement of individual objects through trajectories. However, those interfaces restrict the navigation to certain frames and can only work for individual objects.

We defined three requirements for supporting temporal exploration.

We introduced DragLocks, an extension for direct manipulation video navigation systems that allows us reduce the navigation restrictions in the interface. Unfortunately, even with our extension, the direct manipulation interfaces cannot easily handle the exploration of relationships from multiple objects. This led us to define three requirements we aim at for temporal exploration interfaces, such an interface should be ambiguity-free and consider multiple objects as well as historical objects.

We introduced how structure can support navigation.

Instead of attempting to extend existing video navigation systems with these properties, we looked for another approach visualizing structural changes within a medium that could serve as information scent, leaving the interpretation of changes to the user. We then showed the usefulness of using structure over time with our ExamPen prototype. From there, we defined our notion of structure and structural properties that we can use for temporal navigation.

Chronicler is a navigation and exploration interface for source code.

Based on this understanding, we designed Chronicler, a system for navigating and exploring source code histories based on the tree structure of source code. We showed that the system has a strong user preference when it comes to exploration tasks and it also outperforms a non-structure aware timeline slider when it comes to navigation tasks. Such a system can complement more specialized navigation mechanisms that have to make trade offs between navigation speed in specific tasks and restricting navigation for other tasks. Finally, we illustrated how the tree structure and propsosed temporal navigation interface can be applied to other domains.

6.2 Future Work

There are many opportunities to extend on this work. One could look into different structural representation and visualizations, more specific usecases, e.g., interfaces to create user-defined structures, and finally prescriptive use of these methods.

6.2.1 Structural Representations

How can other structures be visualized?

While we have defined structural navigation properties for any kind of structure, we focused on ordered tree structures for the rest of this work. However, there are structures that are not easily transformed into such an ordered tree, especially 2D location data. We have given some insights into how to transform such a structure into a tree, by using special axis transformations or using secondary structures. Solving this issue would probably require a 3D visualization which has to deal with occlusion issues, e.g., when objects are contained within other objects in such a scene.

There are also more restricted structures that could benefit from a special visualization. Consider multilevel hierarchies [Sugiyama et al., 1981] graphs with multiple levels where any node in a level only has edges to higher levels. There could be a way to directly make a history visualization of special graph structure like this. With a history visualization of structures different from tree graphs, we could potentially find better visualizations for problems like the call graph.

6.2.2 Usecases

How can interfaces implement our structural navigation properties?

Considering details of specific usecases would be interesting to further explore the meaning of the structural navigation properties. For example, so far we have only used structural propagation for simple concepts like selection. However, it may be interesting to use it for more complex

annotations. Imagine a user looking at an old version to understand a method. In these cases it may be interesting to actually make annotations of that method available in the future. We could even go as far as [Ginosar et al., 2013], who try to change multiple versions at once. It is interesting to find out how well users are able to do this, what kind of strategies they employ, and what tasks it benefits their task.

6.2.3 Prescriptive Use

In our GI 2012 paper [Walther-Franks, Herrlich, Karrer, Wittenhagen, Schröder-Kroll, Malaka, and Borchers, 2012], we used the information from direct manipulation navigation prescriptively. We extended Blender, a multi-track interface for animation and rendering, with a new technique to define animation timing. The user could first create the path of an object through a scene without adding any timing information to that path. Then, instead of navigating to a time when dragging the object through the scene, the timing of the dragging gesture would define the movement timing of the object. This allowed users to define natural looking motion simply by demonstrating it to the objects themselves.

We could also let the user define how structure changes over time.

There could be similar approaches for the structural part of such an animation editor. Imagine, instead of having multiple static tracks and track groups, tracks can move from one track group into another. This could allow the user to define grouping that makes sense over time. First, an object may move in a group and have attributes like speed, volume, opacity, etc. from that group. Then the user could separate its track at a certain point in time, at which point it gets the properties from itself or its new group. This could increase the understanding of objects belonging together and separating over time in the user interface.

List of Publications

Peer-reviewed

Moritz Wittenhagen, Christian Cherek, and Jan Borchers. Chronicler: Interactive Exploration of Source Code History. *In CHI '16: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3522–3532, 2016.

Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How Tools in IDEs Shape Developers' Navigation Behavior. *In CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3073–3082, 2013.

Thorsten Karrer, Moritz Wittenhagen, and Jan Borchers DragLocks: Handling Temporal Ambiguities in Direct Manipulation Video Navigation. *In CHI '12 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 623–626, 2012.

