Zoomable User Interfaces: Communicating on a Canvas

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 Leonhard Karl Wilhelm Lichtschlag

aus Köln, Deutschland

Berichter: Professor Dr. Jan Borchers
Senior Researcher Stéphane Huot, PhD

Tag der mündlichen Prüfung: 27. November 2015

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, January 2016
Leonhard Lichtschlag

Contents

Abstract xix

Überblick xxi

List of Publications xxiii

Acknowledgements xxv

Conventions xxvii

1 Introduction 1

2 The World on a Canvas 7

2.1 Information Visualization 8

2.1.1 Design Space of Spatial User Interfaces 9

2.1.2 Overview + Detail 10

2.1.3 Focus + Context 12

2.1.4 Zoomable User Interfaces 15

2.1.5 Cue 16

2.1.6 Transparency 17

2.1.7 Fragmentation and Continuity of the Information Landscape 18

2.1.8 Review of the Canvas Design Space . 20

2.2 Evolution of Zoomable User Interfaces and Important Milestones 22

2.2.1 Pad: The Original 22

2.2.2 Portals 23

2.2.3 Semantic Zooming 24

2.2.4 Pad++: Metaphor-free Navigation . . 25

2.2.5 Multi-Level Interaction 26

2.2.6 Early Adaptation: KidPad 27

2.2.7 Limited Application Domains 28

2.2.8 Ubiquitous Use Today 30

2.3 User Interaction of Zoomable User Interfaces 31

2.3.1 Speed Dependent Automatic Zooming 33

2.3.2 Editing Interactions 35

2.4 Implementing Zoomable User Interfaces . . . 36

2.5 Promise of Zoomable User Interfaces 38

2.5.1 What We Already Know about Zoomable User Interfaces 39

2.6 Research Questions for Zoomable User Interfaces 41

2.6.1 Domain Studies 41

2.6.2 Authoring 41
2.6.3 Scaling of Text 42

3 Presenting on a Canvas 45

3.1 The Task and the User 46
3.1.1 History 48
3.1.2 The Speaker 50
3.1.3 The Author 52
3.1.4 The Audience 55
3.1.5 The Reviewer 56
3.1.6 User Base 57
3.2 Slideware 58
3.2.1 Blame for PowerPoint 58
3.2.2 Blame for Authors 60
3.2.3 No Significant Difference 60
3.2.4 Analysis From an HCI Perspective . . 61
Content Cutting 63
Time Dominance 63
Detail Trap 65
3.3 Canvas Presentation Software 66
3.3.1 CounterPoint 67
3.3.2 Fly 69

3.3.3 Prezi 73
3.3.4 DragonFly 74
3.3.5 Comparing and Contrasting 76
3.4 Studies 77
3.4.1 Authoring Lab Studies 78
More Connected Layouts 79
More Diverse Layouts 80
Incremental Revealing 83
Discussion 83
3.4.2 Investigating the Author in the Field . 85
Study Method 86
Observations 86
Layout Strategies 87
Overviews 88
Zooming 89
Rotation 90
Discussion 90
3.4.3 Investigating the Presenter 91
Study Design 93
Evaluation 96
Discussion 100
Limitations 100

3.4.4 Investigating the Audience 101

Study Design 102

Study Results 106

Discussion 108

3.5 Outlook 109

4 The Code Base on a Canvas 111

4.1 The Task and the User 112

4.1.1 Physical Analogy 113

4.1.2 The Relation Between Writing and Coding 115

4.1.3 Why is Reasoning About Code So Hard? 116

4.1.4 Cognitive Models of Software Design 119

4.1.5 User Tasks 120

Authors of a Code Base 121

Navigation in a Code Base 124

Understanding a Code Base 125

Communicating with Team Members 129

4.2 Visualization Approaches to Improve IDEs . 130

4.2.1 Rich Documentation for Human Readers 131

4.2.2 Live Coding 131

4.2.3 Leveraging Programmer Activity . . . 132

4.2.4 Exploring the History of the Code Base 133
4.2.5 Linguistic Analysis 134
4.2.6 Visual Programming 134
4.2.7 Canvas Visualizations 135
4.3 Our Approach 136
4.3.1 CodeGestalt: Our Vocabulary Based Design 138
The CodeGestalt Prototype 140
The Tag Overlay 142
The Thematic Relations 143
Study 144
Quantitative Results 144
Qualitative Results 145
Discussion 146
4.3.2 CodeMixer: Our Design Patterns Based Design 147
Code Mixer Design 151
Study 153
Discussion 154
4.3.3 CodeGraffiti: Our Sketching Based Design 156
The CodeGraffiti Prototype 158
Navigation Study 163

Quantitative Results 166

Qualitative Results 167

Interview 169

5 Excursus: Writing on a Canvas **173**

5.1 Setup 174

5.2 Observations 175

6 Discussion **179**

6.1 What We Learned for Presentations 180

6.2 Next Directions for Presentations 182

6.3 Critique of Our Coding Designs 183

6.4 Next Directions for Sketching 187

6.5 Our Three User Roles Model 190

6.6 Outlook 192

6.7 ZUIs and the Canvas 193

6.8 Limitations 195

Bibliography **197**

List of Figures

1.1 A Canvas Overview of this Thesis Document 5

2.1 Focus and Context 9

2.2 Taxonomies 11

2.3 Visualization Techniques Examples 13

2.4 Our Modified Design Space 20

2.5 Pad 23

2.6 Geometric and Semantic Zooming 25

2.7 Multiple-level Interaction 27

2.8 Features of ZUI Applications 29

2.9 Speed Dependent Automatic Zooming 34

3.1 The Users and Tasks Involved with Presentations 48

3.2 Reported Authoring Activities 54

3.3 Canvas Presentation Tools 67

3.4 CounterPoint 68

3.5 Workflow of Canvas Presentations 69

3.6 Our Second Fly Iteration 70

3.7 Our Third Fly Iteration 71

3.8 Our Mobile Fly Implementation 72

3.9 Prezi 73

3.10 DragonFly 75

3.11 Studies Presented in the Chapter 3. 78

3.12 Fly Paper Prototype 79

3.13 Group Ordering on Slides and Canvas 80

3.14 Fly Layout with Three Groups 82

3.15 Two Common Designs of Canvas Layouts . . 82

3.16 Example of the Revealing Problem in Fly . . 84

3.17 Prezi Canvases with Decorative Layouts . . . 87

3.18 Two Prezi Canvases with Topic Area Structures 88

3.19 Incrementally Developed Presentation 89

3.20 Distribution for Layout Strategies. 89

3.21 Overview of the Presenter Study 94

3.22 SAM Results of the Presenter Study 97

3.23 SD Ratings of the Presenter Study 98

3.24 Audience Study Design 103

3.25 Presentations of the Audience Study 105

3.26 Quantitative Results of the Audience Study . 106

4.1 Big Picture Views for Physical Objects 113

4.2 Two Examples of Text Scaling 114

4.3 The Story of "All You Zombies" Visualized . 118

4.4 Canvas Layout of "Strudlhofstiege" 122

4.5 Model of Program Understanding 128

4.6 Diagram Rendered by CodeGestalt 140

4.7 CodeGestalt Class Box Element on the Canvas 141

4.8 CodeGestalt Tag Overlays and Thematic Relations 142

4.9 CodeGestalt Tag Overlay 143

4.10 CodeGestalt Study Examples 146

4.11 Codelets 150

4.12 CodeMixer Early Design 151

4.13 CodeMixer Final Design 152

4.14 CodeMixer Study Setup 153

4.15 The Two Concepts of CodeGraffiti 158

4.16 CodeGraffiti Mission Control View 160

4.17 CodeGraffiti Sketchbar View 161

4.18 Editing in the Sketchbar View 162

4.19 CodeGraffiti Study Sketch 164

4.20 Quantitative results 166

4.21 Histogram of Glances 167

5.1 Setup of the Writing Experiment 174

List of Tables

2.1 Common Input Mappings for ZUIs 32

3.1 The Different Canvas Tools 77

3.2 Qualitative Results of the Audience Study . . 107

Abstract

Zoomable user interfaces (ZUIs) are an interesting alternative to existing interfaces such as overview + detail designs. ZUIs lay out information on a unified information landscape, whereas overview + detail traditionally promotes fragmented information. The choice of interface paradigm not only influences how the information is presented but also how the software affords to be used. Existing studies of ZUIs have three limitations: They often investigate lab scenarios that do not necessarily translate well to usage domain scenarios. Many studies compare different ZUI designs, but do not compare them to an established domain baseline. Also, many studies focus on navigation tasks primarily and learning tasks secondarily, yet we also have to consider authors. On the side of design, ZUIs have an unaddressed problem of integrating textual content.

We present a model of author, navigator, and learner to investigate interfaces with. We then study ZUIs and traditional overview + detail user interfaces in two domains: presentation support and integrated development environments (IDEs). In the first, we design a ZUI presentation system, Fly, and find that especially authors explore more creative designs than with existing baseline software, while presenters and learners perform on par with the baseline. In the second domain, we explore three design options to project text to spatial information landscapes: vocabulary based, pattern language based, and hand-drawn sketching based designs. We finally study navigation with sketches compared to a traditional IDE and find it a viable alternative that is strongly adopted by the testers.

In summary, we explore the field of ZUIs further and present these contributions: We test ZUIs as the primary metaphor in two practical domains and find them in parts superior to the baseline. We provide studies that underline that one has to study multiple user roles with ZUIs to get a complete image of the benefits in practical applications. Additionally, we show the feasibility to abstract text with sketches, which opens a new design direction for IDEs.

Überblick

Skalierbare Benutzerschnittstellen (ZUIs) sind eine interessante Alternative zu bestehenden Benutzerschnittstellen mit separaten Übersicht- und Detailansichten. In ZUIs werden die Informationen zusammenhängend auf einer einzigen Informationslandschaft ausgebreitet, wogegen klassische Benutzerschnittstellen Informationen eher fragmentiert darstellen. Die Wahl einer Benutzerschnittstelle beeinflusst nicht nur den visuellen Eindruck, sondern auch wie sie von dem Anwender verwendet wird. Bisherige Studien an skalierbaren Benutzerschnittstellen haben mehrere Einschränkungen: Sie betrachten die Interaktion oft in Laborumgebungen und nicht mit realitätsnahen Aufgaben der Benutzer, sie vergleichen die Interaktion nicht mit einer anwendungsbezogenen Kontrollkondition, und sie betrachten kaum die Erstellung der Informationslandschaft. Des Weiteren ist es problematisch Textelemente zu skalieren und somit sind diese schwer in ZUIs einzubauen.

Wir schlagen vor die Benutzerschnittstellen mit drei Blickwinkeln zu betrachten: der eines Autors, der eines Navigators, und der eines Lernenden. Dann untersuchen wir ZUIs und anwendungsnahe Kontrollkonditionen in zwei Domänen: für Präsentationssysteme und für Integrierte Entwicklungsumgebungen (IDEs). Wir stellen ein skalierbares Präsentationssystem vor und schlussfolgern, dass Autoren vielfältigere Präsentationen erstellen als bei der Kontrollkondition, wogegen Navigatoren und Lernende ähnlich zur Kontrollkondition agieren. Für IDEs stellen wir drei Visualisierungen vor. Diese basieren jeweils auf dem Vokabular des Quelltextes, auf Entwurfsmustern, und auf handgezeichneten Skizzen. Mit diesen Visualisierungen betten wir Text in Informationslandschaften ein. In einer Studie mit handgezeichneten Skizzen stellen wir fest, dass diese eine echte Alternative für Navigatoren in IDEs darstellen.

Insgesamt liefern wir drei Beträge zu der Erforschung von ZUIs: Wir testen ZUIs in zwei Anwendungsdomänen und stellen einige Vorteile gegenüber den Kontrollkonditionen fest. Wir führen drei Blickwinkel ein und untermauern mit den Studien, dass diese Blickwinkel helfen das Benutzerverhalten genauer zu verstehen. Zuletzt zeigen wir, dass unser Ansatz mit handschriftlichen Skizzen geeignet ist, Text in Informationslandschaften einzubetten und dies eröffnet neue Gestaltungsmöglichkeiten für IDEs.

List of Publications

- Leonhard Lichtschlag, Philipp Wacker, Martina Ziefle, and Jan Borchers. *The Presenter Experience of Canvas Presentations*. In INTERACT 2015: Proceedings of the 15th IFIP TC.13 International Conference on Human-Computer Interaction, September 2015.
- Leonhard Lichtschlag, Lukas Spychalski, and Jan Borchers. *CodeGraffiti: Using Hand-drawn Sketches Connected to Code Bases in Navigation Tasks*. In VL/HCC 2014: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pages 65–68, July 2014.
- Leonhard Lichtschlag, Thomas Hess, Thorsten Karrer, and Jan Borchers. *Canvas Presentations in the Wild*. In CHI EA 2012: Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts, pages 537–540, May 2012.
- Leonhard Lichtschlag, Thomas Hess, Thorsten Karrer, and Jan Borchers. *Fly: Studying Recall, Macrostructure Understanding, and User Experience of Canvas Presentations*. In CHI 2012: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1307–1310, May 2012.
- 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 2011: Extended Abstracts of the CHI 2011 Conference on Human Factors in Computing Systems, pages 1357–1362, 2011.
- Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, Florian Heller, and Jan Borchers. *Pinstripe: Eyes-free Continuous Input on Interactive Clothing*. In CHI 2011: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1313–1322, Vancouver, Canada, May 2011.
- Leonhard Lichtschlag and Jan Borchers. *CodeGraffiti: Communication by Sketching for Pair Programming*. In UIST 2010: Adjunct Proceedings of the 23nd annual ACM symposium on User Interface Software and Technology, pages 439–440, New York, NY, October 2010.

- Thorsten Karrer, Moritz Wittenhagen, Leonhard Lichtschlag, and Jan Borchers. *ExamPen: How Digital Pen Technology Can Support Teachers and Examiners*. In CHI 2010: Workshop on Next Generation of HCI and Education, Atlanta, USA, April 2010.
- Leonhard Lichtschlag, Thorsten Karrer, and Jan Borchers. *Fly: a Tool to Author Planar Presentations*. In CHI 2009: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 547–556, Boston, MA, USA, April 2009.
- Leonhard Lichtschlag. *Fly: An Organic Authoring Tool for Presentations*. Diploma thesis, RWTH Aachen University, Aachen, November 2008.

Acknowledgements

While there is only one author on the front page of this document, I would not have been able to finish it, if I had not been supported so generously. Bettina, Michael, and Karl are my greatest supporters, and I will have more time for you from now on, I promise!

Prof. Dr. Jan Borchers helped me at all stages of my path to become a researcher. He first got me interested in Human Computer Interaction with the DIS lecture series and encouraged me to pursue a PhD thesis topic. He provided invaluable support in all our publications, from improving my argumentation to relentlessly pointing out that I have no idea how comma rules work. I also want to thank Prof. Stéphane Huot for supporting this thesis as a co-advisor.

My colleagues and co-authors at the media computing group are exceptional and I detail their contributions to this thesis in the respective chapters. Above all, I want to thank Florian Heller, Thorsten Karrer, Jan-Peter Krämer, Maximilian Möllers, Chatchavan Wacharamanotham, and Moritz Wittenhagen for the invaluable feedback and discussions. Without them, this would be a lesser work. Many student projects helped shape and realize my ideas—I am still astonished that Claude Bemtgen, Christian Corsten, Thomas Heß, Christopher Kurtz, Torben Schulz, Lukas Spychalski, Ardi Tjandra, and Phillip Wacker have trusted me enough to throw in their luck with me. Thank you for trusting me and thank you for all your efforts.

All work in HCI relies studying software with potential users. This is a very time-consuming process that is all to often not adequately rewarded. I am very thankful to the time and effort poured into this thesis by the testers of the user studies. Their names remain anonymous, but I have not forgotten them. It is a great gift to get the time, the resources, and the opportunity by RWTH Aachen University to freely pursue the topics such as this one. The media computing group is a special place to work at—and to stay for a late night tabletop session after. It is this great community that convinced me that this is the right place to plant my roots for a while. I can only hope that I have adequately returned the favor.

Conventions

Some of the software and studies described in this thesis has been previously published by me and by students that worked under my instruction—I detail these instances at the beginning of each chapter. In cases where material was as a peer-reviewed paper at a conference, I give preference to this citation.

I use the singular “they” as the gender unspecific pronoun. Technical terms are set in *italics*.

Chapter 1

Introduction

"I haven't worked a two-dimensional control panel in a long time...how did we manage...?"
"We always seemed to muddle through somehow..."

—Julian Bashir talking to Jadzia Jax
Star Trek DS9, Episode 4x03, "The Visitor"

The quote above comes from a story in which characters travel back in time and have to interact with (to their eyes) old computer interfaces. Luckily, nobody sends us back in time. Most of us have already experienced stark changes in how one interacts with computers. Even though the history of human computer interaction is rather short compared to other disciplines, such as architecture or manufacturing, changes in human computer interaction are plentiful. The dominant interaction style changed from written interaction on terminals to windows/icons/menus/pointer interaction around 1990 and once again began changing to touch interaction around 2007. Even younger computer users find themselves in comparable situations: would not today's students be alienated by 1980 style command line interfaces?[1] And small children are often learning touch interaction before 'traditional' mouse and keyboard input. Human computer interaction researchers search for im-

[1] We invite the reader to search for the respective tests on video sharing websites.

provements to user interfaces, challenge our current working environments, and look for alternatives. We hope that in challenging the established interaction paradigms we find alternatives that will make us look back on today's interfaces similarly to the characters in the story. In this thesis, we document the application of one such challenge to two user domains, both of which are closely related to everyday working environments of knowledge workers.

We investigate zoomable user interfaces.

To investigate this challenge we have to be aware of the metaphors and paradigms that shape our interfaces today and compare them to other metaphors that could provide alternatives. The paradigm we work with in this thesis is called *zoomable user interfaces* (ZUIs) and was put forward by Perlin and Fox [1993] to reform graphical user interfaces (GUIs) in 1993. As the name implies, this metaphor presents an environment that can be zoomed in and out as the user moves a camera over the information landscape. We detail the interaction mechanics in depth in the next chapter. The reader might be irritated, that we refer to such an old metaphor—surely its pros and cons have been investigated in depth already? Also, since we do not see widespread adoption, can we not conclude that it does not provide sufficient benefits to warrant adoption? Both of these questions turn out to be unanswered for now. Critical changes in user interfaces have often had a long incubation time before they were adopted widely. The most prominent examples are digital mice ([Engelbart and English, 1968]), adopted with ~14 years delay) and touch interaction ([Buxton, 2010], widely adopted with a ~37 years delay with the iPhone in 2007). As Buxton [2010] details in great length, it can take a long time for innovations to reach the market and practical applications.[2] There is no reason to count Zoomable User Interfaces out on the basis of age. Instead, the problems it addresses are still ever present in today's interfaces and in recent years, the ZUI paradigm and its related *canvas paradigm* have been applied in some domains and challenges the status quo. We discuss zoomable user interfaces, the canvas metaphor, their roots, and their rich history in chapter 2 "The World on a Canvas". On the ques-

ZUIs are old, but many questions are unanswered.

[2] If you doubt him, William Buxton will show you a smartwatch with touch input from 1984 [Casio, 1984], that was 31 years before the introduction of the Apple Watch this year.

tion of studies, not all promises of zoomable user interfaces have been investigated and this thesis does precisely this in the following areas: The first application in the area of of *presentation support* started in [2001]. We present iterations on the design and an in-depth study from many angles with our own prototypes. The second application in an *integrated development environment* (IDE) is is a design exploration by us and investigates first steps in applying canvas ideas to this new domain.

In 2009, we investigated [Lichtschlag et al., 2009] the question whether a canvas interface could be an alternative to the status quo in presentation support software: Microsoft PowerPoint. Unhappy with the (to our eyes) form-filling nature of slide show authoring, we found the canvas metaphor to hold a great opportunity to thinking visually about arguments in a talk and to visually structure a presentation. Previously, we restrict the investigation to users authoring a presentation aid before a talk. This left questions open, that now this thesis is tackling in chapter 3 "Presenting on a Canvas": In particular, we are investigating other actors—presenters and audiences—and their interaction with the canvas metaphor. With the aforementioned move to touch interactions, we also have a look at interactions on tablet devices for the presenter during a talk. Also, in the timespan of this thesis, the research field has moved quite a bit: the canvas metaphor has moved out of the lab environment to commercial software and thus allows us to confirm our previous findings by investigating real world use. Chapter 3 "Presenting on a Canvas" presents our first contribution: iterative designs and a thorough evaluation of the canvas format in a specific field where it had already time to mature have market success.

Previously, we investigated presentation document authors.

Now, we investigate presenters and audiences.

In the next chapter of this thesis, we take the canvas design and apply it to another domain that complements (quite self-servingly) the essential software environment of a computer scientist: writing, and in particular writing source code. When investigating an existing source code base or planning an implementation, it has always been our first approach to sketch and visualize the inner workings of the software in hand written form and to present them visually. Likewise, the reader will be familiar with the sight

of whiteboards in a software developer's office on which they draft outlines of a source base or present their understanding of an existing source base to a colleague. That behavior is completely unsupported by current IDEs, so it is a good opportunity to bring the ideas that improved our communication from presentation software to source code software: with our prototypes, it is possible to build a canvas landscapes that are connected to source code elements. We design three prototypes based on hand-drawn sketches, on software design patterns, and on the vocabulary used in the code base. Our designs are not the only research direction for more canvas layouts of source bases: several approaches take on the old workflow of reading and editing source code line by line and provide reference points in a developing field. Comparing them to our prototype, we explore future directions with decidedly different working environments for software developers. Our contribution in chapter 4 "The Code Base on a Canvas" focusses on different design directions for a new field of canvas designs and includes a study to use our sketching prototype for navigation.

We investigate canvas interfaces for software developers with three prototypes.

Armed with the results from our application of the canvas metaphor to the two domains mentioned above, we can qualify the promises of zoomable user interfaces in chapter 6 "Discussion". We present the results broken down with our model of three usage scenarios: authoring, navigating, and understanding. Users actively engage with the metaphor and authors construct more diverse layouts. The findings for navigation and audiences are more complicated. We analyze our designs for abstraction of source code and find a combination of design patterns and sketching to be most promising for the next design iteration. Following that, we appropriated our sketching prototype to write this thesis and imagined it on a canvas from the beginning. See figure 1.1 for a high level overview of this thesis as we see it. We tell the whole story of how this canvas is constructed in the penultimate chapter 5 "Excursus: Writing on a Canvas".

We appropriated our sketching prototype to write this thesis.

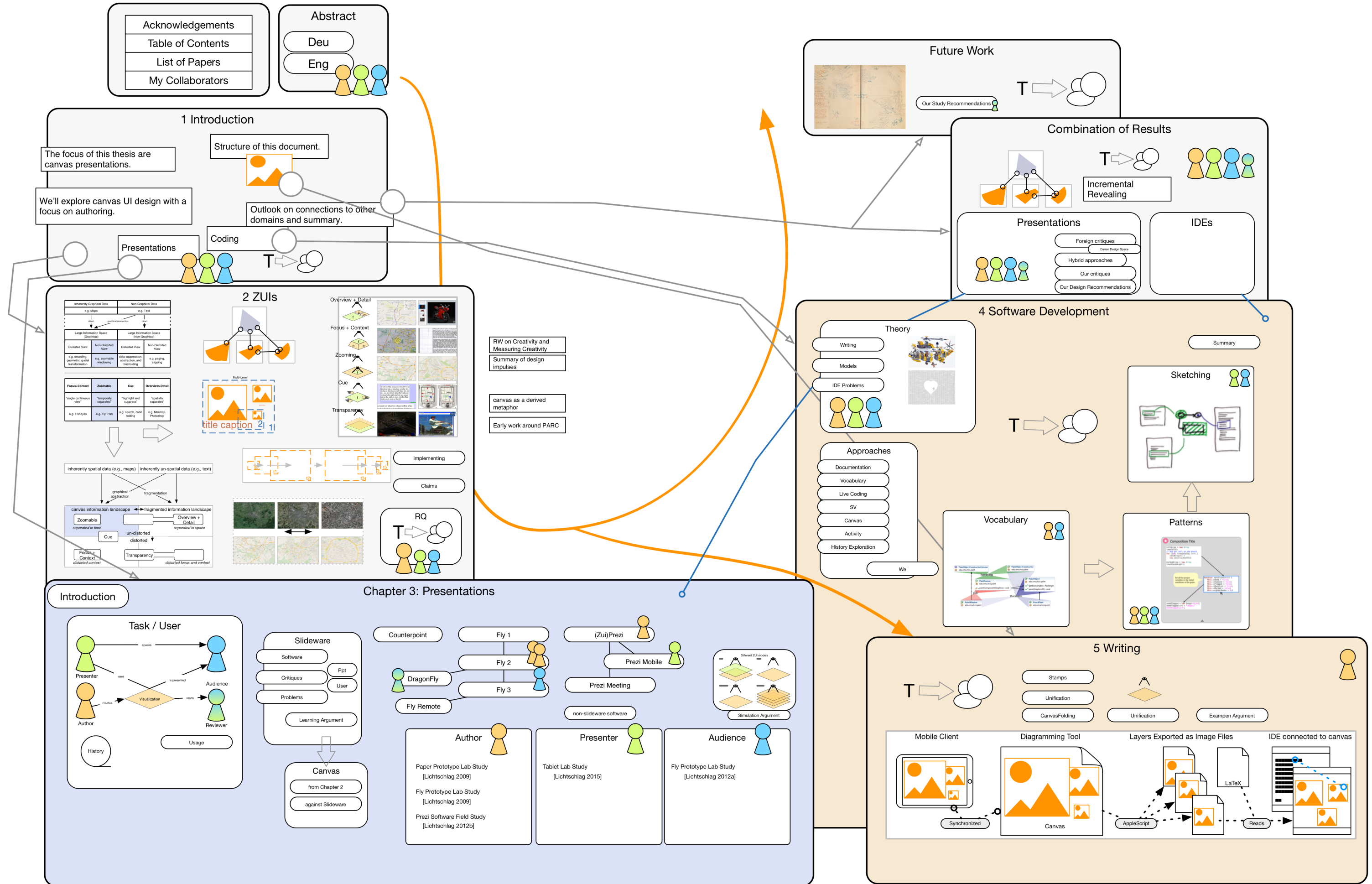


Figure 1.1: A canvas overview of this thesis document as created during the writing process. We detail how we designed this canvas in chapter 5 "Excursus: Writing on a Canvas".

Chapter 2

The World on a Canvas

> *"My imagination makes me human and makes me a fool; it gives me all the world and exiles me from it."*
>
> *—Ursula Le Guin, Unlocking the Air and Other Stories, 2005*

In the introduction we used a couple of terms—*zoomable user interfaces* (ZUIs), *canvas interfaces*—and we have yet fill these terms with meaning. These terms are not invented by us and they have a rich history. We have to understand this background before we can get to our applications of ZUIs in the following chapters.

As alluded to in the introduction, ZUIs are quite old with the first prototype presented in [1993]. But, it is likely that the reader came into contact with ZUIs only much later, e.g., with Google Maps in [2005], when fast server to client turnarounds allowed for interactive responsiveness. Today, ZUI applications and the their interaction style are widespread where ever there is a touch interface, e.g., in mobile interaction. In this chapter, we also look at the evolution of zoomable user interfaces from the first prototypes to today's interactions, how one would go about implementing them today, and how the interaction styles changed over time.

In this chapter, we present the background, the history, and the studies on zoomable user interfaces.

Of course, with such a long history, researchers have pursued variants for ZUIs. Likewise, different domains warranted further developments from the original idea. We explore a design space of spatial interfaces and how canvas interfaces and ZUIs fit into that picture. With this design space we have the foundation to place our prototypes and studies in relation to other research and related work. We then see what makes them particular and how they improve our understanding.

Many claims have been made about the usefulness of ZUIs for organization and learning, e.g., "[...] the concept is very natural since it mimics the way we continually manage to find things by giving everything a physical place." [Perlin and Fox, 1993]. Over time, researchers have investigated some of the claims about ZUIs usefulness and usability, we have a look at their results in turn. And as it is so often the case, a close look at the studies is most important: our summary and a summary by Bederson [2011] show that more studies are needed. We define a model of segmenting the user tasks into three fields: authoring, navigating, learning. This model gives us a framework to guide our studies in the field of presentations, and outlines how this thesis contributes by filling some of the blanks in the space of studies on user tasks.

2.1 Information Visualization

Many visualization techniques address the need to display a subset of content.

One of the most important information visualization problems is that there is more information to be presented than fits on the screen. This fundamental problem is here to stay: modern displays may have very high resolutions, but new devices with ever smaller screens are developed all the time. Even a hypothetical super high resolution, the human eye sight remains limited to resolve roughly 1mm at a 25cm distance, and our field of vision is limited.[1]The problem of

[1]A current tablet (iPad4) has a screen with a 2048 x 1536 resolution at a 25cm physical diagonal. This is clearly a multitude of the screen resolutions that ZUIs were first built for, yet the physical form factor is very comparable. On the other end of the spectrum, very large wall sized displays, e.g., Nancel et al. [2011] still run into similar issues.

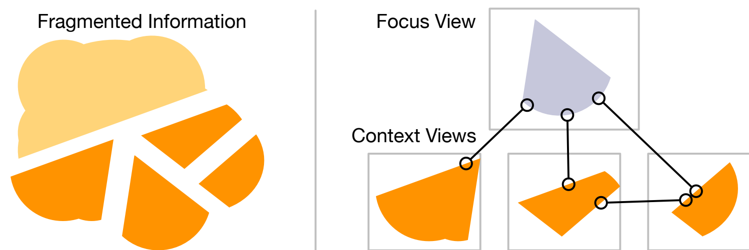


Figure 2.1: Selecting a subset of information to display fragments the information. The focus view shows the selected subset, the context view(s) can show related elements and reconstruct a part of the landscape.

limited space remains and will remain current for many applications areas, not just the ones presented in this thesis.

Solutions to the problem of limited screen space employed in Human Computer Interaction (HCI) include the selection of subsets of information, transformation the information so that is fits in the view, or spreading content over time. HCI research labels the views on the data as *focus views* and *context views* (cf. figure 2.1). The focus is a look at a (high-resolution) detail of information, the context represents connected information and embeds the focus in the environment. Often, research in information visualization defers the investigation of user interaction [Yi et al., 2007]. Clearly, this is a stance that Human Computer Interaction researchers cannot take. And we have a wealth of interactions at hand to switch between these subsets and representing them in the original context (cf. figure 2.1). Below, we have a look at taxonomies of information visualization techniques by Leung and Apperley [1994] and by Cockburn et al. [2008]. These two taxonomies present different strategies to present focus and context in conjunction.

Visualizations have focus and context views.

We also have to consider how one interacts with these visualizations.

2.1.1 Design Space of Spatial User Interfaces

In 1994, Leung and Apperley [1994] presented their “Review and Taxonomy of Distortion-Oriented Presented Techniques”. They summarized the approaches that had

emerged in the graphical user interfaces and especially focussed on the distortion techniques that had been brought forward by Furnas [1986]'s seminal paper "Generalized Fisheye Views". Their taxonomy (cf. figure 2.2, top) separates the visualization techniques in four quadrants along two questions: the first question divides the underlying data into inherently graphical and non-graphical, the second question divides into distorted and non-distorted techniques. They describe the graphical solutions as a dynamic selection on a continuous information landscape (e.g., fisheyes, ZUIs) and the non-graphical solutions as static separations (e.g., paging).

Two summary papers outline the approaches.

In 2007, Cockburn et al. [2008] presented their "A Review of Overview+Detail, Zooming, and Focus+Context Interfaces". This design space summarizes their experiences with research in spatial interfaces, especially zoomable interfaces. Their design space is close to the one by Leung and Apperley [1994] as they also organize systems into four categories according "varying uses of space, time or visual effect" (cf. figure 2.2, bottom). Three of Cockburn at al.'s categories closely mirror Leung's quadrants and describe similar solutions: *focus + context*, *zoomable*, and *overview + detail*. They also name *cue* as a technique to hint at contextual elements. Below we have a closer look at the techniques.

Both papers agree on many approaches.

HCI authority Shneiderman has also published on visualization techniques [Shneiderman, 1996]. He formulates an common guideline for approaching Information Visualization with regards to usability. He argues for a "Visual Information-Seeking Mantra": first, display an overview, secondly, zoom and filter, and then, display details on demand. Again we see the same building blocks of different views employed to display information in relation to its context.

2.1.2 Overview + Detail

The *overview + detail* design is the most straight forward approach—it just shows two separate views for focus

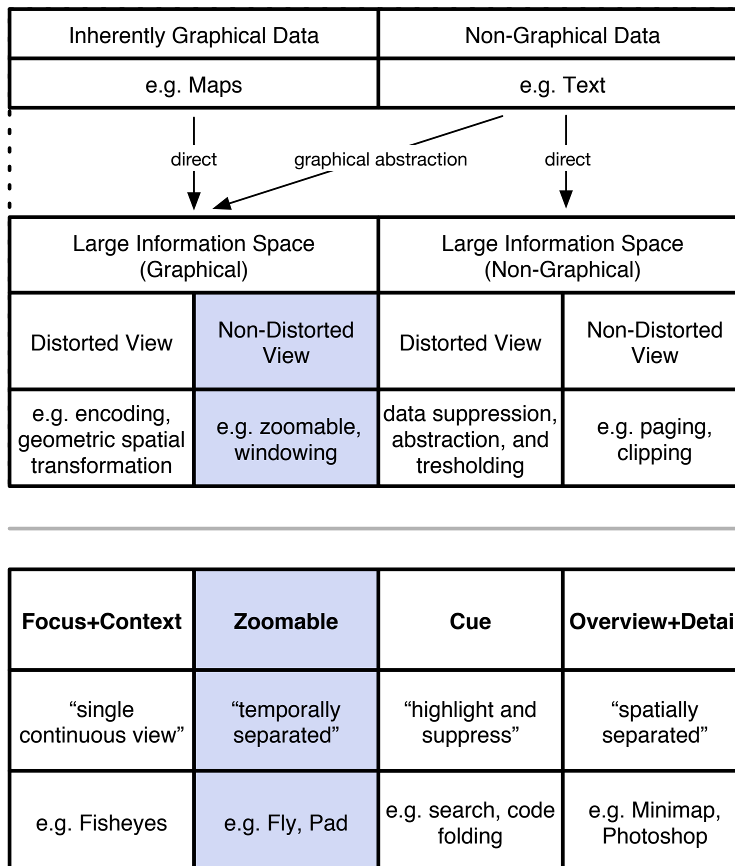


Figure 2.2: Top: taxonomy by Leung and Apperley [1994] (recreated figure). **Bottom:** categories described by Cockburn et al. [2008]

and context each: "An overview + detail interface design is characterized by the simultaneous display of both an overview and detailed view of an information space, each in a distinct presentation space." [Cockburn et al., 2008]. These two views are then either shown sequentially or in parallel, and they are *spatially separated* (cf. figure 2.3, top row). They also have no direct spatial relation—the context view might be placed to the right or to the left of the focus without change in meaning. Often the overview view gives the user the possibility to select a focus and gives feedback on the selected focus. Leung and Apperley [1994] refer to this with pagination techniques for non-graphical data and windowing for graphical data.

Overview + detail separates focus and context spatially.

Figure 2.3 (top row) shows two examples from commonplace software. The first example shows a map interface from Bing Maps [Microsoft, 2010] with a large view on the inner city of Aachen in focus and in the top left, a context view on the surrounding regions. The context view also highlights the selected focus region. The second example show a slide deck in Apple Keynote [Apple, 2003]: here the overview shows that the second slide out of a collection of five is selected and viewed in detail on the right side in the focus view. The overview + detail design is prevalent in all types of desktop software that has any kind of thumbnail views. Another example is a navigator window that shows the whole document in a graphics software (e.g., Adobe Photoshop) while the user edits the document in the main view.

Overview + detail can be used for continuous and for fragmented information spaces.

Here we can already note that the two examples presented are different from each other: by nature of the data, the maps shows the overview as a single continuous display, while the presentation software segments the data into slides or pages. Leung and Apperley [1994] made this distinction by separating the data into inherently graphical and non-graphical. We discuss this distinction further in section 2.1.7 "Fragmentation and Continuity of the Information Landscape".

2.1.3 Focus + Context

Furnas [1986] developed the idea of "Generalized Fisheye Views" on textual data and influenced many researchers to adopt this idea to visualizations. A series of prototypes was developed at Xerox Parc Research trying different strategies of distortion to show focus and context at the same time in a single continuous view [Mackinlay et al., 1991, Rao and Card, 1994]. Card et al. [1999] defines the technique as follows: "Focus + context starts from three premises: first, the user needs both overview (context) and detail information (focus) simultaneously. Second, information needed in the overview may be different from that needed in detail. Third, these two types of information can be combined within a single (dynamic) display, much as in human

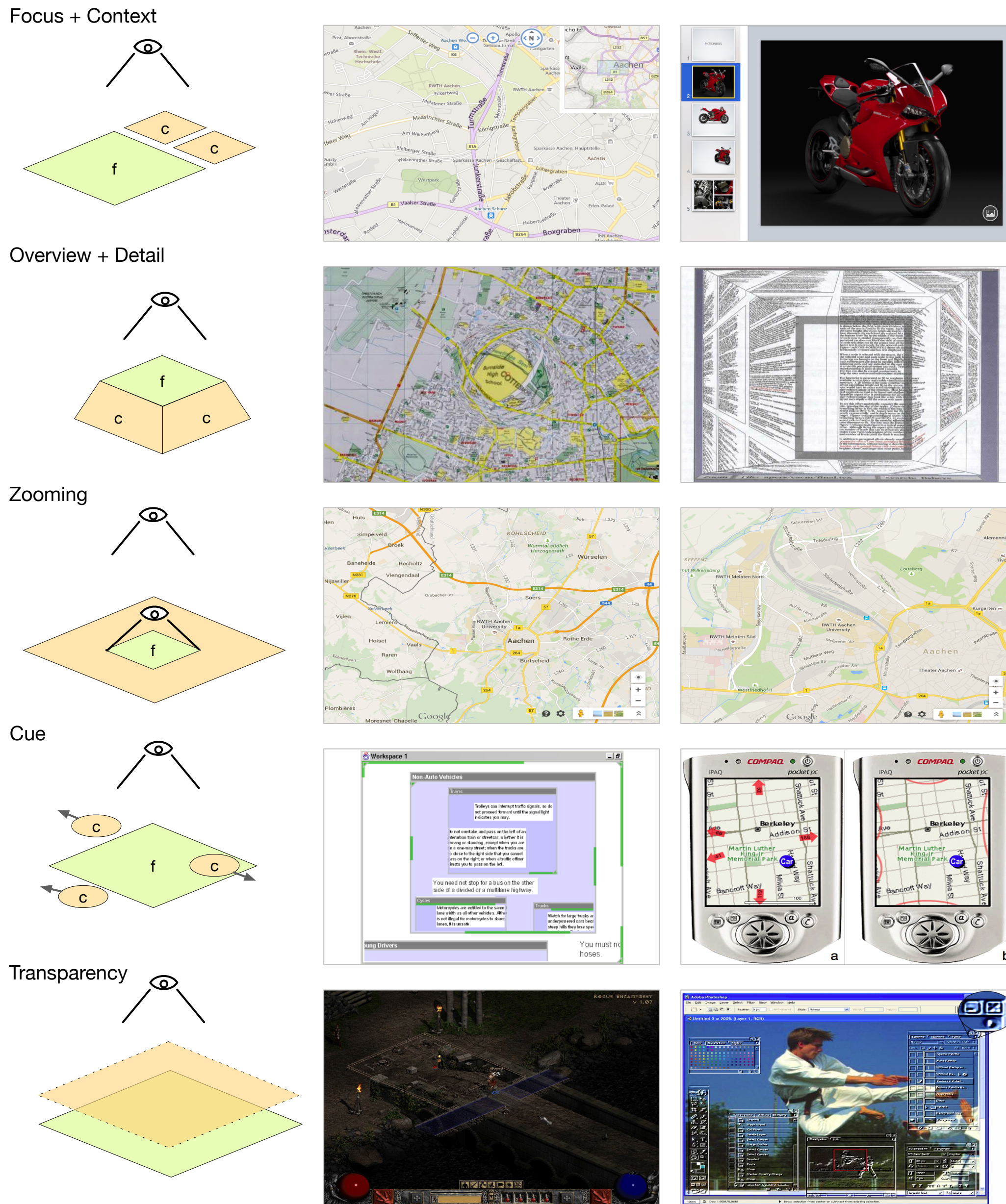


Figure 2.3: Examples for the visualization techniques described in the related work. Left schematic: f denotes a focus view, c context views.

vision.". Leung and Apperley [1994] refer to this design as "distorted" in both the graphical and non-graphical domain.

Focus + context integrates focus and context and distorts the context.

The *focus + context* design distorts the information landscape, so that the focus region is embedded into the overview. The focus region reproduces data accurately in detail, while concessions are made for the overview region to fit it to the display. Often this is done by applying a physical metaphor like a fisheye lens effect or the perspective rendering of a wall. This is sharp reproduction in the center is often compared to the human eye-sight. Other non-graphical approaches are closer to Furnas's original paper: they drop some of the information, e.g. code folding [Jakobsen and Hornbæk, 2009] or accordion menus[2], yet still keep the overall form of the overview intact. Leung and Apperley [1994] refers to this as "data suppression".

Figure 2.3 (second row) shows two examples. The first example again shows a mapping application, but this time a fisheye effect is applied to show the detail as selected by the user. The second examples shows a research prototype from [Mackinlay et al., 1991]: here the reader views a large written document which pages are laid out in a six by three grid. Again the view is distorted, so that the display can show all pages simultaneously and give the reader a sense of place in the document. The focus region remains undistorted and renders the text large enough to be readable.

Occurrences of focus + context designs are rare in consumer software: the Apple Dock comes into mind and the aforementioned code folding, but these are rare exceptions. Also, common user interface toolkits do seldom provide widgets for this type of interaction, so that programmers cannot easily integrate this into their software (e.g., jQuery UI[3] features an accordion widget).

[2] https://jqueryui.com/accordion/
[3] https://jqueryui.com/

2.1.4 Zoomable User Interfaces

Zoomable user interfaces also present content that has a fixed position in a single unified two-dimensional world—a spatial arrangement of information.[4] Zoomable User Interfaces use zooming to select a window to present a dynamic subset of information (cf. figure 2.3). The size of the viewport depends on the amount zoomed in: At the highest zoom level ,one can see the whole information landscape at the same time. At the lowest level, one can see only a single information element or only a part of it. Bederson [2011] defines "ZUIs to be those systems that support the spatial organization of and navigation among multiple documents or visual objects". The reader will note that this definition neither refers to zooming as the navigation mechanism, nor differentiates against other techniques with a single spatial information landscape.

Cockburn et al. [2008] define ZUIs to display information that is "temporally separated". That means that ZUIs are simpler than the other approaches as there is only one undistorted viewport at the same time. But, it also means that all switches between focus and context cost time and need a user interaction. Another way to view this is to say that focus and context are separated through user interaction.[5]

Zooming designs separate focus and context through time and interaction.

A prominent example given by Cockburn et al. [2005] is Google Earth [Google, 2001]: Here one can zoom into geographic information of the world. Depending of zoom level the camera is either a distant observer in space or very close and focussed at street level. To move the camera to another focussed place on the globe, e.g., from Paris to New York, the user would zoom far out to the level of continents, pan westwards until the American east coast is in the view and

[4] ZUIs have also been called "2.5D" to denote that information can change its shape based on zoom level. More on that in 2.2.3 "Semantic Zooming". This is unrelated, however, to 3D visualizations with sprites that always orient themselves towards the camera, which is also often referred to as "2.5D".

[5] In practice, software often mixes visualization approaches, e.g., in Photoshop one can both zoom and pan (ZUI interaction), but also open a smaller window with an overview map (overview + detail interaction).

then zoom in again. Would the user only use panning actions she would have a hard time to do that only on the focussed level, yet at the zoomed out level, the interaction can be very fast as the panning is amplified by the zoom level.

Panning and zooming are both needed in ZUIs.

We can already see from the examples that the panning is in contemporary ZUIs equal in importance to zooming. The term zoomable user interface has stuck to define the whole combined interaction. We suspect the reason is historical, as the first approaches had no panning interaction. The first mention we found of edge panning interaction was in Druin et al. [1997], an early application of ZUIs. We investigate the multitude of interactions that a user might be faced with below.

Bederson [2011] says that there were three core ideas in the original Pad prototype: zooming, semantic zooming, portals. We have presented zooming as the standout feature and naming feature, we discuss semantic zooming in 2.2.3 "Semantic Zooming" and portals below in 2.2.2 "Portals", as they are not universally adapted in all ZUIs.

2.1.5 Cue

Cue design presents proxies to off-screen objects.

The idea for cue designs is to reserve all of the screen for the focus view and only "[...] introduce proxies for objects that might not be expected to appear in the display at all." [Cockburn et al., 2005]. These proxies are rendered on the edges of the focus view, hinting at direction and distance of the context objects and maybe some of their characteristics. While Cockburn et al. [2005] gives only graphical examples, his description of "highlighting and suppressing" is very similar to Leung and Apperley [1994] describing distorted views on non-graphical data. Yet, for direction and distance to be meaningfully incorporated into the cue display, there has to exist a distance measure in an information landscape.

Figure 2.3 (fourth row) shows two examples, both research prototypes. The first shows 'City Lights' [Zellweger et al., 2003], where bars at the edge of the focus view hint at con-

textual items. The second again shows a mapping application and highlights the main motivation to use this technique: small screens. The Halo prototype [Baudisch and Rosenholtz, 2003] uses arcs or arrows to indicate search results on a map, yet the whole screen remains available for the immediate focus display and interaction.

Cue based techniques are seldomly used in desktop software, yet at times used in map-based games to indicate off-screen events. Only some of the implementations refer to user interaction with the off screen proxies. Again, no implementation support exists in common user interface toolkits.

Cue designs are mostly non-interactive.

2.1.6 Transparency

The transparency design is not classified by Leung et al. or Cockburn et al., even more, hardly referenced ever in the HCI literature. Cox et al. [1998] is the only paper that we are aware of that investigates this pattern. Here, both the context and the focus view exist at the same time and are rendered in the same place (cf. figure 2.3, bottom row). The context view is composited transparently on top of the focus, each pixel displays a 50/50 mixture of both views to the user.

Transparent compositing of focus and context views is very rare.

One commercial approach we are aware of is in map-based gaming, e.g., in the game Diablo II [Entertainment, 2000] where the map of the level is can be permanently represented as a thin overlay on the game view (cf. figure 2.3, last row). Here, the focus view takes over all of the screen, and a grey schematic overlay of the map gives the position in the level. Sometimes, the overlay is reduced to full transparent in the immediate surrounding of the player character as to not falsify the information at the user's locus of attention. Also, the second example from [Baudisch and Gutwin, 2004] gives an example on how one can use hard blending with the contours of the overview to keep the color information of the focus intact, albeit at the cost of a small distortion.

Picking up on Cockburn et al. [2008]'s naming scheme, one could say that focus and context are *not separated* at all. This comes at the cost of clarity and true representation of the data, similar to the drawback of distortion in focus + context techniques. The model also has similarities to overview + detail, as one could see it as a variation where the two views just happen to be rendered in the same place.

The user only interacts with the focus layer.

As each position on the screen has not clear attribution to either focus or context, it is not clear how a user would discern between them in pointing interactions. In the limited instances that we are aware of this technique, no user interaction is possible with the overview layer, all interaction is always on the focus layer.

2.1.7 Fragmentation and Continuity of the Information Landscape

Clearly, the aforementioned design spaces do not include all types of visualization. We are missing 3D interfaces, linear interfaces, augmented reality approaches, virtual reality approaches, and so much more. Why should we limit the discussion to the ones above, and why are these techniques the ones that are mentioned in overview papers?

All of the approaches present information landscapes.

All of the approaches have in common that they present an "information landscape" (cf. [Robertson and Mackinlay, 1993]). This landscape is two dimensional in nature, each item in the space has a defined two dimensional position. This position is either fundamental in the nature of the data (e.g., maps) or is used to convey meaning and relation between the items (e.g., a mind map). This landscape model adapts well to most interfaces, not only two dimensional screens, but also to tables, walls, flip boards, etc.—all two-dimensional areas to present information on. All of the design approaches detailed above therefore strive to keep this two-dimensional information intact, although they might deform and stretch it.

The aforementioned taxonomies of Cockburn et al. [2005] and Leung and Apperley [1994] did not include explic-

itly but alluded to a differentiation between designs that present a single landscape of a multitude of landscapes. Leung and Apperley [1994] touch this by differentiating between geometric and non-geometric data, when they discuss designs for non geometric data and use paging as an example with a fragmented space. Also, Cockburn et al. [2005] refer to overview + detail examples from PowerPoint, a design that features information fragmented into multiple slides. An interesting observation is that neither the zooming nor the cue examples use a fragmented landscape and we are not aware of a zoomable design that fragments its data.

We have to differentiate between fragmented and continuous landscapes.

This now finally brings us be to the title of this chapter: "The World on a Canvas". We first came across the term *canvas* for unfragmented interfaces in PreziMeeting [Laufer et al., 2011] (where it referred to an adoption of their zoomable presentation software to note taking in meetings). We liked this term because it moves the discussion from the technology to a human centered approach. Talking about these interfaces as canvas casts them in the image of a surface to be creative on. Is not canvas a much better term to talk about creative interface like image manipulation, vector graphics, and—yes—-presentations? A canvas asks the user to build, draw, sketch, design something on it.

The term canvas describes continuous landscapes.

Below we see that the act of authoring on canvas interfaces (as opposed to traditional, fragmented interfaces) is underexplored. And since this is a main study focus of this thesis, this differentiation is very important. Therefore, we use the name *canvas* from now on to refer to designs of a single continuous information landscape, differentiating against fragmented designs. With this definition, canvas includes map designs, no matter if they are build with a zoomable user interface, a overview + detail interface, or a mixture of both. Now, it is opportune to examine if the design space can be recast with that distinction in mind.

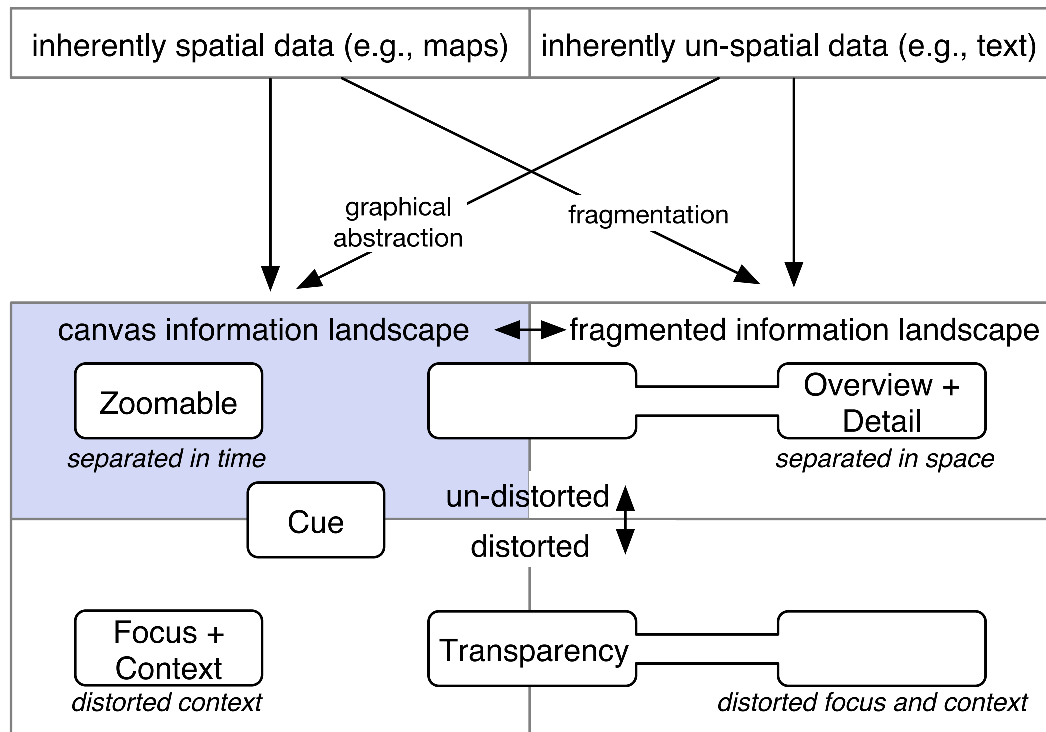


Figure 2.4: We modified Leung and Apperley [1994]'s design space. Our modified space differentiates between canvas and fragmented designs and leaves some designs ambiguous depending on their implementation.

2.1.8 Review of the Canvas Design Space

We differentiate between fragmented and canvas designs.

We first kept Leung and Apperley [1994]'s design space with the the differentiation between canvas designs and fragmented designs, and then inserted the approaches as categorized by Cockburn et al. [2005], adding the rarely used transparency approach. Zoomable, focus + context, and cue designs need the focus to be spatially embedded in the context to build the visualization. Thus, they cannot be fragmented and are on the left side of the space. As we have seen before, overview + detail can be used in both fragmented (e.g., paginated) information landscapes but also in continuous information spaces. In most of the applications it is applied to fragmented or paginated information spaces. Similarly, transparency designs lend themselves to both fragmented and canvas spaces, since focus and con-

text are not embedded there. The few examples we found were all canvas designs. Thus, we placed both designs on the border, with a weight on right and left, respectively.

As for distortion, the separation is straight forward for all but cue and transparency techniques, as we have discussed before. Since cues abstract the context similarly, we placed it on the border between distorted and undistorted. Both the rendering techniques presented for the transparency design falsify part of the information. Accordingly, we placed it firmly in the distorted lower half of the design space. To be precise, here both focus and context are distorted, where the focus + context design only falsifies the context (albeit often quite strongly).

We differentiate between distorted and undistorted designs.

Many of the designs are often mixed in practice. E.g., many graphics manipulation software (e.g., Photoshop) present the canvas in both an overview + detail and a zoomable interaction. Similarly, our Fly prototypes use both zooming and cues (and even overview + detail for the presenter view). Also, in chapter 4 "The Code Base on a Canvas" we visit some prototypes of information landscapes in IDEs and see that all approaches can be used there. We can summarize that it would be unrepresentative of actual practice to say that a software belongs to only one of these techniques. Rather, the border between the approaches is very much fluid and allows for mixtures.

Often, designs are mixed.

We have to consider the data we want to present: does it already come with inherent spatial mappings or not? Consequently, matching designs to the type of data can be straight forward. As Leung and Apperley [1994] pointed out, one can consider projecting un-spatial data in a graphical space. And, one can also consider the reverse: fragmenting a spatial information into pieces.

We consider two transformations: projecting un-spatial data in a graphical space and fragmenting spatial information.

The investigations in the following two chapters can be seen as investigations of these two transitions and the distinction between fragmented and canvas designs. First, in chapter 3 "Presenting on a Canvas", we investigate designs to display information on a canvas where traditional software fragments it. We then study these designs comparing them to the traditional fragmented software and estimate

This thesis investigates these transformations.

the costs and benefits of the transition. Second, in chapter 4 "The Code Base on a Canvas", we investigate how one can graphically abstract from inherently un-spatial data (text) and build a information landscape. Here, traditional software represents the information in its textual form and keeps it fragmented, and we explore canvas designs that allow alternatives.

With this modified design space, we have a way to describe and categorize the software mentioned in the next chapters and understand the different approaches in a unified matter. But, since zoomable user interfaces are such an important part of the canvas designs in the coming chapters, we first have a detailed look at important milestones in zooming interaction below.

2.2 Evolution of Zoomable User Interfaces and Important Milestones

In the following, we present a couple of seminal ZUIs with a focus on the studies that they accompanied and the differences in interaction styles they brought. When opportune, we delve deeper into some of the lower level issues and techniques of zoomable user interfaces as they arise. For a detailed account of the history of zoomable interfaces up to 2011, we recommend the study of an overview paper [Bederson, 2011] by one of the principal researchers of ZUIs.

2.2.1 Pad: The Original

Pad is the original zoomable user interface.

No investigation of zoomable user interfaces is complete without mentioning Pad [Perlin and Fox, 1993], the paper and prototype that introduced the idea. Pad is the original ZUI implementation, although they never actually referred to their implementation as a *zoomable user interface*. They write: "A good approximation to the ideal depicted would be to provide ourselves with some sort of system of 'magic magnifying glasses' through which we can read,

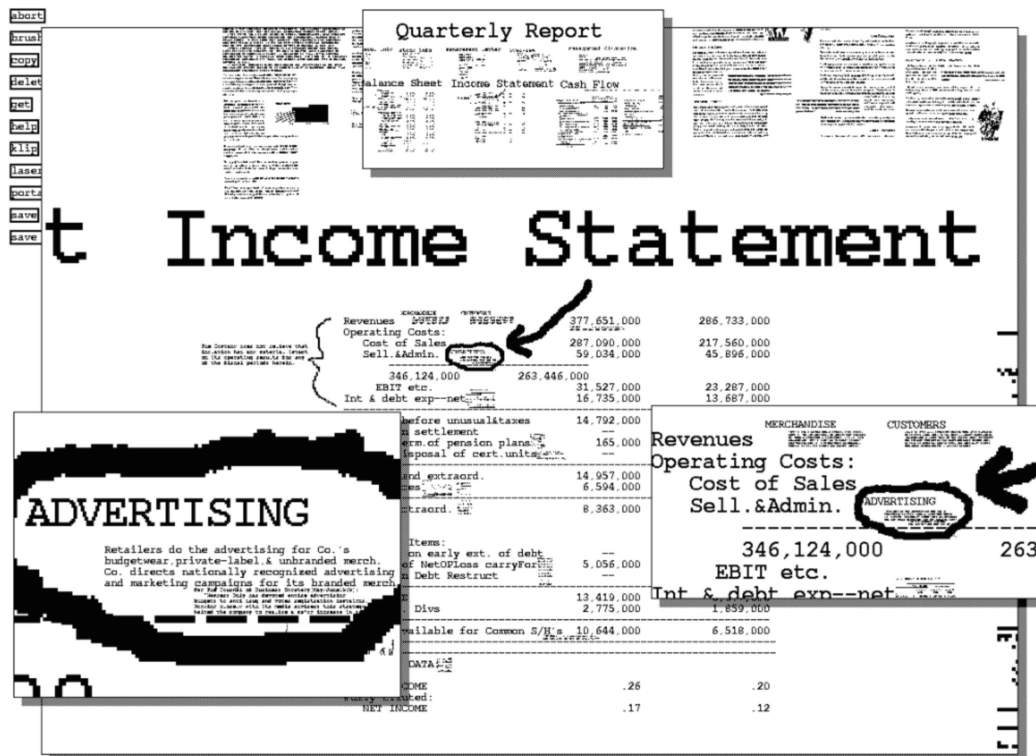


Figure 2.5: The earliest ZUI by Perlin and Fox [1993]. We see a view on the canvas with four portals: one for the main canvas and three smaller viewports.

write, or create cross-references on an indefinitely enlargeable ('zoomable') surface." Additionally, they present two new ideas—*semantic zooming* and *portals*—which we describe below. Figure 2.5 shows a view on the canvas with three portal filters overlaying the canvas. Zooming and jumps with the portals are the ways the user can navigate. Due to limitations in graphics power, none of these navigations are animated and all visuals are monochrome.

Pad introduces semantic zooming and portals.

2.2.2 Portals

Portals are introduced in Pad and allow the user to have another viewport on the canvas. Figure 2.5 shows how one can see the whole canvas in the top portal, and a very small

detail on 'Advertising' in the left portal. With these portals, the user can view and interact with objects of varying size at the same time, and overcome the different levels of magnification. Perlin and Fox [1993] also defines portal filters that change the appearance of objects in its view: "For example, a portal may show all objects that contain tabular data as a bar chart [...]". With portal filters, the user can toggle a mode on the displayed data, e.g., enable editing.

Portals open views to different camera positions on the canvas.

Interestingly, the portal technique is not picked up by the following zoomable user interfaces, not even by the direct successor Pad++ [Bederson and Hollan, 1994]. Bederson [2011] writes: "[N]one of these applications that focus on zooming as a core organizational and navigation technique use portals in a way similar to how they were envisioned." We suspect this is because they complicate the user interaction compared to a system with a single viewport and because all following systems allowed fluidly animated zooming, thus allowing the user to overcome different levels of magnification easier through zooming.

2.2.3 Semantic Zooming

Again, *semantic zooming* is introduced in the original seminal Pad paper [Perlin and Fox, 1993]: "As the magnification of an object changes, the user generally finds it useful to see different types of information about that object. For example, when a text document is small on the screen the user may only want to see its title. As the object is magnified, this may be augmented by a short summary or outline. At some point the, entire text is revealed. We call this semantic zooming.". This is an extremely helpful design technique to build zoomable user interfaces and is widely adopted. Its benefits are clear when compared to *geometric zooming* (cf. figure 2.6). Geometric zooming treats the items in an "information landscape" purely physical, but semantic zooming allows the objects to change their shape, visibility, and representation according to the distance to the viewer. Perlin and Fox [1993] also envision semantic zooming to enable mode on the displayed data, e.g., zoom in to edit.

Semantic zooming transforms the data as the camera moves.

Figure 2.6: Geometric and semantic zooming applied to the same region in Google Maps [Google, 2005]. Geometric zooming directly scales the satellite image as the camera moves closer. Semantic zooming shows different content on different camera distances. Smaller streets are only revealed in close up, text is rendered at legible size in all views.

Figure 2.5 shows how text is especially limited by geometric zooming: most of the elements on the canvas are much to small to be readable. Their position and shape still convey some meaning, but the much of it seems just noisy. We build our canvas design for presentations in part with semantic zooming because titles get transparent on close zoom and bullet points get blurred far away. The whole point of our design is to abstract references to source code to scalable representations to support storytelling of source code.

2.2.4 Pad++: Metaphor-free Navigation

Pad++'s design philosophy is a *physical user interface* as opposed to a metaphor based user interface. Bederson and Hollan [1994] write "Our goal is to move beyond mimicking the mechanisms of earlier media and start to more fully exploit the radical new mechanisms that computation provides [...]". Founding the design in physics promises ben-

Pad++ presented zoomable user interfaces as an antithesis to metaphor based user interfaces.

efits: familiarity, scalability, but most of all, no reliance on old metaphors. They write: "Any restrictions that are imposed on the behaviors of the entities of the interface to avoid violations of the initial metaphor are potential restrictions of functionality that may have been employed to better support the users' tasks and allow the interface to continue to evolve along with the users increasing competency.". We certainly felt restricted by the slide model of mainstream presentation systems and agree wholeheartedly.

We used the examples of maps to introduce the reader to different visualization strategies. There are metaphors hidden in the model of a 'information landscape' and 'canvas' design directly alludes to a physical canvas on an easel. We can accept that because it promotes the kind of interaction we want to empower in the user. Attaching physical behavior to digital items cannot be really free of an allegory to a non-digital element. Bederson and Hollan [1994] clarify: "There are certainly metaphorical aspects associated with a physics-based strategy. Our point is not that metaphors are not useful but that they may restrict the range of interfaces we consider."

2.2.5 Multi-Level Interaction

Multi-level ZUIs allow content elements to be presented at different scales.

Bederson [2011] introduces the quality *multi-level* to differentiate between objects that appear on the canvas at significantly different sizes, even though they might be of comparable importance. This requires a user to navigate between different zoom levels in order compare these objects. And objects may appear in the current view at ill-suited sizes (cf. figure 2.2.5 "Multi-Level Interaction")—this can happen very easily in presentation documents. This is an instance where a portal could overcome this problem. But some ZUIs system avoid the whole problem by adopting a single level strategy. E.g., Fly [Lichtschlag et al., 2009] is single-level because every object is either a title or a content element, but Prezi [Prezi, 2008] and CounterPoint [Good and Bederson, 2001] are multi-level. Our study in chapter 3.4.2 "Investigating the Author in the Field" looks at the

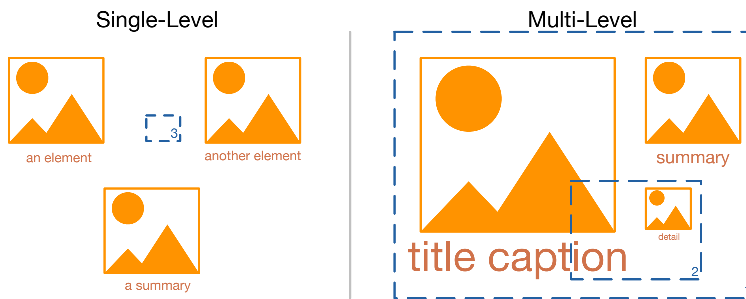


Figure 2.7: ZUIs can be differentiated by whether they allow content to be laid out over multiple levels. With content on multiple levels, content can easily be of inappropriate size to the current viewport. Viewport 1 shows only a tiny scaling of the detail, viewport 2 shows a text fragment from the title. Viewport 3 is within the bounds of the scene but too close to show any content ("desert fog").

typical amount of levels used in Prezi presentations.

2.2.6 Early Adaptation: KidPad

KidPad [Druin et al., 1997] is an early adoption of the zoomable interaction paradigm to a practical application. Young children (aged 8–10) use it at school to train storytelling skills. This is considered a key metric in education research to investigate children's cognitive, social, and emotional development [Boltman, 2001]. In participatory design with the children, Druin et al. [1997] develop the first mention of panning in ZUIs and a tool palette called *Local Tools* [Bederson et al., 1996] that directly embeds editing control into the canvas as zoomable elements.

KidPad is an early ZUI for children to train storytelling skills.

Boltman [2001] picks up on the KidPad investigation and performs a controlled experiment in which 72 children in England and Sweden use either pictures in KidPad, pictures on paper, or an un-animated ZUI of the same material as reference material to tell stories. Boltman [2001] concludes: "Results illustrated that the spatial computer presentation assisted in many storytelling areas, with greater benefits in elaboration than in recall. Children's stories

KidPad was evaluated in a large scale study.

showed more complex story structure and a greater understanding of initiating events and goals." We see that it is interesting to contrast the role of the storyteller (presenter) and learner (audience) when investigating the influence of spatial capabilities of canvas designs. The studies do not investigate authoring in KidPad.

2.2.7 Limited Application Domains

ZUIs are rarely used for window systems.

Perlin and Fox [1993] and Bederson and Hollan [1994] envisioned zoomable user interfaces to be full replacements for *window managers*, with the potential to augment windows/icons/menus/pointers (WIMP, [Wikipedia, 2015a]) graphical user interfaces in general. E.g., [Bederson and Hollan, 1994] describes a directory browser in Pad++ and "Raskin" [Raskin, 2015] is a window system replacement product available today. On mobile devices, zooming techniques are adapted to launch applications, e.g., the springboard on iOS9 can be seen as a single-level canvas and users zoom in and out of applications.

Many approaches use presentations as a lens to study ZUIs.

Presentation support software is one of the domains we study in this thesis and has received considerable attention with research prototypes. Notable examples from the literature are Good and Bederson [2001]'s CounterPoint, our Fly [Lichtschlag et al., 2009], Prezi [Prezi, 2008], and Microsoft's pptPlex [Microsoft, 2008]. Of course, we discuss them in more detail in the next chapter. Bederson [2011] reports on the characteristics of early zoomable user interfaces in table 2.8 which we amended with systems important to the discussion in this theses.

Although (up to now) ZUIs have seen a much stronger representation in software than distortion interfaces, they did not replace window managers as the primary interaction metaphor or widely bring 'physical' interaction to user interfaces. Bederson [2011] writes: "I would argue that ZUIs have never reached the level of broad use envisioned by their original creators." but also "Zooming has been successful in that some kind of zooming is commonly used in a wide range of interfaces".

Year	Name	Description	Layout Flexibility	Mult-level	Navigation Mechanism For Zooming	Citation
1993	*Pad*	*Original formulation of ZUIs*	Unconstrained. Drawings, images, text positioned manually.	*Yes*	*Mouse Buttons or Portals*	*[Perlin and Fox,1993]*
1994	*Pad++*	*Poses ZUIs as an alternative to metaphor based User Interfaces.*	Unconstrained. Drawings, images, text positioned manually.	*Yes*	*Middle button zooms in, right button zooms out or right button + movement direction*	*[Bederson and Hollan,1994]*
1997	KidPad	Uses "local tools" to enable children to create stories.	Unconstrained. Drawings, images, text positioned manually.	Yes	Hyperlinks for path. Multiple Zoom in/out modes. Click to fly in/out.	[Druin et al., 1997]
1998	PadPrints	Creates a visual hierarchical map of web pages visited.	Dynamically generated tree-based node-link diagram	No	Right click-and-hold flies in/out.	[Hightower et al., 1998]
2001	CounterPoint	A plug-in to PowerPoint that enables presentation authors to lay out slides in zoomable space.	Contains PowerPoint slides. Small number of layout algorithms wth manual override.	Yes	Click on slide for next. Right click-and-hold flies in/out.	[Good and Bederson, 2001]
2001	PhotoMesa	A personal photo browser, desktop and mobile.	Contains photos in a dynamically generated group of grids using a treemap algorithm.	No	Left click zooms in. Right click zooms out.	[Bederson, 2001]
2002	Seadragon	Client/server high perf. ZUI for images over a network (web and mobile)	Contains photos laid out by a small number of fixed layout algorithms.	No	Click to zoom in. On-screen button zooms out.	[Microsoft Seadragon. 2002]
2002	Spacetree	Hierarchy exploration tool	Dynamically generated tree-based node-link diagram	No	Right click-and-hold flies in/out.	[Plaisant et al. 2002]
2006	Dynapad	Supports organization of personal collections.	Dynamic, very flexible.	No	Multi-touch	[Bauer, 2006]
2007	Apple iPhone	Application launcher for Apple's mobile phone.	Contains icons laid out in a fixed grid.	No	Tap to zoom in. Physical button to zoom out.	[Apple, 2007]
2008	Microsoft pptPlex	A plug-in to PowerPoint that enables presentation authors to lay out slides in zoomable space.	Contains PowerPoint slides. Small number of layout algorithms with manual override.	No, but with manual override	Click on slide for next. Click to zoom in. Esc to zoom out. Scroll wheel flies in/out.	[Microsoft, 2008]
2008	Prezi	Website for creating and making free-form presentations.	Unconstrained. Drawings, images, movies, text positioned manually.	Yes	Tab for next. Scroll wheel, on-screen buttons and mouse/keyboard combo to fly in/out.	[Prezi, 2008]
2009	*Fly*	*Application for creating and making free-form presentations.*	*Unconstrained. Images, movies, text positioned manually.*	*No*	*Mousewheel*	*[Lichtschlag 2009]*
2009	Microsoft Canvas for OneNote	Plug-in for Microsoft OneNote shows visual overview of all a user's notes.	Dynamically generated grid of grids containing images of notes from Microsoft OneNote.	No, but with manual override	Click on note to zoom in. Esc, right click, or background click to zoom out.	[Microsoft, 2009]
2010	*CodeCanvas*	*Example of a zoomable map in an IDE*	*Generated Layout is hand-tuned by user*	*No*	*unknown*	*[DeLine and Rowan, 2010]*
2014	*CodeGraffiti*	*Hand-drawn sketches in a zoomable map in an IDE*	*Contains images that are placed manually and annotated with hyperlinks.*	*No*	*Mousewheel*	*[Lichtschlag et al., 2014]*

Figure 2.8: Some ZUI applications and their traits. Reproduced from [Bederson, 2011] and amended with newer systems. Added systems in cursive.

2.2.8 Ubiquitous Use Today

ZUIs are often used for maps and map-based games, ...

The navigation of ZUIs is very much adopted in certain domains, even a standard user interface element. The first of these are *maps*. Map interaction in a zoomable user interface is almost obvious, maps and the metaphor directly reference landscapes. Similarly, games often use ZUI elements when they are map based, e.g., levels in a strategy game. Interestingly, the most sophisticated user interaction for navigation is found in games as well. Semantic zooming is widely used in maps and games, and interfaces are typically single-level since all objects reside on a 'ground'.

...in digital image manipulation, ...

Digital image manipulation is the second domain in which ZUIs are almost ubiquitous. Notably, both imaging and mapping use mostly geometric zooming (e.g., gray text is an exception, but the reasons to use it may only be performance in rendering). Digital imaging software has developed a model of selection frames, interactions with the corners of objects, etc. which is almost standardized and outliers easily irritate the user. Digital image manipulation is interesting, because here the authoring of the canvas landscape is the primary use case. Again, we see that the user role is key in investigating ZUIs.

...and especially touch systems.

The third domain with much use of ZUIs are *mobile devices*. As mentioned above, the iOS9 window manager has zoomable elements (cf. figure 2.8) and many applications implement zoomable navigation. Also, the calendar and photos applications on iOS9 zoom to content elements almost in the same fashion as outlined by [Perlin and Fox, 1993]. One could argue (cf. [Bederson, 2011]) that the touch interaction model brought ZUIs into commercial applications, because touch gestures are an almost natural mapping to zoom and pan. In mobile systems, space is very limited and ZUIs separation of focus and context through time allows the user to keep a conceptual model of the landscape even though they only see a small part. This is also the first area in that vendor supported user interface toolkits included zoomable widgets. Touch interaction on other larger devices (e.g., tables and wall) also often uses zooming interactions, which is probably again due to the natu-

ralness of mapping touch to zoom and pan gestures.

2.3 User Interaction of Zoomable User Interfaces

In our discussion up to now, we mostly focussed on the question on how to arrange the views and different techniques to present the information landscape across zoom levels. We have not discussed how a user navigates or modifies content in the landscape. This seems to be a second grade concern in the whole discussion in the literature, as most papers do not particularly dwell on the issue. The first paper [Perlin and Fox, 1993] just says "Pad objects receive events from the user's mouse and keyboard".

If we limit ourselves to navigation for a moment, how does the user exactly move the camera? Let us consider the possible actions: the user can *pan* the camera laterally, they can *zoom in and out*, they may *edit objects* in the scene, and very rarely systems allow *rotating the camera*. The straightforward way is to use on screen buttons (cf. figure 2.3, top). Similarly, keyboard mappings are mostly used for panning, sometimes the page up or down buttons zoom in or out. Almost all systems map mouse events directly to camera actions. And this is where the interaction becomes ambiguous.

ZUIs require mappings for navigation and editing actions.

Table 2.1 lists some of the input mappings we encountered in ZUIs. As one can see, there is no standard of interaction for theses systems. Each mouse buttons is used somewhere as a panning command and the user has to try out before they can be sure of the mappings. The mouse wheel, which may feel like a straight forward mapping to the camera zoom, is sometimes mapped to the vertical pan because this is the typical behavior for a system with a vertical scroll bar. Some systems map the rate of the camera zoom to the distance of drag with a middle mouse button. Clicking or double clicking the button to zoom is also problematic, because the user cannot know resulting amount of zoom. E.g., in Pad a click halves the zoom distance, but another

Mouse and keyboard mappings are often problematic because of lacking interaction standards.

	Input	Panning	Zooming	Note
Mouse	left click and drag	yes		Sometimes depends on the click point
	right click and drag	yes		
	middle click and drag	yes	yes	Sometimes rate based
	double click		yes	Sometimes only zoom in
	right button double click		yes	Sometimes only zoom in
	scroll wheel	yes	yes	
Multitouch	one finger drag	yes		
	two finger drag	yes	yes	confused with scrolling
	pinch gesture		yes	
	double tap		yes	second tap reverses action
	two finger double tap		yes	
Keyboard	cursor keys	yes	yes	used in Pad for zoom

Table 2.1: Table of common input mappings for standard input devices in ZUIs.

system may chose a different factor. The problem of clear input mappings is already mentioned in Bederson and Hollan [1994] and has not been standardized since.

But it gets worse when one considers the direction of change, since some systems have a model of 'moving the landscape' and some system have a model of 'moving the camera'. E.g., a simple mouse drag to the right can either drag the landscape to the right (and thus move the view to the left), or move the camera to the right (and thus move the view to the right). On Mac OS X, the default scroll direction is reversed for the y direction, which may make sense for a scrolling interaction, but causes problem when that is mapped to zooming. Even more, some systems map scrolling forward to 'moving the camera' closer, and some map the same gesture to moving the 'camera up' (since the user 'pushes the scroll wheel up'). Again, it is an unstandardized mess.

The reversal of mouse zooming actions is difficult to design.

A key problem with zooming with mice is that the action is hard to undo. Most systems zoom towards the location of the cursor, so that the element at the user's focus of attention stays under the cursor location, which is understandable as the user points on the position they want to zoom to. To reverse this zoom action, the cursor has to be placed

at the same location again. But this time the user can not make the connection with the location, and this can have disorienting effects as the camera begins to pan as part of the zoom. Another option is to make the camera focus the screen center during zoom out navigation, but then the actions are not reversing each other any more. This is a tricky decision to make as a system designer, and we have sometimes seen this as an option in the settings of the application.

Touch controls map naturally to zooming and panning.

The mappings are much better with touch controls. On touch devices like smartphones and tablets, the pinch gesture has always been mapped to the zoom/rotation interaction and there is not ambiguousness in the direction of the mapping. With touch controls on trackpads and multitouch mice, this clear mapping is now also available on desktop systems but less common. Panning gestures are less clear: the mount of fingers used for panning depends on the application. Apps that allow editing of canvas elements either reserve the single tap for editing of canvas elements or decide on the tap location. There are also interesting interaction techniques with post desktop interaction, e.g., Gummi [Schwesig et al., 2004].

If not designed carefully it is easy to get lost in a ZUI.

Bederson [2011] refers to another problem that he observed which he calls "desert fog" The information landscape may have 'gaps' between objects that show nothing, and when the user navigates there they may lose their orientation (cf. 2.7). This can be mitigated somewhat by limiting the navigation to the convex hull of all objects, but on systems with unrestricted zoom levels (cf. 2.2.5 "Multi-Level Interaction"), there are always such positions. Therefore, some systems (e.g., Prezi) include a patterned background that the user to orient themselves to.

2.3.1 Speed Dependent Automatic Zooming

Often, a ZUI software performs a camera movement over a set piece of time, e.g., when clicking a hyperlink or when advancing to the next stop in a presentation. When the focus view is at a very detailed level (that is, if the camera is

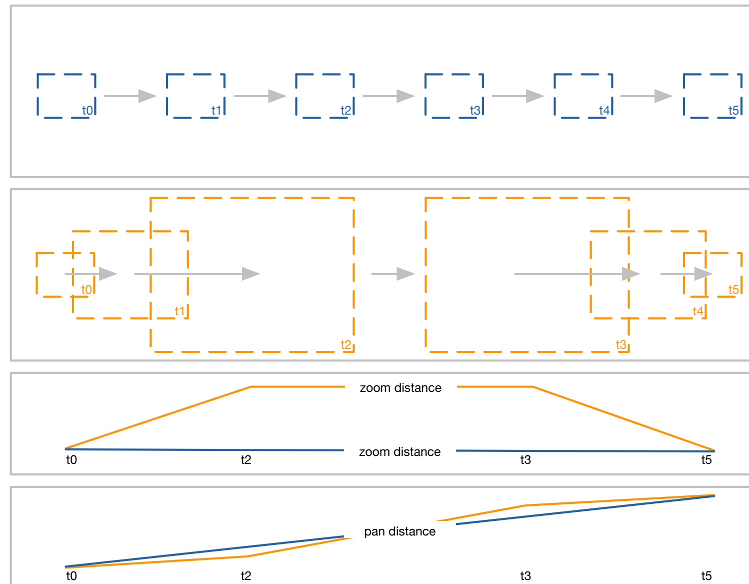


Figure 2.9: Speed dependent automatic zooming (SDAZ, orange) lifts the camera as part of the pan action. This is more pleasing to the viewer when compared to straightforward panning (blue), because the screen elements do not change so rapidly.

very close), a large panning movement can be quite jarring to the viewer. Figure 2.9 explains this problem: since the camera shows only small views on the information space, the contents scroll by very quickly over the time from t_0 to t_5.

Speed dependent automatic zooming automatically moves that camera out on large panning actions.

Speed dependent automatic zooming (or SDAZ) [Cockburn et al., 2005] provides a way around this problem by automatically lifting the camera up during the transition (cf. figure 2.9). To achieve this, the algorithm has to spend some of the available time to adapt the camera height, then move faster on the high level, and then again spend time to lower the camera before arriving at the destination. This way, the camera shows a bigger part of the landscape at the middle of the transition and even moves faster at the apex. Yet, the movement is more pleasing to the viewer as each part of the movement seems slower because more of the context is visible.

Cockburn et al. [2005] studies differentiates between implementations of automatic zooming and performs studies with users. The evaluation shows that users strongly favor SDAZ in navigation tasks and also perform faster. SDAZ ties the lateral and zoom degrees together, thus limiting degrees of freedom, but also simplifying the interaction by automatically choosing a pleasant path. Bederson [2011] argues that SDAZ is not good in precise and time dependent tasks, e.g., games. He argues that this is hindering the real time *perceive think act* loop [Card et al., 1983].

SDAZ tests well.

Overall, navigation in zoomable user interfaces has more freedoms and consequently can be considered more difficult than traditional user interfaces. This is especially apparent when compared to the pick-one-out-of-n style interaction of overview + detail designs, e.g., a discrete selection from folders. These drawbacks are somewhat mitigated with SDAZ and multitouch interaction, but the system designer still has to take care to implement clear and consistent mappings.

2.3.2 Editing Interactions

Many systems do not allow much editing in the scene (e.g., maps or games), but those that do mostly follow the standard interactions of graphical manipulations programs. The editing actions of Adobe Illustrator, Microsoft PowerPoint, and Prezi all allow multiple selections, dragging, resizing, etc. of elements with the primary mouse button. The single touch is mapped similarly on touch systems. This is a form of direct manipulation of the canvas elements [Hutchins et al., 1985]. Notable exceptions to this are early ZUI systems: the second iteration of KidPad [Druin et al., 1997] used *Local Tools* [Bederson et al., 1996], a tool palette that is integrated as objects on the canvas—a particularly metaphor heavy implementation.

Editing actions in ZUIs follow graphics manipulation software.

Interestingly, zooming almost never implies a mode on the authoring, as suggested by the original Pad paper (cf. chapter 2.2.3 "Semantic Zooming"). Semantic zooming often transforms the style for fidelity of the elements in the can-

vas, but does not change the way they are interacted with. For the systems in which is the case, one has to consider authoring the objects dependent on the zoom level. Furnas and Zhang [1998] presented the *Multi-scale Editor* (or MUSE) for such cases. But, as far as we are aware, no such implementation has made it to commercial applications.

2.4 Implementing Zoomable User Interfaces

While toolkits for classic windows/icons/menus/pointer (WIMP) widgets exist in many forms, toolkits that include ZUI elements are very rare. This is unsurprising since ZUI systems are much younger than the WIMP metaphor, and we can attribute that the rarer usage of zoomable designs originates in part in the lack of toolkit support. The first ZUI system, Pad, was only able to work by being powered by a predecessor to *OpenGL* from Silicon Graphics. In 1993, it had to run on dedicated hardware and yet was not able to achieve fluid animations during a zoom animation [Perlin and Fox, 1993]. Luckily, graphics powered rendering OpenGL and similar graphics accelerated hardware and drivers are ubiquitous today, and even a mobile device the size of the Apple Watch can now support full screen redraws animating with 60Hz. So, what options exist to implement a zoomable user interface today?

Few toolkits help the developer build ZUIs.

In common UI toolkits, ZUI interaction is still a second grade citizen but a few out of the box solutions exist, or other widgets can be appropriated with little effort. E.g., Apple's *Cocoa UIKit* does not contain a dedicated ZUI widget. The NSScrollView is what comes closest to a ZUI widget. It added scaling support with Mac OS X 10.8 in 2012 after *UIScrollview* on the iOS side (*CocoaTouch toolkit*) had supported this for 3 years. In using *NSScrollView*, the implementer still has to take care to make the zoom animate fluidly, rather than changing the zoom level instantaneously. In the absence of these (partial) solutions, canvas elements that allow for custom drawing commands are available in many toolkits, e.g., Pad++ used the *TclTk* Canvas Widget.

By nature of potentially changing all pixels on the screen during a redraw in a non-translation movement, ZUIs are computationally costly. A service that makes use of accelerated graphics is mandatory to achieve interactive results once the scene becomes a bit more complicated. Luckily, technologies like *Open GL*, *Vulcan*, *DirectX*, or other similar technologies are available on all platforms and even low end graphics hardware. With *shader language* support, it is also straightforward to build distortions for focus + context designs, e.g., Fly makes use of blurring for contextual views. But, a developer can also appropriate game engines, e.g., *SpriteKit* on Apple platforms or 3D engines with orthogonal projection, and thus leverage the low-level implementation of the engine developer.

Accelerated graphics enable ZUIs.

A problem in common with all these approaches is that they lack the feel of the widget set. Most of the technologies take care of presenting the view, but they do not help with input, e.g., entering text. The developer most likely needs to implement all the input facilities from basic events like mouse clicks and touch events. In Fly, we took advantage of our single scale design that allowed editing only at a particular scale. We then overlaid the scene with a standard system widget for editing text at the right zoom level. That allowed for a native feel without taking the user out of the zoomable scene, but this trick cannot be generalized to all ZUI interactions.

Input mappings must be recreated manually.

When planning to deploy on the web, the amount of technologies available is more limited, but the following building blocks are still fine options. *HTML5* includes a canvas element which allows drawing primitives, and *WebGL* allows to tap the graphics power of the hardware directly. With *cascading style sheets* (CSS3) many animation primitives are added that can be used to transform conventional HTML elements to ZUI building blocks.[6] And of course, there is *Adobe Flash*, which allows Adobe's multimedia platform to be hosted in the browser. The web and desktop versions of Prezi are implemented in Adobe Flash.

As only a part of the information landscape is rendered on

[6] See https://bartaz.github.io/impress.js/ for a particular impressive implementation. Last accessed April, 2015

the screen, some form of pruning is necessary and the developer has to plan for that with appropriate data structures. Structures have two (competing) desires: pruning objects in the scene to the needs of the current view and fast fetch of said objects. Again, one can leverage a data structures from graphics programming, where similar problems arise, e.g., a *space partitioning tree*. Jazz [Bederson et al., 2000] implements a framework for ZUIs and shows also how to introduce nodes into the graph that include semantic meaning, such as the current edited region or semantic zooming. Some platform technologies, e.g., *Code Animation* and *SpriteKit* on Apple platforms, allow to arrange objects in a tree structure, so that the engine takes care of some of the work. Should the designer want to include portals (cf. chapter 2.2.2 "Portals") this can be problematic because they can introduce cycles into the view hierarchy. In this case, one should limit the amount of times a portal can be rendered to ensure a timely halt of the rendering loop.

Data structures from graphics programming can be adapted to ZUIs.

2.5 Promise of Zoomable User Interfaces

Bederson [2011]'s paper "Promise of Zoomable User Interfaces" closes with a review and design guidelines based on his experience. He summarizes three promises of ZUIs: First, that "ZUIs are engaging", i.e., that users like the view them and helps them build a mental map. His opinion is that this holds, when ZUIs are designed well. Second, that "ZUIs are visually rich", i.e., that more degrees of freedom on the canvas allow more creative expression. His opinion is that this holds, but comes with the drawback of complexity in navigation and understanding. Third, that "ZUIs offer the lure of simplicity", i.e., that retrieval and organization are easier and he concludes this is not true for very large datasets. Bederson adds guidelines to use of ZUIs, including the recommendation to only use canvas layouts when they are meaningful for the user, to keep the landscape consistent over time, and to shy away from multilevel layouts. He also stresses that ZUIs must support a small representation of all canvas elements, noting that text and audio recordings are especially problematic to support in a ZUI.

Bederson concludes that the promise of ZUIs is only partially fulfilled.

A couple of claims have been put forward for ZUIs on the organizational qualities of ZUIs: Perlin and Fox [1993] write: "Pad is a good way to store documents with hierarchy and multiple narrative pathways." and "As compared to standard current window models, this system makes it easier for the user to exploit visual memory of places to organize informationally large workspaces". Bederson and Hollan [1994] follow: "The ability to make it easier and more intuitive to find specific information in large dataspaces is one of the central motivations for Pad++."

Claims also refer to the ability of audiences to understand content presented in ZUIs: Good and Bederson [2001] write: "This spatial layout may provide the audience with an additional attribute or memory pathway with which to recall the presentation content." and "[S]tructure is itself revealed to the audience during the normal course of the presentation." and "[A]nimation may improve long-term understanding of presented material." and "ZUIs facilitate a more spatial portrayal of hierarchies.", concluding: "However, future empirical studies are needed to verify these advantages.".

Claims have been made about spatial representation in ZUIs and how audiences and navigators benefit from them.

It is interesting to review the justification of these claims. The references to the spatial memory and structure do not exclude focus + context designs or overview + detail on a single continuous information landscape. One sees that while they are formulated for zoomable user interfaces, the theory that leads to this claims seldomly excludes the other canvas designs might have. We can therefore postulate that the defining feature that entices these promises is not the primary navigation metaphor of zooming, but rather the single continuous information landscape—the canvas.

Most of these claims could also be formed for other canvas designs with the same justifications.

2.5.1 What We Already Know about Zoomable User Interfaces

The studies on zoomable user interfaces have shed some light on some of these claims. We know that zooming costs time, since the animation itself takes time. The promise is that this cost is outweighed by savings due to better un-

derstanding or more targeted navigation actions. Hornbæk et al. [2002]'s studies found that contextual overview (as in an overview + detail design) result in slower performance, but higher user satisfaction. Cockburn et al. [2008] add to that a review of 'low-level evaluations' concerned with simple target acquisition times and 'high-level evaluations' closer to real world tasks. In their summary, no clear trend emerges on completion times, and that the benefit may depend on the domain. For ZUIs, they find that "There is also evidence that users can optimize their performance with zooming interfaces when concurrent, unimanual controls for pan and zoom are supported.".

Some studies find ZUIs beneficial for navigation.

On the question of understanding material that is presented in ZUIs, we have both positive and negative results. E.g., a study on storytelling by children with KidPad [Boltman, 2001] found that the KidPad condition children scored better for recall of their stories. On the other hand, Good [2003] found no statistically significant effect on audience recall. The meta review by Cockburn et al. [2008] concludes that zooming has benefits because "the spatial separation [between focus and context] demands that users assimilate the relationship between the concurrent displays of focus and context", yet also that "[t]he temporal separation of zooming also demands assimilation between pre- and post-zoom states". They note that animations are especially important for ZUI navigations to reduce the problem of reorientation after zooming and that animations are well worth their cost in time. Tentatively, there is an indication that ZUIs improve the recall of a structure [Bederson, 2001], yet not the content of canvas landscape.

Zooming is a cognitive burden for the audience that can be lessened with animation.

Studies indicate that ZUIs improve the recall of structure, but not of content.

When we look at domains and the claims about ZUIs, time and recall are only some of the outcomes that interest us. E.g., not only should presentations to be easy to understand, we also want them to be easy to conduct by the presenter and convincing and engaging for the audience. And we have little information on how they are authored. E.g., the time it takes to author one is important, but not as important as what kind of presentations a zoomable UI affords and how it influences how the author expresses structure of thought.

Many aspects of the authoring of canvas landscapes are unexplored.

In virtually all studies above users reported excitement for ZUIs, underlining Bederson's first promise above. In HCI studies, users often voice excitement for the new conditions, a form of novelty bias. Here, this seems to be a particularly strong feedback—ZUIs are simply exciting.

ZUIs are very engaging.

2.6 Research Questions for Zoomable User Interfaces

Bederson [2011] reviews studies on zoomable user interfaces and notes that most of them deal with navigation in lab environments, e.g., the different implementations for speed dependent zooming [Cockburn et al., 2005]. He asks for more studies considering how different users interact with ZUIs, better design with standardized controls, and an exploration of the design space considering a comparison to non-spatial ground truths. Bederson [2011] also specifically mentioned that ZUIs have problems scaling text elements (cf. chapter 2.1.8 "Review of the Canvas Design Space").

2.6.1 Domain Studies

Many of the early papers envisioned ZUIs to be a replacement for fundamental system services, e.g., replacing the window manager. Our studies in the 'mundane' application areas of presentation support and IDEs offer us an opportunity for more focussed designs with the needs of the users in that domains in mind. The existing software solutions in this domains are very standardized overview + detail interfaces and offer us a ground truth to evaluate against.

We study ZUIs in two application domains: presentation support software and IDEs.

2.6.2 Authoring

When considering ZUIs as a helpful technique to build a UI, the first studies that come to mind investigate naviga-

tion. And most of the studies mentioned above investigate this issue. Yet, authoring and learning has always been brought forward as an argument when talking about user interfaces that use ZUIs as a guiding concept for the user. The author is the role that interacts most with a ZUI, e.g., in the domain of presentations, audience only experiences the resulting animation and the presenter does not interact much with the document and their experience is much shorter.

We consider three main activities with ZUIs: authoring, navigating, learning.

We can rudimentary segment the user tasks into three fields: authoring, navigating, understanding. This model gives us a guide to our studies in the field of presentations, and outlines how this thesis contributes by filling some of the blanks in the space of studies on user tasks. We look at all three different user groups in chapter 3 "Presenting on a Canvas" and develop this model further.

Of these user tasks, authoring is almost not explored at all, with the notable exception of Good's dissertation [Good, 2003] and participatory design of KidPad [Druin et al., 1997]. Good [2003] found that benefits for organizing information over folder structures in a lab study with an abstract content. And we know from the KidPad study [Boltman, 2001] that the KidPad condition helped presenters ("strongly supported the creation of non-linear stories"). We can hope that ZUIs also help authors plan such stories. So this looks like an interesting field to study further with our presentation system to investigate how authors create content on a canvas and if we can see improvements in the resulting documents. We do this in chapter 3 "Presenting on a Canvas".

We study authoring with presentation support systems.

2.6.3 Scaling of Text

Another open question is the handling of non-graphical content in a ZUI. The primary problem involved with that is that it does not scale well geometrically: large text is irritating in a multi-level ZUI and small text becomes unreadable and is a poor representative of its content. For presentations this is not all that bad, as a certain dramatic effect is

desirable and text heavy presentations should be shunned anyway. But in other domains, such as writing or coding, this bars us from employing ZUI techniques.

We iterate on designs to abstract text with our IDE prototypes.

Good tried this with automatic text summarization in his Niagara prototype [Good, 2003] but did not report on an evaluation. Another way to approach the problem by also distorting the text is a reduction the content in another way to pick the lines that are the most important as done in fisheye code folding [Jakobsen and Hornbæk, 2009]. In chapter 4 "The Code Base on a Canvas", we investigate the issue further and present three different designs to deal with bodies of text and evaluate one that is based on hand-drawn sketches that project the text onto a canvas.

Chapter 3

Presenting on a Canvas

Power Corrupts.
PowerPoint Corrupts Absolutely.

—Edward Tufte, 2003

We set out to apply our canvas metaphor as a guiding design principle to the domain of presentation support software. We actually encountered the term *canvas* first when it was picked by Laufer et al. [2011] to describe the application of the ZUI metaphor in presentation support software. *Presentation support software* are a good opportunity to investigate the promises by ZUI proponents, because it deals with the core task of ZUIs: the visualization of information. We can investigate if indeed ZUIs present structure and hierarchy better and if indeed audiences can hope for improved learning effects. Also, presentations is one of the most ubiquitous and most frequent domains that computers are used for, that alone is reason enough to have a good look at them. We review data on their use in section 3.1 "The Task and the User".

Studying ZUIs in the realm of presentations is a natural fit.

From the critical analysis of the most prevalent product of *presentation support software*—Microsoft PowerPoint—and the type of software type it represents—*slideware*—we develop an analysis of its problems (3.2 "Slideware"). We then use this analysis to review how this guided our alternative design (originally presented by this author's diploma thesis

[Lichtschlag, 2008] and published as a paper [Lichtschlag et al., 2009]). We called that alternative *Fly* and its type of software *canvas presentations*. Our design builds on previous designs in alternative presentation software and limits the ZUI principles to a subset applicable to this domain. Furthermore, we present a design study on mobile use, and a design study of use for asynchronous reviewers, and we want to thank Claude Bemtgen [Bemtgen, 2012] and Christian Corsten [Corsten, 2009] in supporting these projects. With Fly, we use the greater freedom (compared to slideware) to empower all users of presentation support software: authors, presenters, audiences, and reviewers. Or so we hope—to find out if this really works is the job of the studies that follow.

We present an iteration of our original artifact contribution.

We quickly review the two studies from this author's diploma thesis [Lichtschlag, 2008, Lichtschlag et al., 2009] which dealt with the authoring scenario in two lab studies and found that the canvas condition allowed the author express a great diversity of designs. In this thesis, we contribute three new studies (section 3.4 "Studies") to this: we revisit the original scenario of authoring again, but this time use data from real world use and confirm our results. We then investigate the other two roles, namely the presenter and the audience. This part has been supported by Phillip Wacker [Wacker, 2014] and Thomas Hess [Hess, 2011] respectively, and previously been published at international conferences [Lichtschlag et al., 2012a,b, 2015]. Together, these five studies look user interaction with zoomable presentations from all angles.

We three additional studies so that we have an understanding of all user roles.

3.1 The Task and the User

First, we have to establish our model of whom we design for and what their task is—as it is good practice for work in the field of Human Computer Interaction [Dix and Finlay, 2004]. While probably all readers have held and heard many presentations, it is easy to forget tasks that may lie out of our personal experience.

The *presenter* is, of course, the first role that comes into

mind, and has historically received the most attention. If we have a closer look, we see that the reason to present can be diverse and has implications on the design of our software. Giving a presentation is only one aspect: a good presentation often takes many days to research, structure, plan, prototype and rehearse. It is this task of *authoring* a presentation that we focussed on in our first study. It may require handouts for the audience, video recordings for online delivery, and all these materials may need to be reused at another date for a different audience. These do, however, not necessarily have to be done in this order or strictly one after the other—building presentations is often an iterative process. The presenter and author are often not the same person, as it is commonplace to present a joint project, to delegate the production of slides, or to inherit a slide deck from another employee.

Presenters research, structure, plan, prototype and rehearse talks

Since most studies on presentations come from a background of education, improving the experience for *audiences* has received the most investigations in studies, e.g., by looking to software to improve learning. While it is often thought of as passive, attending a live talk allows for interaction with the presenters, e.g. through question and answer segments. And, finally, as presentation materials are often published, be it in the form of recordings or as slide decks, we have to consider an audience that does not attend the talk itself, which we call *reviewers*. An example for such an audience is a student that views a slide deck later on to learn the material.

Many studies investigated *audiences* in order to find ways to improve learning.

So, we define four tasks for people involved with presentations: the *author*, the *presenter*, the *audience*, and the *reviewer* (cf. figure 3.1). The first three directly correspond to our author, navigator, learner roles (cf. chapter 2.6.2 “Authoring”). We understand the reviewer as a mixture of navigator and learner. Before we have a closer look at them in turn, we take a short look at the history of presentations and presentations aids.

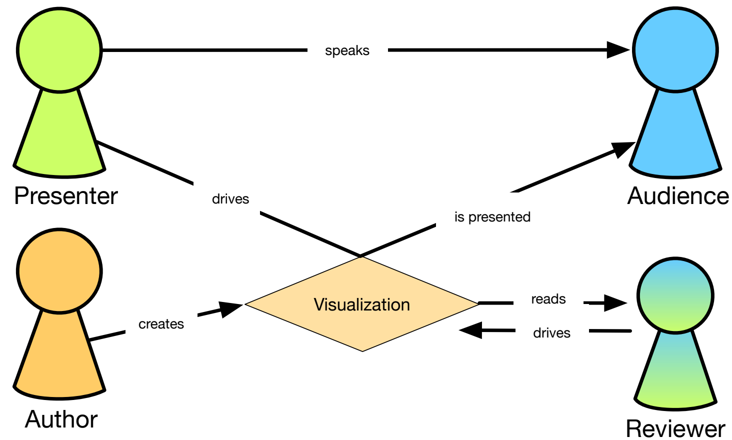


Figure 3.1: The users and tasks involved with presentations.

3.1.1 History

To understand a task and our tools, it is beneficial look at their history. As explained in the introduction, presenting is an important task today, but is far from new. Speaking publicly has always been important—in fact, discussions of public speaking techniques is among the oldest literature that we know of (e.g.,[Aristotle, 350 BCE]). Consequently, rhetoric, the art of professional talking and presentation delivery, has been studied extensively ever since. The heritage of presentation aids that accompany today's talks, however, is much shorter than that. The discussion we present here is limited so that we can understand the effect on today's tools. For a more detailed discussion of presentation history, especially concerning literature, we suggest the study of Earnest [2003]'s review.

Public speaking has always been a key skill.

Earnest [2003] illustrates a detailed chronology of presentation literature and presentation aids: The first books appear in the classical era: Greek (e.g., [Plato, around 380 BC]) and Roman (e.g., [Cicero, 55 BC]) literature are inherently focussed on the spoken word. Yet, they also explain how to use body language and clothing to underline the speaker's intent. Medieval era literature is strongly influenced by religious practices and focusses on rhetoric for sermons. It

Body language can be a presentation aid.

was only after the industrialization and the spread of organized education and literacy, that the form of *presentation visualization* emerged. It is this part of the history that we are interested in this thesis.

We can look at the physical presentation aids as precursors to present digital presentation aids. Several technical innovations began to support public speaking and all of them remain in use today: Blackboards were introduced in 1801 and became widespread in the middle of the nineteenth century; 35mm physical slides appeared in 1936 and carousel slide projectors in 1961; overhead projectors were used by the military in 1945; Television was widespread in the United States by 1960. This time constitutes a major change in how public speaking is conducted, Earnest [2003] writes: "With the advent of mass media and the dawning of the Information Age, speech teachers would find that what constituted the 'available means of persuasion' was expanding in new and dramatic ways."

Modern presentation aids appear in the second half of the 20th century.

Today, slide and overhead projectors are diminishing is use and are replaced by digital technologies. Yet, institution with less funds available or extensive existing slide collections may keep using the existing materials and devices. This transition started with software solutions to develop physical sheets for overhead projectors, users designed the slides digitally and then printed them on physical sheets. Later, the need to print out on slides vanished, and the user controlled the digital display directly from the computer. Microsoft PowerPoint [Microsoft, 2015] is not only the present market leader of such presentation software and one of the most used programs, it is also the first program of its kind [Parker, 2001].[1] It was introduced to the market in 1987 by Forethought for the Apple Macintosh—Microsoft bought Forethought in the same year and in 1990 PowerPoint was released as part of Microsoft Office for Windows. PowerPoint is a first generation presentation aid software, and is firmly rooted in the model of physical slides. While we have transitioned slide shows from the physical to digital, and similarly have copied the software

Digital tools are ubiquitous.