Benjamin Walther-Franks, Marc Herrlich, Thorsten Karrer, Moritz Wittenhagen, Roland Schröder-Kroll, Rainer Malaka, and Jan Borchers. Dragimation: Direct Manipulation Keyframe Timing for Performance-Based Animation. *In GI '12: Proceedings of Graphics Interface*, 2012.

Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, Florian Heller, and Jan Borchers. Pinstripe: Eyes-Free Continuous Input on Interactive Clothing. *In CHI '11 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1313–1322, 2011.

Stephan Deininghaus, Max Möllers, Moritz Wittenhagen, and Jan Borchers. Hybrid Documents Ease Text Corpus Analysis for Literary Scholars. *In ITS '10: Proceedings of the International Conference on Interactive Tabletops and Surfaces*, pages 177–186, 2010.

Demos, Posters, and Workshops

Florian Heller, Leonhard Lichtschlag, Moritz Wittenhagen, Thorsten Karrer, and Jan Borchers. Me Hates This: Ex- plor- ing Different Levels of User Feedback for (Usability) Bug Reporting. *In CHI EA '11: Extended Abstracts of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 1357–1362, 2011.

Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, and Jan Borchers. ExamPen: How Digital Pen Technology Can Support Teachers and Examiners. *In CHI '10: Workshop on Next Generation of HCI and Education*, 2010.

Thorsten Karrer, Moritz Wittenhagen, and Jan Borchers. PocketDRAGON: A Direct Manipulation Video Navigation Interface for Mobile Devices. *In MobileHCI '09: Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–3, 2009.

Bibliography

Christopher Ahlberg and Ben Shneiderman. The Alphaslider: A Compact and Rapid Selector. In *CHI '94: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 365–371, 1994.

Paul André, Max Wilson, Alistair Russell, Daniel A Smith, Alisdair Owens, and M C Schraefel. Continuum: Designing Timelines for Hierarchies, Relationships and Scale. In *UIST '07 Proceedings of the annual ACM symposium on User Interface Software and Technology*, pages 101–110, 2007.

Apple. *The Swift Programming Language*. Apple, 2014.

D Archambault, T Munzner, and D Auber. Visual Exploration of Complex Time-Varying Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):805–812, 2006.

F Arman, R Depommier, A Hsu, and M Y Chiu. Content-Based Browsing of Video Sequences. In *MULTIMEDIA '94: Proceedings of the international Conference on Multimedia*, pages 97–103, 1994.

Yuji Ayatsuka, Jun Rekimoto, and Satoshi Matsuoka. Popup Vernier: A Tool for Sub-Pixel-Pitch Dragging with Smooth Mode Transition. In *UIST '98: Proceedings of the annual ACM Symposium on User Interface Software and Technology*, pages 39–48, 1998.

Annie Barrows. Behind the Books with Annie Barrows. `http://www.randomhousebooks.com/articles/behind-the-books-with-annie-barrows`, 2015.

Thomas Berlage. A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects. *Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High Accuracy Optical Flow Estimation Based on a Theory for Warping. In *Computer Vision - ECCV 2004*, pages 25–36. Springer Berlin Heidelberg, 2004.

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

Weiwei Cui, Shixia Liu, Li Tan, Conglei Shi, Yangqiu Song, Zekai Gao, Huamin Qu, and Xin Tong. TextFlow: Towards Better Understanding of Evolving Topics in Text. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2412–2421, 2011.

Ruwanee de Silva, David Tyler Bischel, WeeSan Lee, Eric J Peterson, Robert C Calfee, and Thomas F Stahovich. Kirchhoff's Pen: a Pen-Based Circuit Analysis Tutor. In *SBIM '07: Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling*, 2007.

Stephan Deininghaus. An Interactive Surface for Literary Criticism. Diploma Thesis, RWTH Aachen University, 2010.

Stephan Deininghaus, Max Möllers, Moritz Wittenhagen, and Jan Borchers. Hybrid Documents Ease Text Corpus Analysis for Literary Scholars. In *ITS '10: Proceedings of the International Conference on Interactive Tabletops and Surfaces*, pages 177–186, 2010.

Alan Dix, Janet Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction*. Pearson Prentice-Hall, 3rd edition, 2004.

Pierre Dragicevic, Gonzalo Ramos, Jacobo Bibliowitcz, Derek Nowrouzezahrai, Ravin Balakrishnan, and Karan Singh. Video Browsing by Direct Manipulation. In *CHI '08: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 237–246, 2008.