PowerPoint is the undisputed market leader.

PowerPoint is the oldest presentation software and firmly rooted in its physical heritage.

[1] A man named Whitfield Diffie was involved in its first prototypes, luckily for him, he is much better known for his contributions to public-key cryptography.

from desktop computers to tablets and smartphones, the interaction style remains the same to this day and firmly rooted in the slide metaphor.

3.1.2 The Speaker

The speaker's primary goal is to communicate to the audience. What she wants to communicate and for what reason is a wide field and the field of presentation styles is equally wide: E.g., a speech given at a wedding embellished with pictures of the couple is fundamentally different form a talk asking for funding at a founder's meeting, which is in turn fundamentally different from a lecture given at university—the first of these talks seeks to entertain, the second seeks to convince, the third seeks to inform. Talks can be speaker centered, in extreme cases without visualizations at all or with very minimalist slide content that underlines the sentence spoken at the moment, often with the exact same words.[2] Another example is the proverbial thirty second *elevator pitch* can determine funding, employment, or rejection for the speaker's operation. Or think of PowerPoint Karaoke, a friendly gathering where presenters give presentations without preparation or knowledge of the visual aids. One format is called *PechaKucha 20x20* [PechaKucha, 2015] that shows 20 images, each for 20 seconds, with images advance automatically. Similarly, advice on good presentation practice varies greatly: popular advice includes a rigorous format with 6 bullet points with 6 words each per slide (or variations thereof with 5 or 7), 10 slides in 20 minutes with 30pt font [Kawasaki, 2005], and presentations that use only headlines with images [Reynolds, 2011][3]

There are many different ways to present.

Presentation performance can make a difference.

Public speaking is often thought of as very intimidating and a stressful situation [Moscovich et al., 2004]. Not only do most people avoid situation where they are judged by others, but also their career may hinge on her performance.

Many people do not enjoy public speaking.

[2]This method is sometimes referred to as *Lessig method*, named for Lawrence Lessig, who made often use of this style.

[3]Our personal advice is always that there should be no text on slides except for headlines, the important data points, or take home messages.

And this is true before one introduces technology into the mix by asking the presenter to handle the controls of the presentation aid. Software commonly allows temporal navigation forwards and backwards, for example, by using the arrow keys, and random access to points in the planned presentation, for example, by typing in the slide number. The speaker can use many interfaces to do this: keyboard, remote control, touches, laser pointer or gestures [Cao et al., 2005] are just some of the available means.[4] With the added burden of driving the software they have one more thing to worry about, and an error in the slide deck—be it an accidental step backward, or resource that fails to load—can easily bring a presentation to a halt as the presenter has to focus all of their attention on fixing the visualization. Handling the controls of the presentation device only adds to the cognitive load and stress of the situation.

Speakers can dive the presentation with a wide array of input mechanisms.

Presenter's notes are another common feature of presentation aids: annotations that can be seen on a private screen by the speaker, but are invisible to the audience. This can be helpful for the presenter, if she wants to minimize her memory load, but not distract the audience with too much information. In software these presenter notes are typically shown aside the current visualization on her private screen, for example, a laptop. Before computer support, this was often realized with note cards that the presenter held in her hands or annotations upon the sides of physical slides [Churchill and Nelson, 2002]. Thielsch and Perabo [2012] found that only few speakers use this feature: 20% of presenters reported using it often or always, 42% reported to never use it.

Presenter's notes are not used often.

Visualizations are often called visual *aids* for a reason: they hold the promise of making the job easier, as any good technology should. Not only is some data much easier to show with images than to describe in words, also the visualization is often used as a guide through the talk, a lifeline for the presenter, if you will. Excessive appropriation of the visualization as a guiding help to the presenter, which is especially often done by beginners, is often criticized (for example, [Van Pelt, 1950, Tufte, 2003]). Especially in talks

Presentation aids can be a lifeline for presenters.

[4] Let us not forget the obnoxious call to an assistant that interrupts the flow of the talk: 'Slide!'

where a speaker is not or hardly visible to the audience it might questions if the presentation driven by the speaker or the speaker driven by the software? For example, a talk at a large conference or a presentation distributed online might be perceived by the audience like a movie, whose speaker just so happens to also be in the room and available for questions after. We can lament the fact that inexperienced presenters may be looking for an easy way out, but we should not forget that the presenter is indeed finding help in technology.

Public speaking is already hard, our tools must not make it harder.

As we see, the presenter has to do many things at the same time: deliver the speech, present visual aids, drive the presentation in sync with his talk, engage the audience, and prepare for the next argument in the talk. All these tasks at the same time make giving a talk typically a very stressful situation for the presenter and such situations are often feared among inexperienced presenters. It is therefore of utmost importance that we keep in mind that all our design decisions must not hinder this interaction, especially not put the presenter under duress. Any software implementation should prioritize simplicity and error prevention during presentation delivery. We talk a lot about the visualization, but we should not forget that it is ultimately the interaction between presenter and audience that makes or breaks the talk [Farkas, 2009, Garner et al., 2009, Thielsch and Perabo, 2012]. Any software implementation should aid the presenter in giving an engaging speech.

3.1.3 The Author

Authors select and arrange the material in the presentation aid.

Before a presentation aid can be used in the actual talk, it has to be built beforehand towards the intent of the talk. If the author is not familiar with the topic, she has to perform research, gather information, and make sense of it. Then she needs to select a subset of material to include in the presentation document, and finally compose the document. Johnson and Nardi [1996] report on the use of animation software, graphics manipulation software, or even text editors for parts of the editing process, which then are later combined in the main presentation software. Some-

times, composing is only a small aspect, for example, if the presenter is already an expert in the domain, can rely on existing material, if previous documents exist, or if the talk is repeated or adapted from previous talks. Often presentation aids are existing, but they were not prepared by the person who delivers them, in which case the presenter has to examine the topic to prepare for the talk [Johnson and Nardi, 1996]. When authors spend work on the presentation aids, they often want to reuse them in other formats or other talks. Is is very common to give the same or a varied talk about the same subject, maybe in a different setting. Then, most parts of the talk can stay the same, while others might be trimmed, recast, or have new materials added. Drucker et al. [2006] implemented a sophisticated prototype for version control and comparison among PowerPoint documents of related talks. For the remainder of this investigation we concern ourselves with authors that understand the talk domain well and do not have any false understandings. Yet, they may not have finalized mindset on how to present the material.

The authors work may be used more than once.

Similar to the presenting task, the authoring of the presentation document can be very diverse and individual: presentation documents are often created by multiple authors or the author is not the speaker [Johnson and Nardi, 1996]. This finding is in line with data gathered with semi-structured interviews by Spicer and Kelliher [2009], who also described delegation of parts of the authoring to collaborators and strong reuse for presenters in administrative and academic roles.[5] Collaboration and reuse are clearly the a very common behavior: Thielsch and Perabo [2012] report that only a third of presenters never reused foreign presentations, 16% never reused own presentations, and only 10% never collaborated. Currently most presentation software does not support collaboration, with the notable exception of web tools such as Google Documents [Google, 2007], Apple iCloud [Apple, 2015a], and Prezi [Prezi, 2008]. Figure 3.2 shows the activities of authors during authoring: text, tables, and images are the most often used media types. His study also finds that amateurs tend to use few programs, whereas experts use a larger set of tools; es-

Authors often collaborate on presentations.

Only web tools support collaboration.

[5] For example, often university lecture materials are adapted from year to year and are authored by the lecturers in turn.

	Never	Seldom	Once in a while	Often	Always
Preparation together with colleagues	9	30	33	24	4
Creation based on own former presentations	16	23	33	26	2
Creation based on presentations from other people	36	34	21	9	0
Import from other applications: text	8	29	25	32	5
Import from other applications: tables	8	26	35	29	2
Import of pictures and images	2	6	20	57	17
Import of videos	38	38	18	6	1
Import of audio content and sounds	47	35	14	4	1
Export of presentation as website	72	19	6	3	0
Export of presentation as PDF	27	22	20	24	8
Printing out presentation as handout	7	14	28	35	16
Sending presentation via e-mail	8	21	27	37	7
Use of presenter view	42	21	17	14	6

Figure 3.2: Results from a study by Thielsch and Perabo [2012]. The table shows what activities users reported during authoring. All values are rounded percent to the nearest whole number. The answer anchors were explained to the participants as follows: "never" means less than 5% of presentations, "seldom" means 5–35% of presentations, "once in a while" means 35–65% of presentations, "often" means 65–95 % of presentations, and "always" means more than 95% of presentations. Table from Thielsch and Perabo [2012].

pecially for important, very polished presentations, slides are designed with general graphic software and only composed with slideware.

Authoring is an iterative process.

Although the author may be informed about the topic, the actual selection of content and the gestalt of its visualization is not determined, but rather evolves through a process of re-casting ideas [Schön, 1983]. Good [2003] analyses authoring approaches and states that four parts form the authoring process: generating, organizing, composing, and revising. The author performs these stages very loosely in this order and often more than once. Top-down and bottom-up strategies are both frequently employed. Similarly, Thielsch and Perabo [2012] found that users spent 59% of their time on preparing content, 28% on design, 9% on animation. As we can see tool use takes up a considerable part of the time. Good [2003] also stresses the importance of flexibility: "The more difficult it is to explore alternatives, the fewer alternatives the presenter is likely

to consider." and "In addition, formal structures can introduce modification costs that reduce the chances that an author will explore alternative organizations.". Thus, we arrive at our requirement for presentation authoring: software should support this process of exploration and spark creativity.

3.1.4 The Audience

When we talk about presentations, clearly we cannot forget our audience members. Unfortunately, in most talks they find themselves in a very passive role and one might ask why we would consider them users of the software in the first place. With the notable exception such as *Classroom Presenter* [Anderson et al., 2003, 2004b,a] contemporary presentation software does not allow the audience to interact with or through software. The audience might ask questions to the speaker during or after the talk. In response to that question the presenter they may decide to display a certain position in the visualization, or bring up backup material that has not been planned to be presented but is deemed helpful for to answer the question [Spicer and Kelliher, 2009]. We will see that this is a use case in which zoomable interaction metaphors promises to simplify the task for audience and presenter (cf. section 3.2.4 "Analysis From an HCI Perspective").

Most audiences do not get to interact with the presentation software.

Presenters often want to give handouts to the audience before or during the talk. These handouts can be a written script, but increasingly, are print-outs of the presentation aids. He [2000] explored the usefulness of such handouts and found that students with highlighted transcripts and video recordings performed better. In contrast, simple print-out of the slides performed worst and were less accepted by audiences. This has also been observed by Norman [2005]. Unfortunately, once slides are produced for the presentation, it is all to easy to just print them out.

Audiences may receive presentation aids in the form of printed material or recordings.

The most critical aspect, however, is for the audience to receive the information in a way that the author and presenter intend to. Most of the literature is concerned with

teaching and therefore considers learning of material the important aspect of the presentation and the visuals (e.g., [Brown, 1992] or [Garner et al., 2009]). Lanir et al. [2008] and Slykhuis et al. [2005], examine lectures in a classroom setting as very content oriented presentations: they find that content in such lectures can be divided into *rich content* and *support content*. Rich content gradually built up and referenced throughout the lecture, whereas support content has only a short time value. Eye-tracking studies show that students are adept at differentiating the two and spend significantly more time on the rich content [Slykhuis et al., 2005]. When audience members are unhappy with the presentation, technical aspects are named in 41% of the responses, while 48% of the responses mentioned the performance of the presenter [Thielsch and Perabo, 2012]. This clearly shows we can hope to improve presentation software quite a bit.

Most studies consider improved learning for the audiences to be the most important metric.

3.1.5 The Reviewer

We already decried the use of slides as handout material. As visualizations become more and more prominent in presentations, the reuse of presentation aids after the talk becomes more prominent as well. A well built presentation aid can live a second life. It is quite common to expect slides as learning aids in school and university. Similarly, the slides from a talk often get retained as a means for documentation, when composing a written report or an essay on the matter is deemed too laborious. Hence, we have to consider an audience that is not present during the actual talk for which the presentation aid was composed and yet tries to gain knowledge from it—we call this user the *reviewer*.

Presentation aids can live a long life after the talk.

It is increasingly common with the development of digital video and increased bandwidths to offer recordings of the talk itself instead of paper handouts. Additionally, some talks are built to be viewed without interactions from the audiences at a later time—they are built for this communication channel in the first place. For example, Linda.com, Slideshare.net, Youtube.com are popular online communi-

We see more presentations that are only built for asynchronous viewing.

ties to not only share talks and materials after they have been recorded, but also to share the talks only this way in the first place.

The reviewer cannot ask questions to the presenter or get to the presentation aids that were not recorded such as backup slides. But, they can consume the document in their own time, location, and with the speed that they desires.[6] E.g., it can be useful to rewind to a complicated issue or to speed up a boring part. In chapter 3.3.4 "DragonFly" we discuss an adoption of our Fly prototype to the needs of a reviewer.

3.1.6 User Base

Presenting is ubiquitous, so one might like to say: everybody is a potential user. And that would most likely be true for anyone who has been to school in a highly developed nation in recent years.[7] Higher education institutions and business are similarly completely permeated with presentation use [Thielsch and Perabo, 2012]. Even the recent leaks from the NSA and military use are almost completely in the form of PowerPoint slides [Gallagher, 2014].[8]

PowerPoint permeates most administrative activities.

As of 2013, Apple offers its presentation programs for free with the purchase of new hardware, similarly, Windows machines often come preinstalled with PowerPoint. Free software solutions are available for all platforms, e.g., from OpenOffice Impress [OpenOffice, 2012] or Google Documents [Google, 2007]. We can summarize without exaggerating: if one buys a computer today, it comes with free access to presentation software.

Slideware is readily available.

Thielsch and Perabo [2012] have one of the most re-

[6] An unfortunate side effect of this is that students often skip class if they feel that the published presentation aids or recordings will suffice to learn the material.

[7] e.g., in Baden-Württemberg, a presentation is scheduled in the syllabus at 2nd and 4th grade.

[8] "One military advisor from Duke University said that the U.S. military, instead of getting our allies to use PowerPoint, should give it to the Iraqis. *We'd never have to worry about them again*.", Wall Street Journal, April 26, 2000.

cent studies and report a detailed look at how pervasive computer aided presentation use is and they write that PowerPoint remains an almost undisputed market leader in 2012, being used by 96% of all participants, and predominantly used by 83% of all participants. Apple Keynote and OpenOffice.org are used by 10%. Adobe Acrobat (29%) and Excel (33%) are also often used for handouts or preparation of parts of the presentation. Simons [2004] concludes: millions of PowerPoint-based presentations are given around the globe each day.

3.2 Slideware

The reason of bad talks is very much debated.

PowerPoint has such a bad image that it almost hard to not find an article that rails against it. And we are sure that nobody who regularly attends presentations is unaware of problems such as too much text on slides or presenters that pay more attention to their private screen than to the audience. But is it justified to blame a piece of software for bad experiences in the classroom? And furthermore, if it were true that PowerPoint makes talks worse, can we not follow that there is software that improves them? These questions are part of a debate that is as old as presentation aids. We could probably fill the entire thesis with this debate. In interest of the reader and because we have already presented this debate in this author's diploma thesis [Lichtschlag, 2008] at great length, we will just shortly outline the main criticisms and arguments.

3.2.1 Blame for PowerPoint

Presentation literature appears shortly after the first presentation aids become widespread. And so does the critique of presentation aids: Van Pelt [1950] writes: "We have fidgeted, mentally if not physically, as the remarks of a renowned scientist came to a dead stop while he readjusted some ill-arranged piece of apparatus or hunted for a scientific specimen to illustrate his point. The habit of using bad visual aids is rampant among those who 'speak

to inform"'. This reads like a criticism handed out about PowerPoint today, but this quote is from 1950. One almost fears that we have achieved nothing with new presentation aids. A contemporary critique by Kaminski [2001] reads: "[PowerPoint] too easily becomes a replacement for the presenter, not a reinforcement. Instead of a visual aid for the speaker, the speaker becomes an audio aid for the slides. This strips the presentation of some of its most essential appeals." and "It wastes time. You can suck up precious time tweaking a presentation". We can confirm the critique that the creation of presentation aids takes time. Thielsch and Perabo [2012] report that 36% of preparation time is spent on graphic design and animation of slides alone.

Critique of presentation aids appears soon after them.

Much time is spent in graphics.

The critics of PowerPoint argue that PowerPoint changes the delivery of presentations—for the worse. Tufte [2003] approaches the problem humorously and compares PowerPoint to Stalin, imposing a totalitarian regime on the presentation: all content must fit into the style of bullet points, slide after slide and pretty 'chartjunk'. In his view, "PowerPoint style routinely disrupts, dominates, and trivializes content", and instead of augmenting the talk, it substitutes the talk itself. This might even make talks, that used to employ different styles, look the same after 'enhanced' with slideware [House et al., 2005]. If the presenter talks about the slide, rather than the slide backing her arguments, then the slides implicitly set the pace of the presentation—this can be seen when the presenter has to orient themselves after a slide change [Farkas, 2005, 2008]. Similarly, Johnson and Sharp [2005] and Craig and Amernic [2006] argue that PowerPoint creates a finalized mindset, inhibiting spontaneous discussion or impromptu changes to the talk. Therefore, slideware is also seen as unfit for education, because it shows the results rather than the process of obtaining them. [Parker, 2001, Johnson and Sharp, 2005] Also an often voiced concern is the 'perfection fault': instead of thinking about high level decisions, the author is supposedly more inclined to get distracted and beautify low level content [Wright, 1983, Parker, 2001, Tufte, 2003, Good, 2003, Li et al., 2003]. The criticism that PowerPoint diminishes the presenter's ability to prepare a good talk can best be summarized by the quote from Tufte [2003] at the

Tufte argues that the design of PowerPoint degrades presentation quality.

beginning of this chapter.

3.2.2 Blame for Authors

Those who defend PowerPoint against this criticism argue that PowerPoint is merely a tool—a tool that can be used to create good and bad slides, but the outcome depends of the author's skill. Shwom and Heller respond to Tufte: "Having read hundreds of poorly worded business letters in our consulting practice and teaching, as well as many dense and impossible-to-decipher engineering reports, would we be fair in saying that word processing software is just 'not serious'?". Consequently they ask together with Holmes [2004], Brown [2007], Norman [2005], and Kjeldsen [Kjeldsen, 2006] for proper training of students in presentation visualizations or "Media Rhetoracy". Norman [2005] also argues that personal notes, handouts and slides are different documents that should not be mixed, since they have distinct features that make them not interchangeable. The view that PowerPoint and other slideware should not be held responsible for the failings of authors [Hardin, 2007] is best summarized by Kjeldsen [2006]:"PowerPoint does not give bad presentations, People do.".

The responsibility to deliver a good talk is with the presenter, not the software.

3.2.3 No Significant Difference

Thielsch and Perabo [2012] write: "As easy as computer-based presentations are to create, presenters seem to be seriously challenged to create good presentation slides and to deliver a good talk.", outlining the duality of the problem: fixing either software or presenter education will not solve our problems. Unfortunately, this thesis can only tackle the first problem—or can it?. Well, even this is open to debate. Over a series of articles especially Clark [1983, 1994, 2001] and Kozma [1994, 1991] argue about the general possibility or impossibility that specific media can ever influence learning beneficially. The discussion reached a pinnacle in 1994 when Clark and Kozma sharpened their respective views in 'Educational Technology Research and Develop-

Many studies show that different media to indeed not influence learning.

ment'. More recent research Hoyt [1999], Russell [1999], Ramage [2002], Joy and Garcia [2000], Clark [2001] strengthens Clark's line of thought: rigorous testing, especially canceling the side effects and separating method and media, has found little evidence of learning benefits with media as an explanatory variable. Evidence of increased learning are at best unconvincing and insignificant against a vast body of studies that do, in fact, find no influence on learning. Russell [1999] summarizes in his book *The No Significant Difference Phenomenon* [1999] 355 studies, of which the vast majority finds no benefits. A study comparing varying setups of slideware [Earnest, 2003] came to the same result. We close this view with Clark [1983]'s argument "[...] media do not influence learning under any conditions." and "[...] media are mere vehicles that deliver instruction but do not influence student achievement any more than the truck that delivers our groceries causes changes in our nutrition."

This reads like bad news for presentation support software, slideware or otherwise—until we remember that the learning outcome based on media is not the only measure here. Authors and presenters of talks have to build and drive the presentation well before they can hope to teach well. If software hinders or helps them in these regards, nothing in the no significant difference debate can argue against that. And if we empower authors to understand their own material better or to have more time to spend, then we can even hope to influence learning outcomes. Finally, when we consider learners that review material on their own, e.g., learning for an exam with the presentation materials of the course, we already saw that we cannot cast them as passive audiences anymore. They navigate the materials on their own—a different *method* of learning.

Clark's argument predicts that we will find no evidence of increased learning.

3.2.4 Analysis From an HCI Perspective

We see that the criticism of slideware is a heated debate. Everyone seems to agree that many presentations are bad, but the opinions differ as to what or who should be to blame. We do not want to take sides in this fight. Clearly a presen-

ter is ultimately responsible for himself. But that does not mean that we cannot strive to build better tools for him to use and to prepare his talks. Hence, below we analyze how users actually deal with PowerPoint and slideware from an HCI perspective. And we add to that an outline how the identified hurdles are overcome with a canvas presentation tool. We identify three basic problems: *content cutting*, *time dominance*, and *detail trap*. We first brought this analysis forward in this author's diploma thesis [Lichtschlag, 2008] but since then the basic metaphor of slideware has not changed and the criticism still holds. Farkas [2008] has since then formulated a similar argument. Below, we roughly revisit the three basic problems so that the reader can follow our design of Fly and the motivation for our studies. The full argument is published in [Lichtschlag, 2008, Lichtschlag et al., 2009].

We present our own analysis of slideware together with an argument for canvas presentations.

As we saw, slideware has been repeatedly criticized to be the reason for a degrading the quality in presentations [Gopal and Morapakkam, 2002, House et al., 2005, Parker, 2001, Tufte, 2003]. So, let us revisit the basic conceptual model of slideware: is based on the notion of rectangular slides shown in a linear, predefined sequence. We outlined in 3.1.1 "History" how this is rooted in the history of slideware aids, it is a continuation of a metaphor for physical slides. However, the constraining technical possibilities of traditional slide and overhead projectors that created this model are no longer valid for computer visualizations—yet they still shape our understanding of the nature of presentations. An author structures their thoughts around slides and talks about slides, because that is what the tool affords them to do [Lovgren, 1994]. Canvas presentations propose to change this dynamic using an underlying metaphor and a user interface that allows the author to keep his mental model intact and reify it without fragmenting it (cf. chapter 2.1.7 "Fragmentation and Continuity of the Information Landscape").

The problem of PowerPoint is an outdated metaphor.

Content Cutting

Slides separate content into discrete chunks of equal size, each bucket effectively determined by what can be seen comfortable from a distance. Apart from the necessities of presentation delivery, the size of these chunks is arbitrary. It does not relate to the shape of the content, moreover, it would be quite the coincidence if the information that should be presented came in exactly slide-size chunks. This leads to common problems in slide preparation when sizes do not match: When content cannot span boundaries of slides, the author has to make a choice. Do they reduce the amount until it fits, or do they stretch it over multiple slides by cut it into parts? If content does not fit, it is likely to be dropped from the talk [Parker, 2001] or the problem is battled with tiny font sizes. There is no 'half' slide for less content, or a good way to compare two slides next to each other. When a consistent topic is spread over many slides, it is an additional burden on the audience to reassemble the whole from the fragments, and the presenter's burden to help them [Good, 2003]. This is why we call our first criticism of the slideware metaphor *content cutting*: the slide metaphor acts like a stamping machine that cuts the content into equal sizes, no matter if it actually fits.

The slideware cuts the content into pieces.

Canvas presentations propose to change this, because this is actually the way we defined the term 'canvas' in chapter 2.1.7 "Fragmentation and Continuity of the Information Landscape". A zoomable user interface or a focus + context interface keeps a single continuous information landscape intact. An author that arranges the presentation content on the canvas has more degrees of freedom to do so, and no borders. But this does not solve the problem of presenting the content. A screen that is driven by the presentation aid still can only show a certain amount to content before is becomes unreadable. Which leads us to the second criticism.

Canvas tools promise to change this.

Time Dominance

Astonishingly, is is possible to build a software that aids presentations no matter what the reason to present. As we

saw slideware is employed to show everything from lectures to wedding photos. The common denominator of all presentations is makes this possible: talks are given over a set amount of time, and slideware puts this consideration front and center. Here, the timeline of the talk is hard-coded into the document at the moment of creation. Any non-linear content has to be projected onto the timeline, losing its original shape unless reconstructed via clever overviews by the presenter. Again, this leads to common problems: connections other than to the adjacent slides are lost (unless the author goes to great lengths to define hyperlinks). Individual items are either included in the talk or left out, creating a "finalized mindset" that hinders prototyping and exploration of alternatives [Good, 2003, Gopal and Morapakkam, 2002]. Optional material has to be put at the end, rather than close to the topic which it refers to. Since all slides have exactly one position in time, duplicates are needed to revisit ideas. Arbitrary access to slides is hard to do for the presenter, and jumping to the other end of a talk is usually accomplished by the visually rather jarring experience of rapidly flipping through all slides in between [Moscovich et al., 2004]. The resulting document is only valid for its original timeframe: content that is not anticipated cannot be presented [Anderson et al., 2004b]. Reusing of the document for a different talk will most likely require projecting the contents onto a new timeline all over again—even if both share most of their content. We call this criticism of the slideware metaphor *time dominance*: the slide metaphor puts the consideration of timing first.

With slideware, the author has to think about mapping the objects to time from the beginning.

Instead, we propose to postpone this consideration. In canvas presentations we can wait until the last minute before the talk and not have a mapping from time in the talk to content. Instead, the presenter can just navigate through the information landscape according to the presented content. Since content is laid out according to a structure that is meaningful to the presenter, most navigations are close, e.g., zooming or panning to an adjacent item when they follow the natural hierarchy of the content. Ok, we do not really want to burden presenters with closely controlling the navigation while they give a talk, but this scenario is helpful to see the difference to slideware: on the canvas no object has an inherent timing, it is not in a linear sequence.

Canvas tools promise to change this as well.

When the author wants to prepare a sequence, they can do so by preparing a path through the landscape before the talk. Or more than one when there are different ways or occasions to present. And if the content comes up again later in the talk, then the path loops around and follows the landscape. Following, that the animations between the items on the landscape are all consistent with each other. Where in slideware the content swoops in from the right if the presenter chose this effect, in canvas presentation software the landscape moves in from the right, if and only if the next camera position is to the right. Such considerations are hardly possible in slideware, where each slide is, in a way, a world of its own—which brings us to our last criticism.

Detail Trap

Conceptually, each slide acts like a folder into which the author has to sort their content [Good, 2003] and also their attention. Slides are limited in absolute size dimension, and the presenter is also implicitly limited in scope to editing on a detail level. They cannot 'step back' meaningfully, as there is no more context on the current slide [Gopal and Morapakkam, 2002]. Instead, an author is more likely to beautify the individual slide than to think about its place in the overall shape of the talk, a common criticism we have seen above. [Good, 2003, Li et al., 2003, Parker, 2001, Tufte, 2003]. Current software limits authoring to the smallest level—there is no support for designing a 'big picture' of the topic other than manually drawing it on a special slide that resides in between the rest. They are effectively locked in a *detail trap* of the slideware interface. The only remaining inter-slide connection is the sequence with its transition; anything else is suppressed by the format. The best thing available in slideware are interfaces called *slide sorters*, yet they also do not allow the slide to be understood in relation to each other other than a linear list. The author of a slide deck is required to anticipate the need for overviews for the audience, they have to generate separate overview slides or to explicitly name interconnections of subtopics. It takes experience to know that this is consid-

Slides are an overview + detail view, and the visualization of overviews is poor.

ered good practice and of great help to the audience [Good, 2003].

Again, canvas tools promise to change this.

With canvas visualizations this problem does not exist, the context and the surroundings on the unified information landscape are only a zoom action away. Any scope between an individual item and the simultaneous view of all items is possible. Furthermore, it is not only possible, the author is even afforded to think about the 'big picture' when he navigates to a different location in the canvas. Any navigation brings the content back into view and discourages them to spend time on small details. The net effect is that the shape of the layout on the canvas is in a structure that follows the content, and thus a presenter can simply zoom out during a talk and easily create an overview. They can do so without previously planning to do so because zoomable user interfaces build the overviews by themselves.

3.3 Canvas Presentation Software

Here we look at all the canvas tools.

There are three main canvas presentation tools that we consider when we reason about the research in this space. *CounterPoint* was developed as a research prototype at Xerox Parc by [Good, 2003] and is the earliest approach to specifically build a tool for presentation support. In 2008, both *Prezi*[9] as commercial approach and our *Fly* as a research approach joined the line-up. Both Prezi and Fly present a new way to approach presentations, because they work completely without slides. Both tools then added mobile clients, Prezi added support for concurrent collaboration with *PreziMeeting* and iterated on their authoring experience. We developed a new iteration of Fly [Hess, 2011] and investigated a combination of the canvas with video recordings with *DragonFly* [Corsten, 2009]. See figure 3.3 for the heritage of canvas tools. Here, we only present the design of canvas presentation tools as we need them to understand the following studies. For a detailed exploration of the complete related work of presentation support software, this author's diploma thesis [Lichtschlag et al., 2009]

[9]Formerly, *ZuiPrezi*.

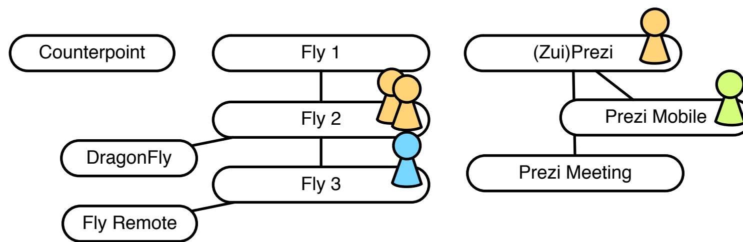


Figure 3.3: Canvas presentation tools. Attached figurines indicate user studies presented in this thesis (orange: authors, green: presenters, blue: audience)

gives an in-depth report. We discuss comparisons to new hybrid approaches in chapter 6.2 "Next Directions for Presentations".

3.3.1 CounterPoint

CounterPoint [Good and Bederson, 2001, 2002, Good, 2003] approaches the canvas from a practical side: the author takes an existing PowerPoint presentation as a basis for their layout and exports each slide as an image. Then, in CounterPoint, they import these images as the new atomic elements of the canvas. The authoring is split into two phases: one for the within-slide layout in PowerPoint, and one for the inter-slide layout in CounterPoint. The canvas itself is a multi-level zoomable user interface and slides can be placed at very varying distances from the camera (cf. figure 3.4). The user can group the slide elements into a hierarchy (similar to how Keynote [Apple, 2015a] does it with slides), and then CounterPoint automatically spreads the canvas layout based on this hierarchy. When the author plans for the presentation delivery, they create a path of camera positions. Each camera position corresponds to a stop in a traditional slideware presentation, but the camera can also zoom out a bit and show multiple slides next to each other, or zoom out all the way to show the whole canvas. To define such a path, the author navigates to the desired camera position and 'takes a photo' of the scene. This records the location to the path and creates a thumb-

CounterPoint is the oldest approach to bring zoomable user interfaces and presentations together.

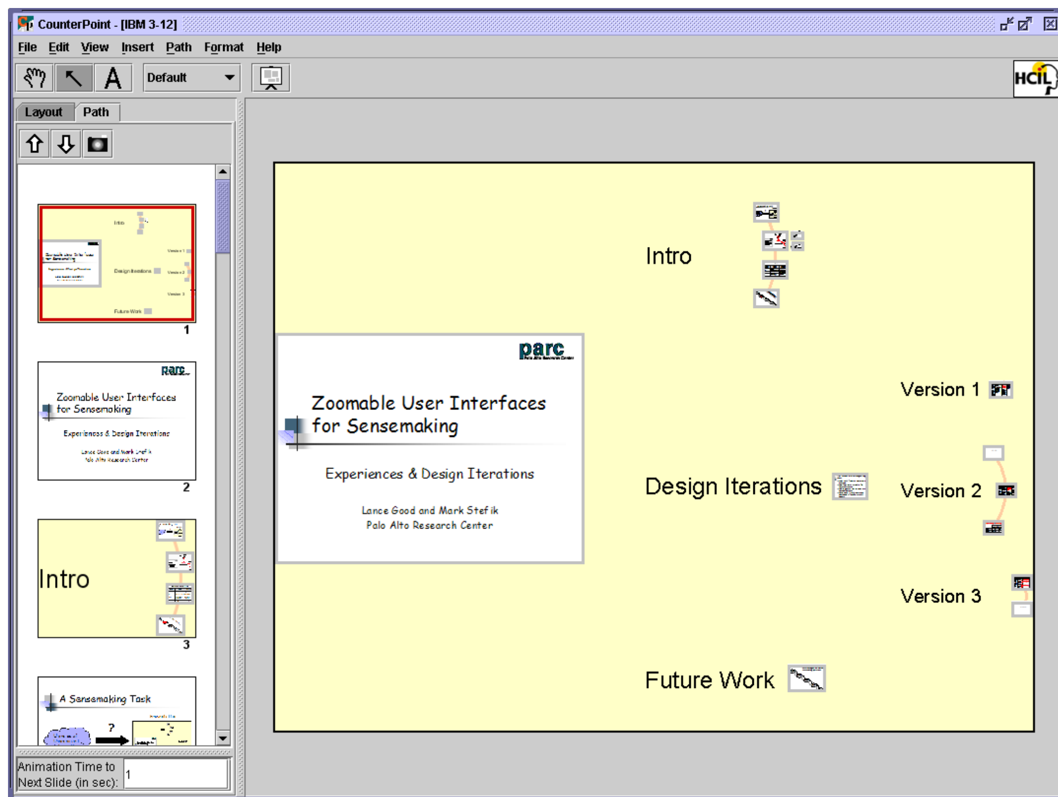


Figure 3.4: The inter-slide authoring view of *CounterPoint*. **Left:** sequence of camera stops in the path; **right:** current camera position. [Good and Bederson, 2001]

nail to the left of the interface (cf. figure 3.4, left). During the presentation delivery, the camera animates smoothly between these stops with speed dependent automatic zooming (cf. chapter 2.3.1 "Speed Dependent Automatic Zooming").

CounterPoint retains slide elements.

Their idea to build a path of camera positions in the scale space of the ZUI is a new addition to the design of zoomable user interfaces and is adopted by all of the canvas presentation tools. It is a critical innovation because it allows the user to defer the reasoning about the timeline of the presentation and it allows the same content to be referred to again, without replicating the slide. In Fly, we also adopted the 'take a photo' metaphor to insert a reference of the current position into the path. We are somewhat critical of the lingering effects of slideware by the inclusion of slides at the atomic elements in the canvas. What we are

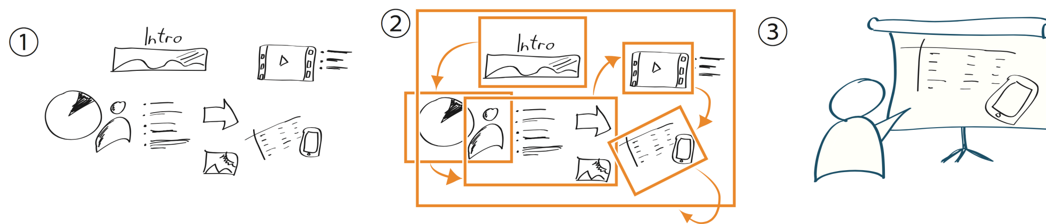


Figure 3.5: A workflow of canvas presentations. (1) The author arranges content in a spatial way. (2) They add a series viewports over the canvas that form the presentation sequence. Varying zoom depths show different amounts of content at once. (3) A viewport during presentation delivery. [Lichtschlag et al., 2012b]

sure is a consequence of practical prototyping lead to a split of the authoring into two domains and we would argue content still gets cut into pieces with this format (although the author can piece it together again in the canvas). A similar design was built as a plug-in for PowerPoint [Microsoft, 2008].

3.3.2 Fly

Fly went through multiple design iterations.

Fly has been developed over three iterations, starting with an early sketch by Holman et al. [2006], followed up by this author's diploma thesis [Lichtschlag, 2008, Lichtschlag et al., 2009] and a continuation by Hess [2011]. Here, we outline the characteristics as we need them to understand the following study section, the original publications details on the artifact designs. Holman et al. [2006]'s first Fly prototype started similar to CounterPoint with exported slides and a hierarchical approach to the canvas layout (Fly did optionally layout the slides automatically). The current design shares little of this heritage.

Fly places elements atomically.

Lichtschlag [2008]'s diploma thesis started from the approach of CounterPoint, but made a couple of key distinctions. First, we removed any reliance and reference to slides. All authoring takes place on the canvas directly and elements on the canvas are not bound by any boxes (cf. figure 3.5). The motivation was that with this design we expected authors to be able to place content on a canvas just as they had done in a paper prototype study during the de-

Figure 3.6: Our second Fly iteration. The canvas shows elements at two levels and also displays the current path as a green line. [Lichtschlag et al., 2009]

velopment of Fly (We outline the results of this study below in 3.4.1 "Authoring Lab Studies"). The second alteration is that we did not build a multi-level zoomable user interface but rather a single-level layout. In Fly, all content is on the 'ground' of the canvas, except for topic labels which are rendered above, a form of semantic zooming. When a users adds content (text, image, video) to the canvas, the content appears on the ground level or on the topic level, depending on his current zoom distance. Our reasoning was that this way, a fully zoomed in canvas always shows the elements at a proper size, especially the text, because they are on the same level. We selected the text size to be comfortably read from a distance and allowed no changes in font size.

Fly places all elements on a 'ground' level.

When a text label is placed on the topic level, it lies over the content elements of the ground level and is opaquely rendered. This is a form of hierarchy because the elements on the topic level potentially overshadow a group of im-

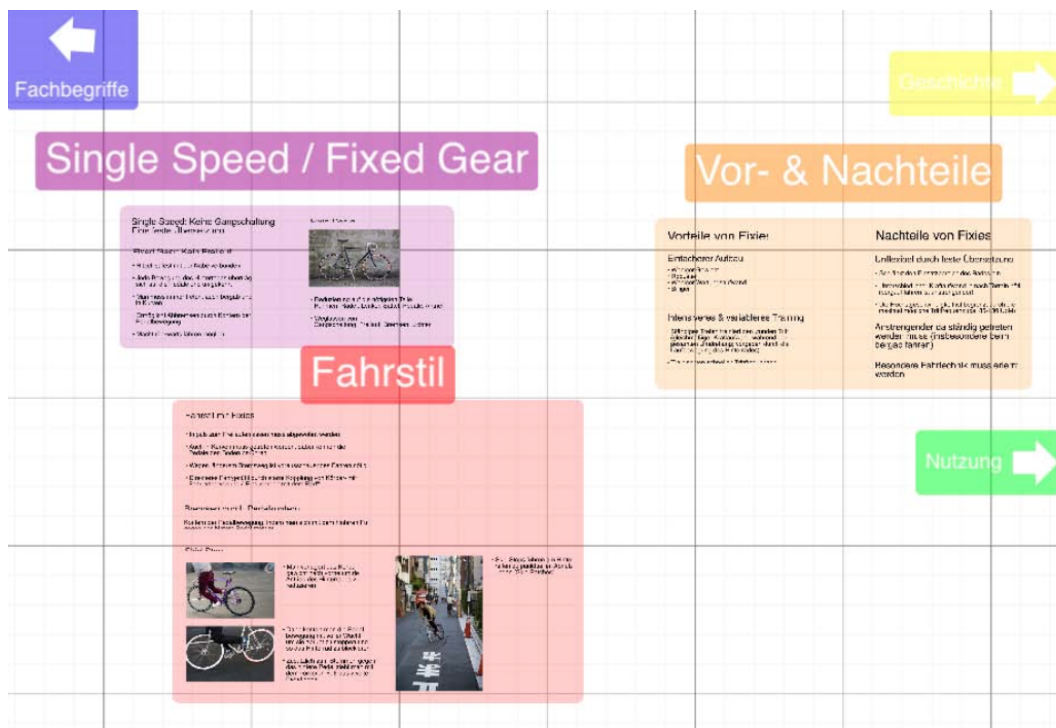


Figure 3.7: Our third Fly iteration. The canvas now automatically colors topics and shades their background. [Hess, 2011]

ages on the lower level. But this is not a formal relationship between the objects, they just happen to be placed at the same location on different distances to the camera (cf. figure 3.6). The last distinction to CounterPoint is that we included indicator for off-screen elements (cf. chapter 2.1.5 “Cue”) because we wanted audience members to have a sense of place in the presentation even when the camera is zoomed in. Just as CounterPoint, Fly documents can have multiple paths through the scene and they are also created by iteratively navigating to the camera positions of choice and snapping a picture.

Fly has cues.

In the last iteration of Fly [Hess, 2011] we reacted to feedback from our second authoring study. Since almost no authors placed images on the topic level we changed topics be be text only. And we opted into creating a shallow hierarchy in the way the user creates these topic nodes. In the current Fly, the author selects the elements on the ground level and groups them with the topic command, this auto-

Fly 3 changes how topics are created.

Figure 3.8: Our Fly Mobile in portrait and landscape orientation. [Bemtgen, 2012]

matically creates a text node as the topic level and shades the background of selected elements and the label with the group color. This also allowed authors to visually group elements together, similar to the camera frames in Prezi, but without having a hard slide frame as it was the case in CounterPoint.

Fly Remote extends the metaphor to a tablet device.

The last implementation of Fly we want to mention is the mobile presenter application *Fly Remote* developed together with Claude Bemtgen [Bemtgen, 2012]. It is a companion application to the current Fly on an Apple iPad (cf. figure 3.8). The Fly presentation documents get shared to the tablet, which can be carried by a presenter during a talk. This allows the presenter to move freely in the room when giving a presentation, and not be reliant on a stand with a presenting computer. Navigation of canvas presentations is potentially more complex than when using slideware. A handheld presentation remote with two buttons would not give the presenter the ability to zoom and pan to a different location. In our design of Fly Remote, the tablet can be held in any orientation, the 'next' and 'previous stop' buttons are always close to the thumb location. The presenter can also interact with a miniature canvas when they want to leave the presentation path. We added two buttons to

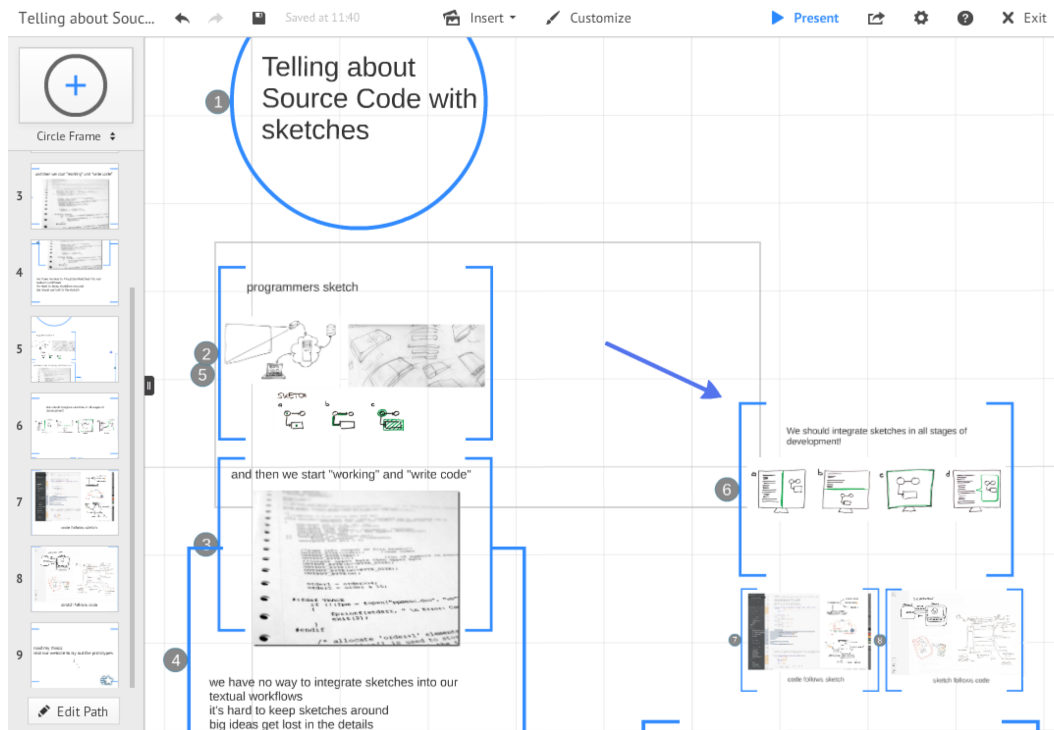


Figure 3.9: The Prezi presentation web application. On the left: the list of path stops similarly to Counterpoint, and the author can also directly interact with the path on the main canvas. Blue brackets designate the position of camera stops.

lock the navigation and to return to the last position on the path. See [Bemtgen, 2012] for details on Fly Mobile design and implementation.

3.3.3 Prezi

Prezi [Prezi, 2008], formerly ZuiPrezi, came out in parallel to our Fly, and also iterates on the ideas of CounterPoint. Just as Fly, Prezi completely abandons slides and has only one editing environment. Content gets placed directly on the canvas without intermediate arrangement on slides. Similar to all other canvas tools Prezi also uses the idea of a path through the information landscape to plan presentation delivery. In contrast to Fly and Prezi, the path is not only available in the list view on the side, the path is also directly manipulatable. After an author has created a path

Prezi has no slide metaphor either.

Prezi integrates the path just like any other element of the canvas.

stop, the stop gets rendered with a frame in the scene and is visible to the author, presenter, and audience. There are multiple different styles for frames to choose from and they can also be turned invisible, similar to how the path can be turned invisible in Fly. The path can be directly interacted with, e.g., the author can grab the rendered path line and drag it to another frame and change the flow of the presentation. Similarly, frames can be moved, scaled, and even rotated just as any other element on the canvas.

Prezi allows for collaborative editing.

As the only purchasable product, Prezi includes support for many more formats of media: diagram elements, video, sound, etc. Prezi on the web is built with Adobe Flash, but also runs on mobile devices (*PreziMobile*), but there it allows only very simple interaction: the presenter can log in, select a presentation, and follow the path. *Prezi Meeting* shares the same canvas amongst multiple authors and Laufer et al. [2011] report on their experiences with showing the location of each author reified on the canvas. The collaborative interaction has since been integrated into the main product. Critical differences to Fly are that Prezi allows multiple levels of content (just as CounterPoint, cf. 2.2.5 "Multi-Level Interaction") and they value creative expression by adding rotations to camera and content elements as well as large background images that serve as backdrop to the scene.

The design differences between Fly and Prezi lead to different results.

In contrast to that, Fly was was more focussed on presenting content in relation to each other and in our original Fly publication [Lichtschlag, 2008], we argued against rotation. Since we perform authoring studies with both tools below, we can see the results of these decisions.

3.3.4 DragonFly

DragonFly navigates video recordings with the canvas.

We already noted that the *reviewer* of presentations becomes more important as more recordings of talks are available (cf. chapter 3.1.5 "The Reviewer"). We developed *DragonFly* together with Christian Corsten [Corsten, 2009] and investigated how we can support watching video recordings of talks on a canvas. The video recording of the talk and the canvas document that was presented are displayed jointly next to each other in DragonFly. But, instead of navigating

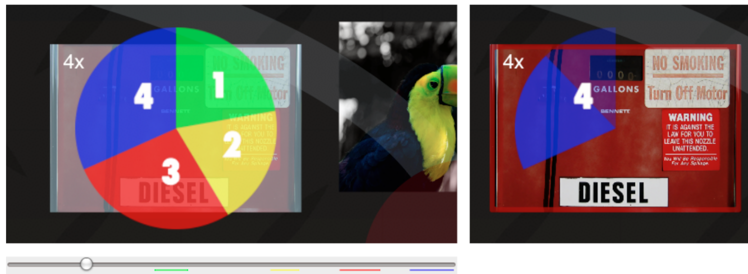


Figure 3.10: *DragonFly*. **Left:** the canvas displays a multiple choice circle each segment indicating a reference to the material. **Below:** the timeline slider also refers to the same colored regions. **Right:** the video recording plays the respective timespan. Figure by Corsten [2009]

the videos with a traditional progression slider (cf. [Karrer et al., 2008]), we also allow the *reviewer* to directly interact with the canvas to navigate the video. We built four ways to navigate the video: canvas elements, camera positions, edge indicators, and bookmarks. The reviewer can click on canvas elements that were mentioned in the recording and on the camera positions that were part of the talk. Just as in Prezi, we elevated the camera stops to elements in the canvas. Additionally, the edge indicators are clickable if they refer to a position that was mentioned and the user can save and load bookmarks. In each case, the camera navigates to the referred position on the canvas, and the video recording navigates to the timespan in which this topic was presented. The user can also pan and zoom the canvas normally as in Fly without triggering a navigation in the video, e.g., to compare the currently presented material to other material. Since the canvas positions may be referred to multiple times in the talk, there may be non-consecutive segments in the video that would be applicable when this position is activated. In this case, the respective navigation element displays a multiple choice circle ordered by time and duration in the recording. The user selects a segment, and the recording navigates to this section (cf. figure 3.10).

DragonFly is he only tool that addresses the reviewers tasks.

This is the only canvas presentation design that directly targets our ominous *reviewer* task. We see how we envision interactions for the reviewer that both fill the learning and

the navigating role. A user study with 14 participants who attended the talk showed that DragonFly users were 1.5 faster when navigating to a specific position in the recording than with a standard video player. The audience members remembered the spatial structure of the talk and were able to use it to improve their performance when reviewing the lecture. Corsten [2009] reports on the details of the implementation and the study.

3.3.5 Comparing and Contrasting

Surprisingly, even though all the tools have such a similar idea of how the canvas should be put to use to support presentations, each of them is a bit special in their approach. We compare the differences in table 3.1. We did not mention in the description above that all the tools allow the presenter to quickly get to an overview during a presentation, even if there is not a stop planned for that. This simply follows naturally from the canvas metaphor and is built into all the presenter controls.

Slideware can simulate canvas presentations...

...and canvas presentations can simulate slideware.

We made quite an argument against slideware, but we can also make an argument for it. Slideware can simulate any of the tools above[10]: one simply has to create the presentation as one big high-resolution canvas image (e.g., in a graphics tool) and then place the same image onto each slide. By scaling and moving the image the audience gets the same impression as if the the camera were moving in a canvas tool. This also works the other way around—canvas presentations can simulate slideware (to a large degree) and as we see in the studies, some authors do: one simply places the elements in nice rectangular shapes from left to right, ignoring one of the dimension of the canvas. The resulting animation will resemble a slide-in animation from the side.

So, if both tools can simulate each other, why can we assume that the outcome would be any different? If they can simulate each other, are they not equally powerful? This is the argument a computer scientist can make, but not one a researcher in human computer interaction can. We did not

[10]Which we did for a conference talk to test this argument.

	CounterPoint	Fly 1	Fly 2	Fly 3	Prezi	DragonFly
Multiple Paths	Yes		Yes	Yes		Yes
User Defined Paths	Yes		Yes	Yes	Yes	Yes
Atomic Placement of Content			Yes	Yes	Yes	Yes
Multi-Level Layout	Yes	Yes			Yes	
Formal Hierarchy	Yes	Yes		Yes		
Automatic Layout	Yes	Yes				
Path Stops Interactable					Yes	Yes
Animated Builds					Yes	
Rotations					Yes	

Table 3.1: The different canvas tools have distinctive features.

make the claim that PowerPoint is not powerful enough to move the pixels around. We make the claim that slideware affords poor use. It limits the interaction to the necessities of thin plastic film strips although hardly anyone uses them anymore for presentations. It traps the thought processes of users in the *detail trap*, and *content cutting* and *time dominance* actively separate where the human associates.

3.4 Studies

We present five studies on authors, presenters, and audiences.

Now that we have presented the argument against slideware and for the use canvas presentation software, it is ample time to back up our claims with proper user studies. In the following, we look at the domain though the lens of our user role model (cf. figure 3.11). We outlined three activities for interacting with zoomable user interfaces: authoring, navigating, and learning. Here, study each activity with the respective user task for presentations (cf. chapter 3.1 "The Task and the User"). First, we review the results from our two authoring studies in the lab, which were part of this author's diploma thesis [Lichtschlag, 2008, Lichtschlag et al., 2009]. Then, we follow up with an authoring study in the field, a presenter study investigating the emotional impact of presenting on a canvas, and we close with an investigation of learning effects that an audience might have when viewing a canvas presentation. We compare a canvas pre-

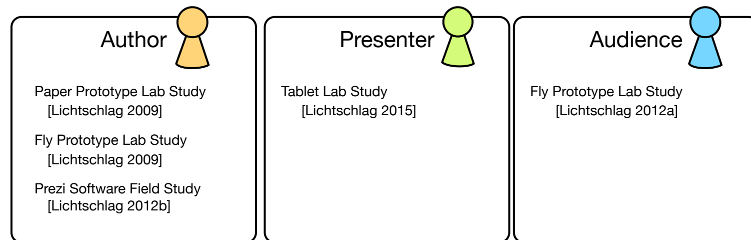


Figure 3.11: The studies presented in this chapter.

sentation tool against a ground truth of a slideware presentation tool in all of the studies except the field study. Since slideware is the de-facto standard of presentation support software (cf. 3.2 "Slideware"), this is a good baseline to evaluate zoomable user interfaces to.

3.4.1 Authoring Lab Studies

We compared authors twice during the initial design of Fly.

In this author's diploma thesis [Lichtschlag, 2008] and the accompanying paper [Lichtschlag et al., 2009] we presented the Fly design and prototype as well as two studies on authoring presentations with the canvas metaphor. Studying the author is particularly interesting, because we hypothesized (cf. chapter 3.2.4 "Analysis From an HCI Perspective") that zoomable user interfaces would be able to overcome the key drawbacks of slideware. Better visuals do not necessarily lead to a better talk, since speaker performance remains the dominating factor of presentation quality. Our hope was that authors would make use of the canvas format to build presentations that are more creative, that allow the content to be presented without fragmentation, and are able to convey the macrostructure of a talk. During the development of Fly, we did two studies to proof this claim: First, we conducted a simulation of a canvas with a paper prototype of Fly and slideware, exploring the different presentation documents that authors create. Then, we did the same with the finished Fly prototype, comparing it with PowerPoint. Our two studies looked at both quantitative and qualitative aspects of the authoring process, of which we outline the main findings below. We previously pub-

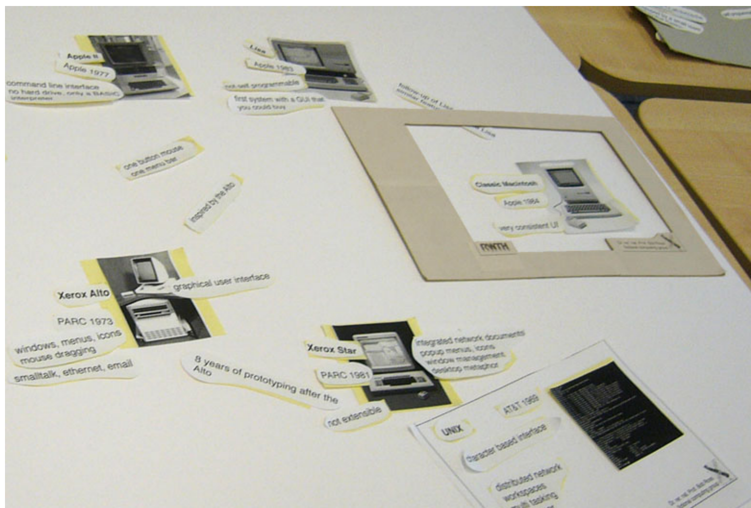


Figure 3.12: Fly paper prototype. [Lichtschlag et al., 2009]

lished these two studies in [Lichtschlag, 2008, Lichtschlag et al., 2009].

More Connected Layouts

The most important finding is that we could observe authors change the way they present content on a canvas. We saw this already when we simulated the user interface with paper versions of an imaginary typical slideware application and Fly respectively (cf. figure 3.12). We asked testers to prepare visual aids for an two upcoming talks to the best of their ability with materials we supplied, one talk for each condition, and then we investigated the resulting documents. Testers were not asked to give the actual talk, but to shortly outline how they would use their document. We found that the canvas presentations are more connected that slideware documents, indicating that authors could overcome some of the content cutting drawbacks of slideware. We evaluated the documents with respect to how many connections in the topic they embodied visually through the layout of elements on the canvas or slides. At the same time, we did not measure increased time to author canvas layouts. Authors achieved this by

Canvas documents were more connected but not slower to create.

abandoning the linear structure that is set by the time dominance. Instead, the canvas invited authors to think about the structure of the talk first, leading to very diverse layouts.

We found this result with a low-fidelity and a high-fidelity prototype.

The goal of our second study with the software prototype was to find out if the concepts that worked in the paper domain would carry over to an interactive application. Two major problems make it hard to transport the easy paper handling to the computer: limited screen space, and the indirect manipulation through mouse and keyboard. Yet again, we saw that the topics were more connected with Fly than with PowerPoint, however, the effect size was smaller that in the paper prototype.

More Diverse Layouts

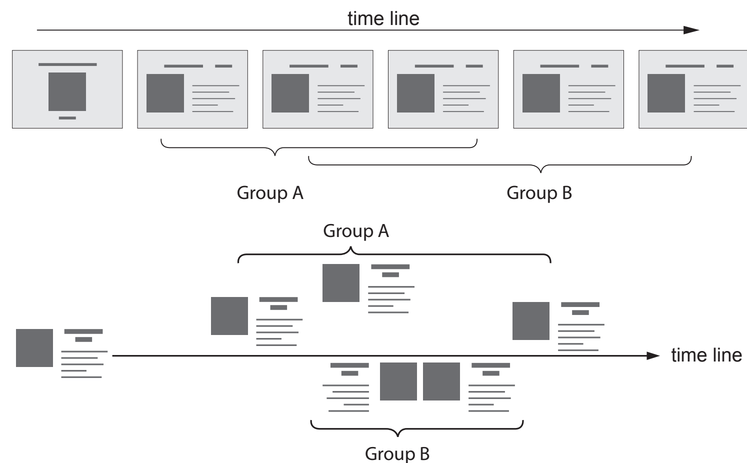


Figure 3.13: Top: Time and group ordering conflict in the linear case. **Bottom:** The problem solved in a planar layout. Lichtschlag et al. [2009]

Slide results were very linear and looked dull.

The layouts of the documents in the slide and PowerPoint conditions were very similar, almost as if normalized. The only variations in this theme were the position of elements in the slide frame with practically identical results otherwise (cf. figure 3.13, top). If one subject spanned two slides, the image was often repeated. Users were very observant of the problems of time dominance and often commented

accordingly: "It is hard to get a good order", and "I will present non-profits first, and then make a jump back in time and start with Apple's systems." Comparing that to how authors addressed the same task when presenting content in the canvas condition, we found them to use the second dimension to avoid the problem. Figure 3.13 shows how the problem of conflicting order criteria was addressed elegantly: the vertical dimension makes it trivial to group subjects without breaking the timeline into segments, or, as one tester put it: "The Apple II should go here chronologically, but it does not fit—I see that's why we have the plane." This was very illuminating to witness how the overview + detail nature of the slideware approach afforded users to fragment their model of the information landscape. We also observed two other main layout strategies: a second design (cf. figure 3.15, left) starts by constructing 'pillars' of a common idea and then spreads them out horizontally. A third design (cf. figure 3.15, right) revolves around a 'central idea' of the talk, in this case an important computer system perceived as the origin of the remainder. We observed these designs 5, 4, and 3 times during our study respectively and they seem equally capable of communicating the topic's features.

Authors make use of the canvas in three different ways.

The plane visualizations exhibited more variation on the detail level. Often the whole material for one subject was not visible simultaneously. For example, testers positioned text to the different sides of the image at the same time, thereby sharing the image between two viewports and strengthening the context. The more flexible layout facilitated dynamic local comparisons with and without zooming. Nearly all testers saw this possibility, and planned their layout accordingly.

Authors use small panning movements when content does not fit one screen.

When we investigated the documents in the software prototype, we one again saw that slideware affords very linear talks: of the 18 PowerPoint documents, 14 were strictly linear. Three clustered all content on less than three slides, and only one created a manual overview slide before sequentially discussing each topic in detail. In contrast to that, only three Fly presentation documents were linear, nine divided the topic into two or three clusters (i.e., figure 3.14), and two structured the characters in two pillars

We see similar results in the software prototype study.

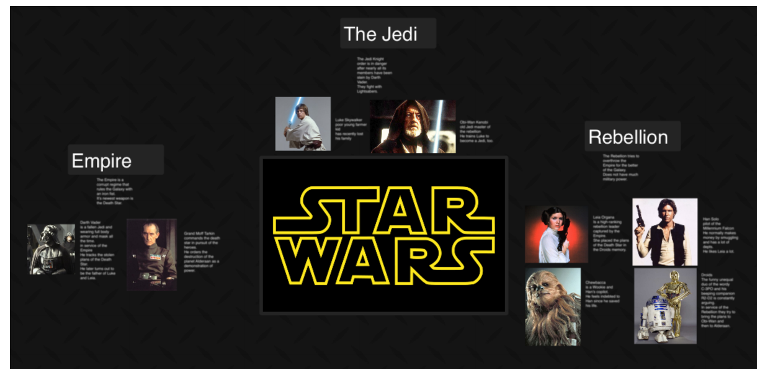


Figure 3.14: An example document from the high-fidelity study with three groups. [Lichtschlag et al., 2009]

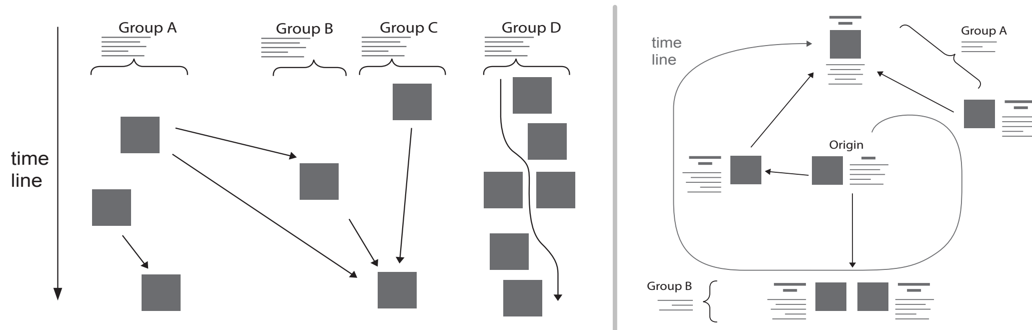


Figure 3.15: Two common designs of canvas layouts. **Left:** A canvas layout with groups along the horizontal, and time along the vertical axis. **Right:** A canvas layout where the central topic serves as an origin for the talk. [Lichtschlag et al., 2009]

(cf. figure 3.15, left). Two layouts were circular (cf. figure 3.15, right), discussing the connection of all characters to the main character in the center. The last two arranged information like a collage, but without hierarchies, relying on proximity alone. All fifteen non-linear talks had meaningful overviews, and fourteen presentation documents used zooming as a to facilitate this in their paths.

The canvas creates overviews by itself.

We gathered strong feedback considering the three problems of slideware: seven testers each stated that they see a benefit in the creation of overviews over PowerPoint, or, as one tester put it: "[It] creates overviews by itself." Seven testers saw an improvement upon PowerPoint in creativ-

ity, and six liked the ability to place elements without restricting slide frames, underlining the *content cutting* problem. Two stated slideware makes "run-of-the-mill" presentations whereas Fly was considered more flexible. Three considered the slide framework harmful, one said it helped them.

Incremental Revealing

When testers used the paper frame in the low-fidelity study to indicate which path they wanted to take, often snippets were half visible or information of a subject not currently in focus could be seen. This is very uncommon in slide presentations where only immediately relevant information is shown. On inquiry, our testers stated that they did not consider this to be a problem, as long as the information did not disorient the audience or was already been presented.

This was the first time we observed the challenge of incremental revealing.

When laying out their path with the software prototype, four users were concerned this problem of *incremental revealing* (cf. figure 3.16): They tried to hide the upcoming parts, but since Fly has no mechanisms for revealing, they had to place them at greater distance to achieve this. They did not perceive this as a problem, if the half-visible information was part of previously discussed topics. While some content, such as answers to questions for discussion, will always require hiding, in many cases the preview of upcoming content might actually be helpful to the audience. In any case, if authors want to control the the visibility of the next topic, we do not want them to distort the information layout on the canvas to achieve this.

We saw it again in the second study.

Discussion

Users stated that it was easier to express themselves on the canvas, and the possibility to define paths by demonstration was consistently considered positive. When asked whether they were satisfied with their results, testers gave more positive answers for Fly in the high-fidelity study and the canvas layout in the low-fidelity study. And when

Authors accept the canvas format.

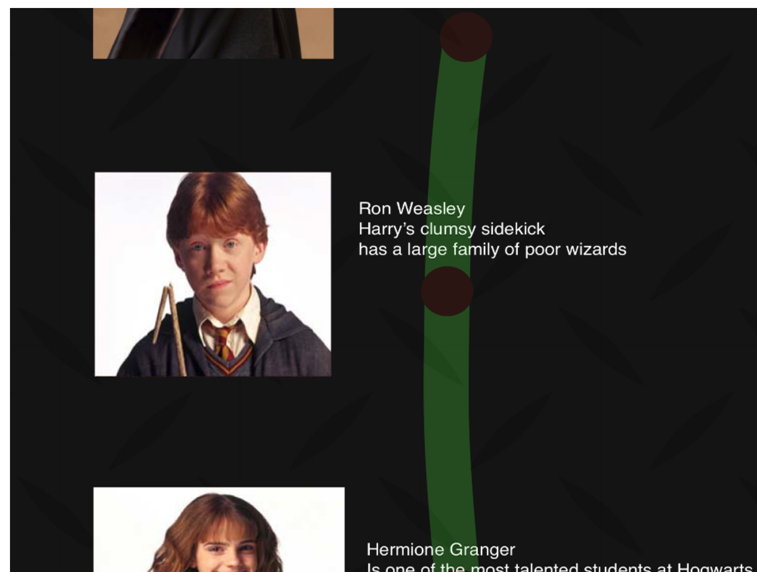


Figure 3.16: An example of the revealing problem in Fly where the heroes are presented after another. The path begins at Harry Potter, and leads via Ron to Hermione. Authors often found the half-revealing of upcoming content (Hermione, bottom) troublesome, but not of already presented information (Harry Potter, top). [Lichtschlag et al., 2009]

asked which software it was easier to express themselves in, and which they preferred for real talks, most testers chose Fly and the canvas.

We observed problems with the creation of topics and mouse scrolling.

In the software prototype we tried out our single-level ZUI design, restricting object placement to the two topic and detail layers. This model worked well for our users and two layers turned out to be enough for the scope of the test materials. But, we could observe them to encounter a problem with the effective modality of our implementation. Since the level at which new content was created depended on the current zoom level, sometimes they created a new element on the wrong layer. As a result of this observation, we changed how topics are defined in the following design iteration. Another recurring problem was mouse-centered zooming, many users did not see that the position of the cursor had an influence on the outcome of the zoom oper-

ation (cf. chapter 2.3 "User Interaction of Zoomable User Interfaces").

Our user tests provided strong evidence that users not only easily understood the new interface but were able to capture the structure of strongly connected topics in their presentations much better than when using the traditional slide interface. We were able observe these results both with the low-fidelity paper prototype early in the design of Fly and with the high-fidelity software prototype. Likewise, users commented positively on the ability to express their mental models of the material more freely and generally preferred Fly over interfaces based on the slide metaphor. These findings support our hypothesis that the canvas interface is better suited for the task of illustrating non-trivial topics than slideware.

We conclude: authors benefit from canvas presentations.

3.4.2 Investigating the Author in the Field

Our previous published studies already examined the process of *authoring canvas presentations* with Fly compared to using the traditional slide deck format (cf. section 3.4.1 "Authoring Lab Studies"). And we were happy to find that the resulting Fly documents tended to be more diversified and had a better representation of the structure of connected topics. We saw that authors abandon linear structures and switch to two-dimensional layouts. But these two studies were conducted in a lab environment, they do not necessarily represent everyday practice of canvas presentations. The claim could very well be made that authors would build their documents differently when presenting their own topics and when they do not know that we evaluate the documents. So this is why we follow up with a field study investigating the authoring again with these concerns in mind. The following study was done in collaboration with Thomas Hess [Hess, 2011] and has been previously published as a peer-reviewed paper Lichtschlag et al. [2012b].

We revisit our earlier authoring questions, but investigate them under different circumstances.

Study Method

On Prezi authors share canvas presentations with the public.

We examined a pool of publicly available Prezi documents to see how authors use a canvas-based presentation format for real world tasks. With Prezi, documents that are created on their web service can be shared with other and are are publicly accessible on the 'Explore' section of the Prezi website [Prezi, 2008]. Authors can choose to share them either as read-only or even available for reuse by others. For this evaluation, we considered the most popular 73 of the 308 presentations listed on July 1, 2010. While this may not be representative for all canvas presentations, it helped us to concentrate on documents that were considered well-authored. We excluded documents that were either clearly not created as live presentation support, were not finished yet, or served as instructions for Prezi, and after that 50 presentations remained. We examined these presentations with regards to their use of layout strategies on the canvas, overviews in their presentation paths, their use of zooming, and their use of rotations. It is important to keep in mind that Prezi follows a multi-level ZUI layout strategy for its elements and allows for rotation of both documents and the camera.

We examined 50 such presentation documents.

Observations

Authors create unique documents.

Authors use zooming to emphasize individual elements.

The first thing to note is that every document studied had a unique canvas layout; there were no recurring designs as it is common with slide presentations. Nearly all authors used scaling and zooming to achieve varying viewport resolutions, like we expected from the lab user study. All presentation paths zoomed in on single or very few elements to focus on the currently relevant information. Other common focusing practices were to zoom in on details of large graphics (so that the graphic was larger than the viewport), such as diagrams and screenshots, or on single words and phrases of larger texts for emphasis. These presentation paths are always possible in Prezi because it does not limit the zoom distance of the camera.

Figure 3.17: Prezi canvases with *decorative layouts*. **Left:** a large photo of a desktop is used in the background. **Right:** viewports are embedded into a film strip. [Lichtschlag et al., 2012b]

Layout Strategies

Often documents follow a decorative layout.

A recurring strategy we observed are *decorative layouts*, which use a large background graphic with content elements placed at lower scales into the gaps of the graphic (cf. figure 3.17). In these cases, the content is primarily arranged along the graphic shape and not necessarily according to the semantics of the topic. A distinction can be made between this layout and structural layouts, where the arrangement of the content elements reflects the macrostructure of the topic. Out of the examined documents, 36 had structural layouts and 14 had decorative layouts (cf. figure 3.20). For the documents with structural layouts, we identified three subtypes: *topic areas*, the *incremental development of an idea*, and *slide-like* linear layouts.

Many layouts are similarly structured to our earlier results.

The majority of the documents (n=29) organized content into *topic areas* (cf. figure 3.18), and the presentation paths explored these areas sequentially. Starting from an overview to preview the upcoming content, such a path drilled into one topic and then, after covering it completely, zoomed back out—either showing a repeated overview of the past topic for recapitulation or directly moving on to an overview of the next topic—and then drilled into the next topic. This kind of structure was often built recursively with topics that contained subtopics, which were traversed in the same way. This type of grouping is very similar to the group ordering layouts we already saw in our lab study.

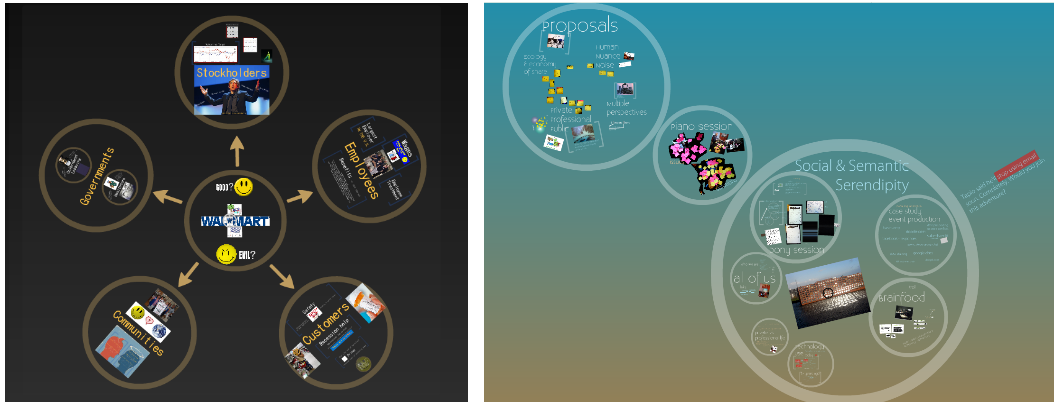


Figure 3.18: Two Prezi canvases with *topic area* structures. [Lichtschlag et al., 2012b]

Few documents create very deep zoom structures.

Three documents had structures that *incrementally developed* an idea (cf. figure 3.19). Their presentation paths started by showing content on a detail scale. Then, all steps in the presentation zoomed out, incrementally revealing more content, ending with a view of the whole canvas. Accordingly, the content was scaled larger the later it occurred in the presentation sequence. Overviews were mainly used to recapitulate.

Few documents are linear.

Four documents had structures *similar to slide decks*. All the content elements shared a small range of zoom levels and the paths traversed them sequentially while constantly remaining on the detail zoom levels with none or little overviews. The content of these presentations indicated that their purpose was to tell a story as opposed to inform about a topic.

Overviews

Overviews preview and/or review the structure of the material.

The majority of presentations (33) had positions in their presentation paths that facilitated an overviews: 17 used overviews to preview and recapitulate content; 16 used overviews only to preview; one used overviews only to recapitulate (cf. figure 3.20). We see that the Aristotle's mantra "Tell your audience what you're going to tell them. Tell them. Then tell them what you told them." is often followed closely [Aristotle, 350 BCE, chapter 19].

Figure 3.19: This presentation *incrementally develops* of an idea by zooming out. This shows a position late in the presentation, where the initial views have almost vanished. [Lichtschlag et al., 2012b]

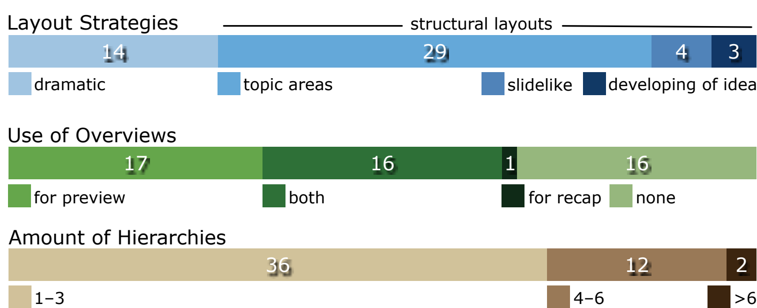


Figure 3.20: Distribution for layout strategies, amount of zoom levels, and use of overviews for the examined presentations. [Lichtschlag et al., 2012b]

Zooming

Because of the unlimited zooming capabilities of the Prezi canvas, there was no restriction on how deep the content hierarchies could be nested or how small content elements could be created (cf. chapter 2.2.5 "Multi-Level Interaction"). However, 36 of the examined documents did not

create hierarchies deeper than three levels (e.g., all content, topics, and subtopics). Among the others, 12 had hierarchies between four to six hierarchy levels deep (cf. figure 3.20). Only two of the presentations that developed an idea had more than six levels as a result of their design approach. Another recurring pattern we observed is using unlimited zooming to hide content, such as a footnote to a text, by scaling it down smaller than a recognizable scale. Then, a dramaturgic zoom-in-movement in the presentation path reveals the previously easily overlooked content.

Most presentations use relatively shallow layouts.

Rotation

Three of the presentations used rotation to follow the semantic structure of the material, e.g., when the content had circular arrangements. However, 29 presentations used rotation primarily to achieve decorative canvas layouts or to provoke impressive viewport transitions—elements were often rotated by 90 degrees or more or in opposed directions. In decorative layouts, content elements were often rotated to make them fit into the intended shape (cf. figure 3.17).

Discussion

Compared to our earlier lab studies 3.4.1 “Authoring Lab Studies”, we found similar results. We again noted a use of diverse layout strategies grounded in the macrostructure of the content. We saw that a majority of the presentations group content into clusters according to topic structure. In Prezi this can be visually underlined with frames around the elements. We did not observe the circular layouts of our earlier study, this may be due to the given topics in that study. Yet, we also saw new layout strategies which we could not observe in the lab study: decorative layouts and incremental development of an idea by zooming. We are a bit concerned that the former overly conforms content to the graphics of the decoration instead of the content structure.

The field study replicates results from the lab study.

Previously we found that the canvas format facilitates employing an expressive layout for the purpose of overviews. We could confirm this observation with our selected body of presentations and are happy to see this claim about the affordances of ZUIs validated again. The frequent use of hierarchies (cf. figure 3.20) indicates that a canvas presentation tool should allow the nesting of content—either via the use of zooming or through explicit layers. In our body of presentation documents investigated, we see that the majority produces a 'shallow' multi-level structure. This indicates that authors are conservative in their use of multi-level designs even when the tool allows unlimited depth. Rotation, which is not possible in Fly and consequently was not investigated in the lab studies, was used in the majority of presentations, but mainly for decorative reasons. This may come at a cost when presented to the audience, Schacter and Nadel [1991] write that since spatial knowledge acquired from maps is not robust against orientation manipulations.

We learned how authors use the canvas format with unrestricted zooming.

Overall, this study validates the results from the lab setting. And we show that canvas presentations do, in fact, influence everyday authoring of presentations.

3.4.3 Investigating the Presenter

Our next study looks at our second user group, the presenter (cf. section 3.1.2 "The Speaker"). When one thinks about presentations, the presenter (and for many people the dread of being the presenter), is the most important user. The presenter gives the talk, they are the arbiter of the interaction. And they also drive the presentation visuals. To our knowledge, this is the first investigation of the interaction of presenters with canvas presentations. If all goes well, the presenter can follow the prepared path and just drive the presentation one step at a time. But when interruptions occur, the navigation facilities and the metaphor of the presentation software matter. The canvas format could be especially helpful for navigations that deviate from the planned presentation delivery, e.g., in response to a question. The presenter can quickly pan-and-zoom to create an

One can hypothesize that presenters benefit or are hindered by the canvas layout.

impromptu overview and show the macrostructure of the talk to the audience. Contrasting that, we can also hypothesize that the freedom of the format may be too demanding for a presenter who is preoccupied on his main job—talking. Here, we present a lab study with presenters who gave short talks in both the canvas and the slideware format to investigate these issues. We measure the emotional state of presenters during a presentation delivery in which several kinds of interruptions occur. The following study was done in collaboration with Philipp Wacker [Wacker, 2014] and has been previously been published as a peer-reviewed paper [Lichtschlag et al., 2015].

We study the impact of canvas presentations on the presenter's emotions during unforeseen demands.

Defining emotion is very difficult, and there are numerous attempts in the literature. A popular approach is to characterize emotion using a component model where expressions, bodily reactions, and the subjective experience have "long-standing status as modalities of emotion" [Scherer, 2005]. There are different ways to combine these components; here we use a dimensional approach (e.g., [Russell and Mehrabian, 1977]) with *valence* and *arousal* as the main components [Scherer, 2005]. Valence dimension contrasts pleasure and displeasure, while the arousal describes intensity. The term *feelings* is defined in the component model as the subjective experience of an emotion and can occur separate from bodily reactions (e.g., [Frijda, 2000]). As many people are anxious about public speaking it is the presenter's feelings that we are interested in. Scherer [2005] writes that these feelings can only be accessed through a person's self-reporting.

Valence and arousal denote the direction and the intensity of emotions. Feelings are subjective experiences.

The *self-assessment manikin* (SAM) [Bradley and Lang, 1994] (cf. figure 3.21, right) is one way to elicit ratings from subjects. Each row represents change along one dimension of emotion as pictorial depictions of a person. In our study we used SAM without the dominance domain. While this method requires fewer ratings from a person, the two dimensional rating does provide little insight into the aspects that produced the emotion Scherer [2005]. A more detailed technique is the *semantic differential* (SD) which is used to measure the meaning of words [Osgood et al., 1957] (cf. figure 3.21). A semantic differential consists of point rating scales with bipolar word pairs on either end of the scale

The *self-assessment manikin* and the *semantic differential* are techniques to elicit the emotional state.

(e.g., *good–bad*). It has been used to measure attitude and feelings, for example in a classroom scenario [Evans, 1970]. A person rates a concept by indicating for each word pair where they place the concept between the limits.

Study Design

We conducted a lab study with every tester giving two presentations for about 7 min each (cf. figure 3.21, left). One presentation was given with Apple Keynote [Apple, 2003], representing slideware, and one with Prezi [Prezi, 2008], representing the canvas condition[11]. In this format, we were able to have comparable talks for all users and were also able to to include interesting tasks in the presentations as well as simulated technological problems. In each presentation, we first asked the tester to give their presentation normally by *stepping forward* through the presentation to model a successful flow of the talk. Then we interrupted them and asked to *move to a well defined position* in the presentation (e.g., going back to a specific previous slide as part of an audience question) and then to *skip forward* towards the end (e.g., due to time constraints). Finally, we asked them to *search for a loosely defined position* (e.g., the presenter has to find the matching answer in the materials). One of the two talks also included simulated errors: a *misinterpreted input* (simulated by a step backwards on a forward command), and an *unresponsive program* (simulated by no action on the first command). We were able to simulate these problems by modifying the slide or canvas structure. We counterbalanced the delivering software and the order of the error condition.

Our study tests presentations in the canvas and the slideware format.

Our study tests presentations with and without errors.

Our setup excludes many confounding variables (e.g., varying documents, audiences, stakes, presentation occasions, length, etc.). With this restricted setup we build a baseline understanding in the lab with a high control of explanatory variables. As such, the results of this setup are limited to a lab setting until a field study can investigate

Our study setup limits confounding variables.

[11]During presentation delivery, the differences between the different canvas tools are negligible, similarly for slideware. We do not expect our choice of software to impact the study.

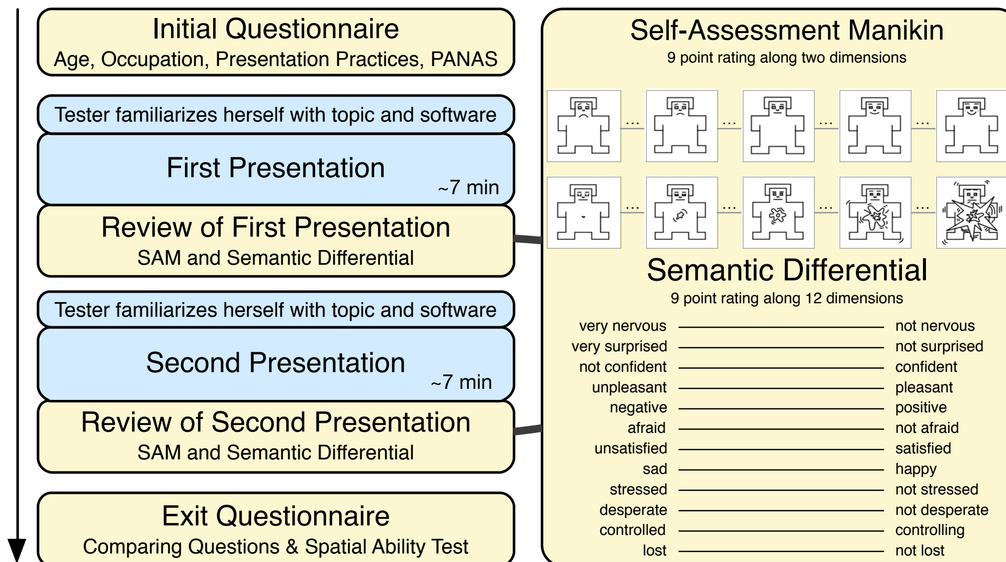


Figure 3.21: Overview of the presenter study. **Left**: each participant presents two talks, and after each they rate their feelings with the Self-Assessment Manikin (SAM) and the Semantic Differential (SD) described on the right. **Top Right**: One row changes from an unhappy person to a happy person (the valence dimension), one row from a person who has the eyes closed to a person experiencing intense feelings (arousal dimension). **Bottom Right**: The SD dimensions used to quantify more precise feelings. [Lichtschlag et al., 2015]

their generalizability.

In both talks, testers presented the 2014 Soccer World Championship, a topic presented heavily in the media at the time, so that we could expect our testers to have at least some prior knowledge. The topic was also a good fit for a spatial layout by placing players on their actual position on the pitch. Therefore, the topic could be presented spatially in the canvas condition in a manner that was approachable to our testers. Additionally, we introduced the participants to the documents and the tools, then they used the software on their own to familiarize themselves with the matter. Documents were created by us. While is common to present foreign slides, this remains a limitation of the lab setup. All presentations were driven by an iPad carried by the presenter. Using the same input device for all presen-

All presentations were driven by a tablet.

tations lowered the possible differences in interacting with the different software. Both software animated transitions, canvas with the inherent flyover, slideware with a slide-in from right to left between each slide. During the presentation delivery, the input modalities are step forward, step backward, as well as zoom and pan.

During the presentation, the moderator acted as an interested audience member that smiles and acknowledges the information given. Two cameras recorded each talk, this increased the stakes for our testers as playacting the presentation in this manner gives similar results to the "real" situation [Kern et al., 1983, Higgins et al., 1979]. Secondly, the cameras allowed us to watch the recording afterwards together with the presenter. Although memory of feelings lessens over time [Robinson and Clore, 2002], the participant could relive the situation and assess their feelings, and we avoided interrupting the presentation. We used the self-assessment manikin with a nine-point rating scale [Bradley and Lang, 1994] (cf. figure 3.21, right) to measure the valence and arousal of the participant in the task situations. We also used the semantic differential [Osgood et al., 1957] to ask them to rate twelve feelings on a nine-point rating scale to measure feedback on specific feelings. To find out which ones to include as these twelve dimensions, seven weeks before our study, we had asked in an online survey among presenters which ones they had experienced themselves or observed in others. We combined the reported emotions with directions from literature [Good, 2003] and produced the dimensions in figure 3.21. Accounting for the possibility that underlying moods affect the feelings of the participants, we used the PANAS test, a reliable and valid method to measure mood over various periods of time [Watson et al., 1988]. Finally, in the exit questionnaire, we asked for informal feedback on the experience with the software, differences to their regular presentations, and how their feelings in the study related to feelings in real presentations. To see whether spatial ability influence the experience of presenting, we measured the participants spatial ability using a paper folding test [Ekstrom et al., 1976]. We formulated these hypotheses:

We recorded all talks to increase the stakes for the presenter and to review the situations with them afterwards.

We constructed the SAM dimensions according to a survey prior to the study.

H1: Feelings in canvas presentations are rated differently

than feelings in slide presentations.

H2: Presentations with technical difficulties are rated differently than presentations without technical problems.

H3: Order of presentation and presence of errors does not influence ratings.

H4: Participants experience the same feelings during the study compared to a real world presentations.

Evaluation

We used the presenter's experience, their technical expertise, their mood, their general stance to presenting, and their spatial ability as explanatory variables.

We recruited 21 participants for the study with varying proficiency in presentation skills in general and technological skills in particular. The participants were 8 teachers, 7 students, 6 other professions, none familiar with the lab, aged 17–66 (mean=37.09, SD=16.02). To quantify the presentation experience we calculated a presentation age by subtracting the age at which a participant had given their first presentation from their current age. This *presentation age* (*PAge*) had a mean of 18.33 years and standard deviation of 11.73. The *technological expertise* (*TE*) was assessed by calculating the mean between how often the participant uses canvas tools and slideware respectively (rated on a five-point scale where higher values mean more often). *TE* had a mean of 1.33 and a standard deviation of 0.56. We also asked participants how much they *liked* to present (*L*) on a five-point rating scale (1–5, 1 = most enjoyment, mean=2.19, SD=0.93). Other gathered characteristics were *spatial ability* (SA, 0–20, number of correct solutions in the paper folding test; mean=12.76, SD=4.77) and mood (PANAS test, separated for *positive affect* (*PA*) (mean=31.14, SD=4.95) and *negative affect* (*NA*) (mean=12.57, SD=2.34)). A correlation of *PAge*, *SA*, *TE* and *L* showed that the presentation age and spatial ability of participants had a significant negative correlation. Hence, we could not analyze them separately, and when we report on the experience of the presenter below, their spatial ability or a combination of both can also be an explaining factor. We categorized the *PAge*, *TE* and *L* each into two groups for the evaluation (e.g., high and low technological expertise).

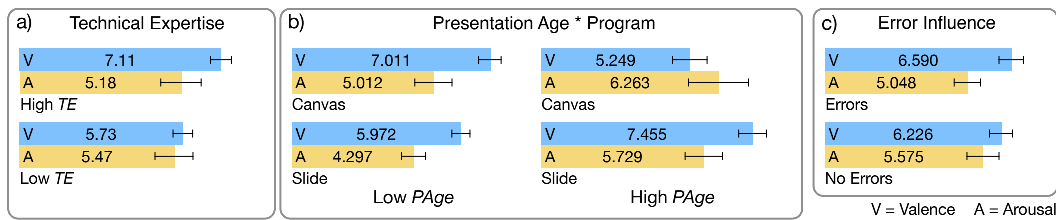


Figure 3.22: a) SAM ratings for technical expertise show significant difference. b) Presentation experience influences emotional response. c) Arousal rating is influenced by presence of errors. [Lichtschlag et al., 2015]

As for the hypothesis of program influence (**H1**) we conducted two repeated-measures MANCOVAS with the Valence/Arousal ratings (SAM) and the semantic differential (SD) ratings as dependent variables respectively. *PAge*, *TE* and *L* were taken as between-subjects factors and positive affect (*PA*) and negative affect (*NA*) as covariates. For the valence/arousal ratings we found no main effect of the delivery method ($F(2,10)=3.00$, not significant) but a significant between-subjects effect of *TE* ($F(2,10)=6.78$, $p<.05$) and a significant interaction effect of Program**PAge* ($F(2,10)=7.82$, $p<.01$). Between-subjects, *TE* had a significant effect on the valence ratings ($F(1,11)=11.24$, $p<.01$) with more *TE* leading to higher ratings (cf. figure 3.22a). The interaction effect of Program**PAge* was significant for the valence ratings ($F(1,11)=14.66$, $p<.01$), and an analysis of the means showed that less experienced presenters gave higher valence ratings for the canvas presentation (5.97 to 7.01) while more experienced presenters gave higher ratings for the slideware presentation (5.25 to 7.46) (cf. figure 3.22b). The results from the analysis of SD ratings indicated a main effect of the delivery method ($F(11,1)=825.93$, $p<.05$), an interaction effect of Program**NA* ($F(11,1)=1481.85$, $p<.05$), an interaction effect of Program**PAge* ($F(11,1)=2249.68$, $p<.05$) and an interaction effect of Program**TE* ($F(11,1)=349.55$, $p<.05$). While the individual SD dimensions did not differ significantly between the programs, the overall trend was that the slideware presentation received more positive emotional response. The interaction effect of Program**PAge* was significant for *pleasantness* ($F(1,11)=8.4$, $p<.05$), *positivity* ($F(1,11)=10.11$, $p<.01$), *afraid* ($F(1,11)=14.67$, $p<.01$), *satisfaction* ($F(1,11)=20.38$, $p<.01$), *stress* ($F(1,11)=13.91$, $p<.01$),

Technically experienced presenters gave higher valence ratings.

Less experienced presenters gave higher valence ratings for the canvas presentation.

More experienced presenters rate their feelings as more positive in the slideware condition.

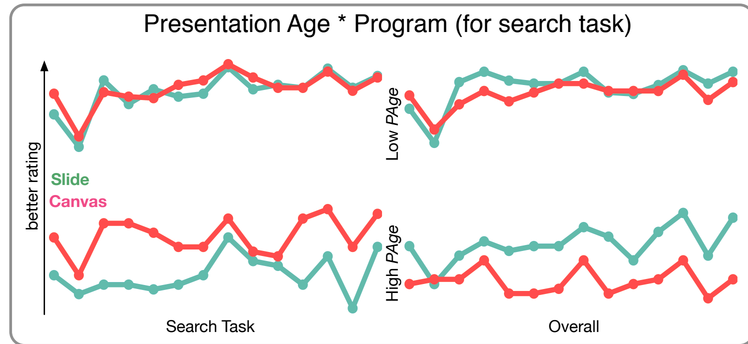


Figure 3.23: SD ratings show experienced presenters rated the search task differently from the trend. [Lichtschlag et al., 2015]

desperation (F(1,11)=5.81, p<.05), *controlled* (F(1,11)=6.01, p<.05), and *lost* ratings (F(1,11)=16.83, p<.01). More experienced presenters gave positive ratings for slideware on all these dimensions, while less experienced presenters showed only minor differences. The interaction effect of Program*TE was significant for surprise (F(1,11)=9.99, p<.01), unsatisfied (F(1,11)=9.21, p<.05), and lost ratings (F(1,11)=5.47, p<.05). Presenters who had less TE gave higher ratings for slideware on all these dimensions while presenters with more TE showed only minor differences. In conclusion, we accept H1.

More experienced presenters rate their feelings as more positive in the slideware condition.

Exploring the data, we noted that the ratings for the *search for a loosely defined position* task showed a flipped behavior. An interaction effect Program*$PAge$ occurred once again and valence values were significantly different for this task (F(1,11)=9.31, p<.05). Further analysis showed that there was a great difference between slideware and canvas presentations for experienced presenters, with canvas presentations having a better rating, while no difference was found for less experienced presenters (cf. figure 3.23).

To explore the error hypothesis (**H2**) we conducted two repeated-measures MANCOVAS with the Valence/Arousal ratings and the semantic differential (SD) ratings as dependent variables respectively. $PAge$, TE and L were taken as between-subjects factors and

positive affect (*PA*) and negative affect (*NA*) as covariates. The analysis of valence/arousal ratings indicated a main effect of the error condition ($F(2,10)=5.55$, $p<.05$), an interaction effect of Error*PA ($F(2,10)=6.62$, $p<.05$) and an interaction effect of Error*TE ($F(2,10)=5.88$, $p<.05$). Arousal ratings were significantly different between the error conditions ($F(1,11)=7.90$, $p<.05$), and examination of the means showed that arousal was rated higher in the condition without errors (cf. figure 3.22c). The interaction effect of Error*PA was significant for the arousal ratings ($F(1,11)=8.19$, $p<.05$), and the plot indicated that while the positive affect rating had no effect on the arousal rating in the no-error condition, it had a positive effect on the arousal ratings in the error condition. The interaction effect of Error*TE was significant neither for valence nor for arousal ratings. The results from the analysis of SD ratings indicated between-subjects effects of negative affect on stress ratings and of *TE* on nervousness, pleasantness and positive-negative ratings. Further analysis showed that higher *NA* ratings correlated with more experienced stress, and that lower *TE* participants felt more nervous, more unpleasant and more negative across both conditions. In conclusion, we accept H2.

Errors had an influence on the emotional state.

Checking the quality of counterbalancing (**H3**), we compared presentations by order and found no differences between ratings for the programs (Canvas: $F(2,18)<1$, *ns*; Slide: $F(2,18)<1$, *ns*) or the errors (error: $F(2,18)=1.73$, *ns*; no-error: $F(2,18)<1$, *ns*). Thus, we accept H3: the order of error or program condition did not have an influence.

As for **H4**, almost all (20) participants expressed that they felt similar to a real presentation. 14 mentioned that they felt less pressure since they had less stakes in the presentation, 10 mentioned an additional burden (e.g., the unfamiliarity of the topic), while 2 felt that the study was outright harder than their own presentations because of that. With this, we cautiously accept H4: the limitations of the lab study were manageable, and our setup was comparable to a real presentation.

Almost all expressed that the presentations felt real.

Discussion

Less technical expertise and high spatial ability preferred the canvas condition.

Our evaluation shows that presenters experience canvas and slide tools differently. Technically experienced presenters report a more positive emotion, independent of tool used. Furthermore, participants of our study that scored high on spatial ability or were less experienced preferred the canvas condition, while experienced or lower spatial ability presenters preferred classic slideware. Due to the strong overlap in our tester population between experience and lower spatial ability, we cannot attribute this effect to a single or a combination of these factors. We expected lower spatial ability to interact with the canvas condition due to its ZUI nature, but we could also explain that more experienced presenters are well versed in slideware and hence feel right at home. Interestingly, this difference is lessened in the *search for a loosely defined position*, a task that benefits particularly from the canvas format as the presenter can quickly zoom out to get an overview and pinpoint their target. Experienced presenters reported more positive feelings when using a canvas tool for this task.

The design of canvas presentation tools has to take into account that not all presenters prefer canvas navigation tools.

In summary, we have improved our understanding of canvas presentations and gained insight into the emotional state of the presenter. We found that the presentation format is again influencing the user's experience. The same freedom of navigation for which we found beneficial effects for authors turns out to be ambivalent while presenting. Here, time and attention are limited and, as we have seen, a simpler linear format can be easier to handle for some presenters (cf. Good [2003]'s concern). One could conclude to always limit the format during presentation delivery, but we have also shown that for some presenters this would be unfavorable. Future design could try to lessen this effect by giving the presenter a choice to give the presentation strictly linearly.

Limitations

By design, our study was a lab study, and therefore, it is a controlled situation that might not be representative of real-

life use. A field study with presenters presenting their own presentations, with their own agendas, on their own topic of expertise, and own audiences could corroborate the results of this paper or bring new results. We were unable to attribute the interaction effect of program use to spatial ability or presentation age, as both independent variables correlated. Other limitations are the short nature of presentations and the novelty of the canvas condition.

3.4.4 Investigating the Audience

Our next study looks at our last user group, the audience (cf. section 3.1.4 "The Audience"). All talks are given because they want to affect the audience an a way—entertainment, emotional state, knowledge. Here, we investigate presentations that aim to teach the audience about a subject. Previous research suggested that canvas presentations may be more beneficial for the the audience than traditional slideware [Good and Bederson, 2002]. Content macrostructure is more visible in canvas documents, and may be easier to grasp. This claim had been investigated by Good [2003] and he found that there was no significant effect in audience recall. Since the design of the canvas is Fly is quite different from CounterPoint, we decided to conduct a similar investigation. We compared canvas-based presentations and their slideware counterparts, measuring recall of facts, measuring understanding of macrostructure among audience members, and gathering results on their subjective assessment of these presentations. The following study was done in collaboration with Thomas Hess [Hess, 2011] and has been previously published as a peer-reviewed paper [Lichtschlag et al., 2012a].

We study the interaction of the audience with presentations authored in canvas tools.

There is an argument to be made for improved learning effects with the canvas layout. Plausible reasons for this are the increased visibility of the macrostructure, the potential to leverage the spatial abilities of the audience, and less fragmentation of the material. Content macrostructure and its relations can be incorporated into the spatial layout, and can be communicated to the audience implicitly through spatial overviews and animated viewport transi-

tions. The presenter hence does not need to express them verbally through written or spoken text, as is necessary in slide-based presentations. This may reduce cognitive load for the audience [O'Donnell et al., 2002]. In canvas presentations, successive viewports can overlap, so content can be presented in a less fragmented way: related topics stay together, and more relationships are represented spatially. Especially animations between viewports can offload some of the viewers cognitive burden to the human perceptual system by exploiting the perceptual phenomenon of object constancy that enables viewers to track element relationships without thinking about it [Robertson et al., 1991]. Good and Bederson [2001] argued that by shifting load from the verbal to the visual cognitive channel, the audience can exercise a larger portion of their memory resources, which would be especially useful in a presentation scenario in which the oral narration must be followed and processed continuously. On the other hand, there is an argument to be made against improved learning effects with the canvas layout. Audience members less skilled in spatial orientation might be overburdened by a canvas presentation. Following Clark [1994]'s argument that different media will not improve learning, a large corpus of studies exists that has not been able to document significant effects of different media on learning (cf. chapter 3.2.3 "No Significant Difference").

There are arguments for and against improved learning effects with different presentation software.

Study Design

We studied with slideware and canvas presentations on two topics.

In our study, we showed two instructional talks to two audiences (cf. figure 3.24). Each group separately attended two presentations of 15 minutes each on two different topics. One of the presentations used PowerPoint, the other Fly, representing the *slide deck* and *canvas* conditions. We formulated the following hypotheses:

H1: There will be no significant difference for fact retention between the canvas and slide deck condition.

H2: In the canvas condition, participants will perform better for macrostructure recall and transfer questions.

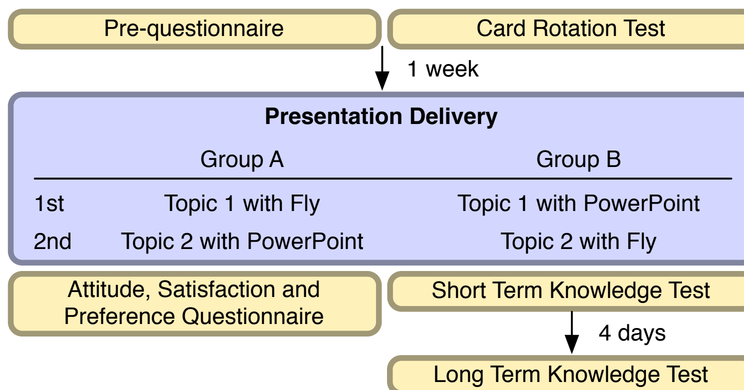


Figure 3.24: Audience study design. All participants performed the pre- and post-experiment activities (yellow). The main experiment (blue) split participants into two groups, switching the order of tools (not topics). [Lichtschlag et al., 2012a]

H3: The canvas visuals will provide participants with a better orientation of talk progression.

H4: Participants will find the structure of the canvas presentations easier to comprehend.

H5: Participants will find the amount of content shown on the screen at a time more adequate in the canvas condition.