Ryan Eccles, Thomas Kapler, Robert Harper, and William Wright. Stories in GeoTime. *Information Visualization*, 7 (1):3–17, 2008.

Stephen G. Eick. Graphically Displaying Text. *Journal of Computational and Graphical Statistics*, 3(2):127–142, 1994.

Stephen G. Eick, Joseph L Steffen, and Eric E. Sumner Jr. Seesoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18 (11):957–968, 1992.

Cesim Erten, Philip J Harding, Stephen G Kobourov, Kevin Wampler, and Gary Yee. Exploring the Computing Literature Using Temporal Graph Visualization. In *Proceedings SPIE 5295, Visualization and Data Analysis*, pages 45–56, 2004.

Ben Fry. Watching the evolution of the "Origin of Species". `http://benfry.com/writing/archives/529`, 2009a.

Ben Fry. On the Origin of Species: The Preservation of Favoured Traces. `http://benfry.com/traces`, 2009b.

Helen Gibson, Joe Faith, and Paul Vickers. A Survey of Two-Dimensional Graph Layout Techniques for Information Visualisation. *Information Visualization*, 12(3-4):324–357, 2013.

Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Björn Hartmann. 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*, pages 485–494, 2013.

Dan B Goldman, Brian Curless, David Salesin, and Steven M Seitz. Schematic Storyboarding for Video Visualization and Editing. *ACM Transactions on Graphics*, 25(3):862–871, 2006.

Dan B Goldman, Chris Gonterman, Brian Curless, David Salesin, and Steven M Seitz. Video Object Annotation, Navigation, and Composition. In *UIST '08: Proceedings of the annual ACM symposium on User Interface Software and Technology*, pages 3–12, 2008.

Tovi Grossman, Justin Matejka, and George Fitzmaurice. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *UIST '10: Proceedings of the annual ACM symposium on User Interface Software and Technology*, pages 143–152, 2010.

Björn Hartmann, Mark Dhillon, and Matthew K Chan. 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*, pages 2207–2210, 2011.

Susan Havre, Elizabeth Hetzler, Paul Whitney, and Lucy Nowell. ThemeRiver: Visualizing Thematic Changes in Large Document Collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, 2002.

J Heer and D Boyd. Vizster: Visualizing Online Social Networks. In *INFOVIS '05: IEEE Symposium on Information Visualization*, pages 32–39, 2005.

Florian Heller, Leonhard Lichtschlag, Moritz Wittenhagen, Thorsten Karrer, and Jan Borchers. Me Hates This: Exploring Different Levels of User Feedback for (Usability) Bug Reporting. In *CHI EA '11: Extended Abstracts of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 1357–1362, 2011.

William C. Hill, James D. Hollan, David A Wroblewski, and Tim McCandless. Edit Wear and Read Wear. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3–9, 1992.

Orland Hoeber and Joshua Gorner. BrowseLine: 2D Timeline Visualization of Web Browsing Histories. In *IV '09: Proceedings of the international Conference on Information Visualisation*, pages 156–161, 2009.

Reid Holmes and Andrew Begel. Deep Intellisense: A Tool for Rehydrating Evaporated Information. In *MSR '08: Proceedings of the International Working Conference on Mining Software Repositories*, 2008.

D Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

Wolfgang Hürst and Georg Götz. Interface Designs for Pen-Based Mobile Video Browsing. In *DIS '08: Proceedings of the 7th ACM conference on Designing Interactive Systems*, pages 395–404, 2008.

Edward W Ishak and Steven Feiner. Content-Aware Scrolling. In *UIST '06: Proceedings of the annual ACM symposium on User Interface Software and Technology*, pages 155–158, 2006.

A D Jepson, D J Fleet, and T F El-Maraghi. Robust Online Appearance Models for Visual Tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(10): 1296–1311, 2003.

Brian Johnson and Ben Shneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Visualization '91: Proceedings of the IEEE Conference on Visualization*, pages 284–291, 1991.

S Jürgensmann and H J Schulz. A Visual Survey of Tree Visualization. `http://vcg.informatik.uni-rostock.de/~hs162/treeposter/oldposter/treevis_lores.pdf`, 2010.

Huzefa Kagdi, Maen Hammad, and Jonathan I Maletic. Who Can Help Me with This Source Code Change? In *ICSM '08: IEEE International Conference on Software Maintenance*, pages 157–166, 2008.

Thomas Kapler and William Wright. Geotime Information Visualization. *Information Visualization*, 4(2):136–146, 2005.