We recruited our participants from the students of an introductory HCI course. Each participant was asked to fill out a pre-questionnaire and to assess their spatial ability using the card rotation test [Ekstrom et al., 1979]. According to this data, students were divided into two groups with a counterbalanced mix of different ages and spatial vs. verbal learners. Students with prior knowledge in any of the two topics or those that were not very proficient in the language of the presentation were excluded from the experiment. This resulted in 26 participants in total, 23 male, 3 female, 13 per group. Ages ranged from 23 to 35 (median 27).

We made sure that audiences had no prior knowledge of the topics.

To understand the talks no prior knowledge about the content was needed; both topics, "Fixed-Gear Bicycles" and "Convergent Evolution", were uncommon and independent from each other. We counterbalanced the order of the canvas and slideware conditions, but not the order of topics, since the latter was unlikely to create learning effects.

Presentation documents were authored to be of equal quality, yet with their innate characteristics.

An important factor in the experiment was the authoring quality of the presentation documents. Since presentation visuals for each of the two topics were needed in both formats, the challenge was to ensure that the documents on the same topic contained the same content. Because of the fundamentally different formats of Fly and PowerPoint, there was no exact way to match document content. To reduce the risk for bias, an external and experienced presentation author (31 years) who was not otherwise involved with the study created all four documents.

Canvas presentations tended to be more graphical, slideware used textual overviews.

In the resulting documents, the Fly visuals contained more unique layouts compared to an image with bullet points, and were more verbose for some sections. Some layouts were only possible because of the canvas-based format and could not be adapted to PowerPoint. The biggest layout difference occurred in the convergent evolution talk: the development history and present-day distribution of marsupials was integrated into a big timeline layout with an illustration of geologic eras. In PowerPoint, for the same topic, the development history was covered with a series of text-based slides that showed one era each with the illustration on an extra slide (cf. figure 3.25 top). Naturally, the overviews in Fly used more graphics and were structured spatially, while the overviews in PowerPoint used more text and were structured linearly. For example, for the introductory section of the fixed-gear bicycle presentation, a large graphic of a bicycle was used for the background on which the explanations of the basic concepts of bicycle technology were placed (cf. figure 3.25 bottom). The Fly visuals contained additional overviews that previewed and recapitulated single topics.

Personality, mood, and performance of a speaker and the interaction between speaker and audience can have a strong impact on the quality of a presentation. To mini-

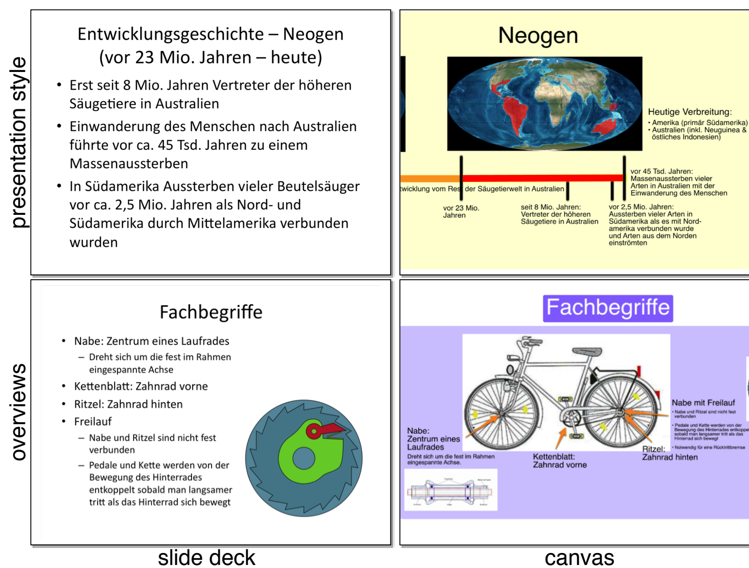


Figure 3.25: Presentation style and overviews in the slideware (left) and canvas (right) condition. [Lichtschlag et al., 2012a]

mize bias through speaker/audience interaction or different speaker performances, we presented all talks as prerecorded video presentations. Without the speaker physically present, it was important to have an engaging narrative, hence a professional broadcast speaker recorded the spoken commentary. Spoken texts were kept simple and informal to match a face-to-face presentation. To avoid differences in pronunciation, emphasis, and elaborateness, both conditions shared the same audio material; the recordings were split into segments that could be mapped to the visuals of both formats. Ellis and Mathis [1985] showed that using recordings instead of a live presentation does not have a significant influence on learning.

All talks were recorded previously to reduce presenter bias.

After both presentations were over, participants immediately filled out a *short term knowledge* test and a *preference and commentary* questionnaire (cf. figure 3.24). Four days later, they filled out a *long term knowledge* test. To measure knowledge transfer, the two knowledge tests asked for content and macrostructure facts with retention questions, and for content understanding with problem-solving trans-

We measured outcome variables with knowledge tests and a questionnaire.

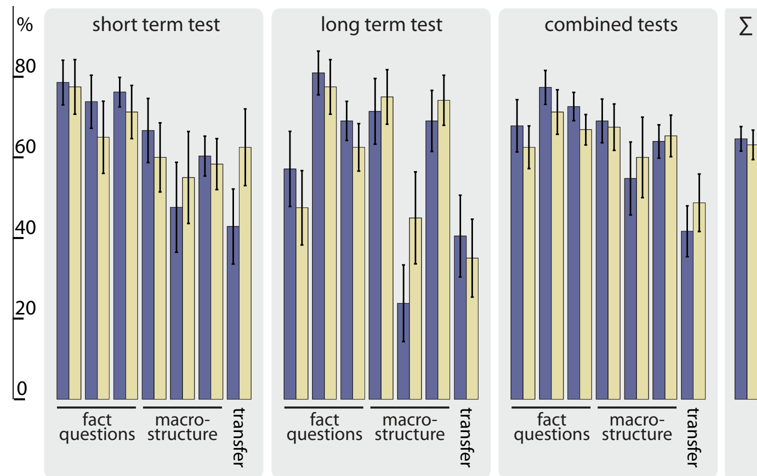


Figure 3.26: Percentage of correct answers to questions by presentation method for the two tests by question categories, combined tests, and all questions (canvas=dark blue, slideware=light green, error bars ±1 *SE*). Both techniques performed equally well in terms of retaining facts, structure, and transferring knowledge. Only the short-term knowledge transfer question shows a significant difference. [Lichtschlag et al., 2012a]

fer questions. To gain insight into the participants' attitude towards and satisfaction with the presentations, the questionnaire asked several Likert scale questions (cf. table 3.2).

Study Results

We found only one instance of presentation condition influence.

Figure 3.26 shows knowledge test results for both conditions. For the short term transfer questions, the mean score is higher in the slide condition (paired t-test, p=.029). However, this is the only significant difference for all question categories.

Regarding the differences between short term and long term recall, the only significant result was that group A performed worse in the long term test for the slide condition (paired t-test, p=.003). Regarding the comparability of topics, paired t-tests did not show significantly more correct

answers for any topic in the short term tests (p=.071), long term tests (p=.352), and in total (p=.145).

Table 3.2 lists the responses to our attitude and satisfaction questionnaire. A related samples Wilcoxon signed rank test showed that participants significantly preferred the canvas over the slide condition in questions A4, S5, S6, and S7. No other differences were significant.

The audience preferred canvas presentations.

In the spatial cognitive ability test, participants received a mean score of 127.32 (SD=21.011) out of 160. An independent samples t-test showed no significant difference between the mean scores of both groups (p=.837). We found no correlation between spatial ability and percentage of correct answers for any groups, talks, or conditions. Interestingly, the higher spatial cognitive ability, the more individuals found the amount of content on the screen (S2) too much (Pearson's r=.469: p=.003). However, we found no correlation between spatial ability and format preference.

Spatial ability showed no influence on learning outcome.

	Question	*Mdn* (canvas / slide)	*p*
A1	The presentation was interesting.	2 / 2	0.756
A2	I liked the presentation visuals.	2 / 2	0.204
A3	I liked the narration of the presentation.	2 / 2	0.248
A4	I liked the presentation overall.	**2 / 3**	**0.047**
S1	The speed of the presentation was too slow.	3 / 3	1.000
S2	The amount of content shown on the screen at once was too much.	3 / 2	0.058
S3	The visuals distracted me from the narration.	4 / 4	0.755
S4	I had sufficient time to look at all the content.	2 / 3	0.805
S5	The structure of the talk was easy to understand.	**1 / 3**	**0.048**
S6	I always knew which part was currently shown.	**1 / 3**	**0.006**
S7	I always knew how far the talk had progressed.	**2 / 3**	**0.001**

Table 3.2: The questions (Likert scale 1–5, 1 for strongest agreement) from the attitude and satisfaction questionnaire. A related samples Wilcoxon signed rank test shows significant difference in four cases, all in favor of canvas presentations. [Lichtschlag et al., 2012a]

Although we tried to balance members between groups, a t-test found group B performed slightly better than group A in both questionnaires. However, the difference is only significant for macrostructure retention (p=.080). Also, group A found the presentation visuals less distracting (p=.043).

Discussion

Based on the fact recall results, we can accept H1. As the factual information is represented similarly in both formats, it seems that the presentation form alone does not influence fact retention. Although we expected the spatial arrangements in the canvas condition to help participants understand relations between topics (H2), the results do not support this. Consequently, there is no evidence to suggest that either canvas presentations or slide based ones are better suited to convey information to an audience. This is very similar to Good [2003]'s results with CounterPoint. However, the results support H3; there were significant differences in favor of the canvas condition for the statements about orientation in content (S6) and temporal progress (S7). We also accept H4; participants found the canvas structures easier to understand (S5). On average, participants found the amount of content on the screen at a time more adequate in the canvas condition (H5), but this result was not significant (S2).

We replicate Good's results.

We also find that the audience preferred the canvas presentation.

We evaluated the canvas presentation format against a baseline slide deck format to investigate the effects of canvas presentations on the learning performance and preferences of a student audience. Learning performance was largely the same in both cases, underlining Clark [1994]'s stance on learning once again. We found no influence of spatial ability. As also found that students clearly preferred the canvas-based presentation. Several limitations have to be kept in mind when interpreting these results. First, canvas-based visuals are still new and exciting, which may have influenced participants. Second, our study used educational presentations with a focus on knowledge transfer. Other talks primarily focussed on conveying motivation, emotion, etc., may benefit even more from a canvas layout, again partly due to its novelty. Third, the talks were rather short and author and audience were informed beforehand of the study design and the knowledge tests. They may therefore have put more effort into their performance (cf. “Hawthorne effect”).

3.5 Outlook

Now we have presented the results from five studies on the effects of canvas presentations on the various users. We review what we have learned about canvas presentations and zoomable user interfaces in chapter 6 "Discussion". In the next chapter, we investigate our second domain, integrated development environments, and investigate if we can also find a positive influence from canvas designs on user actions.

Chapter 4

The Code Base on a Canvas

We have a code annotation tool at work, but the only thing I ever want to write with it is "See me" in red ink.

—John Siracusa (@siracusa), 2014

As promised, we also investigate a second domain: *integrated development environments* or *IDEs*. IDEs are software suites that allow the developer to build and investigate software. Before we can make suggestions to IDE design, we must understand what it takes to build and investigate software. We need to have a closer look at what code is, and what is means to work with it.

Source code text is not only used as the language to the machine. Code is also subject of discussion and communication between developers. Therefore, code has a dual function: is communicates our intentions to the compiler and it also documents our intentions to other humans (or our future self). Of course, the communication between humans is more important to us. And it is our goal of this chapter to investigate how one can foster this communication on a canvas.

Two problems make hard to bring canvas designs to IDEs. First, the task domain is tough: software design is particularly hard for a number of reasons we outline below. This problem is constant for all approaches to visualize code bases and reason about them. Second, text is traditionally hard to present in zoomable environments. Which makes IDEs a good testbed for the designs we propose.

In this chapter we apply canvas designs to code bases.

We present three approaches with our prototypes to address this second problem: Our approaches are a design based on the *vocabulary* used in a code base, a design based on *software design patterns*, and a design based on *hand-drawn sketching*. We base each prototype in the related work and then present a prototype for each in the following chapter. Finally, we review and evaluate these approaches and our designs.

4.1 The Task and the User

When we consider programming, is it not curious that the product of our work is named *"source code"*? The word stems from Latin, "code" referring to early books—written stories. In French, "code" means rules and laws. In modern use, when not specifying the source of a program, one would use code to mean something that is ciphered.[1] All of these meanings describe parts of the developers work: it is a lot of writing in a specific set of rules. And it is also often very hard to understand and hard to decipher as the work code suggests.

Source code is often very unapproachable.

Reading code can be unsatisfying, even for experts, we can describe this by an example. We developed a software at the media computing group, *Personal Orchestra*. The complete code base has has 845 files and roughly 72000 lines of code. We are in the process of handing the software over to a new team and training them on the designs embodied in the code base. When a new team member opens the code base in the IDE, they will find no entry point, no narrative,

[1]Or a treasure that can be sought, as in "Indiana Jones and the Crystal Codex".

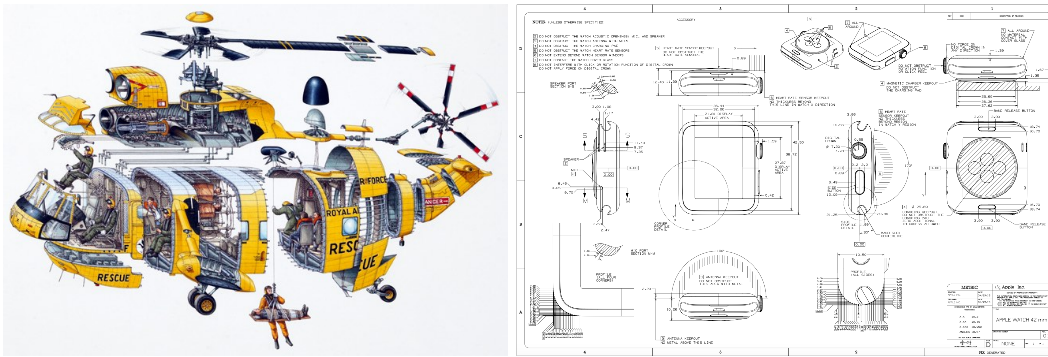


Figure 4.1: Big picture views for physical objects. On the left, a cross-section from [Biesty and Platt, 1992], on the right, a technical drawing of the Apple Watch [Apple, 2015d]. In both cases the reader gets an high-level overview and can get closer for the details.

and no story that is told to them. The way the software is presented in the IDE will only reveal the text, not the structure of its design. Each time one reads the source code one literally reads a 'code' that has to be deciphered, its meaning deconstructed and integrated into a model of the inner workings of the software in the developers head.

4.1.1 Physical Analogy

We can compare code visibility to physical engineering domains. Consider to a car: A car is a very high tech product and it has a visual hierarchy to it. Changing the car from one generation to the next allows the spectator to immediately see the high-level differences (e.g., shape or seat configuration). The engineer can also 'pop the hood' and see that parts of the engine laid out, focus on the part that interests him, and then look closer at the details. The physical properties give the product a form and a hierarchy. As one steps away from a physical part, it scales naturally to a smaller representation and reveals the parts it is connected with (cf. figure 4.1). Furthermore, the complexity of connections to other components is bounded to its surface area and the amount of miniaturization is bounded by the engineering process.

Physical objects have an inherent form to them.

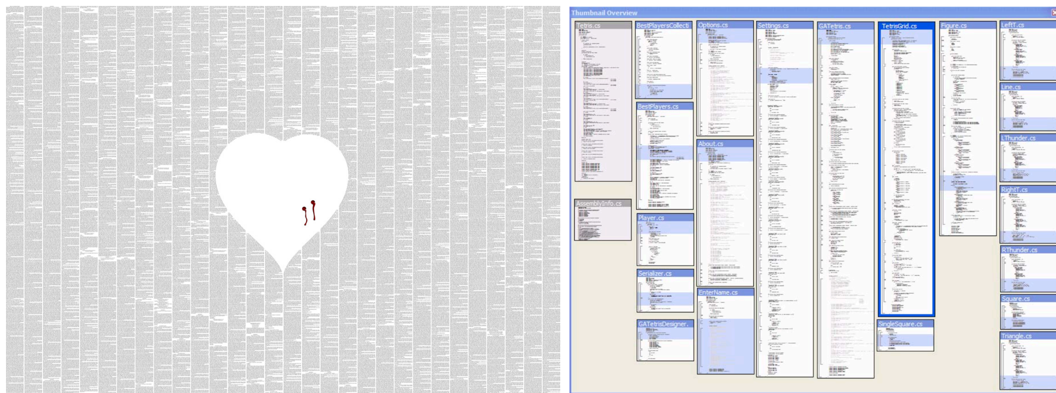


Figure 4.2: Two examples of text scaling. On the left, the complete text of Bram Stoker's novel "Dracula" is rendered as a single poster for artistic effect (from [SpinelessClassics, 2015]). On the right, a whole code base rendered in a similar fashion (from [DeLine et al., 2006])

Text is without inherent form.

Compare that to code and text: it scales poorly (cf. 2.6 "Research Questions for Zoomable User Interfaces"), rendering a small representation reveals little about its structure and looks mostly the same. Even though this can be a nice artistic device, it does not allow one to grasp much of the structure of the text (cf. figure 4.2).[2] To make things worse, the complexity is neither bounded in the amount of pieces it can be connected to or the parts it can be composed of.

IDEs give little form to a code base.

No physical parallel is immediately available for code. Files and class hierarchies come to mind. Files are often divided on technical, not semantic grounds. The aforementioned project has 845 files and this is a poor way to structure, it is not apparent how the files interact with another, where a design is located, etc. Class hierarchies describe the way the classes derive behavior from another, but do not talk about their composition at runtime. Each class is a prototype for an implementation, but only the instantiated objects describe the behavior of the program. E.g., it is of less use to know that an *NSView* is also an *NSResponder*, when the configuration of views and their customization is the way we describe how the program works.

As the diligent reader is certainly aware, ZUIs were con-

[2] Little care is taken to make a novel browsable other than just reading it from front to back.

ceptualized be 'physical' according to Bederson and Hollan [1994] (cf. chapter 2.2.4 "Pad++: Metaphor-free Navigation"). Their motivation was to bring the sense of place and hierarchy to the digital world. They write: "an effective informational physics might arrange for useful representations to be a natural product of normal activity". When we apply ZUI designs to code bases, we can hope to leverage some of the physical benefits.

Zoomable user interfaces strife for form through physical analogy.

We can also draw a comparison to our investigation of presenting in the previous chapter. Presenting our dear colleagues with 845 files is akin to taking the reference material to a talk an dumping it to the reader's feet and say: 'go work on that'. Or we can compare it to the author of this thesis zipping a folder of the related work and say 'everything is in here', but not tell a story about it. Very justified, a reader would be unsatisfied and decry the talk as bad. Why do we tolerate such behavior for source code, and why have we not produced better tools to communicate and reason about software?

4.1.2 The Relation Between Writing and Coding

In investigating how source code and how users interact with it 'regular' written text is often used as a reference. Détienne [2002] presents an in-depth investigation and a history of cognitive models for written texts. She describes how they have been applied to software development and reviews studies that evaluate the models. The *structural*, *functional*, *mental model* explanations how coders work have been developed for narrative writing before they were adapted to coding. We present them below to motivate our work and cross reference with newer studies.

Writing is often used as a reference for coding.

Surface and *deep structures* are two key concepts from models about narrative text that are easily adapted to code texts. Surface structures are the textual structure as it is presented to the reader, the letters and the paragraphs rendered in the IDE. This surface structure provides a poor visibility of the schemas in the code base (cf. figure 4.2). The deep structure "corresponds to the relations that are not explicit in the sur-

Surface and deep structure are concepts to reason about narrative texts and code texts.

face structure" [Détienne, 2002]. An example of deep structure in software is easy to find: control structure such as a loop. In this case the surface structure is a linear list of statements, but the deep structure is non-linear. Another example is a variable that is defined earlier and has has to be kept in mind for later statements. In these examples, the flow of the execution and the state of the variable are the deep structure of the code. Since these terms were defined for narrative texts first one can find examples there as well, e.g., one can consider a story that jumps to different locations (or times) between scenes, or a character that is only characterized by his actions and the reader has to deduce his intentions[3]. This 'unwritten' knowledge makes reasoning about code and narratives hard and a great deal of work has been spent on investigating how people approach this deep structure. We present a short overview of this investigation below and will continue to draw comparisons to narrative storytelling.

4.1.3 Why is Reasoning About Code So Hard?

Surface structure in narrative texts (and presenting) addresses the audience. Not so for code. Here, the surface structure addresses the computer and human readers alike. One could hope that with higher level programming languages and abstraction over machine code, the discourse might have shifted more to the human side. For example, the naming of variables and other *natural language* text only addresses to the human reader and not to the compiler. But, for the developer the main problem is clearly communicating to the machine. Any concern about communicating to colleagues or a future self is secondary to a bug-free implementation.

Code is hard to understand because it primarily addresses machines.

We can consider an example object in a program. An object describes a range of actors and is often rather bland without customization either by subclassing or by specification after instantiation. In writing and reasoning about the class one has to keep in mind all the ways that the class might be instantiated at the same time. One does not build

[3] Often considered a mark of good writing.

a model of the surface structure (the code text), but the deep structure (all the ways that the algorithm can be applied). The user reasons on a very abstract level. Compare this to a narrative text: a variable actor is boring, even 'bland'. A good writer wants to characterize the actor, make him unique and concrete. Where code has the potential to talk about whole classes of actors in an abstract way, a story is most likely dealing with only an instance of that.

Code is hard to understand because it describes abstract objects.

We can consider a second comparison: a program that sorts an array, e.g., 'sort(array)'. A program has to deal with the ambitious input variable 'array': it can be a short or a long array, in might be nil, the objects in the array might be comparable or not. Maybe there are assumptions about the variable array that are spelled out in the header or the documentation such as it being not null, only used in certain contexts. The experienced programmer will undoubtedly notice a range of possibilities that the use of this function could go wrong.[4] One has to think about a wide field of input possibilities, hence the name 'variable'. Comparing that to telling a story in a talk or writing, the question of sorting would likely not come up: The author of the talk or story has already done the work, they present an unvariable amount of content and they present them in a defined order. Again we see that programming is easily more *abstract* than storytelling.

Code is hard to understand because one has to consider all the boundary conditions.

Another important problem for code comprehension is that it is very *multiplexed*. Reasoning about objects and operations often follow multiple paths at the same time, e.g., when plans about code intersect and cross over each other. Any code of relevant size is inherently multiplexed through the use of classes, jumps between functions, concurrent operations, etc. Such a code 'tells' many stories at the same time. Détienne [2002] gives one example for written texts where narrative text is similarly multiplexed and that is murder mysteries. One could also think about time travel stories, or stories with many actors[5] in concurrent storylines that could overwhelm a reader. Consider the figure 4.3 of the short story "All You Zombies" by Robert Hein-

Code is hard to understand because is multiplexed.

[4] Especially in the JavaScript style we presented the syntax of the function, lacking crucial information about its assumptions.

[5] Looking at you, Lord of the Rings and Game of Thrones.

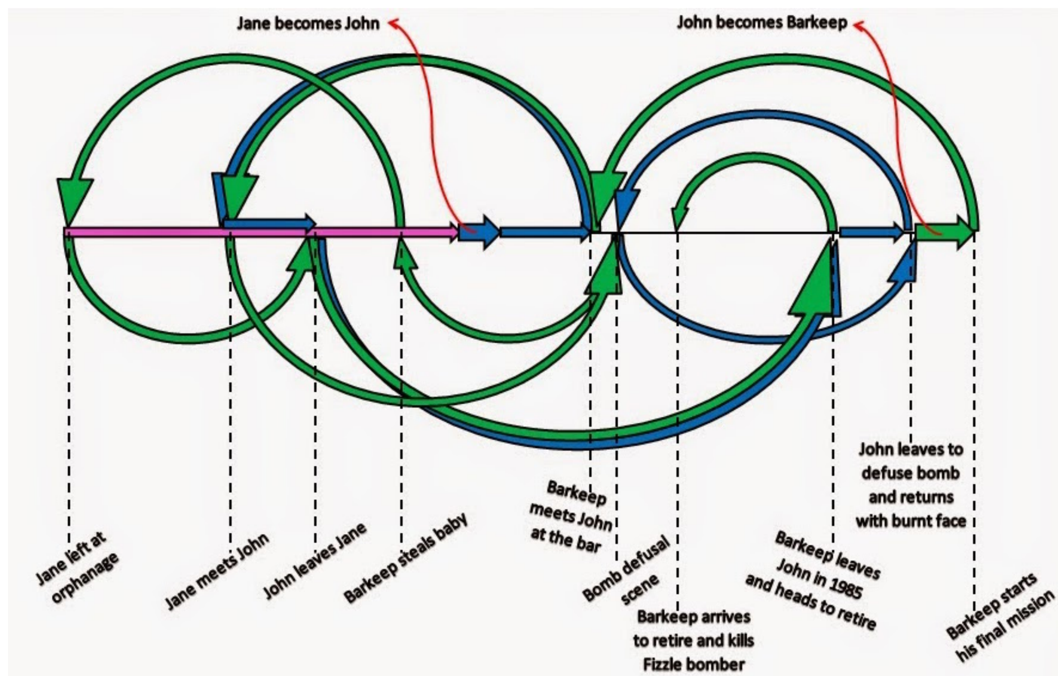


Figure 4.3: The story for the short story "All You Zombies" by Robert A. Heinlein [Heinlein, 1959]. A series of time travels happen and three characters turn out to be the same person. This intentionally overwhelms and confuses the reader. In code such a call graph would not be uncommon. Explanation from [Pyrotechnics, 2014]

lein: here the story is intentional complex, it is part of the allure of the story. Many readers of the story or viewers of the film will want to review it again to get all the details and this figure is a nice high-level explanation of its events. But those narrative texts are clearly outliers. The closest thing available for code bases are call graphs, but based on the sheer amount of calls these get much out of hand if not filtered.

Code is hard to understand because the interesting parts are hard to find.

Also, in investigating a code base, the reader has to contrast between important and unimportant information, and then select the right content to read. Since one has to be diligent in talking the machine and leave no details undefined, a huge deal of the surface structure of a code base is often 'glue code': architecture that is needed to get to the interesting statements. In investigating a code base, a developer has to identify the important parts but often the only approach to do so, is to read everything first. Huge parts of

understanding code deal with reading boring parts. A narrative does not have this problem, at least when it is expertly told: the writer will take care to tell about the parts that are important to the story they want to tell, talk only about them, and leave the rest to the imagination.[6]

4.1.4 Cognitive Models of Software Design

Détienne [2002] presents multiple theoretical approaches to studying software design, each of which is appropriated from studying the design of narrative texts. We present a short summary of the results here, and the reader should look at Détienne [2002]'s excellent analysis for a detailed discussion.

Central to the formulation of expert knowledge is the term *schematic knowledge*, which describes a reusable approach of solving a problem in the memory of the developer. In that way, the definition is very similar to the definition of *software design patterns*: "a general reusable solution to a commonly occurring problem within a given context in software design" [Gamma et al., 1994, Borchers, 2001]. For example, a familiarity with a *model view controller* design pattern is a schema and allows the developer to formulate their software design with that vocabulary, or allows them spot the schema in an existing code base and quickly parse larger parts of the code base. But schemas are less formalized than design patterns, they are not written down to communicate between developers, and Détienne [2002] notes that the assumption that multiple developers would build the same schemas of the same text is false. We can also interpret much of basic programming teaching in computer science to teaching students about basic schemas as reusable components of problem solving, e.g., linear lists or binary trees.

Knowledge about schemas describes an reusable approach of solving a problem.

The idea of *strategic knowledge* builds on that and further describes expert knowledge that extends beyond the individual schemas. It describes a hierarchy of high level schemas that group schemas to a strategy towards solving a prob-

Strategic knowledge describes a hierarchy of solutions.

[6] If only we could program this way.

lem. In that sense the idea is close to the definition of a *pattern language* [Gamma et al., 1994, Borchers, 2001], but again, less codified and not necessarily written down. And again, it makes a high difference in the model of the cognitive processes of a developer, when one considers an expert with experience codified in strategic knowledge compared to a more novice developer without that code knowledge in their head. For example, this leads to different strategies in software authoring: an expert is more likely to plan the software top down (strategic schema), a novice is more likely to start with a part of the solution (a single schema) and build bottom up from there [Détienne, 2002].

4.1.5 User Tasks

We previously presented three prototypical user groups in 2.6.2 "Authoring" and applied this lens to look at the users of presentations in 3.1 "The Task and the User". Similarly, Détienne [2002] develops the theoretical models for the three following user groups: *design of software*, *reuse of software*, and *understanding of completed software*. LaToza et al. [2006] investigates the roles that programmers at Microsoft fill and classifies as six groups: *designing*, *writing*, *understanding*, *editing*, *unit testing*, and *communicating*. We see that the range of tasks is much more complex than with presenting. In their definition, editing and unit testing refer to the same activities as Détienne's reuse activities. The highlight of the paper is the focus on collaborative interactions (designing, communicating) in the developer team and how the team records (or does not record) knowledge about source bases. These tasks deal with communicating the mental models of the code base, ensuring the same schematic and strategic knowledge between collaborators. These tasks are often done away from the computer, e.g., in conversations or with whiteboards and seen little consideration in the earlier literature. The navigation in the code base happens in all of the tasks that interact with the code and has been studied extensively for a wide range of tasks (e.g., LaToza and Myers [2010b], Storey et al. [2007], Piorkowski et al. [2011]). The amount of time spent on the activities is very dependent on the state of the software and

Collaboration between team members is an important task.

its release state [LaToza et al., 2006].

Comparing to our investigation of presenting, the tasks one encounters when dealing with a source base are much more varied. E.g., Ko et al. [2005] studied time spent during a *change task* (reuse, editing categories respectively) and recorded 28% time spend on reading code or documentation, 20% on editing, 34% on navigation or searching, and the rest on other activities. So all three roles (author, navigator, audience) that we laid out in 2.6.2 "Authoring" are represented in the observed behavior: users create new content, navigate in the source base, and reason about existing content. But, a key distinction between presenters and coders is that we assumed the presenter to have near perfect knowledge about the presentation document. Clearly, we cannot assume the same for coders in a source base except for the smallest programs. E.g., a bug-fixing task is likely to include understanding the code base, navigating to the problematic section and its references, and authoring a solution. We see that a user potentially personifies all roles during a user task. Below we explore the tasks and problems in these groups in more detail.

Our lens of author, navigator, audience does not directly map to the tasks in code bases.

Authors of a Code Base

Authors of a code base build a solution to a translation problem according to Détienne [2002]: They have to keep in mind a model of the problem domain and a model of the computing domain, and translate between them. Often the problem domain is not completely defined and thus software design (and narrative writing) is iterative cyclical between planning, executing, reviewing. All of this is accompanied by note-taking in which the developer or author of a narrative text externalizes their schematic knowledge and builds a representation of that knowledge in the world [Détienne, 2002]. Program design then consists of activating schemas and applying them, be it individual schemas in what is called the *knowledge centered approach* for isolated problems or the use of strategic knowledge in building a hierarchy of schemas (*strategy centered approach*). These cognitive models of developer activity have been verified by a

Two cognitive models describe authoring code.

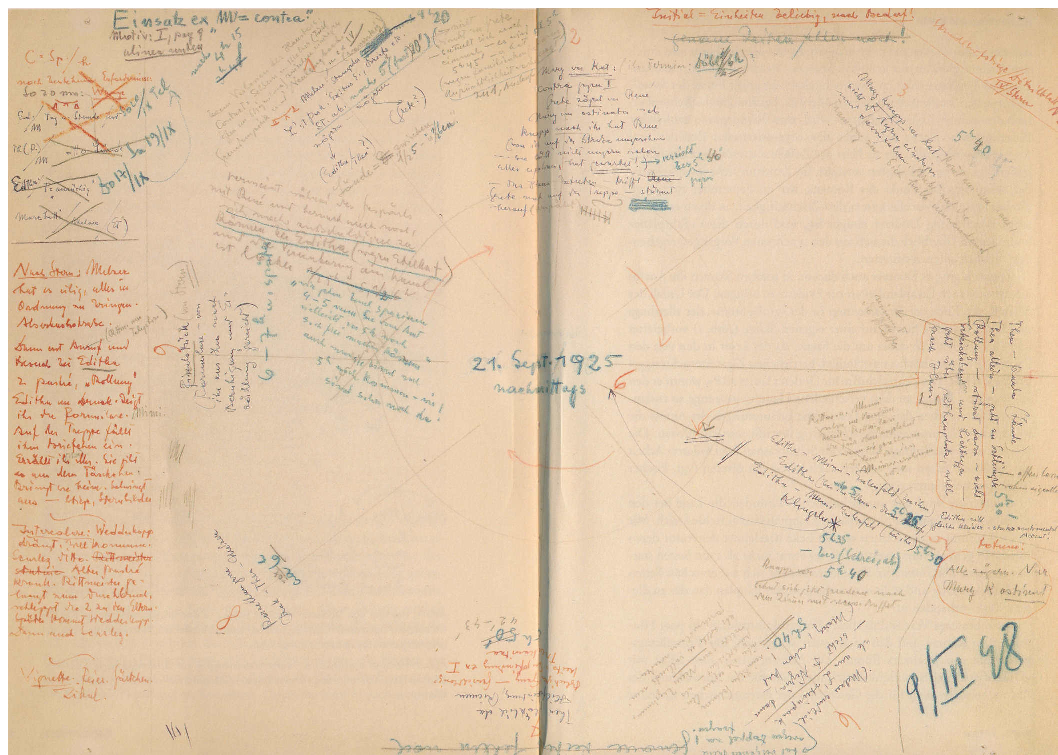


Figure 4.4: Canvas layout of "Die Strudlhofstiege Oder Melzer Und Die Tiefe Der Jahre" by Heimito von Doderer [Von Doderer, 1995] as developed by the author while writing it.

series of studies (cf. [Détienne, 2002] for more details).

As mentioned above these activities have originally been studied with narrative texts. Plachta [1997] goes into much detail how these fragments of externalized knowledge can be understood to reason about the author's intention. Authors sometimes leave fragments of their externalized models for the construction of the narrative and literature science studies them to understand the author's work process. See figure 4.4 for a canvas layout for a novel depicting character plot points over time [Von Doderer, 1995, Fetz and Kastberger, 1998].

Authors externalize mental models.

LaToza et al. [2006] studied the design of code and found that "[...] despite the availability of high-level views of code and visual editors such as tools for UML, developers remain focused on the code itself. Developers reported using

a source code editor the most for design while paper and whiteboards were perceived most effective". There is an interesting contradiction between the fact that developers clearly spend a great deal of their time externalizing mental models while authoring code (∼42%), yet the means they choose are predominantly volatile—–whiteboards drawings, paper drawings, notebooks. They almost never use UML or similar structured design tools (cf. [Cherubini et al., 2007]). The developer develops their mental model but the knowledge is not permanently recorded other than their memory: "Lots of design information is kept in people's heads.". When performing work on 'foreign' existing code, e.g., to fix a bug, the author is often trying to find the 'owner' to discuss changes. Rarely do they seek out design documents, because they are perceived to be out of date. [LaToza et al., 2006, LaToza and Myers, 2010c]

Authors look for the code 'owner'.

Détienne [2002] formulates a series of recommendations for IDEs to better support the design of software. She recommends tools that are based on the notion of structural schema, to help structuring programs on both the micro and macro level. Her second recommendation is the inclusion of knowledge schemas that note their preconditions, examples of use, and alternate schemas for the same problem. And this is almost exactly what design patterns are. She imagines both these schemas to be visualized in the IDE so that is allows "semantically connected but noncontiguous elements of the code to be grouped together visually". This overlaps nicely with LaToza et al. [2006]'s recommendations. They suggest hyperlinking code to design documents to promote their use, and a way to capture informal hand drawn designs.

Studies suggest integration of schematic knowledge in the IDE by hyperlinking design documents.

A situation that is studied separately from a clean slate authoring of new code is the reuse of existing code, e.g., as part of a library or API (cf. [Duala-Ekoko and Robillard, 2012] for a newer investigation). Here, the developer maps between the model of the target situation and the existing source situation. In this case, *new code reuse* describes the development of new code fragments to bridge between the existing code and the target, when they are not directly applicable. But how does the developer identify a proper source solution? Again, Détienne [2002] recommends to

Visibility of schemas in a code base helps reuse.

clearly identify schemas in existing code with preconditions and applicability, and examples to help the developer pick a source. Object oriented languages allow the developer to build reusable components in the form of classes. Studies showed that solutions get more similar when developers use a common object oriented API and help build schemas. But, it has also been indicated that *objects-oriented programming* leads to more multiplexed code (a more complex deep structure) and that the objects do not map to the strategic knowledge of the programmer but should rather be seen as orthogonal to their plans [Détienne, 2002, Davies et al., 1995].

Navigation in a Code Base

Many studies investigate the way developers navigate code. We present representative findings from two recent studies below. Ko et al. [2005] investigate which facilities of the Eclipse IDE are used to navigate and they report *syntactic navigations* (e.g., declarations, type definitions) that are directly supported by the IDE at 24% of all navigations, usage of symbols (e.g., where a variable name appear in the code base) performed with text searches at 22%. A dominant 42% of navigations are unsupported at all because they follow an indirect relationship, that is a relationship that represents a deep structure of the code, but is not available to the IDE to follow. 25% of the recorded navigations returned to the code they had just navigated from ("there and back again") and a total of 35% all time on the task was spent navigating.

Studies suggest that IDEs do not aid that navigation following schemas or strategic knowledge.

The second set of studies [LaToza and Myers, 2010a,b,c] clearly identifies that in trying to understand the code base, developers "ask reachability questions". They divide the questions into 'downstream' (e.g., "How do calls flow across process boundaries?", "How does this code interact with libraries?") and 'upstream' questions (e.g., "What is the 'correct' way to use or access this data structure?", "When during execution is this method called?"). In the language of cognitive models these questions correspond to searches along the structural model of the code base, in-

Developers ask questions about strategic knowledge.

vestigating connections between schemas or objects. LaToza and Myers [2010b] note that tool support to navigate to answers to these questions is sometimes lacking. They recommend that "developers could benefit greatly from diagrams that are more focussed on task-relevant items".

Since developers often reported getting lost when navigating code, LaToza et al. suggest that the IDE should allow the context of the task to be externalized—methods they have examined, decisions in progress, and other information. This information then can help the author to resume their work after interruptions, and also can serve as a starting point for documentation for other team members.

LaToza et al. [2006]'s studies suggest to visualize the task of the programmer.

Understanding a Code Base

Again, understanding of narrative texts is used as a template for understanding source code. Readers use three sources to understand source code: the surface structure of the source code, external representation (e.g., documentation), and knowledge stored in memory. Détienne [2002] presents three models adapted from narrative texts: the *functional*, the *structural*, and the *mental model* approaches.

The *functional approach* explains understanding code as applying previous knowledge in the form of schemas to the code base, either in a top down or bottom up approach. This model works well to explain expert behavior and recognizable patterns, and again is a good motivation visualizations based on design patterns. Presenting schemas in the IDE would be beneficial to understand the code. In narrative texts, we find these schemas in archetypes of stories, e.g., a hero's journey or a formalized way to write papers. Studies validate this model up to the point where one can show that the schemas replace the surface structure of the code as the way they are stored by memory by the developer. This can lead errors, e.g., when a running variable in a loop is remembered as 'i', even though the real name is actually different (*distorted recall*). We should keep in mind that according to the schema model, one has to understand the patterns to understand the software. Hence, readers

The *functional approach* explains understanding code as applying previous knowledge in the form of schemas to the code base.

have to know the schemas already, otherwise they are not able to detect them in code. The functional model is apt to differentiate between novices and experts. [Détienne, 2002]

The *structural model* approach explains references between code elements, forms a hierarchical model of structures. E.g., a developer might follow the input data through entry to the system, processing on the high level, an processing algorithm on a middle level, to a method on the low level. In doing so, they link the schemas of the program together along the lines of communication. This is similar to investigating a dynamic hierarchy at runtime, rather than the static class hierarchy. This model of a structure of schemas has a resemblance to a pattern languages comprised of individual patterns. In writing, an example of applying this model would be the formalism of a scientific paper, e.g., ways to perform a evaluation, or to reason about limitations, and how keywords communicate references between the sections. Unfortunately, this model has seen less validation through studies. [Détienne, 2002]

The *structural model* approach explains references between code elements.

The *mental model* is the most complicated model and builds a series of models for different views of understanding. Without going into too much detail, it can be easily understood as a combination and extension of the functional and structural models. According to it, the developer builds a *program model* for the schematic structure of the text (cf. functional model), and a *situational model* reflecting relationships between entities (cf. structural model). The program model includes both reasoning about high and low level schemas, and the situational model includes the reasoning about dynamic and static relationships between objects. The mental model is well evaluated, is well suited to understand reasoning with object oriented languages, and can take the programmers task into consideration. [Détienne, 2002]

The *mental model* is a well evaluated extension and combination of the functional approach and the structure model.

Looking closer at how task and presented structure ease or hinder understanding, traditional models for narrative text comprehension name *reading-to-recall* and *reading-to-do* as two tasks [Détienne, 2002]. A person reading-to-recall wants to understand the whole, e.g., to write a summary, whereas reading-to-do models tasks where one

wants to make a change and only reads as far as needed. Reading-to-do looks for direct relevance (the text that needs change), the surrounding (what sections are impacted by the change), strategic relevance (e.g., the closest superstructure). This works well to understand most of programmer's tasks, e.g., bug fixing, an extension of functionality, etc. But since text is linear, yet mental and situational models are not, three discontinuities have been identified: *temporal*, *spatial*, and *causal*. Temporal discontinuities describe execution that is not happening in procedural order, such as call jumps or multithreading, spatial discontinuities describe calls that leave the view of the reader, like a jump to another file, and causal discontinuities describe unclear relations, e.g., a method does two unrelated tasks. All of these have analogues in narrative text when characters and locations change or a scene does not causally follow the previous one. In each discontinuity case, the reader has to delinearize the text to build the multidimensional model that is the deep structure of the code base, such as multiple references to an instance, handover of data between classes. Clearly, with complex code and multithreading, there is no way to avoid this problem, and one has to find ways to deal with it. Textual structure has been shown in multiple studies to have an effect on comprehension of code. Both preceding comments (on file, class, group of procedures) and indention of control structures help understanding as they document part of the schemas in code. Unfortunately, object oriented programming has found to lead to more discontinuities. [Détienne, 2002]

Written texts have temporal, spatial, and causal discontinuities.

The reader has to overcome discontinuities and reconstruct the deep structure.

Looking at real world practice, LaToza et al. [2006] also note the need to overcome the discontinuities and many participants in their study agree that they ask "how design decisions are scattered across code". Their participants remark that tasks in code understanding are among most problematic: "understanding the rationale behind a piece of code" (66%), "understanding the impact of changes I make on code elsewhere" (55%) and "being aware of changes to code elsewhere that impact my code" (61%). In investigating these issues developers mainly use the IDE, the debugger. Especially the quality of the code is often a concern as developers ask "why the code is implemented the way t is" (82% agreement) and "if the code was written as a tem-

Studies show that developers work according to the mental model.

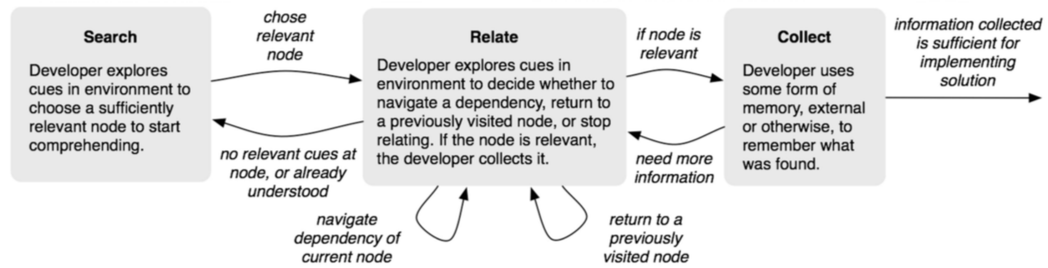


Figure 4.5: A model of program understanding from Ko et al. [2006].

porary workaround" (73%). Again, written documentation ("a write only media") is seen as outdated and code owners are seen as the go-to authority on the concerns above (cf. [Lethbridge et al., 2003]). Interestingly, in learning the answers to their questions, the investigators primary goal is fulfilled, and the answers remain unrecorded—the next person to answer that same question has to perform the work again.

Ko et al. describes code understanding as a series of searching, relating, and collecting actions.

Ko et al. [2006] follow their earlier navigation study [Ko et al., 2005] and build a model of the observed behavior of developers (cf. figure 4.5). The three activities they found are *searching* for information, *relating* to prior knowledge, and *collecting* said knowledge. Their model describes collecting information both in memory and in recorded information—this nicely resonates with the theoretical approach [Détienne, 2002] and the the study evidence from [LaToza et al., 2006]. They call for more help for developers to relate to "purpose and intent" of code in the IDE based on the observation that the code by itself is often insufficient.

Documentation can help overcome the discontinuities in code and should be integrated in the IDE.

Détienne [2002] concludes that exposure of the superstructure in a documentation is of critical importance and asks for two approaches. First, to present relationships in the IDE because that would be beneficial to understand the code, e.g., call graphs, storyboards, and visualization of class hierarchies (part of the situational model). Second, to bridging the discontinuities in code by documenting so "that it makes explicit the information that is implicit in the program code". Thereby, the documentation would bring

together de-localized plans and presents them in a localized way to the reader.

Communicating with Team Members

One key activity one has to consider when reasoning about he activities round code bases is how developers interact with each other to exchange and synchronize their mental models, even though the actual code artifact may not be part of that process. We mentioned in part already the results from LaToza et al. [2006] on communication between team members with regard to design and understanding. An interesting observation is that teams have a *team historian* who is the "go-to person for questions about the code".[7] Finding the right person to talk to is reported by 39% of participants as a serious problem, and 34% report difficult getting enough time with the senior developer that is knowledgeable about parts of the code base (cf. [Kagdi et al., 2008] for an approach to automate this search for the right person). LaToza et al. [2006] also observe that new team members in their process of joining the group (*on-boarding*) are given jumpstart problems and documentation that in specifically selected or designed to introduce them to the group knowledge. We have reported the same behavior in our investigation with development teams [Lichtschlag et al., 2014, Schulz, 2014].

Teams collect relevant knowledge in the minds of colleagues.

Developers are very likely to use written notations to externalize their mental models when talking to team members, yet these notations are volatile and get quickly wiped off the whiteboards. A reason stated by the participants is that overhead of integrating the information into the source base—even as simple text comments—is to big. Also they remark that they do not feel confident enough to claim to be an expert on that issue by providing documentation in the code base [LaToza et al., 2006]. The practice and need to seek other team members to understand code combined with the volatile recording of knowledge is a considerable drain on the resources of a team. We introduced this chapter with a fitting tweet by John Siracusa, highlighting the

Teams often discard notations of mental models.

[7] This is certainly true for all the projects we have been involved in.

distaste for reifying one's thoughts.

Pair Programming is exceptional collaboration technique.

Pair programming is a newer 'extreme' collaboration technique, where two programmers work very closely on a task: only one—the *driver*—interacts with the computer, the other—the *navigator*—advises. In discussion they can produce a higher quality product than if they were working on their own in parallel. This interaction has been investigated in great length [Begel and Nagappan, 2008, Bryant et al., 2008, Chong and Siino, 2006, Chong and Hurlbutt, 2007] although it is not often practiced (LaToza et al. [2006] reports 16% of the teams at Microsoft use this technique). We published a design [Lichtschlag and Borchers, 2010] on how a sketching can give the driver a way to draw and record hand drawn sketches in the IDE to foster the team communication—this was a precursor to the sketching design we outline below.

4.2 Visualization Approaches to Improve IDEs

Class hierarchies, package explorers, and user interface builders are commonplace commercial visualizations.

Often, a great part of the code base describes user facing artifacts. Then, advanced IDEs have facilities to build the user interface with graphical elements, for example, the Visual Studio Builder or the storyboards in Xcode [Apple, 2015c]. And commonplace in IDEs are static class hierarchies, e.g., a package browser in Eclipse or the class browser in Xcode. But, as outlined above, this only brings the deep structure forward in isolated cases. Some try three-dimensional views such as Code Cities [Wettel and Lanza, 2007], Metric View [Lange and Chaudron, 2007], or VisMOOS [Fronk et al., 2006]. But these visualizations are not used widely [Cherubini et al., 2007, Sensalire and Ogao, 2007]. Developers refer to code as the "king" [Cherubini et al., 2007] and the primary object of communication, and suggest that visualizations should be linked to the code itself [Sensalire and Ogao, 2007]. Bassil and Keller [2001] and Kurtz [2011a] both have excellent investigations and studies of the design space of software visualizations, and Murphy et al. [2006] investigates their use in the Eclipse IDE. Bassil and Keller

[2001] find that time savings, better understanding of code, and hierarchical representations are the most desired features by programmers (cf. [Park and Jensen, 2009]). And they find that call relations, inheritance relations are elements of deep structure that are already well visualized in the evaluated tools.

4.2.1 Rich Documentation for Human Readers

As discussed above, the surface structure of code communicates the program intent to the machine first, and the human reader second. Comments alongside can offer a side channel that only addresses human readers. While many scorn at such practices (e.g.,[Fowler and Highsmith, 2001]), a couple of approaches iterate on them. Borchers [1995] presented a tool to write comments in HTML so that the code file can also be rendered as documentation directly. The rich comments of Xcode Swift Playgrounds [Apple, 2015f] bring this idea into a modern mainstream IDE and also allow adjunct files to be packaged to the code text file, for example images to be rendered inline with the code. These approaches also reintroduce headers, emphasis, and other typographic features to text that is aimed at humans. Détienne [2002] also notes that rich text comments are more helpful than plain text comments.

Rich documentation can be integrated into the source code.

4.2.2 Live Coding

A key feature of source code we outlined above is that it often operates on abstract data. Thus, another strategy is to use example inputs in the IDE so that the readers and designers can closely follow the effect of the code. Victor [2011]'s article "Ladders of Abstraction" wonderfully describes how a learner can understand complex systems by starting on example input first. In research approaches this design is referred to as 'live coding' [Tanimoto, 2013], offering immediate line by line output as if the program was observed with a variable watcher, yet without ever leaving the writing environment. Unit tests offer the possibility to

test fine-grained behavior, and debugging with example input allows the same on the big level. But such execution is removed from the code itself, the unit test is defined spatially separated, and debugging interrupts the flow of the developer. Prototype systems for live coding exist, such as [Krämer et al., 2014], and approaches exist that transform the trace automatically into unit tests for later reference or testing [Ulmen, 2014]. Again, Apple's Xcode brings some of this into the IDE [Apple, 2015f], albeit so far this feature is only available in playgrounds and for custom interface builder elements. Another market application is web development: often changes to the DOM tree can be rendered immediately. Tanimoto [2013] defines a hierarchy of liveness, and these approaches would be on level 3–4, even more advanced approaches could try to predict authors edits and provide feedback on these predictions. Krämer et al. [2014] evaluated a live coding environment and found that it "significantly decreases the average total fix time of bugs introduced while creating software" but "no decrease in task completion time when working live.".

Live coding uses example inputs to overcome the abstraction problem of source code.

4.2.3 Leveraging Programmer Activity

A series of prototypes are build around the idea of a *working set* of the developer. The assumption is that the salient parts of the code base and the relationships that are important are revealed through the activities of the people that work on the code base. CodeBubbles [Bragdon et al., 2010] is the most interesting one and is a major reference point for our prototypes. Their interface completely forgoes file and folder structure as an organizing principle, instead functions are laid out in 'bubbles' on a canvas and these bubbles connected through call relations. The developer opens up new bubbles by following the relations and over time the canvas displays a working set of their current task. No function bubbles are laid out that the developer did not explicitly call up, the display lays out functions according to their deep structure, and not according to their location in the files, bridging the spatial discontinuity. In their evaluation they show that navigators are much faster in CodeBubbles because of reduced need to navigate to off-screen

Recording the working set of a developer can identify related items and visualize at schemas.

code locations. Hartmann et al. [2011] has another example how the activity of the developer can be leveraged: in HyperSource, web searches by the developer are observed and the links inserted as comments in the IDE at the current editing position. This way, a future investigation of that source code has access to the web activity of the author and use it to explain the code. An evaluation by Fritz et al. [2007] found that frequency and recency of interaction of a programmer with parts of the source can be used to identify experts on these parts (cf. 4.1.5 "Communicating with Team Members").

4.2.4 Exploring the History of the Code Base

Understanding of the current state of a code base can be aided by investigating its history.

Software development almost always includes some form of change tracking with a revision control system so that concurrent work can be merged and errors recovered. This offers a wealth of information on the formation of the code base that can be used to reason about the current state. Wittenhagen [2015] deals with this argument in depth, and he suggests that one can better understand the deep structure of the current state of the code base by looking at the history of how it was created. For example, if one starts writing the algorithm from the 'inside out' starting from the most salient part of the program and builds the infrastructure around it. Then, a look back in the history can often reveal that the early check-ins of a repository carry less cruft and have a better signal to noise ratio (cf. the concern above). An in depth description of the idea is given in [Wittenhagen, 2015]. Other studies [Atkins, 1998, Yoon and Myers, 2012, Kuttal et al., 2014, Yoon and Myers, 2014] and prototypes [Telea and Auber, 2008, Hattori et al., 2011, Servant and Jones, 2012, Maruyama et al., 2012, Yoon et al., 2013] of history exploration open a wide field of possible approaches to deal with the history. Schulz [2014] explored how their approaches can be adapted to handle visual elements such as a sketch on a canvas.

4.2.5 Linguistic Analysis

Since the vocabulary used in variable names are often very consistent and even subject to conventions (cf. [Détienne, 2002]), one can hope to draw connections to distant code elements (deep structure) if the same vocabulary is used. For example, if multiple classes are involved interprocess communication, they all might refer to the data passed around with names like 'message' or 'packet'. The *vocabulary problem* [Furnas et al., 1987] is basic problem of HCI work and points out that often different terms are used to refer to the same object. Ko et al. [2006] investigated this consideration with vocabulary used in bug reports, and found that developers use very consistent naming (cf. [Storey et al., 2007]). Kuhn et al. [2008, 2010] uses the vocabulary in code bases to build 'thematic maps' of the code base. These maps have the nice property that the layout is robust for small iterative changes to the code base, therefore, developers can build a spatial memory of the map and follow call traces on it (cf. [Speicher and Nonnen, 2010] for another approach based on vocabulary.). Their study indicated that automatic layout based on lexical similarity may be confusing because users cannot map it to their mental model and they recommend user direction in the layout of the map.

Vocabulary used in a code base reveals related code locations.

4.2.6 Visual Programming

Many approaches take the definition of the program out of a textual domain and try out ways to define the program visually (e.g., [Apple, 2015e]). Myers [1990] has a great article on the early history of these approaches. Scratch [Maloney et al., 2010] is a very advanced contemporary approach addressing novices, and see [Asenov and Müller, 2014] and [Zinenko et al., 2014] for prototypes that address experts. Visual programming has often been used for the definition of the user interface of a program, e.g., Hartmann et al. [2010] built a prototype that includes visual layout of animations and state changes. Modern IDEs like Xcode include storyboards [Apple, 2015c] that serve a very similar function.

Visual Programming avoids the problems of textual representations.

4.2.7 Canvas Visualizations

The reader surely has noticed that the studies that evaluated user tasks often advocate for changes to IDEs that could be served with canvas user interfaces. We already discussed a couple of software visualizations that are building a *single continuous landscape* representing the code (Kuhn et al. [2008], Bragdon et al. [2010], Apple [2015c]) resulting in visualizations that can be described as a canvas interface (cf. chapter 2.1.7 "Fragmentation and Continuity of the Information Landscape"). Seesoft [Eick et al., 1992] and CodeThumbnails [DeLine et al., 2006] are two approaches that scale the text geometrically, with the problems as discussed before.

Modern visualization approaches often include canvas designs.

Code Canvas [DeLine and Rowan, 2010] is a recent approach that directly describes itself as a single landscape zoomable user interface with the intent of supporting spatial memory, rendering debugger traces similar to [Kuhn et al., 2008]. They contrast that approach against fragmented views in most software visualizations, which they call "bento box designs", mirroring our argument (2.1.7 "Fragmentation and Continuity of the Information Landscape"). Recently, they combined their efforts with the CodeBubbles team and produced a series of refinements and studies [DeLine et al., 2010, Bragdon et al., 2011, DeLine et al., 2012], also publishing it as a free expansion to Microsoft Visual Studio [Microsoft, 2013]. Their approach has the rare feature that it gives the author or investigator of a code base to decide how much of the structures should be included on the canvas. Their studies indicate that the canvas is well suited for large code bases and was preferred for complex call relations. They recommend to use the canvas as a mode, not as a replacement to standard user experience (in contrast to [Bragdon et al., 2010]). We also want to highlight Relo [Sinha et al., 2006] that also allows the creation of a UML canvas with user directed selection of elements. Relo was an inspiration for our vocabulary based approach below 4.3.1 "CodeGestalt: Our Vocabulary Based Design".

Canvas designs in code bases promise to overcome discontinuities in the written representation.

4.3 Our Approach

The related work has not only produced quality models to understand programmer behavior, it has furthermore outlined directions for future research. We can summarize the key weaknesses of the tool support that current IDEs provide: First, current IDEs offer little support for visualizing the *deep structures in the code that are not readily parseable by compilers and model checkers*. We have both many and excellent tools at our disposal to model class hierarchies and call relations, but those tell only 'part of the story'. Their narratives are founded in the computer specification, which only addresses the human developer only secondarily. Second, current IDEs give *no visibility to the big picture* of the code base, but users desire such a view [Cherubini et al., 2007]. A programmer can plan a project by outlining views in a visual editor for the UI, or they can start developing their class hierarchy in UML. But these are again only the deep structures that are parseable by the tools, but planning a project is not primary about defining a specification for a computer, it is about reifying one's ideas, iterating on them, and comparing notes with colleagues. Formalized tools like UML modeling are hard to change and do not invite experimentation, so it is no wonder many programmers often use them informally at best [Cherubini et al., 2007]. An investigator of a code base can get lucky and find a code base where files and folders structure, a visual database model, and a visual UI description give a good overview, but they are more likely to find answers to their questions from the team historian. Third, current working environments are *poor to capture informal notations* as they are used by programmers to reason or communicate about code. All to often, they are produced outside of the IDE and never recorded in a permanent way.[8] While a developer might write a rough textual description, our tools offer no way to capture informal graphical descriptions. Fourth, current IDEs only capture *what is, not what should be*. There is no way to have an informal or unfinished plan in the IDE and build on that. An author can develop a design document

Existing IDEs can visualize only parts of deep structures,...

have no big picture overview of a code base,...

cannot record informal notations of mental models, ...

and can only visualize realized solutions.

[8] An interesting facet of programmer behavior since software developers are otherwise not known for poor data retention, and build cool tools like revision control systems.

or a project plan outside of an IDE, but there is no way to integrate it into their workspace and connect it to the elements of code that implement it. Only the parts of the mental models and structures are available in the IDE that have already been realized.

Three PhD thesis projects are currently investigated at the media computing group: Wittenhagen [2015] explores the change of the code base over time and explores ways for investigators to build a mental model by traveling back in time, thus answering a very commonplace question when dealing with source code: "How did this code come to be?"[9]. The second axis of investigation [Krämer et al., 2014, Krämer, 2011] explores ways improve our tools to answer the question "What exactly does this code do?". With live coding and stack exploration one can investigate the status quo of the code base without entering a debugging mode: the IDE always reasons about the code and gives possible answers to the author without him specifically asking for it.

In our approach we try to answer the question about wishful thinking: "What is this code intended to do?". So how can we address this question, and how can we build a canvas that invites experimentation and informal planning? We would like to have a 'pop the hood of the car' overview of the code base like one can have it with physical artifacts. What if one is not to follow the information already present in the code base, dominated by technical specification language and instead were refer to human communication as the source to this question? We support the recent exploration of canvas approaches (cf. 4.2.7 "Canvas Visualizations") to build an 'pop the hood of the car' scalable overview of the code base because we already found the canvas model is well suited for presentation visualizations. We essentially try to build presentations about code bases here—ways for developers to tell stories about the source code to another (or to oneself if one builds a mental model while investigating foreign code).

Our approach is to present the mental model of developers on a canvas.

ZUIs offer exactly the physical metaphor of a unified information landscape that can provide a summary of parts of

[9] Feel free to add "What were they thinking/smoking?"

Zooming out of the code brings the 'big picture' of the code forward.

the landscape or the whole by zooming out. Thus it seems a good match to bring a 'big picture' into the IDE. And as we have seen in the domain of presenting, authors can provide rich information about connections between elements on the canvas and communicate them to an audience. An investigator trying to building a mental model of a code base is often best served by talking to the original developer. Below we formulate three approaches with which we hope to allow software developers to externalize their mental models on a canvas and communicate them to their peers in a language they already use.

We rely on developers to author a canvas visualization.

In relying on human input to direct the layout of the canvas and to define the elements that should be included on the canvas to tell a story, we avoid any form of 'one click solution' as described by Kurtz [2011a]. The one-click invocation style is typical for a wide range of software visualizations (e.g., [Kuhn et al., 2008, Wettel and Lanza, 2007]), but these layouts produce always the same visualization for the same state of the code base. If this does not happen to be the same as the mental model of the designer, then the visualization cannot do the job of communicating the mental model between colleagues. Instead we look for 'human driven/machine assisted' designs (cf. [Kuhn et al., 2010]).

We experiment with vocabulary in the code base, software design patterns, and hand-drawn sketches to record the mental models.

We developed three prototypes to explore different ways to achieve this. First, and approach based on the *vocabulary of the code base* inspired by [Kuhn et al., 2008] but with human direction to the layout and inclusion of elements to the canvas. Second, an *pattern based* prototype that allows the developer to reify their *schematic* and *strategic knowledge* (cf. 4.1.4 "Cognitive Models of Software Design"). Third, we developed a prototype that takes *hand-drawn sketches*, e.g., from a whiteboard or a sketchbook, and arranges them on a canvas connecting them to the code base. Below, we look at each design and prototype in depth.

4.3.1 CodeGestalt: Our Vocabulary Based Design

Our first design is based on using the vocabulary of the code base to build an abstraction of the code text. We

present *CodeGestalt*, design for a source code visualization that uses the vocabulary to assist the developer in building and recording a mental model. The vocabulary is interesting to mine because we can use the source code author's existing design effort of careful selecting natural language identifiers [Ko et al., 2006, Kuhn et al., 2008, Speicher and Nonnen, 2010, Storey et al., 2007] such as variable or method names (cf. 4.2.5 "Linguistic Analysis"). We designed two user interface elements that amend class diagrams (e.g., [Umbrello, 2015]), the *tag overlays* and *thematic relations*. Both display similarities in the vocabulary used in the underlying source code. Our design goal is to explore with these elements if one can visualize structures in the code base and build a graphical representation of the text base. This design has been supported by Christopher Kurtz [Kurtz, 2011a] and previously been published at as a poster [Kurtz, 2011b].

CodeGestalt brings the vocabulary in the code base to the canvas.

As we discussed before (cf. 4.2 "Visualization Approaches to Improve IDEs"), numerous tools build software visualizations. We base our design on the Relo editor [Sinha et al., 2006] which supports the creation of partial class diagrams and automatically indicates structural relations such as method calls and inheritance of existing elements on the canvas. The user decides which of these relations to expand by clicking contextual buttons. That way, only code artifacts the user selects are visible, and the user controls the layout and scope of the diagram as it grows. Our second inspiration is the *thematic software map* by Kuhn et al. [2008] for the use of vocabulary to build a canvas. It represents classes as hills that are placed close to each other if they share a similar vocabulary. The resulting landscape produces a recognizable shape of the code, but it is always generated the same way, the layout is not really authored by the developer. Our design sits in the middle of the two approaches above: CodeGestalt uses class diagrams as a familiar baseline and lets developers create and customize graph-based visualizations. The developer can then add more semantic information by adding relations between the canvas elements with based on the vocabulary of the underlying source code.

CodeGestalt values human driven layout of the canvas.

An example workflow could be as follows: a developer

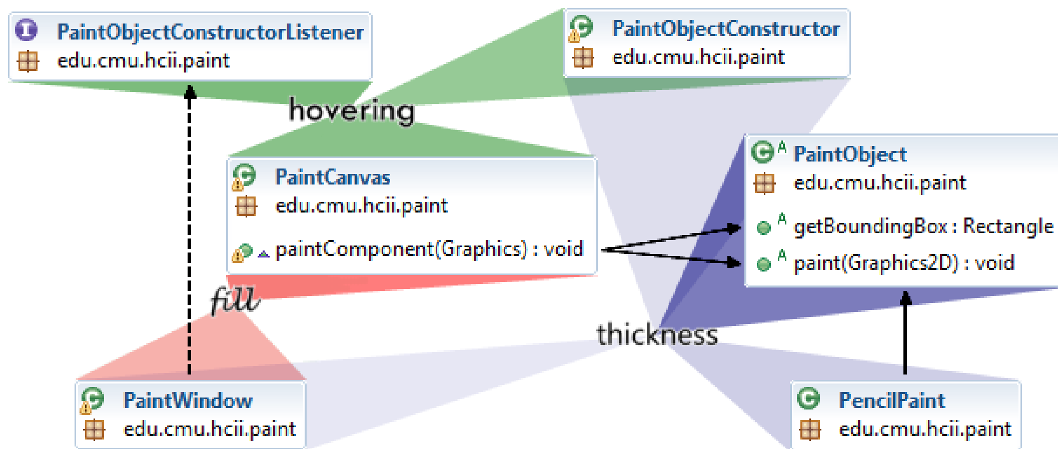


Figure 4.6: Diagram rendered by CodeGestalt. The tags *fill*, *hovering* and *thickness* connect classes using these terms in their source code vocabulary. Color intensity indicates the relative frequency of each term. Adapted from [Kurtz, 2011a]

identifies classes responsible for certain graphics operations (cf. figure 4.6) in the class diagram. They then investigate the vocabulary of these classes and finds that all classes that manipulate the stroke width use the term "thickness" in their implementation. The developer then groups these classes with the common vocabulary and produces an element in the canvas of the class diagram that visualizes this relation and is labeled with the vocabulary selected. Another colleague investigating the code base can then use this visualization to help him understand the author's design decisions. Or they can invoke the class diagram and use the vocabulary elements themselves to search for related structures in the code. Making this source of information easily accessible promises to build an overall gestalt of a code base that is memorable and a conversation piece.

We envision the vocabulary to enable developers to better understand the code base and to use the vocabulary when reifying their mental model.

The CodeGestalt Prototype

CodeGestalt is implemented as an Eclipse plug-in and introduces canvases as a new document type.

Kurtz [2011a] developed our design iteratively with a series of prototypes and then evaluated them in discussions with software architects and HCI experts. One of our first prototypes raised our awareness to the *vocabulary problem* [Furnas et al., 1987] and that lead us into the direction of tag discovery, so that the investigating user does not need to guess

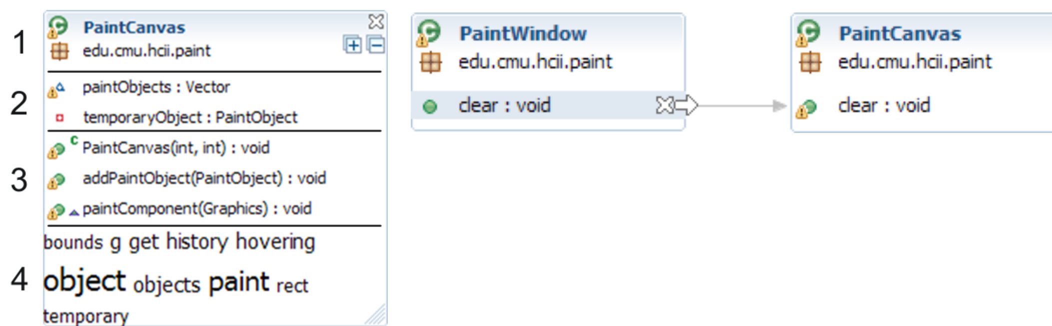


Figure 4.7: Left: A class box element on the canvas with name and package of the class (1), fields (2), methods (3), and a tag cloud (4). **Right:** Preview of a call relation between methods. Clicking the transparent arrow makes it persistent. [Kurtz, 2011a]

the right terms. The final prototype is implemented as an editor plug-in for the Eclipse IDE [Eclipse, 2015] and uses the Cultivate API [Speicher and Nonnen, 2010] to obtain the tag metrics. CodeGestalt canvases are integrated as a new document type with Eclipse and can be placed anywhere in the project structure a source code file can. The user can position the file view just like any other source file in Eclipse (e.g., side by side). The user generates diagrams via drag and drop from source file to the canvas. All kinds of code artifacts (files, classes, methods, etc.) can be dragged to the canvas from any Eclipse view, and are visualized in a way similar to class diagrams. Types are represented as boxes, where fields and methods can be added to a list of members (cf. figure 4.7, left). At the bottom of each box is a tag cloud displaying the ten most frequent terms. When a user selects a diagram element relations to other classes are displayed as context-sensitive ephemeral previews (cf. figure 4.7, right). The user decides on a case-by-case basis if a relation should become persistent and be included in the diagram. These informed user decisions avoid uncontrolled growth and distracting clutter. Double-clicking on an element opens the respective code location in Eclipse.

Figure 4.8 shows the workflow of adding vocabulary elements to the canvas. The user starts a new sketch on the canvas and adds classes to investigate using drag and drop gestures. These form a *class diagram* (1) and provide hints to other connected classes. When searching for spe-

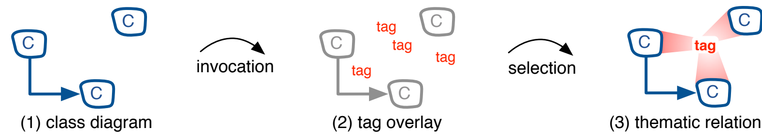


Figure 4.8: *Tag overlays* and *thematic relations*. Given a class diagram (1), the developer first invokes its tag overlay (2), then selects a thematic relation between classes and tags to add to the class diagram (3).

A developer uses the tag overlay to discover structures in the code base and the thematic relations to record them.

cific features and classes of interest, the user toggles a filtering interface called the *tag overlay* (2). By simply clicking on class boxes and tags, the user may visually investigate which classes reference which terms how frequently and vice versa. To record insights from this transient interactive view and share them with others, the user may convert filter results into *thematic relations* (3) that are then saved with the canvas file.

The Tag Overlay

The *tag overlay* is an transient view that visualizes canvas elements with similar vocabulary.

When the user enables the *tag overlay* (cf. figure 4.9), it displays an interactive tag cloud on top of the class diagram. Tags are generated automatically from the identifiers of classes in the diagram. By clicking on classes or tags, the user triggers CodeGestalt to create heat maps that illustrate which classes use which terms, and vice versa. Color intensity indicates term frequency. This way, the tag overlay directs attention to key terms in the code base, and allows the user to quickly find relevant classes for a given task or topic. Tag position and font size are computed from frequency-based weights, and updated dynamically as the user adds, moves, or drops classes in the diagram. The tag overlay is not a permanent part of the visualization, it stays only until user deactivates the mode.

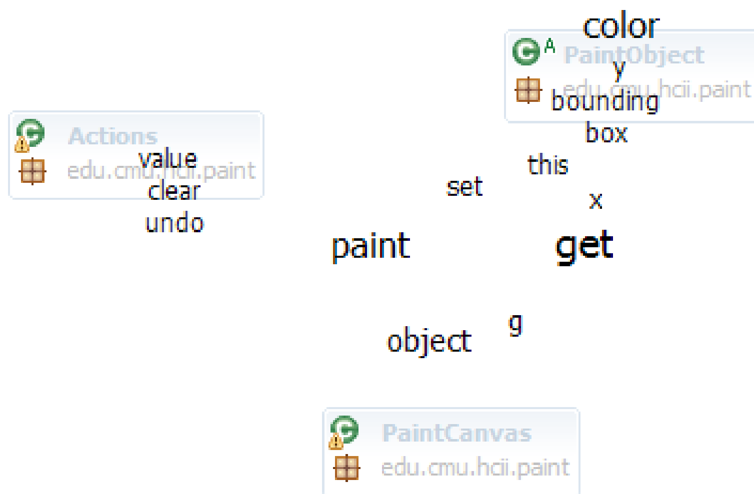


Figure 4.9: A small *tag overlay* for three classes: *Actions*, *PaintObject*, and *PaintCanvas*. The user can identify the 'center of gravity' for concepts, such as *undo*, which is an important term in *Actions*. The size of each tag is determined by the mean of the weights assigned by each class (term frequency in class). Tag size and location indicate how often and where they are used in the classes shown.

The Thematic Relations

While the tag overlay allows developers to quickly orient themselves in an unknown code base, they will still want to record and share insights gained from their exploration using this filter interface. Hence, the tag overlay allows users to create persistent *thematic relations* from their transient tag results. These augment and extend the traditional class diagram and are saved with the diagram file for future reference. Thematic relations are recognizable landmarks that add structure to a class diagram. We visualize them as labeled fans (cf. figure 4.6). The segments of a fan connect all classes using the same term with the corresponding tag. Different transparency levels are assigned to the fan segments to represent different frequency-based weights.

A *thematic relation* is a recognizable landmark that add structure to a class diagram.

Study

We performed a study asking users to investigate and to record their mental modes as if to share their findings with a colleague.

To measure authoring with CodeGestalt, we performed a user study with 16 computer science students and postgraduates. We prepared four tasks that asked participants to learn how a specific feature of the *Paint* source code from [Ko et al., 2006] is implemented, and to share their findings in a diagram as if do document it for a colleague. E.g., the first task is "In Paint the user can choose between three drawing tools: Pencil, Eraser and Line. In order to enable another programmer to create additional drawing tools, you need to communicate to them how the software realizes the support for them." These tasks cover the 'to ground' and 'to share' and touch the 'to manipulate' motivations to make sketches as noted by Cherubini et al. [2007]. Each participant created two canvas diagrams using CodeGestalt and two hand-drawn sketches using pen & paper. We allotted 10 minutes for each task. To familiarize participants with the tools, we gave them a short introduction and allowed them to interact with the tools beforehand. The assignment of tools was counterbalanced, as was the sequence of tasks. We recorded the screen contents and logged manually any user comments or problems they encountered. Kurtz [2011a] reports on the details of the study.

Quantitative Results

Testers used thematic relations in roughly a third of the cases.

Completion rates for sketching and CodeGestalt were not significantly different. However, sketching was significantly faster one of the four tasks ($U = 9.0, z = -2.0, p < .05$). We defined a list of elements we deemed useful (classes, attributes, call relations) for the communication and scored the artifacts according to this by counting how many elements they represented (two authors, interrater reliability $\kappa = .98, p < .001$). Since sketches were created in a multitude of different styles and layouts, we used a lenient interpretation of how sketches could comply with the checklists. In the 20 observed task/error category combinations, CodeGestalt diagrams scored significantly higher in three cases, while hand-drawn sketches did so in one case. We also investigated the use of the different ways to de-

scribe the structure of the solution to the tasks. Participants put thematic relations on 37.5% of the canvases (n=32) with 18 thematic relations in total; they defined inheritance or call relations in 84.4% of the canvases (n=32) with 76 such relations in total; on 9.4% of the canvases only classes but no relations were depicted. In the hand-drawn sketches, they used relations between elements in all cases, with 97 relations in total (The relations cannot all be classified because they are in part unlabeled).

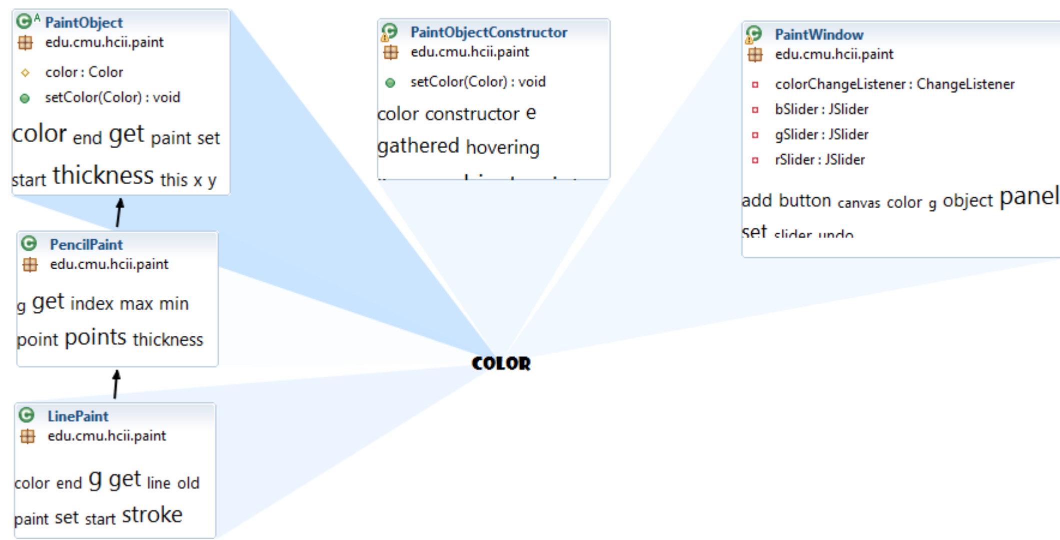
Qualitative Results

After the session, we asked testes to fill out a questionnaire to rate the usefulness of individual features on a five-point (0–4) Likert scale. Participants agreed with the statement that CodeGestalt sketches are a practical alternative to pen & paper (median 4). Similarly, the thematic relation (median 4) was rated very useful. The highlighting features for classes (median 3) and tags (median 3) were rated useful, as was the tag overlay (median 3). We received positive remarks for them from six participants in open-ended questions: E.g., "The tag overlay helped to quickly get a grasp of the overall structure of the program", "The vocabulary connections between classes are interesting and important for the understanding", and "The tag cloud is very expressive for some classes, e.g., the PaintObject".

Testers judge CodeGestalt to be a practical alternative.

Two weeks after the study, we contacted our participants again, asking them to complete an online survey. It asked them to compare four balanced pairs of the previously created sketches and CodeGestalt diagrams (one pair per task, 0..6 Likert scale). We received 14 responses; testers preferred neither sketches nor CodeGestalt diagrams with regard to understandability and usefulness for solving the respective tasks (median 3). For clarity and suitability for documentation, CodeGestalt was slightly preferred over sketches (median 4). See [Kurtz, 2011a] for the details.

Testers preferred neither condition as an audience.



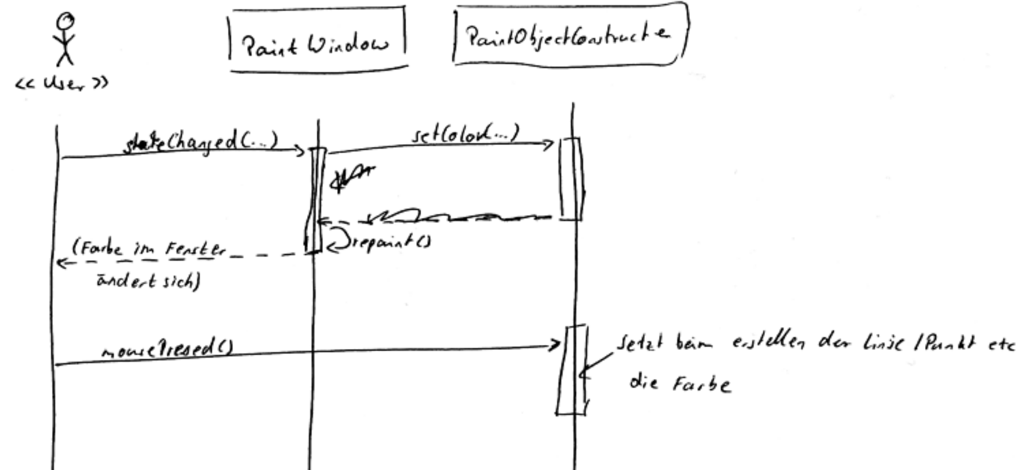


Figure 4.10: Top: a CodeGestalt document from the study. **Bottom**: a hand-drawn sketch result for the same task.

Discussion

Testers did not use the vocabulary elements much.

Participants rated the usability of our prototype highly and the results of the two conditions were seen as comparable by authoring and reviewing participants. We see that the classic class diagram elements make up most of the elements on the canvas and that thematic relations were only used in roughly a third of the cases. This suggests

that while they offer another way to express in class diagrams, they mostly offer redundant capabilities to capture mental models. When we look at the hand-drawn sketches, we see that they are much more informal, use a mixture of visualization styles at the same time, and have more character. The hand-drawn sketching allows the author to quickly switch to a different style without changing the selected tool or enabling the right mode. They even included a human figurine or drawing of a mouse to denote the user input (cf. figure 4.10). Such flexibility is not possible with CodeGestalt. CodeGestalt canvases look more polished, but we think that the hand-drawn sketches capture the mental models better.

Hand-drawn sketches saw more variability.

4.3.2 CodeMixer: Our Design Patterns Based Design

Our second design plays with the idea of integrating visualizations based on the design pattern format into a canvas, with, of course, the same two goals: we want authors be able to record their mental models and communicate them to colleges. And we want to see if such a visualization can be a proper graphical abstraction of text for a zoomable user interface. We present *CodeMixer*, a low-fidelity prototype that was built and iterated by Tjandra [2013] and with which we investigate this design direction.

CodeMixer brings design patterns to the canvas.

When we presented cognitive models of writing (cf. 4.1.4 "Cognitive Models of Software Design"), we already saw that some have a striking similarity with design patterns, and we are not the first to make this observation [Détienne, 2002]. Let us review design patterns for a moment. Alexander et al. [1977] invented *design patterns* as a formal recipe to capture recurring design solutions in architecture. Each design pattern solves a problem of conflicting forces and is described with this background. Each pattern describes the context of its use, the problem it addresses, and then the solution. Other elements of the format include its name, a picture of an example solution, a diagram of a generalized solution, references to other patterns. Alexander et al. [1977] write that each pattern "contains only the essentials

Design Patterns are built for communication.

which cannot be avoided if you really want to solve the problem" and is written down with the intention of fostering communication. Design patterns had their biggest success in software engineering with after Gamma et al. [1994] published "Design Patterns: Elements of Reusable Object-Oriented Software" and captured essential problems and solutions in software design. It is very common to hear software developers to talk about *Singletons* or *Factories* and refer to design patterns, even if maybe they do not know them as design patterns. Lastly, Human Computer Interaction has also used design patterns to capture solutions in interaction design, e.g., *Go Back to a Safe Place*. Borchers [2001] gives an overview of these approaches. There are two differences to keep in mind when we compare design patterns the schematic knowledge of software developers: design patterns are formalized descriptions and they are generalizable. Schematic knowledge is not formalized and not always generalizable [Détienne, 2002].

Design Patterns are very commonplace in the software design.

We see many situations in which an expert software developer shares his knowledge with an audience and formulates a solution in a way that it can be reused and adapted to the context of the reader. Framework vendors do almost always add to their documentation examples of proper use, sample code, tutorials, or video tutorials. The highest rated answers on question and answer sites (e.g., StackOverflow[10]) often contain detailed descriptions of the steps involved in a solution and when it might be applied. Many teachers, be that with in a classroom or on a website for distance learning, provide adjunct code materials, maybe a partial solution to continue working on.

There are many reasons to publish example code.

We imagine an example interaction with CodeMixer as follows. Tim is a teacher at university and blogs about mobile development for iOS. He builds an example to teach about photo filters: the application takes a photo, detects faces and crops the relevant region, and then replaces the head with a random photo of Angela Merkel. His idea is to playfully teach the audience about three tasks: capturing images and face detection with the built in camera, fetching Google results for "Angela Merkel", and image composition. He implements the solution, writes a blog post

[10]https://stackoverflow.com/

about the lesson, and then publishes the blog post linking to an online repository with the source code. His audience may have a hard time extracting his mental model from the written text (for all the reasons outlined in 4.1.3 "Why is Reasoning About Code So Hard?") and has connect code and the blog article to extract the three elements in his solution. In our example, each of the three tasks corresponds to a schema of his mental model, corresponds to a reusable design pattern solution, and corresponds to a *composition* that we consider supporting with CodeMixer.

Now, lets consider how Tim would publish with CodeMixer. With CodeMixer, Tim builds three *compositions*, one for each pattern. For the two low level schemas, image capture and fetch results, he drags the relevant code sections to the canvas. This drag action creates a graphical element on the canvas and links it to the dragged section of the codebase. He connects these elements among another with arrows designating order and data flow (cf. 4.12). He then gives each solution a title and adds comments that describe forces and applicability. This way, he reifies his mental model of the two schema, and he records them similar to the pattern format. Then, he does the same for the high level schema, the image filtering, but since it gets the inputs from the two low level schemas, he links them to their output instead of code locations. Thereby, he established a hierarchy between the compositions and schema, and reifies all this on a canvas. He publishes the canvas of compositions on blog as a figure and interactive element. Julia reads his post and wants to build on his composition. She drags the composition into her IDE. The IDE displays the same canvas as Tim authored it, but it is not yet connected to Julia's code. She then drags a graph element to her code, and this copies the code from Tim's example into her code base to the dragged position, of course, linking to the canvas. Julia then experiments with different filters and other inputs and playfully expands her knowledge about Tim's lesson.

We imagine a workflow in which example code can easily be published with design patterns explaining the structure of the example.

A couple of approaches because they also package reusable solutions visually. Quartz Composer [Apple, 2015e] is a visual programming language that allows the developer to define custom graph elements. CodeBubbles [Bragdon

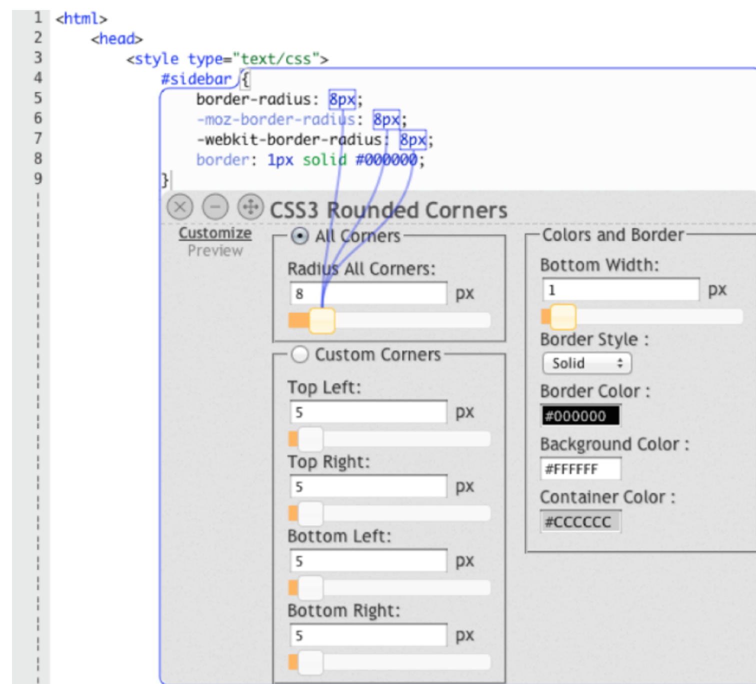


Figure 4.11: *Codelets* by Oney and Brandt [2012]. After integrating the "CSS3 Rounded Corners" pattern the IDE renders a control panel immediately below the pasted example. The developer can customize the general solution to his needs in the control panel. Figure by Oney and Brandt [2012].

et al., 2010] and Relo [Sinha et al., 2006] also connect code fragments on a canvas, the former based on the developer's activity, which in our example above could be very well correspond to the author's mental model. All three connect their canvas elements along syntactic routes, e.g., calls, data flow, which seems to constricting for our design here.

Related work presents multiple approaches for reusable solutions.

Hypersource [Hartmann et al., 2011] and their way of linking to web resources is an inspiration to our idea of importing the canvas from a web publication. Blueprint [Brandt et al., 2010] and Codelets [Oney and Brandt, 2012] ways to directly search in the IDE for reusable solutions and paste example code directly. With Codelets the solution pattern can be graphically interacted with to modify it to the current needs, even after pasting the example (cf. 4.11). In both cases the patterns are localized to only one position in the

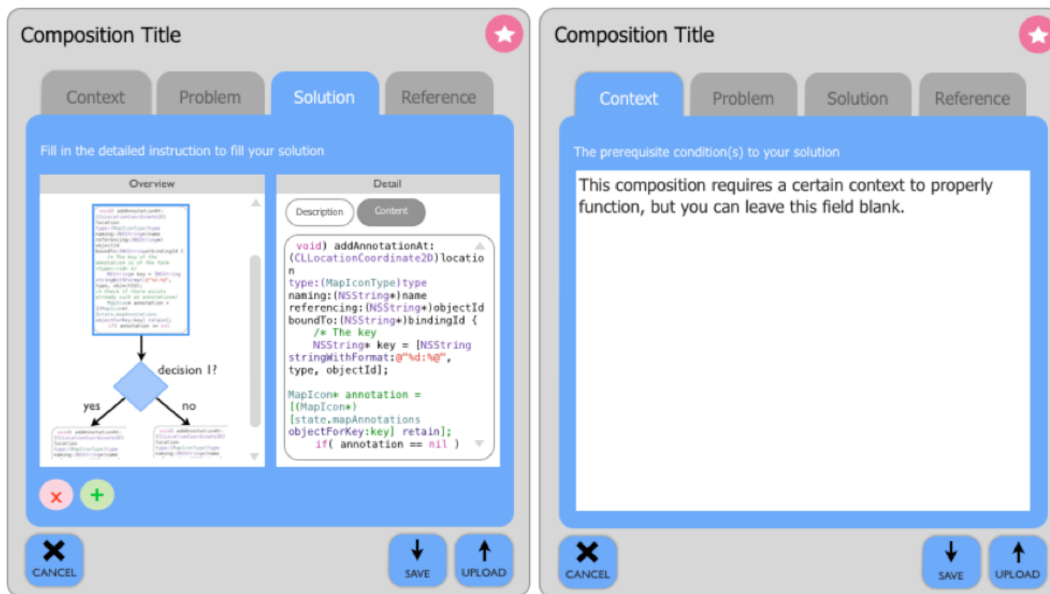


Figure 4.12: CodeMixer early design iteration. We quickly noted that the solution tab is the only one that mattered. Figure by Tjandra [2013]

source code. An accompanying study found that Codelets participants spent more time focusing on the example code and less reading descriptions of the examples.

Code Mixer Design

We built a series of screenshot prototypes to evaluate our design. We describe them here only briefly, Tjandra [2013] reports on them in full. Our first prototype started as a design that included most of the design pattern elements (cf. 4.12). It had a title, four tab views for context, problem, solution, and reference respectively. In the solution tab, a graph of connected code fragments represents our interpretation of the sketch in a design pattern. We were basically asking the author to write a full specification of a formal pattern every time he wants to share a composition. This effort may be acceptable when a person decided to invest time in building a pattern (cf. Wightman et al. [2012]), especially when they have already thought about preconditions and references. But for the interactions we envisioned this seems much to heavyweight, especially when we consider

Our earliest prototype had too much baggage from the formal way to write down design patterns.

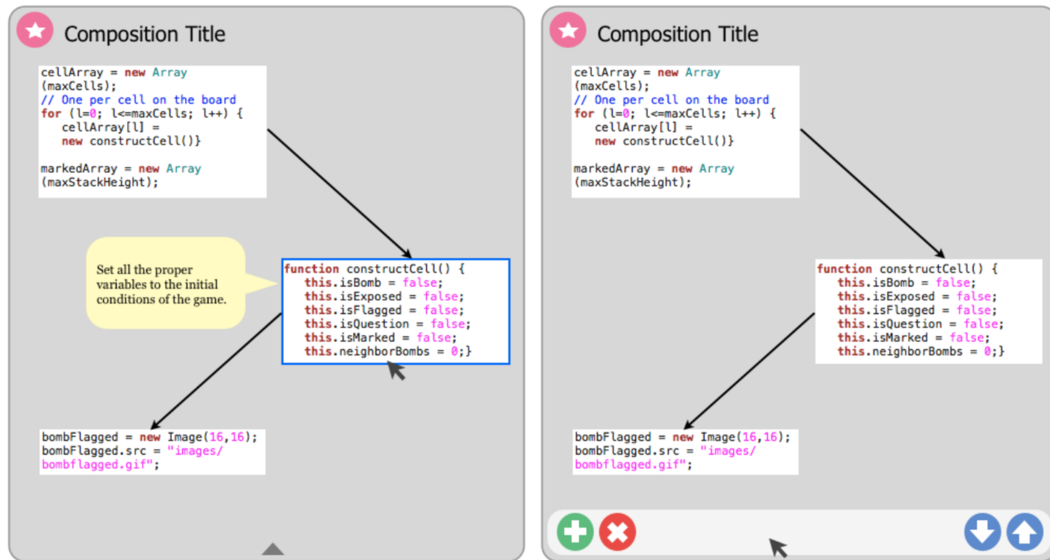


Figure 4.13: CodeMixer final design. Comments and user interface elements are hidden until revealed by the cursor. Figure by Tjandra [2013]

that the schema model is not always a fully thought out pattern, but rather a simple plan with related elements. Reviewing it, we noted that the solution tab was the only one that mattered in the interactions we envisioned.

Our latest prototype promoted the graph as the most important element.

In a later iteration, we further removed the detailed textual description next to the graph, because is became clear that the solution graph is the most important aspect of a composition. Brandt et al. [2010] also noted that developers favor to focus on the various code chunks rather than any other type of information. And once the code is linked to the code base, the description would no longer be needed. Instead the author can leave a comment on a code snippet and a pop-up appears when the cursor is over the respective element (cf. 4.13). We see that in our last design iteration we leave as much room as possible for the composition canvas. All other user interface elements are hidden, expect the title and a colored icon.

Each composition has one unique icon that represents it in the code base. When authors specify an association between a snippet of the composition and a code location, the icon is inserted into the IDE. Since a composition can have

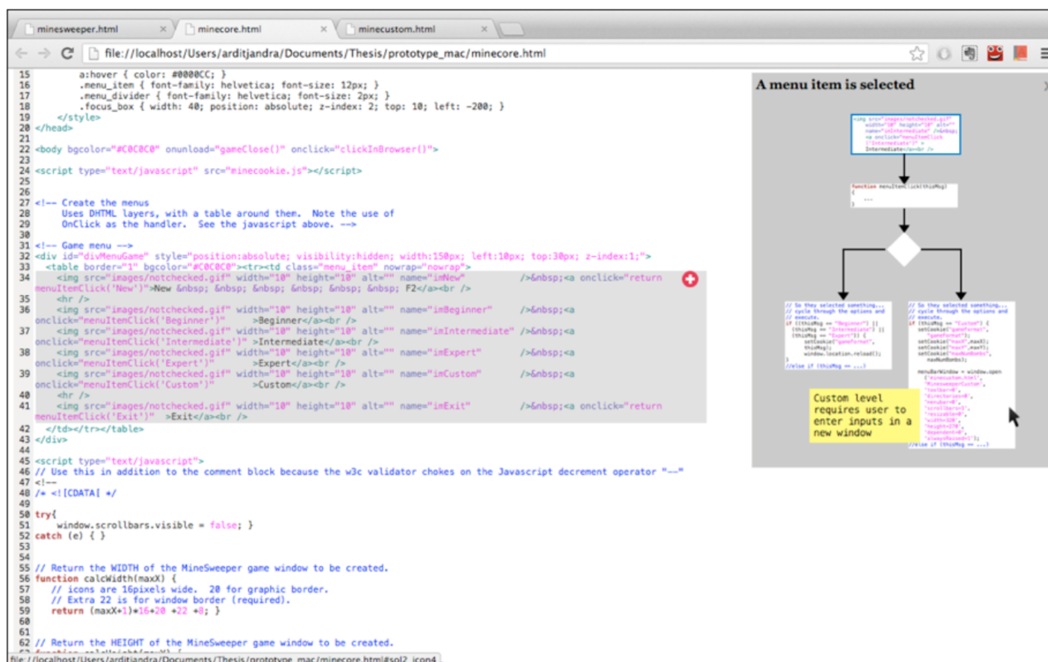


Figure 4.14: CodeMixer study setup. The composition is rendered to the right side of the window similar to a two window layout in an IDE. Figure by Tjandra [2013]

multiple locations, each of them is 'tagged' with the same icon. The idea of multiple icons all across the code base is not unlike colored text highlighting in order to mark interesting parts of the code base. The icons provide a quick access to the composition by clicking on them. Figure 4.14 shows how a composition and a code location are rendered next to each other.

Links to the composition from the code base are symbolized with a colored icon.

Study

We did a qualitative study with ten participants and had three tasks. In the first, we asked them to use the prototype to explain a part of the code base to us, in the second, we asked them to choose one of the compositions to build a new feature, in the third, we asked them to build a new composition. A questionnaire recorded their opinions on the interaction.

We did an explorative user study to test out the design.

Our study prototype was built with web technologies.

A web version of the game Minesweeper[11] served as the test code base for the study. We built seven different compositions, each explaining a multiplexed structure in the code base, e.g., a composition connected menu item, its click function, and its style sheet. For the study, we build an interactive version of CodeMixer with HTML and CSS and presented it in a web browser. Each file of the code base is loaded as its individual page, changing the text though typing is not possible. Clicking on a composition icon in the code base reveals the corresponding composition to the right side of the window (cf. figure 4.14). Clicking on a graph elements focuses the text on that linked code text. Tjandra [2013] has the details on the study and the prototype implementation.

Users remarked positively on the interaction, but had problems distinguishing the visual outline of the compositions.

Although evaluated without a baseline, the testers remarked positively on CodeMixer, agreeing that it would help them achieve real world tasks. Testers also stated that the liked the icons and code text highlights, but were divided about the pop-up comments. They remarked mildly positive on their ability to recognize and differentiate compositions by their shape, but we observed an opposite behavior. On investigation we noted that users particularly payed attention to the icon to differentiate between compositions. They suggested to color the whole background of the graph in order to give it more identity. During the authoring task, we noted that even giving a title to the composition is a step that users would rather skip. Users remarked positively on the switching between canvas and code text through the links, and our observations concur. They also reacted positively to the idea of having a global view for all the compositions in a code base, and one interesting suggestion was that instead of navigating to the composition on click, they just wanted to see a quick preview while hovering over the icon.

Discussion

We learned a lot about our initial hypothesis that such workflows as we described above are worthwhile of sup-

[11]http://www.chezpoor.com/minesweeper/minesweeper.html

porting. Our HTML prototype received good feedback, indicating that users would support this type of interaction, and indicating that this helps user accomplish their tasks. The related work approaches, especially Codelets [Oney and Brandt, 2012], point the way how a high-fidelity implementation can support the search for reusable solutions directly in the browser while keeping the connection to the pattern alive. This way, the author does not just have an easier time implementing, they also leave the pattern directly by the code to document their work. However, to take these pattern approaches to the canvas, they would have to shed their textual structure.

Using the language of patterns for reusable solutions continues to be promising.

We recommend further exploration of reusable solutions that link source code to its origins.

We did not find an answer to our text abstraction problem. Our iterative design lead us to a final design for CodeMixer that only used three elements for the visual identity: the graph of text snippets, a colored icon, and the title. In the study we saw that the users mostly use the icon and title to identify a pattern, not the graph. While participants recommended our design decision of the graph, we could observe them having problems to differentiate the graph elements form another. We have two critiques for our design: first, the design is pretty bland. It lacks ways for the author to express themselves and all graphs have the same 'connected text boxes' look. Second, since text remains the dominant feature, zooming out makes the designs visually interchangeable. We see that this design did not build a visual abstraction of the code—the visual fidelity of the designs of our CodeGestalt approach turned out much better. We are happy with the independence of class and call relations to build the graph (compared to CodeGestalt), that gives the author the freedom to build the graph they envisions. We take two design lessons from this: the elements of a text abstraction should not include code snippets, and graphs should be built out of diverse elements.

We cannot recommend design patterns as a way to abstract text.

In summary, we see this pattern design direction for the visual abstraction as a dead end, but not the style of interaction. We continue in our next design, CodeGraffiti, in which we link to text similarly and iterate on the idea of having code and a small canvas next to each other. We also reuse the idea of highlighting linked code sections.

We keep the interaction style for our next design.

4.3.3 CodeGraffiti: Our Sketching Based Design

Our third design is based on hand-drawn sketching, an established technique for ideation, exploration, and communication [Schön, 1983, Tversky and Suwa, 2009]. Again, our approach is to investigate the use of such rather informal drawings as elements on a canvas and integrate them into the IDE (cf. figure 4.15). We then connect them to code artifacts thus allowing the code to be navigated and understood through the canvas, much as in our previous design, CodeMixer. A possible workflow could be as follows: a software designer begins building the project, mocks up the algorithm on a whiteboard. An engineer takes over the implementation of the algorithm and uses the sketch as an high-level overview and as a guideline of tasks to implement. When they implement the solution, they connect the code fragments to the sketch, thus referencing a distributed design with a common calling card—the sketch. The sketch is part of the team communication and serves as later reference for maintenance work on the algorithm. This way, sketches on the canvas fulfill a role additional to generated visualizations, promising to be more flexible in situations where generated visualizations do not capture the mental model. This design has been supported by Lukas Spychalski [Spychalski, 2013] and previously been published as a paper [Lichtschlag et al., 2014].

In this approach we put hand-drawn sketches as elements on the canvas.

As we noted before, one common way to get the needed information about unfamiliar source code is to ask other team members in short, but interruptive ad-hoc meetings. In such meetings mental models about source code often gets visualized in transient form during short meetings, e.g., on whiteboards or paper [LaToza et al., 2006]. Cherubini et al. [2007] interviewed developers and found that these sketches are also important for understanding existing code, designing, or refactoring. Furthermore, sketches are found to be valuable after the day of creation [Branham et al., 2010], but since sketches are rarely recorded and archived afterwards, the knowledge has to be constantly rediscovered (cf. [LaToza et al., 2006]). Walny et al. [2011] found that software developers use sketches frequently in different phases of the software development process to de-

Developers use sketching techniques to reify their mental models.

pict and convey different views and concepts of the system under development and Détienne [2002] noted that experts take more notes that novices and that sketching often leads to a change in plans.