Thorsten Karrer. *Semantic Navigation in Digital Media*. PhD thesis, RWTH Aachen University, 2013.

Thorsten Karrer, Malte Weiss, Eric Lee, and Jan Borchers. DRAGON: A Direct Manipulation Interface for Frame-Accurate in-Scene Video Navigation. In *CHI '08: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 247–250, 2008.

Thorsten Karrer, Moritz Wittenhagen, and Jan Borchers. PocketDRAGON: A Direct Manipulation Video Navigation Interface for Mobile Devices. In *MobileHCI '09: Proceedings of the international Conference on Human-Computer*

Interaction with Mobile Devices and Services, pages 1–3, 2009.

Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, and Jan Borchers. ExamPen: How Digital Pen Technology Can Support Teachers and Examiners. In *CHI '10: Workshop on Next Generation of HCI and Education*, 2010.

Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency. In *UIST '11: Proceedings of the annual ACM symposium on User Interface Software and Technology*, pages 217–224, 2011a.

Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, Florian Heller, and Jan O Borchers. Pinstripe: Eyes-Free Continuous Input on Interactive Clothing. In *CHI '11: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1313–1322, 2011b.

Thorsten Karrer, Moritz Wittenhagen, and Jan O Borchers. DragLocks: Handling Temporal Ambiguities in Direct Manipulation Video Navigation. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 623–626, 2012.

Anne Kathrein. Event Detection in Videos Based on Object Trajectories. Bachelor's Thesis, RWTH Aachen University, 2011.

Don Kimber, Tony Dunnigan, Andreas Girgensohn, Frank Shipman, Thea Turner, and Tao Yang. In *ICME '07: IEEE International Conference on Multimedia and Expo*.

Jan-Peter Krämer. Stacksplorer Understanding Dynamic Program Behavior. Diploma Thesis, RWTH Aachen University, 2011.

Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How Tools in IDEs Shape Developers' Navigation Behavior. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3073–3082, 2013.

Martin Krzywinski, Inanç Birol, Steven J M Jones, and Marco A Marra. Hive Plots - Rational Approach to Visualizing Networks. *Briefings in Bioinformatics ()*, 13(5): 627–644, 2012.

Joachim Kurz. Blaze—Navigating Source Code via Call Stack Contexts. Bachelor's Thesis, RWTH Aachen University, 2011.

Thomas D LaToza and Brad A. Myers. Hard-to-Answer Questions About Code. *PLATEAU '10: Evaluation and Usability of Programming Languages and Tools*, 2010.

Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *ICSE '06: Proceedings of the 28th international conference on Software Engineering*, pages 492–501, 2006.

Irving Lavin. Picasso's Bull (s): Art History in Reverse. *Art in America*, 81:76–76, 1993.

Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The Perspective Wall: Detail and Context Smoothly Integrated. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 173–176, 1991.

K Maruyama, E Kitsu, T Omori, and S Hayashi. Slicing and Replaying Code Change History. In *ASE '12 Proceedings of the international Conference on Automated Software Engineering*, pages 246–249, 2012.

Justin Matejka, Tovi Grossman, and George Fitzmaurice. Swifter: Improved Online Video Scrubbing. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1159–1168, 2013.

Michael Mills, Jonathan Cohen, and Yin Yin Wong. A Magnifier Tool for Video Data. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 93–98, 1992.

Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt. Selective Undo Support for Painting Applications. In *CHI '15: Proceedings of the annual ACM Conference on Human Factors in Computing Systems*, pages 4227–4236, 2015.

Eugene W Myers. AnO(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

Petra Neumann, Stefan Schlechtweg, and Sheelagh Carpendale. ArcTrees: Visualizing Relations in Hierarchical Data. In *EUROVIS'05: Proceedings of the joint Eurographics / IEEE VGTC conference on Visualization*, pages 53–60, 2005.

Cuong Nguyen, Yuzhen Niu, and Feng Liu. Video Summagator: An Interface for Video Summarization and Navigation. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 647–650, 2012.

Cuong Nguyen, Yuzhen Niu, and Feng Liu. Direct Manipulation Video Navigation in 3D. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1169–1172, 2013.

Michael Ogawa and Kwan-Liu Ma. Software Evolution Storylines. In *SOFTVIS '10 Proceedings of the 5th international symposium on Software Visualization*, pages 35–42, 2010.

Sharon Oviatt, Alex Arthur, and Julia Cohen. Quiet Interfaces That Help Students Think. In *UIST '06: Proceedings of the annual ACM Symposium on User Interface Software and Technology*, pages 191–200, 2006.