With many reasons to sketch, let us look at how developers sketch. Luckily a handful of studies investigated sketching practices recently. In all studies above, the use of hand-drawn sketches outweighed both tool-based visualizations and visualizations created with automated tools, often because sketching is considered unconstrained by formal notations, e.g., UML [Cherubini et al., 2007, Kurtz, 2011a]. Developers depict both microscopic as well as macroscopic views on the code [Cherubini et al., 2007]. More than 75% of the surveyed software developers by Cherubini et al. agreed that hand-drawn, analog sketches are more important than automated tools and state that a macroscopic view cannot be depicted automatically by a re-engineering tool. Walny et al. [2011] traces the life-cycle of sketches and finds that users make use of paper and whiteboards, as well as notebooks, printers, scanners, cameras, photocopiers, hand-held devices, tablet devices and PCs. This indicates that developers establish their own individual workflows in dealing with sketches, mostly in paper notebooks or on tablet devices. The production quality of both digital and analog sketches changes when sketchers are conscious about the possibility that their doodles and scribbles might get reused in the future. Developers might redraw a sketch multiple times in order to get a cleaner version of the original sketch that will be recognizable in the future by others [Branham et al., 2010].

Developers use sketch on many transient materials and depict schemas and structural knowledge.

Several designs have been put forward to integrate sketches into the IDE: ReBoard [Branham et al., 2010] automatically captures whiteboard images and archives them for later reference, so that the images can be accessed through a calendar interface, thereby they address the limited space of a physical whiteboard and document sketches use future use. Calico [Mangano et al., 2010] enhances the software design on electronic whiteboards and tablet devices by introducing multiple virtual whiteboards arranged in a grid, also in order to address limited space. They introduce ‘scraps’, grouped graphical elements, which add the

Recent related work has investigated software solutions to support sketching.

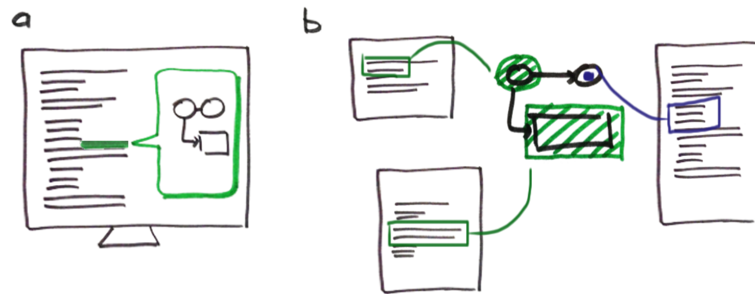


Figure 4.15: The two concepts that guided the design of CodeGraffiti: a, *sketch-follows-code*: the sketch comments on a code fragment. b, *code-follows-sketch*: the sketch refers to many code fragments in multiple files and reveals a structure of the code base. [Lichtschlag et al., 2014]

ability to copy selected parts of a sketch for reuse, e.g., exploring multiple variations of the same initial scrap without the need to redraw. Codepad [Parnin et al., 2010] explores interactions with sketches on touch interfaces that are connected to the IDE and is an inspiration to our *sketchbar view* described below. Plimmer and Freeman [2007] use sketches as a user interface description language and in teaching students. Previously, we proposed sketching on the code as a means of expression for the navigator in pair programming tasks [Lichtschlag and Borchers, 2010].

The CodeGraffiti Prototype

We conceived two ideas to present code and sketches together.

The great strength of sketches is their informality and the ability cover a wide range of contexts with commonplace drawing tools. This same quality, however, makes it both tricky to parse sketches algorithmically and hard to integrate sketches into a formal environment like an IDE without loosing the very qualities that make them so powerful in the first place. Building on designs from the related work [DeLine and Rowan, 2010, Parnin et al., 2010] and our CodeMixer exploration, we developed two concepts in an iterative design process (cf. figure 4.15). In both designs we integrate sketches into the IDE and connecting them to source code: We use *sketch-follows-code* to describe that the

a code fragment is explained by a sketch like it would normally be explained by a comment (e.g., a small drawing that explains a recursive step in a sorting method, cf. 4.2.1 "Rich Documentation for Human Readers"). The sketch is adjunct to the code it describes and provides a concrete example or a graphical description, thereby visualizing a low level schema in the code base. Analogously, we coined the term *code-follows-sketch* to describe a sketch that provides a higher level abstraction or explains a structure of the code base. Such a sketch may be composed of multiple elements referring to multiple code locations, e.g., a workflow of an algorithm that spans multiple methods or files. Here, the sketch brings together parts of a concept and puts them into a big picture, abstracting from the individual code locations. We implemented these concepts in our CodeGraffiti plug-in by providing two new views in the IDE for these two concepts respectively: the *mission control view* for the code-follows-sketch concept and the *sketchbar view* for the sketch-follows-code concept. In both views, our design embodies connections as color coded one-to-one relations between a position on the sketched canvas and a range of code lines or a file in the codebase. Our design does not constrain the semantics of the connections: positions in a sketch are independent of drawing style, formalism, or the IDE's ability to parse the sketch file. Ranges in code can indicate many levels: individual lines, methods, uses of a variable, etc. This way, the user has more freedom to express meaning though connections.

Sketch-follows-code describes a sketch that comments on code.

Code-follows-sketch describes a sketch that summarizes a structure of the code base.

The *mission control view* (cf. figure 4.16) is a semi-transparent, fullscreen zoomable user interface that the user invokes and dismisses by a menu or a keyboard shortcut (cf. 2.1.6 "Transparency"). The canvas provides an overview of the whole project and contains sketches and connections to the code base. These connections allow the user to navigate to the source code instead o regular project navigation (such as its folder structure). The user interface elements of this view are integrated into the view itself, since the mission control view overlaps all other areas of the editor.

The *mission control view* implements a code-follows-sketch view.

The connections between sketch elements and code lines are implemented as *connection dots* (cf. figure 4.17), i.e.,

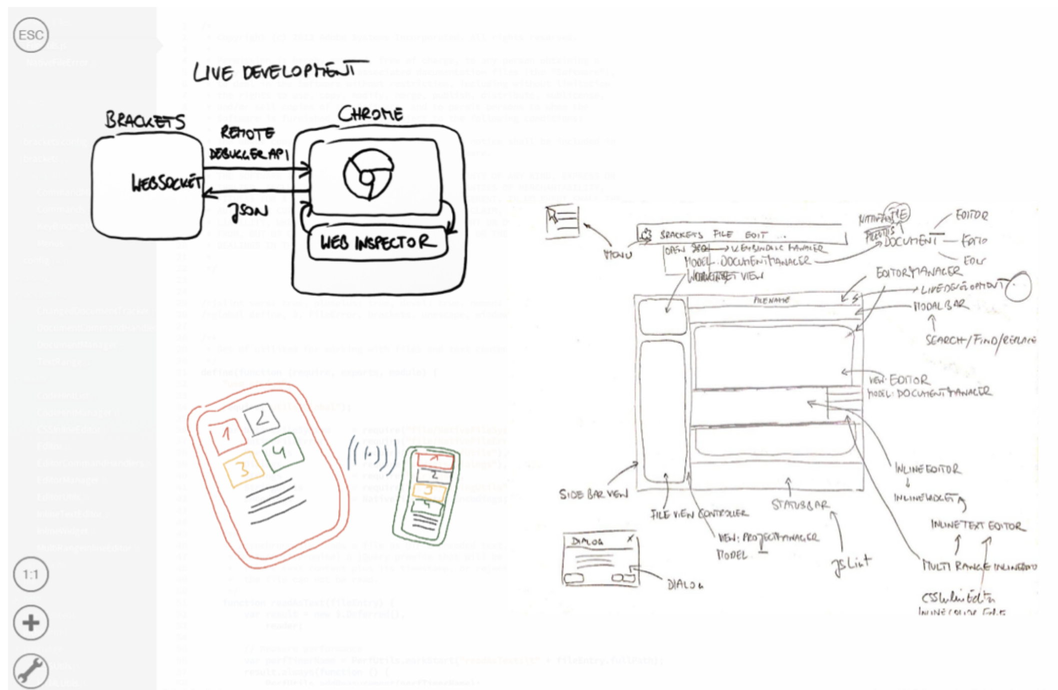


Figure 4.16: The mission control view overlaying the IDE window: three sketches depict an overview of the project. The user can zoom with a mouse scroll wheel and enter edit elements with the buttons on the right. Figure from [Spychalski, 2013]

The *connection dots* invoke a navigation to the linked code location.

buttons embedded into the sketches in the mission control view. These connection dots refer to either a file or to a range of lines in a file, e.g., a method. When the user clicks on them, the mission control view is dismissed and the IDE navigates to the corresponding file and code location. There, the code lines are color in same color as the connection dot in the overlay. If the mission control view is invoked with the keyboard focus in these lines, the mission control highlights the corresponding connection dot. This way a user can navigate in both directions from and to a sketch: They may look at the sketches in the mission control view and decide to see where parts that the sketch refers to are implemented, or they may look at a method and see that it is referred to in the main sketch, so bringing the mission control to the front may allow for insights into higher level abstractions and help form a model of the code base. CodeGraffiti monitors all edit operations in the files because as lines of code move the connections have to be

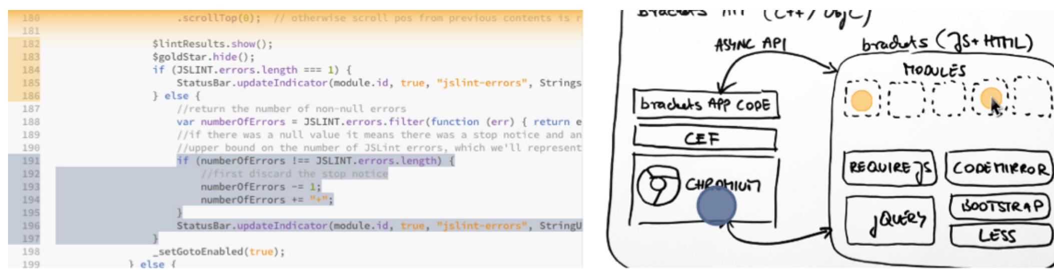


Figure 4.17: The sketchbar view on the right accompanies code file on the left. A blue connection dot and the highlighted blue code fragment indicate a connection. Hovering over the yellow connection dot, the top edge is highlighted, indicating that the connected yellow code fragment is offscreen. [Lichtschlag et al., 2014]

updated as well. In order to edit the sketch in the canvas of mission control view, the user can enter an editing mode by the press of a button in the overlay. Creating a new connection dot automatically links to the currently selected lines of code in the IDE (cf. figure 4.18). The dot can then be moved as any other sketch in the canvas, but its location is in reference to a sketch. Thus, if the sketch moves on the canvas, it drags all it connection dots with it.

The *sketchbar view* implements a sketch-follows-code view.

The *sketchbar view* (cf. figure 4.17) is placed on the right side of a vertical split layout, with the source code residing in left frame. This side-by-side view is common in IDEs, but here, instead of a second source file, we display a sketching space annotating the code on the left side, similar to a wide margin on a paper draft that the editor might scribble on. CodeGraffiti builds one sketchbar view per file, so that each file can have its own sketches displayed alongside the source code. In contrast to the mission control view, the sketchbar view has a limited canvas area: it is of the same 'height' as the corresponding source file, so that sketches are displayed right next to the code lines they embellish. The sketchbar view implemented in the software prototype uses connection dots as well to connect elements of a sketch to lines of code in the partner file (cf. figure 4.18). A developer can add an existing sketch by adding a graphics file; connections are set up in the same manner as in the mission control view.

When the user hovers the cursor over a connection dot in the sketchbar referring to code line that is offscreen (but in

Figure 4.18: A user places a sketch and connections in the sketchbar view (1) by selecting the sketch file (2). They selects a code segment that shall be connected to the added sketch (3), then they adds (4) and places (5) the connection dot on the sketch. While editing the connections synchronous scrolling is deactivated. The sketch stays in view as they then scrolls the content area and selects another code segment (6), again adding (7) and placing (8) a new connection dot for this second connection. Leaving the edit mode (9) reactivates synchronized scrolling. Figure adapted from [Spychalski, 2013]

the current file), this is indicated through highlight at the top or bottom of the source view (cf. figure 4.17). When they then click, the right and left side of the split layout will temporarily lose their strict coupling: the corresponding code scrolls into view and is highlighted, the sketch remains in view as long as the mouse cursor rests over it. This way the user can quickly navigate to multiple locations in the same file, using the sketch in the sketchbar as an index to bookmarked locations in the same file. Analogously, the navigation is reversed when the user starts their navigation from the source code by clicking next to the highlighted code line.

Our CodeGraffiti prototype is build as a plug-in for Adobe Brackets [Adobe, 2013], an open-source and community-driven project, and is built with web technologies such as HTML, CSS and JavaScript. During the implementation process, small feedback loops with participants helped us to define and create a look and feel for the connections between sketches and source code and improve the user experience. Details of the implementation can be found in Spychalski [2013]'s thesis .

CodeGraffiti prototype is build as a plug-in for Adobe Brackets.

In the current version the focus of the design is on exploring navigation through sketches, the user experience of creating sketches remains underdeveloped. Great care in interaction design is needed to make authoring of sketches as hassle-free as sketching with physical pens. With our prototype we can integrate any image file, that is digital sketches, digitalized physical sketches, or just any picture. The user can add image files to the canvas and more them around. But we see this as a crude method to get our investigation going that is good enough for now. We imagine the authoring much more as Calico [Mangano et al., 2010] outlines: a digital capture of individual strokes and editing that allows for quick iteration of ideas. Here, we are most interested in the canvas of the mission control view, and how it helps users navigate and understand a code base. So we did a study that excluded the sketchbar and all authoring actions.

The authoring experience is crude in our prototype.

Navigation Study

To test how the design works for developers, we conducted a between groups user study in which we examined one navigation and understanding task across two conditions. We provided a code base with pre-existing sketches for our testers: the source base used in the study is the source code of the Brackets code editor itself (JavaScript). The sketches were created without any knowledge of our tool or the task of this study by an active Brackets developer who works at Adobe. The connections were created afterwards by us. The sketches did not indicate the task solution by presence alone, because the sketches also referred to schema that

We studied understanding and navigation with CodeGraffiti.

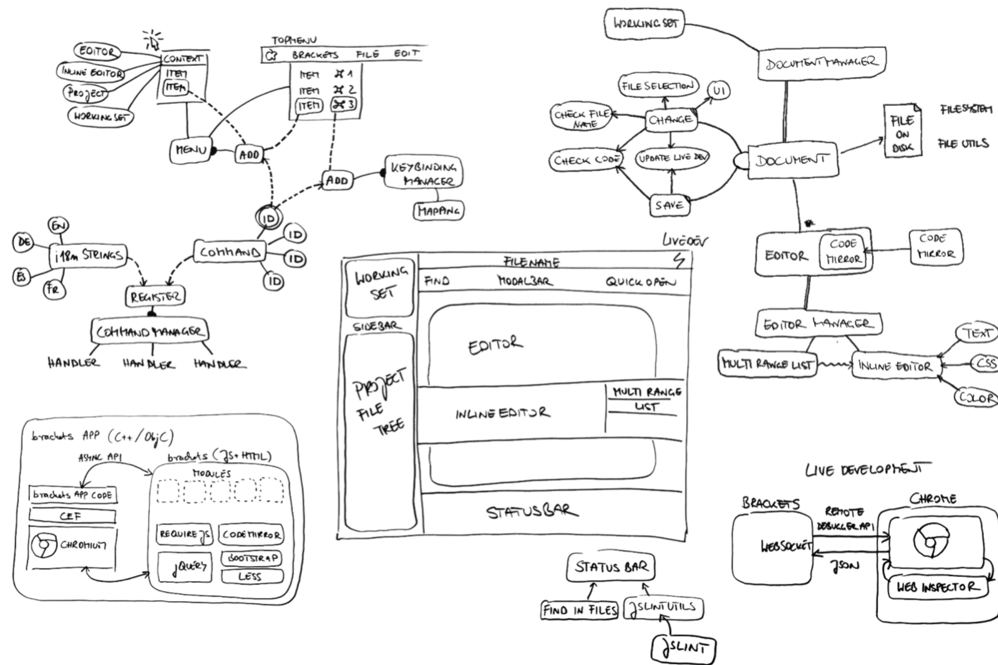


Figure 4.19: The sketch used in the study in the mission control view. [Lichtschlag et al., 2014]

were not part of the solution.

The tasks required the participants to build a mental model of the relevant part of the code base.

We asked participants to locate and identify multiple locations in the code base, to point out the lines of code in which a change should be made, and to verbally outline these modifications. This task was divided into three subtasks T1–T3, which needed to be accomplished in order (adding a menu item, adding the corresponding command, linking both in the controller). These tasks needed the participants to build a mental model of the relevant part of the code base and bridge discontinuities because the solution spanned multiple files. We counted the tasks as successfully completed if the spoken solution would result in a working implementation, but we did not ask participants to actually type the solution.

In the control condition (C1), users worked with a standard installation of the Brackets IDE with the sketches of the organization of the source base presented on a DIN A3 piece

of paper (The sketch is shown in 4.19). In the connection condition (C2), users worked with the Brackets IDE and the mission control view, which showed the same sketch on the canvas and linked with the code. Both relevant and irrelevant sketches were depicted in the mission control view and the printed version to provide a certain level of realism and also so that the presence of a sketch did not indicate the solution. This way, both conditions used the same editor, code base and sketches, only the sketches were presented in different ways. We formed five hypotheses:

We studied a connection condition with the prototype and a control condition with a paper printout.

H1 More programmers solve the task correctly in C2.
H2 Programmers solve the task faster in C2.
H3 Programmers look at sketches more often in C2.
H4 Programmers look at sketches longer in C2.
H5 Programmers (subjectively) find that the connection between sketches and source code is an additional tool supporting their software comprehension process by helping them to understand the conceptual model behind the code.

We recruited a total of 32 participants, 5 female, aged 23 to 36 (average age 28). Twelve participants were graduate and twelve were undergraduate computer science students. Another four graduate students were engineers or physicists with a background in programming and software development. The remaining four participants were professional software developers. Four participants were familiar with the source code of Adobe Brackets. All participants were asked to fill a pre-session questionnaire in order to assess their prior knowledge. We counterbalanced the groups with regard to JavaScript proficiency and source code knowledge. Each participant was given a short introduction to the code editor and its user interface, we introduced shortcuts for *Find* and *Project Wide Search* commands, and we explained the CodeGraffiti plug-in and the shortcut to toggle the mission control view.

32 participants tested our prototype.

We asked the testers to familiarize themselves with the tools and condition by working on a pre-task before the actual evaluation. This pre-task was fixing a bug that was issued within the Adobe Brackets community after the release of Sprint 19 and was fixed with Sprint 20. The Brackets community identified this issue to be suitable for beginners task and therefore we deemed it appropriate for get-

Participants familiarized themselves with a pre-task beforehand.

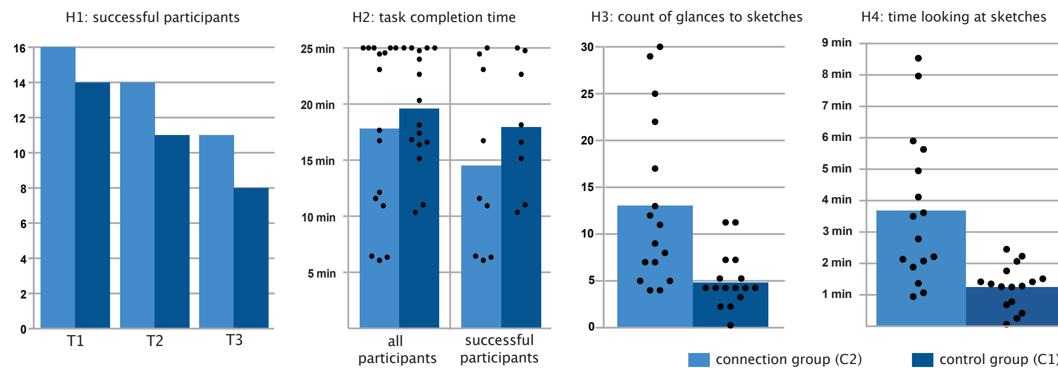


Figure 4.20: Quantitative results of the user study for hypothesis H1 to H4. [Lichtschlag et al., 2014]

We asked about the application of the CodeGraffiti plug-in to participants' actual work projects.

ting used to the editor and the code base[12]. We allotted 20 minutes for this pre-task and 25 minutes for the main task. Participants were asked to think-aloud during the study and sessions were videotaped and annotated with respect to H1 to H4. In order to gather qualitative data with reference to H5, we conducted a semi-structured interview about the potential application of the CodeGraffiti plug-in in actual work projects that the participants are or were recently involved in.

Quantitative Results

We reject **H1**, no subtask showed a significant difference (Fisher's, $p_{T1} = 0.48; p_{T2} = 0.39; p_{T3} = 0.47$). We reject **H2**, users did not perform faster in C2 (Fisher's, $p = 0.32$). We accept **H3**, participants in C2 looked at sketches significantly more often (t-test, $p = 0.003$) than in C1. We accept **H4**, participants in C2 looked at sketches significantly longer (t-test, $p = 0.001$) than in C1 (cf. figure 4.20).

Participants interacted more with the sketches when integrated into the IDE.

Figure 4.21 shows the difference in behavior and interaction with the sketches with the task completion times normalized for each of the 32 participants. Consistent with above observations, the histogram graph depicts that the mission control view sketches were consulted more frequently in C2

[12] https://github.com/adobe/brackets/issues/2930

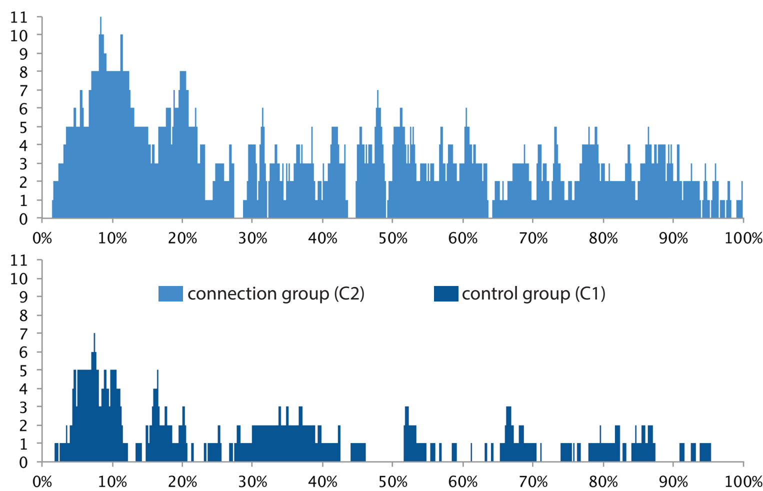


Figure 4.21: Histogram of glances towards the sketches, normalized over task completion time. While most participants of both groups looked at the sketches after reading the task description (peaks at around 10%), participants of the connection group continued the engage the mission control view. [Lichtschlag et al., 2014]

and testers engaged with them over the whole duration of the task.

Qualitative Results

We observed very clear patterns of behavior for both control and connection conditions. Below, we provide an insight into how the average participant made use of the canvas and their connections to the code base in the connection group. The participants of the control group (C1) read the task description and then looked at the sketches provided on paper, they studied each sketch to find potential hints on where to start the task. Most participants stated in think-aloud comments that they could not find anything helpful and moved the paper aside. Eleven participants even moved the paper with the sketches farther to their left and put the paper with the task description right in front of them, so that the sketches disappeared from their field of view. Working with the editor, participants used standard

Participants in the control group used the sketches as a last resort.

IDE navigation operations (tabbed browsing, project tree, scrolling, search, cf. [Ko et al., 2006, Starke et al., 2009]). If participants reached an impasse, they would take another look at the sketches to check for missed clues. At the end of a task, some participants would take another look at the sketch to check if they might have overseen something, e.g., participants pointed out the correct line in the code and described the correct changes they would apply verbally and took a quick look at the sketch to see if they might have forgotten something. Testers expressed statements like: "I will take another look at the sketches, since they have been provided, ... there should be something on them." and "Oh, I completely forgot the sketches, maybe they will help me now ... No, still not helpful." All in all, the sketches were used as a reference, but mainly as a last resort (cf. observations by LaToza et al. [2006]). Most participants looked at the sketches since they were provided, not because they felt the need to. They used standard navigation operations to understand the code, build a mental model, and solve the task.

Participants in the connection group used the sketches as the primary navigation method.

Participants of the connection group (C2) read the task description and then opened the initially closed mission control view. Similarly to the control group, participants in the connection group took the initial glance to get an overview of all sketches. The behavior of this group then deviated from C1: participants constantly switched between the mission control view and the source code in order to navigate the code base. The navigation operations that were used by the C1 members, however, were in part substituted by the navigational facilities of the mission control canvas. Elements of the mission control view turned out to be suitable even if the names of the elements did not coincide with the filename or the method names they were connected to. However, as soon as the participants felt that the sketches and the connections provided by the mission control view would not help them, they fell back into old habits for a short amount of time and, e.g., started to search within files as well as the whole project or navigated via the file tree, only to come back to the mission control view and use its functionality again to continue with the task. Opposite to C1, testers 'defaulted' to sketch based navigation and used other means as a last resort. A few participants stated that

they would not have created some of the connections provided, but rather connected different lines of code or files with the sketches. Interestingly, some of those participants withdrew their statement during the task by saying: "Now that I understand the concept, I guess it makes sense to connect these particular lines of code with that sketch. I'm not sure if it is the best way to do it, but it's OK". A common observation was that when participants of C1 asked: "What was I looking for again?", they immediately opened the mission control view to find the highlighted connection dot in order to see where they were with regard to the sketches, whereas in the same situation most participants of C1 turned to the task description and not to the sketches provided on paper. It is particularly noteworthy that most participants partially or entirely explained the way they understood the tasks and how they worked together by mentally walking through the steps in the sketch to themselves during the session in order to recapitulate their progress in some way. Before they gave their final answer in order to successfully complete the task, they explained their mental model by walking through their individual steps to solve the task: "So I added the menu item here in line 128 and provided the Command-ID that I declared in the Command.js. Now I want to register that Command-ID with the CommandManager and I obviously have to use the register method for that. I have all parameters, but the function that is executed when I click on the menu item and I don't know where this call of the register method has to go". This recapitulation of the progress was made with the opened mission control view, pointing at the canvas and following the sketched lines as well as clicking onto the connection dots to get to the corresponding code segments to prove to themselves, that they had considered every part of the task. We did not observe reasoning about the mental model with the sketch in the control group.

Participants in the control group used the sketches to build a mental model.

Interview

At the time of the interview, 17 participants were working on a solo project, 17 were working on team projects. We initiated the interview session by asking testers how they

would imagine to use the functionality of being able to connect sketches with source code with regard to their projects: Participants liked the idea of *navigation support* through their own projects via the canvas in the mission control view using the connection dots. Some instantly imagined their project affiliated sketches and visualizations and were excited to connect them to the source code. Participants of the control group imagined that the connection dots can be helpful since most participants reported that the printed version of the sketches was not very helpful in finding the correct files or code lines and they had to use the search function instead. Participants imagined the mission control view to provide an adequate *code base overview* of the project and the software architecture, e.g., to see which other team members had to be involved in the task or which other parts of the project had to be considered. Participants of the connection group reported that the mission control view, was a way to not loose track of the task at hand by "zooming out into a kind of meta view".

Participants see value in using sketches on a canvas to navigate, ...

to get a big picture of the code base, ...

Some participants had the idea that the mission control view could be used as a manager's view meaning that a project leader could *capture the development progress* of the project, i.e., new elements that had been added to the view or changes that had been made. The project leader could jump into the corresponding part of the code and have an insight into the work that is already done. Two participants imagined adding 'changed since last visit'-indicators and an overview with a timeline, so that it is possible to scroll through the progress of the project and see the development of new elements and the change of existing elements and their relationships amongst each other. Above all, participants imagined this functionality to be very helpful for *on-boarding* new team members. They reported that it is hard for a new team member to catch up with all the knowledge about the project and the decisions that have been made during the design process and the implementation phase. Sketches created during on-boarding meetings could be an enormous support to get to know the project: "The sketches were like a road map to me. I think using such a map is easier than searching because you don't need to know exactly what you are looking for. The sketches can complete the missing parts or even tell you what to look

for team awareness, ...

for. I think I will start sketching more and archive those sketches.” One professional software developer mentioned that they had special projects made for new team members, that are meant to help the new team members to familiarize themselves with the test project without being at risk of generating any damage to the productive version of the project: “Oh, I can see that implemented in our sample projects and be helpful to new coworkers. Since we already spend time creating these sample projects, adding the connections between sketches and the source code manually wouldn’t be that tragic... as long as the cost-benefit ratio is right, I guess.” One participant in particular had recently joined a software development team and was currently in the on-boarding process. She told of interrupting her mentor constantly and taking notes during these short ad-hoc meetings: “The sketches were like a road map to me. I think using such a map is easier than searching because you don’t need to know exactly what you are looking for. The sketches can complete the missing parts or even tell you what to look for. I think I will start sketching more and archive those sketches. Maybe I can create such a map for our project at work and it could make things easier for the next new team member... Is there a way to connect sketches to source code in the IDE we use at work?” In conclusion, we accept H5: there was a consensus among the participants of C2 that the connections between sketches and source code had helped them formulate a conceptual model. In their statements we see that they want to use the canvas to address problems we identified for code handling (cf. 4.3 “Our Approach”): the missing overview, team communication, reasoning about structures in the code base.

and for team communication.

Participants also identified and confirmed challenges employing connected sketches within their projects: Despite the fact that most participants created sketches or visualizations as a regular part of their work, they still mentioned that this is *time consuming*. Sketches created during team meetings were seen as valuable byproducts without this drawback, but the creation of a sketch for the sketches sake was recognized as an additional burden. It was compared to documenting the source code by some participants: “I guess creating sketches for a project is a nice and helpful thing, but it’s like with documentation: You know you’re

Participants are concerned about the costs of integration, the currentness, and the fidelity of sketches.

supposed to do it, but you still don't do it" (cf. [LaToza et al., 2006]). Some participants considered the quality of their own sketches and were concerned about the *readability of sketches*. They imagined that every team member of their project would contribute and provide sketches and they saw potential problems in how helpful these sketches were if they came below an acceptable level of quality (cf. [Branham et al., 2010]). Very few participants reported to re-sketch their own sketches if out-to-date, the foremost mentioned problem was the *currentness* of sketches and connections. Participants were torn between the fact that they would like to have the connections as well as the sketches created automatically and the fact that the informality of hand-drawn sketches had a "certain charm" of their own and were a "trove of mental work", as one participant phrased it. All testers agreed that maintaining sketches and connections can be realistic for a rather small team of developers or a medium-sized project, but *larger teams* would have problems to maintain the sketches and their quality.

We see that sketches are useful and become even more useful and immediately useable when integrated into the IDE with a canvas and connections to the source code. Users use the canvas to understand and navigate the source code and use it over the standard controls of the IDE. If connections are authored well, they provide allow code to be abstracted to a sketch or to be referenced. Thereby a 'big picture' of the code base emerges and discontinuities in the code base can be overcome. But these benefits are conditional on the cost benefit of integrating the sketches and connections, and maybe conditional on the currentness of the sketch.

Chapter 5

Excursus: Writing on a Canvas

"Being a writer is a very peculiar sort of a job: it's always you versus a blank sheet of paper (or a blank screen) and quite often the blank piece of paper wins"

—Neil Gaiman

In the last chapter we often compared coding practices to writing of narrative texts. We drew comparisons to instances of both writing and coding when we discussed difficulties according to the theoretical models. Last chapter we identified that the authoring is a critical step again for working with source code, but have little data yet, how build sustainable solutions. As it so happens, our Brackets prototype presented in the previous chapter is perfectly fine handling LaTeX texts. So, we decided to write this thesis document with this authoring environment with the canvas as a spatial model of the thesis in mind. The figure at the end of the introduction, 1.1, is the result of this process and as foreshadowed, we discuss details of its construction below.

This thesis document was written with our coding sketching prototype.

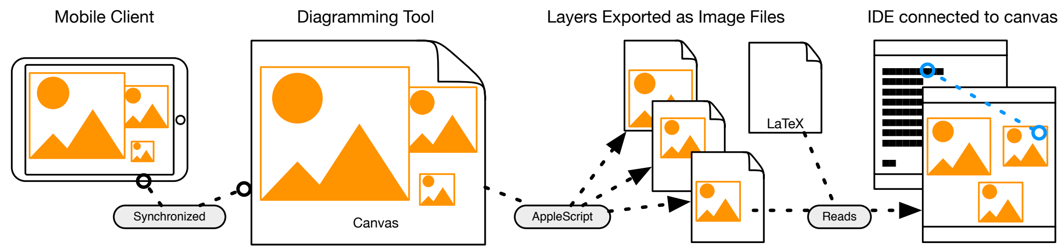


Figure 5.1: Setup of the writing experiment: the canvas elements were edited in the diagramming tool and then exported to image files and displayed in the Brackets IDE.

5.1 Setup

We authored text and connections in the Brackets editor with the CodeGraffiti plug-in.

Our toolchain has three components (cf. 5.1): The thesis is written in LaTeX [Lamport, 1984], a document markup language often used for publication of scientific documents. The Brackets editor treats the LaTeX source files similar so source code, but compiles them to a printable document, rather than an executable program. In the Brackets editor, we run our CodeGraffiti plug-in, with the same capabilities as described in chapter 4.3.3 "CodeGraffiti: Our Sketching Based Design".

We authored sketches in OmniGraffle for Mac and iOS.

Our second component is OmniGraffle [OmniGroup, 2015], a diagramming tool with rich editing options for vector graphics. We designed a single document for the canvas and performed all the edits of the objects in the canvas in the OmniGraffle program. The elements on the canvas are separated into five layers (one per chapter of this document). The diagram was edited on both the iPad and the Mac, the local documents were connected through a synchronization mechanism.

An AppleScript program exported images from the diagramming to to the IDE.

The last component is a custom AppleScript [Apple, 2015b] program. It takes the canvas file, iterates over all layers, displays only the current layer, and exports each into a JPEG file. The new images overwrite the previous version and thus the CodeGraffiti extension displays the new version in the respective views. A typical workflow involves changing the canvas in the diagram editor, invoking the script, reloading the brackets view, and defining connections be-

tween text elements and the new elements of the canvas. By separating the image into multiple layers, the net effect is that each chapter of this document has its own element on the canvas. Each chapter can be moved by itself and keeps the connections anchored to itself.

5.2 Observations

The thesis was written in this environment all the time over a time span of 24 weeks in with the tool setup described above. With the canvas always ready at hand, we build all visual elements of this thesis in this place first. All figures in this thesis were built in this program and at first on the shared canvas. Most figures were kept in the document because they build an abstraction or representative of the written sections of the text. Other figures pull together distant parts of the text, e.g., the figure on the three user roles. However, it should be noted that we trimmed down the elements on the canvas a bit, very late in the production, anticipating the use of the canvas as the figure in the first chapter. Some figures were exported into image files and then moved separate diagram document (In OmniGraffle, it is easy to just export the current selected elements). There are of course also elements on the canvas that connect to text, but are not used as figures.

The canvas was a staging ground for the figures.

Editing all figures on a shared canvas allowed a couple of key benefits to the author: First, figures were naturally visible side by side, and thus it was easy develop a common visual design language and color schemes by reusing elements or styles. Second, on a canvas the constraints of the final placement in the text were deferred until export and therefore kept out of the authors mind until needed. This 'late refinement' of the figures is analogue to the argument against content cutting, cf. 3.2.4 "Content Cutting". Third, most of the time the canvas was in quite a 'rough' shape with placeholder graphics that were 'beautified' late for the final version. This workflow relates to another problem we discussed for presentations, cf. 3.2.4 "Detail Trap". The canvas also included a interactive element that on click directly invoked the exporting AppleScript.

Editing on the canvas has similar considerations compared to presentation authoring.

Long LaTeX source files are hard to navigate because the semantic structure is not represented in the editing view and because the markup, especially in late stages of writing introduces syntactic artifacts to the text structure. On longer documents search for known phrases and quick invocations of the 'mission control view' of the canvas proved valuable to navigate to known positions. We experimented with adding temporary connection dots on the canvas to 'bookmark' a location in the document that were of importance to the current task. Then the connections were removed after that task was done. In some instances the dot was not anchored to a meaningful location, because we fully planned to remove it. A better editing environment could support these 'ephemeral bookmarks' by a separate mechanism.

We added temporary connections as bookmarks.

We planned this as a single scale layout (cf. 2.2.5 "Multi-Level Interaction") anticipating its place in the thesis as a figure. This placed a constraint to the editing that the elements on a roughly similar visual level, leading to a more 'flattened' canvas. It follows that adding new content to the canvas is harder, as one cannot simply zoom until there is enough space between existing content. This would lead to a multi-level layout, which we wanted to avoid. We solved this by manually moving elements to create space (selecting all elements to the right of the anticipated placements and moving them). A better editing environment could automatically push elements to the side or 'reveal a fold the canvas' to make room.

It is harder to edit a single-level canvas.

In our setup, links mostly stayed in place when moving elements on the canvas, since our export script segmented the diagram into layers. But, changes to the elements of a layer needed repositioning of the anchors because the CodeGraffiti cannot anchor to the elements of the canvas directly. This is a limitation of the prototype and underlines that the full syntactic model of the objects on the canvas should be exposed to a more advanced system. That requires either using a more complex (and standardized) file format and/or moving all editing controls of the canvas from the diagramming tool into the IDE. A standardized file format would allow the author to use their digital imaging tool of choice. The second option would addition-

The syntactic model of the objects on the canvas should be exposed to the IDE.

ally reduce the need for any context switches between running applications and is clearly preferable from a usability standpoint.

Often, a change in the image happens soon after or before a change in text markup. E.g., the author transcribed something from a physical notebook to the canvas, and then immediately outlined the text that describes that part. Or, a new section is added to the text and then a visual element is introduced to the canvas to represent the argument. A better editing environment can identify these pairs of editing operations and anticipate that the user plans a connection between them. This in turn would allow the program to simplify the definition of connections in two ways. On invoking the connection operation it could suggest endpoints that are considered likely because they were recently edited. Or it could suggest tentative connections on its own, which the author than affirms or rejects.

The IDE can anticipate connections.

This examination of the author's workflows and needs is hardly a proper evaluation because of the limited sample size of only one author and one project. Also, the author is well experienced with ZUIs and canvas layouts and enthusiastic about them. Yet, the we have identified a couple of points that give design directions to the next prototyping iteration. The prototypical nature of the toolchain is less than ideal because it introduced friction in the form of context switches and more user actions needed to move a change from diagramming tool to canvas. We gained no information on the understanding part of our user role model, only a first time reader of this document will be able to provide feedback on this. One could imagine to use the canvas figure with hyperlinks in the beginning of the thesis instead or adjunct to the table of contents if this thesis document were to be shipped digitally. It will also be interesting to see how we might be able to the appropriate the canvas to the upcoming thesis defense talk.

The study has many limitations.

Chapter 6

Discussion

Beware of the man who works hard to learn something, learns it, and finds himself no wiser than before. [...] He is full of murderous resentment of people who are ignorant without having come by their ignorance the hard way.

—Kurt Vonnegut, "Cat's Cradle", 1963

This thesis started with the aim to explore zoomable user interfaces in practical domains. We presented a model of author, navigator, learner with which we investigated ZUIs in these domains. For presentations, we investigated the drawbacks of overview + detail designs and studied all three user roles. We then analyzed how traditional integrated development environments (IDEs) lack support for communication about code, and designed three canvas approaches to remedy that. Our three approaches explored options to project text to spatial information landscapes: vocabulary based, pattern language based, and hand-drawn sketching based designs. In total, we presented results from multiple studies and one longterm exploration of authoring. So let us integrate the results in our body of knowledge and draw conclusions before we present an outlook on future work.

In this chapter we summarize our findings.

6.1 What We Learned for Presentations

Authoring presentations is impacted strongly by canvas affordances.

So far we see in the domain of presentations that *authoring* has changed drastically. Users adopt to the possibilities of the canvas and make use of it. We see that they create more diverse designs and that they employ the two dimensions to express complexities in the talks. Both in the lab studies and in the field study, the landscapes are not linear as in the slideware condition but used expressive layout facilitating overviews. We also find that the presentation documents are not simply different but have more qualities argued for by presentation advice: richer and more memorable presentations with overviews to support understanding. For the authoring role, our canvas design achieved our design goals and we are able to substantiate the claims about authoring by Good and Bederson [2001] (cf. chapter 2.5 "Promise of Zoomable User Interfaces").

Presentations should allow a 'shallow' multi-level layout, support grouping, and decorative layout strategies.

We have a good insight into how users actually build with the tools with our audience field study. The frequent use of hierarchies indicates that a single-level layout (as with Fly) leaves users wanting, and instead a multi-level design (as in Prezi) should be preferred. The explicit content and topic layer in Fly allowed to build semantic zooming with images on the topic layer, but that was seldomly used. The presentations with geometric zooming in Prezi produce meaningful overviews with less formalism. We can recommend that a tool should support ∼5 levels of hierarchy in a multi-scale environment to capture 96% of the practice by presenters. This value might change for other domains than presenting, but most of the presentations remain at 1–3 levels. With regards to layouts, authors prefer to build groups and decorative layouts, both of them should be supported by the canvas tool. Prezi already supports authoring groups and Fly's third iteration allows the user to group canvas elements to topics, which automatically creates a label and a visual enclosure. With decorative layouts, the author runs the risk to artificially limit the space available for content because it has to fit the background image. It would be interesting to see if content-aware scaling techniques [Wikipedia, 2015b] can be integrated to an authoring environment so that the image can

be retargeted to the presentations needs. In case the author aims for a single-scale layout and pursues a breath-first layout of content, we observed that tools should be able to reveal a free space in between existing content (cf. chapter 5 "Excursus: Writing on a Canvas").

Then, we investigated the *presenter* experience with a controlled lab study and find that presenters experience canvas and slide tools differently. More specifically, participants of our study that scored high on spatial ability or were less experienced preferred the canvas condition, while experienced or lower spatial ability presenters preferred classic slideware. Due to the strong overlap in our tester population between experience and lower spatial ability, we cannot attribute this effect to a single or a combination of these factors. We expected lower spatial ability to interact with the canvas condition due to its ZUI nature, but we could also explain that more experienced presenters are well versed in slideware and hence feel right at home. A repeat of the study with a clear distinction between these two explanatory variables is needed to shed light on this issue. Interestingly, this difference is lessened in the search for a loosely defined position, which is a task that benefits particularly from the canvas format, because the presenter can quickly zoom out to get an overview and pinpoint her target. We conclude that the increased degrees of freedom on a canvas come with a drawback: a simpler linear format can be easier to handle for some presenters. One could follow from that to always limit the format during presentation delivery, but we have also shown that for some presenters this would be unfavorable. We suggest that during delivery, canvas tools should allow the user to limit the navigation to the linear format until they needs the free format. Additionally, they should offer an easy way to get back to the last position on the presentation path.

Presenters experience canvas and slide tools differently.

Canvas presentation tools must keep the degrees of freedom under control during presentation delivery.

Finally, we investigated the *audience* experience with a controlled lab study. We found that learning performance was largely the same in both canvas and slideware conditions, both in short term and long term recall, and also both for content and structure recall. On the other hand, students clearly preferred the canvas-based presentation and felt that they were more oriented and aware of the structure. In

Audience recall was not improved through canvas presentations.

this, our results closely mirror the results by Good [2003]'s audience study, even though the layout of content in Fly is very different from the slide-based canvas of CounterPoint. And, of course, this result is very much inline with the existing evaluations surrounding the 'media learning debate' (cf. chapter 3.2.3 "No Significant Difference"). If there is an effect of canvas presentations on recall by passive audiences, it is probably very small. Considering active interactions with canvas documents by the learner, we can hypothesize a different outcome. We discuss this below in 6.5 "Our Three User Roles Model".

Audiences preferred canvas presentations.

6.2 Next Directions for Presentations

Canvas presentations are in a good place: they are adopted in a niche of the market place, and presenters are enthusiastic about them. If they were to be integrated into a mainstream office package, we believe they would be used much more widely. Based on our experiences and other approaches in the related work of presentation systems, we can outline the next challenges and developments.

A particular point of contention among presenters is the question on whether to hide or show upcoming content in the presentation path. Some do not mind, some go lengths to avoid content in the view of the camera before it is mentioned in the talk. A common desire is to build up content into the view, *incrementally revealing* elements on the canvas without moving the camera.[1] This could be hypothesized be helpful for learners to not be hit with too much content a same time. Prezi has a capability for incremental revealing in their software, this introduces 'mini stops' to the presentation path that do not move the camera. Another desire is to *animate elements* on the canvas, similar to animated builds in Apple Keynote. This is a fundamental challenge to the canvas metaphor, as it breaks with the rule that a canvas element will be at a defined place all the time. Both these desires introduce a temporal dependency to the elements of the canvas. We will discuss the implications

Some presenters desire abilities to animate and incrementally reveal content on a canvas.

[1] Similar to bullet points fading in on a click.

below in 6.7 "ZUIs and the Canvas".

There are hybrid approaches to presentation tools that combine elements of writing with markup languages and outlining of the structure of the talk with canvas layouts for the presentation [Edge et al., 2013, to appear]. This very interesting approach forgoes the visual layout up to the last minute. This is motivated by similar concerns that we outlined as *detail trap* and *content cutting*. The design is very different and challenging to our interpretation of the benefits of the canvas and we are excited to see how further experimentation and evaluation can expand the design space.

Hybrid approaches to presentation tools combine linear definition of the storyline, but present on a canvas.

Apart from the studies that should accompany designs for incremental reveal or hybrid approaches, we can also expand on studies to further investigate our existing research questions. More field studies can affirm our lab results with presenters and audiences, but since real world presentations will hardly be comparable, these studies would most likely be designed as qualitative studies. The next study could be an interview with experienced presenters that have held many Prezi presentations about their experience. Furthermore, we have not investigated how presenters select or switch between paths in a presentation during delivery, this is an interesting field to study and iterate on interaction design.

Future studies should expand on the field studies.

6.3 Critique of Our Coding Designs

'Code' is a revealing word, because its linguistic roots hint at the problem associated with software engineering: it is an hidden knowledge, hard to decipher, and not a natural language. This is obviously not what one wants to have a the basis of one of the most important industries. And, of course, we want to allow more people to begin programming and to be able to build software. In democratizing the skills to design software, code is a problem.

Code is hard to understand.

Code is free of a gestalt, it is hard to recognize its structure, and it is hard to identify and bring together the parts in a code base that are part of a single multiplexed struc-

ture. As with the presentation domain, a linear format of text is too restricting to foster communication between colleagues. This is why software engineers document their mental models outside of the IDE (in unfortunately volatile form). We suggest that one needs to make code a communicable artifact by abstracting from the textual structure to a graphical structure. Our approach closely mirrors how developers work in practice when they sketch on whiteboard or paper while they design or investigate code. The canvas also promises to bring a sense of place in the code base to navigation, placing code in relation to how developers communicate about it and to related but distant code fragments. We show that variable naming, formalized patterns, and hand drawn sketches are ways to build a canvas representation of code. Thus, we both investigate ZUIs in a new domain and also experiment with ways to abstract code to graphical artifacts.

Introducing a high level canvas view promises to make it easier.

We developed *CodeGestalt*, a vocabulary based prototype, and introduced the *tag overlay* and *thematic relations* for software visualizations. With them, developers can find related code artifacts and take advantage of the work that went into the naming of identifiers. But the thematic relations saw little use by participants in our study, the call and heritage relations were sufficient in most cases. This indicates that our vocabulary elements offer little opportunity to reveal structures that are not already present in the call relations. The vocabulary elements on the canvas did abstract from the code, but we saw only few cases of users reasoning with them to understand the structure of the code base. A major drawback of this approach is that the visualization can only be built when there are already code fragments to parse for vocabulary, thus the canvas cannot be designed before code exists. Future iterations on the design should examine different tag cloud metrics and filters to reduce clutter. The mathematical model behind tag weights offers room for improvement. Currently it puts too much emphasis on trivial terms such as 'get'. To avoid this, weights could be normalized based on frequencies in the whole code base and considering word stems. Testers asked for the selection of multiple elements in the tag overlay, using the cross-product of the respective weights for highlighting. This indicates that users expect a combina-

Our vocabulary approach saw little user adoption.

Our vocabulary approach included a lot of cluttered tags.

tion of multiple terms may be better able capture a given concept better than one. Our users also asked to get relation previews for elements not yet included in the diagram, to make the tool a substitute for Eclipse's call and type hierarchy views, and more useful for exploratory tasks.

Design patterns work well as a conversation piece between developers, even when they are authored with little formalism and have only a graph of code snippets as the only required feature. They do however have a drawback that limits their usefulness: since design patterns describe reusable solutions they require expert knowledge of the code base to be recorded [Détienne, 2002]. They already need the author to possess the strategic knowledge of the structures int he code base, and thus, are not so suited to capture design that is still being worked on. We also presented them to be authored by dragging code fragments into the canvas, but that requires the code to already exist, much like the drawback we noted for the vocabulary based approach. We are unhappy with the graphical fidelity of the graphs and cannot recommend them as a way to project text into the canvas. If one desires a more polished look that hand-drawn sketches, e.g., for a canvas such as figure 1.1 or for the example described in chapter 4.3.2 "CodeMixer: Our Design Patterns Based Design", then we recommend using a graphical sketching tool, with emphasis on graphical fidelity and expressiveness, the graph approach of CodeMixer.

We presented a sketching prototype and evaluated it with a navigation task.

In our third design we presented a way to connect source code to sketches that depict anything from the low-level details of the source code to the high-level concepts about the source code. We presented two designs to integrate the canvas into the IDE: a global map (mission control view) and an assistive view on the side of the code (the sketchbar). We reasoned how a code base can be navigated through connected visual sketches and how it helps developers to comprehend the context of source code, to orient within the context, and to support mental walkthroughs. In a between groups user study with the mission control view, participants predominantly used the mission control view for navigation instead of traditional file and search navigation. They quickly invoked it, selected the target navi-

gation, and dismissed it again. Furthermore, testers made more use of the sketches and we could observe that they used the sketches to formulate a conceptual model and finally solve their task. The results of the evaluation showed no significant difference in the task success rates or the task completion times. Even though participants who used our prototype were faster on average, this indicates that the effect size of time savings (if any) is small. Which is a bit disappointing compared to the clear time savings that the canvas design of CodeBubbles achieved [Bragdon et al., 2010]. Testers looked significantly longer and significantly more frequent at sketches on average, with no adverse effects on total time needed and success on the task. We can deduct that the embedding of a sketch in the IDE promotes it to an immediately available source of documentation, whereas the printed version remains an additional, but not directly usable source of information. Hence, this design offers a way to promote sketch use and sketch retention. Which is quite interesting given that we have a clear finding that presently design documents are often out of date or discarded. Given that many developers already create sketches (on paper, whiteboard, or tablets), we can hypothesize that our design can promote not throwing them away and connecting them to the code base. To achieve this, the authoring experience needs to be excellent so that designers do not see it as a burden to integrate sketches.

Testers adopted the mission control view over traditional navigation.

Our design promotes retention of previously discarded documents.

The vocabulary design comes with the drawback that it can only suggest canvas elements that are already present in the current state of the code base and that the compiler can analyze. It does not lend itself to designing software before it is implemented. The design pattern approach also comes with a problem, namely that developers need to know the pattern 'language' and that they need to be able to identify them to use them. Since Détienne [2002] reports that this is far from given, we judge our software pattern design to be an interface for experts. Comparing the approaches, sketching proved most flexible. It allows the design of the canvas independent of prior knowledge of schemas and independent of the current state of the code base. But sketching is also hardest of the approaches to integrate into the code base. All connections are defined 'by hand', increasing the work load of an author and discouraging changes

There are trade-offs between our approaches with regard to flexibility of use and integration into the code base.

to the canvas. In the study, participants were concerned about the costs of integrating the sketches and concerned about keeping the sketches up to date. The vocabulary allows the tool to make the connections itself and a pattern comes with source fragments also brings its own connections to the table.

We think it is most promising to stay focussed on the flexibility and try to integrate second. After all, compliance with the rigidity of unnatural language was identified as a major problem for communication. Picking up developers at their current practices of externalizing mental models, which is sketching, seems like the most promising route to us. Also our testers expressed more enthusiasm for the sketching prototype, although we did not perform a formal comparison between our designs. To overcome the burden of integrating sketching there are some things that one could try to combine the approaches with a focus on quick and lightweight creation (see below).

We suggest sketching as the best design to develop further.

In chapter 2.1.8 "Review of the Canvas Design Space" we motivated this investigation with the search for a proper abstraction to text that is scalable. We presented three novel approaches that produce scalable representations and studied the navigation with our CodeGraffiti prototype, finding that hand-drawn sketches can serve as such an abstraction. Furthermore, we find that users quickly adopted to using these sketches on a canvas instead of the traditional navigation methods. They used them to reason about their mental models when zooming in and out of the text.

6.4 Next Directions for Sketching

We need the experience in all user tasks—authoring, navigating, understanding—to be working well. Our study indicates that navigators and learners are already well supported, but that we have to improve on the authoring experience. In the previous chapter we took our CodeGraffiti plug-in and the Brackets IDE to a test and found a lot of friction in the authoring experience. This is understandable, since the prototype is a first iteration and was build with a

We need a better authoring workflow.

focus on navigation. Before we can perform a study with authoring a canvas through sketching, we have to improve the experience in two critical ways: first, we need a better capture of sketches, second, we need a more lightweight way to create the connections. Related work has investigated support for authoring of sketches for software design [Mangano et al., 2010] and Parnin et al. [2010] presents a workflow for sketch capture with touch interfaces.

The capture of sketches needs to be improved.

A better *capture of sketches* is in large parts a hardware problem, because digital sketching hardware is disappointing. We tried the design of sketches on an iPad in the previous chapter, but it still feels incredibly clumsy[2] when compared to a proper pen on a proper surface. Tablet interaction promises to get much better once more vendors integrate pen as a first class interactive device and do not send events though the touch controls. There is wonderful work by Hinckley et al. [2010] outlining how touch and pen interaction together promise a great authoring experience. Digitalization of non-interactive surfaces (paper sketches, whiteboards) is possible, but we are not aware of a low friction solution. Any capture of sketches should record on the level of the individual stroke, so that the canvas can identify atomic elements on the canvas (as opposed to an image with many strokes). With this information one can automatically detect groups of strokes that are spatially and temporally close and group them as a semantic unit (cf. Karrer et al. [2010] for an implementation of this for written exams). This then allows the IDE to make connections to these semantic units, improving the resilience of connections upon modification (cf. chapter 5.2 "Observations").

The definition of connections to the code should be lightweight.

To improve the experience of creating connections, we already observed that the IDE could predict connections (cf. previous chapter). This would allow two *lightweight ways to create the connections*: First, the user simply confirms a suggested connection or rejects it. Second, on invoking the connection from the current selection the IDE already could suggest endpoints that are considered likely and the user does not have to navigate to the endpoint. We could also try to leverage the benefits of the other two de-

[2]It feels like finger painting.

signs: when designing a label, the IDE could suggest terms from the linked vocabulary. We can imagine a vision where patterns could be integrated with hand drawn sketches to take care of the formal parts that are recognizable building blocks by the API vendor. Then, when integrating a pattern with code examples the links are already present.

Studies could investigate the claim of better documentation.

If authoring the mental models on the canvas is as straightforward as drawing on paper and whiteboards, we can hope to convince engineers to record they documentation connected to the IDE. We already know that a linked canvas promotes it to be used and brings it into the developers context. We could hypothesize that a linked documentation is better maintained than a documentation that is unlinked and thus easily overlooked. Studies should investigate if our canvas design actually promotes better documentation maintenance. Since the currentness of the documentation was a concern (4.3.3 "Interview"), one could hope that sketched documentation still provides value unless the code has been fundamentally refactored. See [Wittenhagen, 2015] for an exploration of designs to browse the history of a code base and [Schulz, 2014] for designs to browse the history of sketches in particular with these concerns in mind. Prause [2013] investigates the use of gamification to promote documentation practices, maybe this is something that can be playfully combined with sketching.

The canvas can 'visualize' the location of a programmer in the code base.

Since the canvas gives the developer a sense of place in the code base, one could investigate designs that trace the navigation of the developer, e.g., to help them resume their work after an interruption (cf. [Fouse et al., 2013]) or to increase group awareness by showing the location of collaborators on the canvas (cf. [Laufer et al., 2011] and 4.3.3 "Interview"). Also, we can imagine other media to be linked to the canvas such as videos. Videos related to code bases include tutorials, API developer talks, and recordings produced for documentation. All these are ways for developers to communicate and when the video shows or refers to source code, one can design with this in mind in two ways: First, the canvas could be used to navigate to positions in the video that talk about the parts of the canvas (cf. [Karrer, 2013]). Second, the canvas could be presented as adjunct material to a video an follow the discussion (cf. [Corsten,

The canvas can be used to navigate adjunct videos, or vice versa.

2009]) In the latter case, again a case could be made for incremental revealing.

6.5 Our Three User Roles Model

The author/navigator/learner model guided our research.

In trying to understand the ways zoomable user interfaces are interacted with, we formed a simple lens to order our knowledge: we proposed to study three prototypical user roles: the author, the navigator, the learner. With this model we could order our knowledge of the previous studies on ZUIs and identify that we have little information on how ZUIs are authored (cf. 2.6.2 "Authoring"). Even though it had previously been claimed that this would be an area that ZUIs would bestow benefits. We applied this model to our investigation of presentation support systems and it proved valuable to form three different kinds of studies. The studies led to results that underlined our claim that ZUIs do actually influence them differently: e.g., we see that authors create more diverse layouts, but we cannot show improved learning for learners. Each prototypical user has different objective and needs. Following that we can recommend to other researchers to clearly differentiate between the roles of users to conduct their studies. In that sense the model is a success because it guided our studies the right path and it helped us build an understanding for our observations.

One could hypothesize that an interaction effect would allow some of the benefits for authors translate to better understanding.

But, by using this model we also detached the users from another and walled off possible interactions between the user groups. E.g., in our audience study, we took the author and the presenter out of the equation in the interest of comparable conditions. But one can also make the argument, that proper presentations need the interactions to truly develop benefits. A very similar argument is made by Brown [1992] for teaching. If we assume for a moment that such an interaction exists, which direction would the effect take? Since we already how authors create different presentations that are closer to what guidelines on good presentation visualizations suggest, we can make an argument, that this interaction (should it exist) can only benefit audiences in the end. We would need a multi stage study to show such an effect, e.g., starting with a task for presentation au-

thors, and then use these potentially biased documents in a learning study. This would be quite an endeavor.

An active learner plays by different rules.

When looking at the interplay of presentations and video navigation (cf. chapter 3.3.4 "DragonFly"), we noticed that we cannot simply treat active learners that navigate a video on a canvas in the same way that we treat a passive audience in a talk. The active learner interacts with the medium, navigates, rewinds, and can build a model through their own actions. Opposed to that that, the passive learner consumes the medium no different than a movie with a particular kind of visuals, but has a direct conversation with the presenter. In our model of presentation users we therefore amended the model with an *reviewer* (cf. chapter 3.1.5 "The Reviewer") modeling active learners, but we did not get to study their interaction with the ZUIs on their own. Also, when we consider an active learner, we are investigating a learner that learns with different methods, and that makes [Clark, 1994]'s claim that media will not influence learning not applicable in this case. Considering that more and more students are learning in non-traditional learning environments, e.g., *massive open online courses* (MOOCS), this role should be studied closely.

The simple model is not able to accurately describe all user tasks.

Then, when studying the roles of software developer, we saw that the activities surrounding an IDE cannot be cast into one role at a time. Where we could claim that authors are familiar with the material and reify their model of the talk, we cannot claim the same software development. A programmer is very unlikely to already have a formed model of the finished code base, he mixes roles of authoring, navigation, and learning as he builds a solution. Similarly, a bug-fix task will involve first an understanding of the code base, then an authoring task (cf. figure 4.5 and [Ko et al., 2006]). We need to understand our model roles as different activities that a user takes on while working on their task (cf. LaToza et al. [2006]). It is also interesting that Cherubini et al. [2007] used a three part classification when organizing the reasons to sketch in IDEs.

6.6 Outlook

We have previously often referenced writing and literature. In the previous chapter, we detailed how we used our canvas tool as authors to write this thesis, so for a moment let us consider how one could expand on the interplay of literature and canvas designs.

Often readers desire a high-level overview of the narrative, not only for non-fictional texts where a good index and table of contents are obligatory, but also for fictional stories. Many stories have strong spatial component to them, e.g., because the protagonists travel a lot (e.g., "The Lord of the Rings" [Tolkien, 1954]). It is not uncommon for those stories to feature a map in the beginning or end of the book to trace the characters' travels. Or fan publications build visualizations of the character interactions on a canvas (e.g., [Munroe, 2009]). With digital publications of novels we can conceive ways for authors to bring these maps to the audience in an interactive fashion.

One can imagine the canvas to function as an index to a narrative.

A benefit for the reader could be that they has an easier time resuming reading after a long interruption, e.g., after waiting for a new installment in the series. (We had a similar argument before for interrupted work for software engineers). We can also think about canvases that do not directly display a geographic map, but rather an network of complex character interactions (e.g., "Game of Thrones" [Martin, 1996]). Interacting with the canvas could allow the reader refresh their memory or reread a particular adventure. But, once again we have to be careful not to reveal to much about the unexperienced narrative. E.g., considering "Around the World in Eighty Days" [Verne, 1999], the reader should not know what path the story takes and how it concludes. We (again) see that incremental revealing becomes a recurring problem in storytelling on a canvas.

Again one has to be careful not to reveal to much.

Formal investigations of literature are very happy to investigate [Plachta, 1997, Fetz and Kastberger, 1998] the author's adjunct material to texts in order to understand their mental models and intentions. Just as software engineers, many authors externalize their mental models. For the au-

thor it is a way to untangle his thoughts and bring structure to a narrative (Cf. Kurt Vonnegut's Shapes of Stories [Post, 2015]). Previous or concurrent versions and notebooks are of prime interest to the literature scientist. For them, these artifacts are view onto the work processes and context of the author at the time of writing (cf. 4.4). So, just as we can use the canvas to understand the software engineer's mental model, in narrative texts, we could also use it to shed light on the question "What was the author's intention?".

Literature scientists are very interested in canvas artifacts by the author.

6.7 ZUIs and the Canvas

We have successfully investigated canvas presentations in more depth and from more angles than before, we have gained insight into who benefits from the format. We have found that the canvas enables authors to be creative, especially when compared to linear slideware. In coding we found again that the linearity of the text cannot represent the multiplexed structure of source code and mental models. With sketches on the canvas, we can support the developer to overcome this limitation. Overall, we are quite positive on the results we achieved with the application of the canvas metaphor.

The canvas is very very valuable when applied to presentations and software.

When we built these aforementioned prototypes, we first thought of them as zoomable user interfaces. As we previously noted, zoomable user interfaces are about much more than zooming (cf. 2.1.4 "Zoomable User Interfaces"). In building the prototypes and in proposing the benefits of ZUIs versus slideware, we noticed that it zooming aspect is not really the explanatory variable. Instead, what enabled us to escape the detail trap, railed against content cutting, we were really making an argument against fragmentation. And yes, it is nice to zoom on it, but the unbroken continuousness of the information landscape is what gives power to our designs. Hence, we introduced the term 'canvas' when we are really talking about this and not a specific implementation to navigating such a space (even though it is zooming most of the time). Revisiting the design space (cf. 2.1.8 "Review of the Canvas Design Space") allows to differentiate the two terms and to pivot the canvas against

Canvas and ZUIs describe slightly different concepts.

fragmented designs.

Leung and Apperley [1994] proposed to add a graphical abstraction to inherently un-graphical data and we did so in our investigation of canvas prototypes for IDE visualizations. But we also got to see the effects of the reverse, when information gets broken up and fragmented onto slides and the communication gets distorted (cf. 2.1.7 "Fragmentation and Continuity of the Information Landscape"). Our studies underline that the author is benefitting from the canvas, just as related work claimed, but we have to refute the claims about improved learning. For us, *non-fragmented authoring* is the promise of zoomable user interfaces.

Incremental revealing challenges the spatial layout of canvas metaphor.

We also got to see the limits of canvas metaphors: our participants ask for incremental revealing and animated builds. Again, when considering future directions for writing (code) we see this problem again. Incremental revealing is potentially a concern that is tied to storytelling on a canvas and thus something that needs further investigation to understand the canvas (and by extension zoomable user interfaces). Implementing a dependency on the elements of the canvas where they are toggled visible, moved, or rotated depending on the progression of the narration extends the two-dimensional quality of the canvas by a timed component. The spatial memory would be in part violated, if elements are only sometimes visible or move around. From a design user experience perspective a presenter would introduce a mode to the presentation document depending on whether it was already visited or not. These modes can easily lead to errors (as they do with incremental revealing of bullet points). Contrasting this opinion, we have to remember how metaphors can be resistant to changes that break with the metaphor and this issue is clearly a point where the metaphor of the canvas is challenged (cf. section 2.2.4 "Pad++: Metaphor-free Navigation"). This is an interesting next design direction for canvas design tools, and studies should investigate its usefulness and a potential impact on user orientation, especially audiences.

6.8 Limitations

Our studies are limited to the framework under which we present and develop software today. Both domains have already seen large changes in how users work (e.g., digital presentation support, object-oriented languages) and will continue to do so. Many presentations and lectures are now targeted at later consumption through video recordings, online distribution, and online education. This could lead to different user tasks to which our studies are not applicable. E.g., Détienne [2002] describes how the reuse of software changed with the introduction of object oriented languages.

Canvas presentations are currently often perceived as novel and this clearly leads to more interest and excitement by users. We have received very positive feedback on our presentation and sketching designs and are very happy that 'users like it', but we have to keep in mind that qualitative feedback on novel user interfaces is often swayed towards the new. And when is comes to engagement with the product, Norman [2004] writes that good emotional feedback is a wonderful thing.

Bibliography

Adobe. Adobe Brackets, 2013. URL `http://brackets.io/`. Last checked: April, 2015.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *Pattern Languages*. Center for Environmental Structure, 1977.

Richard Anderson, Crystal Hoyer, Craig Prince, Jonathan Su, Fred Videon, and Steve Wolfman. Speech, Ink, and Slides: The Interaction of Content Channels. In *MULTIMEDIA '04: Proceedings of the ACM international conference on Multimedia*, pages 796–803, New York, NY, USA, 2004a. ACM.

Richard J. Anderson, Ruth Anderson, Tammy VanDeGrift, Steven Wolfman, and Ken Yasuhara. Promoting Interaction in Large Classes With a Computer-Mediated Feedback System. In *CSCL 03: Proceedings of the International Conference on Computer Supported Collaborative Learning*, pages 119–123, 2003.

Richard J. Anderson, Ruth Anderson, Beth Simon, Steven A. Wolfman, Tammy VanDeGrift, and Ken Yasuhara. Experiences With a Tablet PC Based Lecture Presentation System in Computer Science Courses. *ACM SIGCSE Bulletin*, 36(1): 56–60, 2004b.

Apple. Apple Keynote for Mac, 2003. URL `https://www.apple.com/mac/keynote/`. Last checked: April, 2015.

Apple. iOS Operating System, 2007. URL `https://www.apple.com/ios/`. Last checked: April, 2015.

Apple. Keynote for iCloud, 2015a. URL `https://www.apple.com/iwork-for-icloud/`. Last checked: April, 2015.

Apple. AppleScript, 2015b. URL `https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptX/AppleScriptX.html`. Last checked: April, 2015.

Apple. Xcode Storyboards, 2015c. URL https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/SecondTutorial.html. Last checked: April, 2015.

Apple. Apple Watch, 2015d. URL https://www.apple.com/watch/. Last checked: April, 2015.

Apple. Quartz Composer, 2015e. URL https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/QuartzComposerUserGuide/qc_intro/qc_intro.html. Last checked: April, 2015.

Apple. Xcode Playgrounds, 2015f. URL https://developer.apple.com/xcode/. Last checked: April, 2015.

Aristotle. *Rhetoric, Book III*. self-published, 350 BCE.

Dimitar Asenov and Peter Müller. Envision: A Fast and Flexible Visual Code Editor With Fluid Interactions. In *VL/HCC '14: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 9–12. Citeseer, 2014.

David L. Atkins. Version Sensitive Editing: Change History as a Programming Tool. In *System Configuration Management*, pages 146–157. Springer, 1998.

Sarita Bassil and Rudolf K. Keller. Software Visualization Tools: Survey and Analysis. In *IWPC '01: Proceedings of the International Workshop on Program Comprehension*, pages 7–17. IEEE, 2001.

Patrick Baudisch and Carl Gutwin. Multiblending: Displaying Overlapping Windows Simultaneously Without the Drawbacks of Alpha Blending. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 367–374. ACM, 2004.

Patrick Baudisch and Ruth Rosenholtz. Halo: A Technique for Visualizing Off-Screen Objects. In *CHI '03: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 481–488, New York, NY, USA, 2003. ACM.

Daniel S. Bauer. *The Cognitive Ecology of Dynapad, a Multiscale Workspace for Managing Personal Digital Collections*. PhD thesis, University of California at San Diego, 2006.

Benjamin B. Bederson. PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. In *UIST '01: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 71–80, New York, NY, USA, 2001. ACM.

Benjamin B. Bederson. The Promise of Zoomable User Interfaces. *Behaviour Information Technology*, 30(6):853–866, 2011.

Benjamin B. Bederson and James D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *UIST '94: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 17–26. ACM, 1994.

Benjamin B. Bederson, James D. Hollan, Allison Druin, Jason Stewart, and David Rogers. Local Tools: An Alternative to Tool Palettes. In *UIST '96: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 169–170. ACM, 1996.

Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In *UIST '00: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 171–180, New York, NY, USA, 2000. ACM.

Andrew Begel and Nachiappan Nagappan. Pair Programming: What's in It for Me? In *ESEM '08: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 120–128. ACM, 2008.

Claude Bemtgen. Fly Remote - Reciting Canvas Presentations With an iPad. Bachelor's thesis, RWTH Aachen University, 2012.

Stephen Biesty and Richard Platt. *Stephen Biesty's Incredible Cross-Sections*. Dorling Kindersley, 1992.

Angela Boltman. *Children's Storytelling Technologies*. PhD thesis, University of Maryland at College Park, 2001.

Jan Borchers. HyperSource: Ein Hypermedia-Ansatz für Programmentwicklung und -dokumentation. Diploma thesis, University of Karlsruhe, Germany, 1995.

Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley Sons, Ltd, 2001.

Margaret M. Bradley and Peter J. Lang. Measuring Emotion: The Self-Assessment Manikin and the Semantic Differential. *Journal of Behavior Therapy and Experimental Psychiatry*, 25(1):49–59, 1994.

Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

Andrew Bragdon, Robert DeLine, Ken Hinckley, and Meredith R. Morris. Code Space: Touch + Air Gesture Hybrid Interactions for Supporting Developer Meetings. In *ITS '11: Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, pages 212–221, 2011.

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-Centric Programming: Integrating Web Search Into the Development Environment. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

Stacy Branham, Gene Golovchinsky, Scott Carter, and Jacob T. Biehl. Let's Go From the Whiteboard: Supporting Transitions in Work Through Whiteboard Capture and Reuse. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 75–84. ACM, 2010.

Ann L. Brown. Design Experiments: Theoretical and Methodological Challenges in Creating Complex Interventions in Classroom Settings. *The Journal of the Learning Sciences*, 2(2):141–178, 1992.

Sallyann Bryant, Pablo Romero, and Benedict du Boulay. Pair Programming and the Mysterious Role of the Navigator. *International Journal of Human-Computer Studies*, 66(7):519–529, 2008.

Bill Buxton. A Touching Story: A Personal Perspective on the History of Touch Interfaces Past and Future. *Symposium Digest of Technical Papers*, 41(1):444–448, 2010.

Xiang Cao, Eyal Ofek, and David Vronay. Evaluation of Alternative Presentation Control Techniques. In *CHI '05: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1248–1251, New York, NY, USA, 2005. ACM.

Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., 1983.

Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

Casio. Casio AT-550-7 calculator watch, 1984. URL `http://research.microsoft.com/en-us/um/people/bibuxton/buxtoncollection/detail.aspx?id=227`.

Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J Ko. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 557–566. ACM, 2007.

Jan Chong and Tom Hurlbutt. The Social Dynamics of Pair Programming. In *ICSE '07: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.

Jan Chong and Rosanne Siino. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. In *CSCW '06: Proceedings of the Conference on Computer Supported Cooperative Work*, pages 29–38. ACM, 2006.

Elizabeth F. Churchill and Les Nelson. Tangibly Simple, Architecturally Complex: Evaluating a Tangible Presentation Aid. In *CHI '02: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 750–751, New York, NY, USA, 2002. ACM.

Marcus T. Cicero. *De Oratore*. self published, 55 BC.

Richard E. Clark. Reconsidering Research on Learning from Media. *Review of Educational Research*, 53(4):445–549, 1983.

Richard E. Clark. Media Will Never Influence Learning. *Educational Technology Research and Development*, 42(2):21–29, 1994.

Richard E. Clark. *Learning from Media*. Information Age Publishing, 2001.

Andy Cockburn, Joshua Savage, and Andrew Wallace. Tuning and Testing Scrolling Interfaces That Automatically Zoom. In *CHI '05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 71–80. ACM, 2005.

Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. A Review of Overview+Detail, Zooming, and Focus+Context Interfaces. *ACM Computing Surveys (CSUR)*, 41(1):2–43, 2008.

Christian Corsten. DragonFly: Reviewing Lecture Recordings with Spatial Navigation. Bachelor's thesis, RWTH Aachen University, Aachen, 2009.

Donald A. Cox, Jasdeep S. Chugh, Carl Gutwin, and Saul Greenberg. The Usability of Transparent Overview Layers. In *CHI '98: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 301–302. ACM, 1998.

Russel J. Craig and Joel H. Amernic. PowerPoint Presentation Technology and the Dynamics of Teaching. *Innovative Higher Education*, 31(3):147–160, 2006.

Simon P. Davies, David J. Gilmore, and Thomas R. G. Green. Are Objects That Important? Effects of Expertise and Familiarity on Classification of Object-Oriented Code. *Human–Computer Interaction*, 10(2-3):227–248, 1995.

Robert DeLine and Kael Rowan. Code Canvas: Zooming Towards Better Development Environments. In *ICSE '10: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 207–210. ACM, 2010.

Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *VL/HCC '06: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 11–18. IEEE, 2006.

Robert DeLine, Gina Venolia, and Kael Rowan. Software Development With Code Maps. *Communications of the ACM*, 53(8):48–54, 2010.

Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger Canvas: Industrial Experience With the Code Bubbles Paradigm. In *ICSE '12: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 1064–1073. IEEE Press, 2012.

Françoise Détienne. *Software Design—Cognitive Aspects*. Springer Science & Business Media, 2002.

Alan Dix and Janet E. Finlay. *Human-Computer Interaction*. Prentice Hall, 2004.

Steven M. Drucker, Georg Petschnigg, and Maneesh Agrawala. Comparing and Managing Multiple Versions of Slide Presentations. In *UIST '06: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 47–56. ACM, 2006.

Allison Druin, Jason Stewart, David Proft, Benjamin B. Bederson, and James D. Hollan. KidPad: A Design Collaboration Between Children, Technologists, and Educators. In *CHI '97: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 463–470. ACM, 1997.

Ekwa Duala-Ekoko and Martin P. Robillard. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *ICSE '12: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 266–276. IEEE Press, 2012.

William J. Earnest. *Developing Strategies to Evaluate the Effective Use of Electronic Presentation Software in Communication Education*. PhD thesis, The University of Texas at Austin, 2003.

Eclipse. Eclipse, 2015. URL `https://eclipse.org/`. Last checked: April, 2015.

Darren Edge, Joan Savage, and Koji Yatani. HyperSlides: Dynamic Presentation Prototyping. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 671–680. ACM, 2013.

Darren Edge, Xi Yang, Dan Feng, Bongshin Lee, and Steven Drucker. SlideSpace: Heuristic Design of a Hybrid Presentation Medium. *ACM Transactions on Computer-Human Interaction (TOCHI)*, to appear.

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.

Ruth B. Ekstrom, John W. French, Harry H. Harman, and Diran Dermen. *Manual for Kit of Factor Referenced Cognitive Tests*. Educational Testing Service Princeton, NJ, 1976.

Ruth B. Ekstrom, John W. French, and Harry H. Harman. Cognitive Factors: Their Identification and Replication. *Multivariate Behavioral Research Monographs*, 79(2): 84, 1979.

Lee Ellis and Dan Mathis. College Student Learning From Televised Versus Conventional Classroom Lectures: A Controlled Experiment. *Higher Education*, 14(2): 165–173, 1985.

Douglas C. Engelbart and William K. English. A Research Center for Augmenting Human Intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference*, pages 395–410. ACM, 1968.

Blizzard Entertainment. Diablo II, 2000. URL `https://blizzard.com/diablo2/`. Last checked: April, 2015.

Glen T. Evans. Use of the Semantic Differential Technique to Study Attitudes During Classroom Lessons. *Interchange*, 1(4):96–106, 1970.

David K. Farkas. Understanding and Using PowerPoint. In *Proceedings of the STC Annual Conference on Usability and Information Design*, volume 3, pages 313–320, 2005.

David K. Farkas. A Heuristic for Reasoning About PowerPoint Deck Design. In *IPCC '08: Proceedings of the IEEE International Professional Communication Conference*, pages 1–9, 2008.

David K. Farkas. Managing Three Mediation Effects That Influence PowerPoint Deck Authoring. *Technical Communication*, 56(1):28–38, 2009.

Bernhard Fetz and Klaus Kastberger. *Der Literarische Einfall: Über Das Entstehen Von Texten*. Paul Zsolnay Verlag, 1998.

Adam Fouse, Nadir Weibel, Christine Johnson, and James D Hollan. Reifying Social Movement Trajectories. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2945–2948. ACM, 2013.

Martin Fowler and Jim Highsmith. The Agile Manifesto. *Software Development*, 9 (8):28–35, 2001.

Nico H. Frijda. The Psychologists' Point of View. *Handbook of Emotions*, 2:59–74, 2000.

Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a Programmer's Activity Indicate Knowledge of Code? In *ESEC-FSE '07: Proceedings of the the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 341–350. ACM, 2007.

Alexander Fronk, Armin Bruckhoff, and Michael Kern. 3D Visualisation of Code Structures in Java Software Systems. In *SOFTVIS '06: Proceedings of the International Symposium on Software Visualization*, pages 145–146. ACM, 2006.

George W. Furnas. Generalized Fisheye Views. In *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

George W. Furnas and Xiaolong Zhang. MuSE: a Multiscale Editor. In *UIST '98: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 107–116. ACM, 1998.

George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The Vocabulary Problem in Human-System Communication. *Communications of the ACM*, 30(11):964–971, 1987.

Ryan Gallagher. Operation Auroragold—How the NSA Hacks Cellphone Networks Worldwide, 2014. URL `https://firstlook.org/theintercept/2014/12/04/nsa-auroragold-hack-cellphones/`. Last checked: April, 2015.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

Joanna K. Garner, Michael Alley, Allen F. Gaudelli, and Sarah E. Zappe. Common Use of PowerPoint versus the Assertion–Evidence Structure: A Cognitive Psychology Perspective. *Technical Communication*, 56(4), 2009.

Lance Good. *Zoomable User Interfaces for the Authoring and Delivery of Slide Presentations*. PhD thesis, University of Maryland, 2003.

Lance Good and Ben B. Bederson. CounterPoint: Creating Jazzy Interactive Presentations. Technical report, University of Maryland, 2001.

Lance Good and Benjamin B. Bederson. Zoomable User Interfaces as a Medium for Slide Show Presentations. *Information Visualization*, 1(1):35–49, 2002.

Google. Google Earth, 2001. URL `https://www.google.com/earth/`. Last checked: April, 2015.

Google. Google Maps, 2005. URL `https://maps.google.com/`. Last checked: April, 2015.

Google. Google Documents, 2007. URL `https://www.google.com/docs/about/`. Last checked: April, 2015.

Kreshna Gopal and Karthik Morapakkam. Incorporating Concept Maps in a Slide Presentation Tool for the Classroom Environment. In *ED-MEDIA '02: Proceedings of the World Conference on Educational Multimedia, Hypermedia Telecommunications*. AACE, 2002.

Erin E. Hardin. Technology in Teaching: Presentation Software in the College Classroom: Don't Forget the Instructor. *Teaching of Psychology*, 34(1):53–57, 2007.

Rebecca Harlin and Victoria Brown. Issues in Education: The Power of Powerpoint: Is it in the User or the Program? *Childhood Education*, 83(4):231–233, 2007.

Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. d.note: Revising User Interfaces Through Change Tracking, Annotations, and Alternatives. In *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 493–502. ACM, 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 SIGCHI Conference on Human Factors in Computing Systems*, pages 2207–2210. ACM, 2011.

Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. Software Evolution Comprehension: Replay to the Rescue. In *ICPC'11: Proceedings of the IEEE International Conference on Program Comprehension*, pages 161–170. IEEE, 2011.

Liwei He, Elizabeth Sanocki, Anoop Gupta, and Jonathan Grudin. Comparing Presentation Summaries: Slides vs. Reading vs. Listening. In *CHI '00: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 177–184, New York, NY, USA, 2000. ACM.

Robert A. Heinlein. All You Zombies. *Magazine of Fantasy and Science Fiction*, March 1959.

Thomas Hess. Fly—Expressive and Conveying Planar Presentations. Master's thesis, RWTH Aachen University, Aachen, 2011.

Raymond L. Higgins, Rene R. Alonso, and Mark G. Pendleton. The Validity of Role-Play Assessments of Assertiveness. *Behavior Therapy*, 10(5):655–662, 1979.

Ron R. Hightower, Laura T. Ring, Jonathan I. Helfman, Benjamin B. Bederson, and James D. Hollan. PadPrints: Graphical Multiscale Web Histories. In *UIST '98:*

Proceedings of the ACM Symposium on User Interface Software and Technology, pages 121–122. ACM, 1998.

Ken Hinckley, Koji Yatani, Michel Pahud, Nicole Coddington, Jenny Rodenhouse, Andy Wilson, Hrvoje Benko, and Bill Buxton. Pen + Touch = New Tools. In *UIST '10: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 27–36. ACM, 2010.

David Holman, Predrag Stojadinović, Thorsten Karrer, and Jan Borchers. Fly: An Organic Presentation Tool. In *CHI '06: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 863–868. ACM, 2006.

Neville Holmes. In Defense of PowerPoint. *Computer*, 37(7):100–102, 2004.

Kasper Hornbæk, Benjamin B. Bederson, and Catherine Plaisant. Navigation Patterns and Usability of Zoomable User Interfaces with and without an Overview. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):362–389, 2002.

Richard House, Anneliese Watt, and Julia Williams. Work in Progress - What is PowerPoint? Educating Engineering Students in its Use and Abuse. In *ICSE '05: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 3–15. IEEE, 2005.

Jeff E. Hoyt. Does the Delivery Method Matter?: Comparing Technologically Delivered Distance Education With on-Campus Instruction. Technical report, Utah Valley State College, Department of Institutional Research, 1999.

Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct Manipulation Interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.

Mikkel R. Jakobsen and Kasper Hornbæk. Fisheyes in the Field: Using Method Triangulation to Study the Adoption and Use of a Source Code Visualization. In *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1579–1588. ACM, 2009.

Jeff A. Johnson and Bonnie A. Nardi. Creating Presentation Slides: A Study of User Preferences for Task-Specific Versus Generic Application Software. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(1):38–65, 1996.

Kathy Johnson and Vicki Sharp. Is PowerPoint Crippling Our Students? *Learning and Leading with Technology*, 33(3):6–7, 2005.

Ernest H. Joy and Federico E. Garcia. Measuring Learning Effectiveness: A New Look at No-Significant-Difference Findings. *Journal of Asynchronous Learning Networks*, 4(1):33–39, 2000.

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. IEEE, 2008.

Steve Kaminski. PowerPoint Presentations: The Good, the Bad and the Ugly, 2001. URL `http://www.shkaminski.com/Classes/MNGT5590/powerpoint.htm`.

Thorsten Karrer. *Semantic Navigation in Digital Media*. PhD thesis, RWTH Aachen University, Aachen, 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, New York, NY, USA, 2008. ACM.

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*, Atlanta, USA, 2010.

Guy Kawasaki. The 10-20-30 Rule, 2005. URL `http://guykawasaki.com/the_102030_rule/`. Last checked: April, 2015.

Jeffrey M. Kern, Cindy Miller, and John Eggers. Enhancing the Validity of Role-Play Tests: A Comparison of Three Role-Play Methodologies. *Behavior Therapy*, 14(4):482–492, 1983.

Jens E. Kjeldsen. The Rhetoric of PowerPoint. *International Journal of Media, Technology and Lifelong Learning*, 2(1):1–17, 2006.

Andrew J Ko, Htet Htet Aung, Brad A. Myers, et al. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *ICSE '05: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 126–135. IEEE, 2005.

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

Robert B. Kozma. Learning with Media. *Review of Educational Research*, 61(2):179–211, 1991.

Robert B. Kozma. Will Media Influence Learning? Reframing the Debate. *Educational Technology Research and Development*, 42(2):7–19, 1994.

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

Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. How Live Coding Affects Developers' Coding Behavior. In *VL/HCC '14: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 5–8, 2014.

Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent Layout for Thematic Software Maps. In *WCRE'08: Proceedings of the Working Conference on Reverse Engineering*, pages 209–218. IEEE, 2008.

Adrian Kuhn, David Erni, and Oscar Nierstrasz. Embedding Spatial Software Visualization in the IDE: An Exploratory Study. In *SOFTVIS '10: Proceedings of the International Symposium on Software Visualization*, pages 113–122. ACM, 2010.

Christopher Kurtz. Code Gestalt: From UML Class Diagrams to Software Landscapes. Diploma thesis, RWTH Aachen University, Aachen, 2011a.

Christopher Kurtz. Code Gestalt: A Software Visualization Tool for Human Beings. In *CHI '11: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 929–934, New York, NY, USA, 2011b. ACM Press.

Sandeep K. Kuttal, Anita Sarma, and Gregg Rothermel. On the Benefits of Providing Versioning Support for End Users: An Empirical Study. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(2):1–43, 2014.

Leslie Lamport. LaTeX – A document preparation system, 1984. URL `https://en.wikipedia.org/wiki/LaTeX`. Last checked: April, 2015.

Christian F. J. Lange and Michel R. V. Chaudron. Interactive Views to Improve the Comprehension of UML Models—an Experimental Validation. In *ICPC'07: Proceedings of the IEEE International Conference on Program Comprehension*, pages 221–230. IEEE, 2007.

Joel Lanir, Kellogg S. Booth, and Leah Findlater. Observing Presenters' Use of Visual Aids to Inform the Design of Classroom Presentation Software. In *CHI '08: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 695–704, New York, NY, USA, 2008. ACM.

Thomas D. LaToza and Brad A. Myers. Searching Across Paths. In *SUITE '10: Proceedings of ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, pages 29–32. ACM, 2010a.

Thomas D. LaToza and Brad A. Myers. Developers Ask Reachability Questions. In *ICSE '10: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 185–194. ACM, 2010b.

Thomas D. LaToza and Brad A. Myers. Hard-to-Answer Questions About Code. In *PLATEAU '10: Evaluation and Usability of Programming Languages and Tools*, pages 1–6. ACM, 2010c.

Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *ICSE '06: Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 492–501. ACM, 2006.

Laszlo Laufer, Peter Halacsy, and Adam Somlai-Fischer. Prezi Meeting: Collaboration in a Zoomable Canvas Based Environment. In *CHI '11: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 749–752. ACM, 2011.

Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software*, 20(6):35–39, 2003.

Ying K. Leung and Mark D. Apperley. A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1(2):126–160, 1994.

Yang Li, James A. Landay, Zhiwei Guan, Xiangshi Ren, and Guozhong Dai. Sketching Informal Presentations. In *ICMI '03: Proceedings of the ACM International Conference on Multimodal Interaction*, pages 234–241. ACM, 2003.

Leonhard Lichtschlag. Fly: An Organic Authoring Tool for Presentations. Diploma thesis, RWTH Aachen University, Aachen, 2008.

Leonhard Lichtschlag and Jan Borchers. CodeGraffiti: Communication by Sketching for Pair Programming. In *UIST '10: Adjunct Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 439–440, New York, NY, 2010.

Leonhard Lichtschlag, Thorsten Karrer, and Jan Borchers. Fly: a Tool to Author Planar Presentations. In *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 547–556, Boston, MA, USA, 2009. ACM Press.

Leonhard Lichtschlag, Thomas Hess, Thorsten Karrer, and Jan Borchers. Fly: Studying Recall, Macrostructure Understanding, and User Experience of Canvas Presentations. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1307–1310, 2012a.

Leonhard Lichtschlag, Thomas Hess, Thorsten Karrer, and Jan Borchers. Canvas Presentations in the Wild. In *CHI '12: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 537–540, 2012b.

Leonhard Lichtschlag, Lukas Spychalski, and Jan Borchers. CodeGraffiti: Using Hand-drawn Sketches Connected to Code Bases in Navigation Tasks. In *VL/HCC '14: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 65–68, 2014.

Leonhard Lichtschlag, Philipp Wacker, Martina Ziefle, and Jan Borchers. The Presenter Experience of Canvas Presentations. In *INTERACT 15: Proceedings of the IFIP International Conference on Human-Computer Interaction*, pages 289–297, 2015.

John Lovgren. How to Choose Good Metaphors. *IEEE Software*, 11(3):86–88, 1994.

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. ACM, 1991.

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

Nicolas Mangano, Alex Baker, Mitch Dempsey, Emily Navarro, and André van der Hoek. Software Design Sketching With Calico. In *ASE '10: Proceedings of the International Conference on Automated Software Engineering*, pages 23–32. ACM, 2010.

George R. R. Martin. *A Game of Thrones*. Bantam Dell Publishing Group, 1996.

Kazunori Maruyama, Eijirou Kitsu, Tatsuya Omori, and Shin'ichiro Hayashi. Slicing and Replaying Code Change History. In *ASE '12: Proceedings of the International Conference on Automated Software Engineering*, pages 246–249. IEEE, 2012.

Microsoft. Microsoft Seadragon, 2002. URL `https://en.wikipedia.org/wiki/Seadragon_Software`. Last checked: April, 2015.

Microsoft. pptPlex, 2008. URL `http://www.officelabs.com/projects/pptPlex/`. Last checked: April, 2015.

Microsoft. Microsoft Canvas For OneNote., 2009. URL `https://web.archive.org/web/20100416164157/http://www.officelabs.com/projects/canvasforonenote/Pages/default.aspx`. Last checked: April, 2015.

Microsoft. Microsoft Bing Maps, 2010. URL `https://www.bing.com/maps/`. Last checked: April, 2015.

Microsoft. Debugger Canvas, 2013. URL `https://visualstudiogallery.msdn.microsoft.com/4a979842-b9aa-4adf-bfef-83bd428a0acb`. Last checked: April, 2015.

Microsoft. PowerPoint, 2015. URL `https://products.office.com/en-us/powerpoint`. Last checked: April, 2015.

Tomer Moscovich, Karin Scholz, John F. Hughes, and David H. Salesin. Customizable Presentations. Technical report, Computer Science Dept., Brown University, 2004.

Randall Munroe. Movie Narrative Charts, 2009. URL `https://xkcd.com/657/`. Last checked: April, 2015.

Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006.

Brad A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages Computing*, 1(1):97–123, 1990.

Mathieu Nancel, Julie Wagner, Emmanuel Pietriga, Olivier Chapuis, and Wendy Mackay. Mid-Air Pan-and-Zoom on Wall-Sized Displays. In *CHI '11: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 177–186. ACM, 2011.

Donald A. Norman. *Emotion Design: Why We Love (or Hate) Everyday Things*. Basic Books, 2004.

Donald A. Norman. In Defense of PowerPoint, 2005. URL `http://www.jnd.org/dn.mss/in_defense_of_p.html`. Last checked: April, 2015.

Angela M. O'Donnell, Donald F. Dansereau, and Richard H. Hall. Knowledge Maps as Scaffolds for Cognitive Processing. *Educational Psychology Review*, 14(1):71–86, 2002.

OmniGroup. OmniGraffle, 2015. URL `https://www.omnigroup.com/omnigraffle`. Last checked: April, 2015.

Stephen Oney and Joel Brandt. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2697–2706. ACM, 2012.

Apache OpenOffice. OpenOffice Impress, 2012. URL `https://www.openoffice.org/product/impress.html`. Last checked: April, 2015.

Charles E. Osgood, George J. Suci, and Percy H. Tannenbaum. *The Measurement of Meaning*. University of Illinois Press, 1957.

Yunrim Park and Carlos Jensen. Beyond Pretty Pictures: Examining the Benefits of Code Visualization for Open Source Newcomers. In *VISSOFT '09: IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 3–10. IEEE, 2009.

Ian Parker. Absolute PowerPoint: Can a Software Package Edit Our Thoughts? *The New Yorker*, 2001.

Chris Parnin, Carsten Görg, and Spencer Rugaber. CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In *SOFTVIS '10: Proceedings of the International Symposium on Software Visualization*, pages 15–24. ACM, 2010.

PechaKucha, 2015. URL http://www.pechakucha.org/. Last checked: April, 2015.

Ken Perlin and David Fox. Pad: An Alternative Approach to the Computer Interface. In *SIGGRAPH '93: Proceedings of the Conference on Computer Graphics and Interactive Techniques*, pages 57–64, New York, NY, USA, 1993. ACM.

David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E John, Margaret Burnett, and Rachel Bellamy. Modeling Programmer Navigation: A Head-to-Head Empirical Evaluation of Predictive Models. In *VL/HCC '11: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 109–116. IEEE, 2011.

Bodo Plachta. *Editionswissenschaft, Eine Einführung in Methode und Praxis der Edition neuerer Texte*. Reclam, 1997.

Catherine Plaisant, Jesse Grosjean, and Benjamin B. Bederson. SpaceTree: Design Evolution of a Node Link Tree Browser. In *INFOVIS '02: Proceedings of the IEEE Information Visualization Conference*, pages 57–64, 2002.

Plato. *Gorgias*. self published, around 380 BC.

Beryl Plimmer and Isaac Freeman. A Toolkit Approach to Sketched Diagram Recognition. In *Proceedings of the British HCI Group Annual Conference on People and Computers*, pages 205–213. British Computer Society, 2007.

Washington Post. Kurt Vonnegut Graphed the World's Most Popular Stories, 2015. URL http://wapo.st/1z4pWtd. Last checked: April, 2015.

Christian R. Prause. *Improving the Internal Quality of Software Through Reputation-Based Gamification*. PhD thesis, RWTH Aachen University, 2013.

Prezi. Prezi Presentation Software, 2008. http://www.zuiprezi.com/.

Digestive Pyrotechnics. Predestination: Plot Explained, 2014. URL http://www.digestivepyrotechnics.com/2014/12/predestination-plot-explained.html. Last checked: April, 2015.

Thomas R. Ramage. The "No Significant Difference" Phenomenon: A Literature Review. *e-Journal of Instructional Science and Technology*, 5(1):1–7, 2002.

Ramana Rao and Stuart K. Card. The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus + Context Visualization for Tabular Information. In *CHI '94: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 318–322. ACM, 1994.

Raskin. Raksin—Beyond Desktop, 2015. URL http://www.raskinformac.com/. Last checked: April, 2015.

Garr Reynolds. *Presentation Zen: Simple Ideas on Presentation Design and Delivery*. New Riders, 2011.

George G. Robertson and Jock D. Mackinlay. The Document Lens. In *UIST '93: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 101–108. ACM, 1993.

George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 189–194. ACM, 1991.

Michael D. Robinson and Gerald L. Clore. Belief and Feeling: Evidence for an Accessibility Model of Emotional Self-Report. *Psychological Bulletin*, 128(6):934, 2002.

James A. Russell and Albert Mehrabian. Evidence for a Three-Factor Theory of Emotions. *Journal of Research in Personality*, 11(3):273–294, 1977.

Thomas L. Russell. *The No Significant Difference Phenomenon: As Reported in 355 Research Reports, Summaries and Papers*. North Carolina State University, 1999.

Daniel L. Schacter and Lynn Nadel. Varieties of Spatial Memory: A Problem for Cognitive Neuroscience. *Perspectives on Cognitive Neuroscience*, 1991.

Klaus R. Scherer. What Are Emotions? And How Can They Be Measured? *Social Science Information*, 44(4):695–729, 2005.

Donald A. Schön. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, 1983.

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

Carsten Schwesig, Ivan Poupyrev, and Eijiro Mori. Gummi: A Bendable Computer. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 263–270. ACM, 2004.

Mariam Sensalire and Patrick Ogao. Visualizing Object Oriented Software: Towards a Point of Reference for Developing Tools for Industry. In *VISSOFT '07: IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 26–29. IEEE, 2007.

Francisco Servant and James A. Jones. History Slicing: Assisting Code-Evolution Tasks. In *FSE '12: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 43:1–43:11. ACM, 2012.

Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343. IEEE, 1996.

Tad Simons. Does PowerPoint Make You Stupid? *Presentations*, 26(1):2010, 2004.

Vineet Sinha, David Karger, and Rob Miller. Relo: Helping Users Manage Context During Interactive Exploratory Visualization of Large Codebases. In *VL/HCC '06: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 187–194. IEEE, 2006.

David A. Slykhuis, Eric N. Wiebe, and Len A. Annetta. Eye-Tracking Students' Attention to PowerPoint Photographs in a Science Education Setting. *Journal of Science Education and Technology*, 14(5):509–520, 2005.

Daniel Speicher and Jan Nonnen. Consistent Consideration of Naming Consistency. In *Proceedings of the Workshop on Software-Reengineering*. Citeseer, 2010.

Ryan P. Spicer and Aisling Kelliher. NextSlidePlease: Navigation and Time Management for Hyperpresentations. In *CHI '09: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3883–3888. ACM, 2009.

SpinelessClassics. SpinelessClassics Posters, 2015. URL `www.spinelessclassics.com/`. Last checked: April, 2015.

Lukas Spychalski. Communication Of Source Code Designs Through Sketching. Diploma thesis, RWTH Aachen University, Aachen, 2013.

Jamie Starke, Chris Luce, and Jonathan Sillito. Searching and Skimming: An Exploratory Study. In *ICSM '09: IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 2009.

Margaret-Anne Storey, Lap-Tak Cheng, Janice Singer, Michael Muller, Del Myers, and Jody Ryall. How Programmers Can Turn Comments Into Waypoints for Code Navigation. In *ICSM '07: IEEE International Conference on Software Maintenance*, pages 265–274. IEEE, 2007.

Steven L. Tanimoto. A Perspective on the Evolution of Live Programming. In *LIVE '13: Proceedings of the International Workshop on Live Programming*, pages 31–34. IEEE, 2013.

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

Meinald T. Thielsch and Isabel Perabo. Use and Evaluation of Presentation Software. *Technical Communication*, 59(2):112–123, 2012.

Ardi Tjandra. Code Mixer: A Visual Approach to Code Comprehension and Information Foraging. Master's thesis, RWTH Aachen University, Aachen, 2013.

John R. R. Tolkien. *The Lord of the Rings*. Houghton Mifflin Harcourt, 1954.

Edward Tufte. *The Cognitive Style of PowerPoint*. Graphics Press, Cheshire, Connecticut, USA, 2003.

Barbara Tversky and Masaki Suwa. Thinking With Sketches. *Tools for Innovation*, 1 (9):75–85, 2009.

Tanja Ulmen. Combining live coding and continuous testing. Bachelor's thesis, RWTH Aachen University, Aachen, 2014.

Umbrello. Umbrello Project, 2015. URL `https://umbrello.kde.org/`. Last checked: April, 2015.

J.R. Van Pelt. Lantern Slides and Such. *Quarterly Journal of Speech*, 36(1):44–50, 1950.

Jules Verne. *Around the World in Eighty Days*. Oxford Paperbacks, 1999.

Bret Victor. Up and Down the Ladder of Abstraction, 2011. URL `http://worrydream.com/LadderOfAbstraction/`. Last checked: April, 2015.

Heimito Von Doderer. *Die Strudlhofstiege Oder Melzer Und Die Tiefe Der Jahre*. CH Beck, 1995.

Philipp Wacker. How Does It Feel? Presenter Experience and Evaluation While Using Canvas Presentation Tools. Master's thesis, RWTH Aachen University, Aachen, 2014.

Jagoda Walny, Jonathan Haber, Marian Dörk, Jonathan Sillito, and Sheelagh Carpendale. Follow That Sketch: Lifecycles of Diagrams and Sketches in Software Development. In *VISSOFT '11: IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE, 2011.

David Watson, Lee A. Clark, and Auke Tellegen. Development and Validation of Brief Measures of Positive and Negative Affect: The PANAS Scales. *Journal of Personality and Social Psychology*, 54(6):1063–1070, 1988.

Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *VISSOFT '07: IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE, 2007.

Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. Snipmatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization. In *UIST '12: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 219–228. ACM, 2012.

Wikipedia. WIMP stands for "windows, icons, menus, pointers", a style of interaction using these elements of the user interface., 2015a. URL `https://en.wikipedia.org/wiki/WIMP_(computing)`. Last checked: April, 2015.

Wikipedia. Seam Carving, 2015b. URL `https://en.wikipedia.org/wiki/Seam_carving`. Last checked: April, 2015.

Moritz Wittenhagen. *Temporal Navigation in Hierarchically Structured Media*. PhD thesis, RWTH Aachen University, 2015. to appear.

Joan Wright. Notes from Left Field: Corporate Slide Presentations. *IEEE Computer Graphics and Applications*, 3(4):39–44, 1983.

Ji Soo Yi, Youn ah Kang, John T. Stasko, and Julie A. Jacko. Toward a Deeper Understanding of the Role of Interaction in Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.

YoungSeok Yoon and Brad A. Myers. An Exploratory Study of Backtracking Strategies Used by Developers. In *CHASE '12: Proceedings of the International Workshop on Co-operative and Human Aspects of Software Engineering*, pages 138–144. IEEE Press, 2012.

YoungSeok Yoon and Brad A. Myers. A Longitudinal Study of Programmers' Backtracking. In *VL/HCC '14: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 101–108. IEEE, 2014.

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

Polle T. Zellweger, Jock D. Mackinlay, Lance Good, Mark Stefik, and Patrick Baudisch. City Lights: Contextual Views in Minimal Space. In *CHI '03: Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, pages 838–839, New York, NY, USA, 2003. ACM.

Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. In *VL/HCC '14: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 109–112. IEEE, 2014.

Typeset February 16, 2016