Greg Parker, Glenn Franck, and Colin Ware. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. *Visual Languages and Computing*, 9(3):299–317, 1998.

Charles Perin, Romain Vuillemot, and Jean-Daniel Fekete. A Table!: Improving Temporal Navigation in Soccer Ranking Tables. In *CHI '14: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 887–896, 2014.

Peter Pirolli. Computational Models of Information Scent-Following in a Very Large Browsable Text Collection. In *CHI '97: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 3–10, 1997.

Peter Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, 2007.

Peter Pirolli and Stuart Card. Information Foraging. *Psychological Review*, 106(4):643–675, 1999.

Jef Raskin. *The Humane Interface*. New Directions for Designing Interactive Systems. Addison Wesley, 2000.

Stuart Rose, Scott Butner, Wendy Cowley, Michelle Gregory, and Julia Walker. *Describing Story Evolution From Dynamic Information Streams*. 2009.

H Schulz, S Hadlak, and H Schumann. The Design Space of Implicit Hierarchy Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):393–411, 2011.

Torben Schulz. Sketchassisted Development. Master's Thesis, RWTH Aachen University, 2014.

Francisco Servant and James A Jones. Chronos: Visualizing Slices of Source-Code History. In *VISSOFT '13: Proceedings of IEEE Working Conference on Software Visualization*, pages 1–4, 2013.

Ross Shannon, Aaron Quigley, and Paddy Nixon. Deep Diffs: Visually Exploring the History of a Document. In *AVI '10: Proceedings of the international Conference on Advanced Visual Interfaces*, 2010.

Ben Shneiderman. The Future of Interactive Systems and the Emergence of Direct Manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981.

Anthony Tang, Saul Greenberg, and Sidney Fels. Exploring Video Streams Using Slit-Tear Visualizations. In *AVI '08: Proceedings of the working conference on Advanced Visual Interfaces*, pages 191–198, 2008.

Anthony Tang, Saul Greenberg, and Sidney Fels. Exploring Video Streams Using Slit-Tear Visualizations. In *CHI EA '09: Extended Abstracts of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 3509–3510, 2009.

Alexandru Telea and David Auber. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum*, 27(3):831–838, 2008.

Nuno Vasconcelos and Andrew Lippman. Bayesian Modeling of Video Editing and Structure: Semantic Features for Video Summarization and Browsing. In *ICIP'98 International Conference on Image Processing*, pages 153–157, 1998.

Fernanda B Viégas, Martin Wattenberg, and Kushal Dave. Studying Cooperation and Conflict Between Authors with History Flow Visualizations. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 575–582, 2004.

Fernanda B Viégas, Scott A Golder, and Judith S Donath. Visualizing Email Content: Portraying Relationships From Conversational Histories. In *CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 979–988, 2006.

Romain Vuillemot and Charles Perin. Investigating the Direct Manipulation of Ranking Tables for Time Navigation. In *CHI '15: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2703–2706, 2015.

Benjamin Walther-Franks, Marc Herrlich, Thorsten Karrer, Moritz Wittenhagen, Roland Schröder-Kroll, Rainer Malaka, and Jan Borchers. Dragimation: Direct Manipulation Keyframe Timing for Performance-Based Animation. In *GI '12: Proceedings of Graphics Interface 2012*, 2012.

Nadir Weibel, Adam Fouse, Edwin Hutchins, and James D. Hollan. Supporting an Integrated Paper-Digital Workflow for Observational Research. In *IUI '11: Proceedings of the international conference on Intelligent user interfaces*, pages 257–266, 2011.

Nadir Weibel, Adam Fouse, Colleen Emmenegger, Whitney Friedman, Edwin Hutchins, and James Hollan. Digital Pen and Paper Practices in Observational Research. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1331–1340, 2012.

Moritz Wittenhagen. DragonEye—Fast Object Tracking and Camera Motion Estimation. Diploma Thesis, RWTH Aachen University, 2008.

A T T Ying, G C Murphy, R Ng, and M C Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

YoungSeok Yoon, Brad A. Myers, and Sebon Koo. Visualization of Fine-grained Code Change History. In *VL/HCC '13: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 119–126, 2013.

T Zimmermann, S Diehl, and A Zeller. How History Justifies System Architecture (or Not). In *Proceedings of the international Workshop on Principles of Software Evolution*, 2003.

T Zimmermann, P Weibgerber, S Diehl, and A Zeller. Mining Version Histories to Guide Software Changes. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 563–572, 2004.

Typeset November 2, 2016