HUMAN-CENTRIC TOOLS FOR NAVIGATING CODE

by

Austin Zachary Henley

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Computer Science

The University of Memphis

August 2018

ACKNOWLEDGMENTS

I would like to thank my advisor, Scott Fleming, for sharing his knowledge of research with me over the past 6 years. It is because of him that I aspire to be a great advisor to my students one day, as he has been for me. I would also like to thank Margaret Burnett for her selfless mentorship and unwavering support. I owe you both so much.

My committee members have made a lasting impression on me and have helped me prepare for the next steps in my career. Emerson Murphy-Hill, thank you for all the advice and feedback you have given me, and thank you for talking to me at conferences when I was too shy to talk to anyone. Santosh Kumar, thank you for encouraging me to get involved with the department—those experiences have already proven to be invaluable. Vasile Rus, thank you for your optimism and guidance, and thank you for giving me my first opportunities to lecture in front of a class.

This journey would not have been possible, or as exciting, without my 5 internships. Thank you to my mentors, Maria Christakis, Andrew Dove, Kivanc Muslu, David Piorkowski, and Ali Syed for investing so much time and effort into working with me. I am also grateful for the great friends that I met during those summers: Christopher Chambers, Michael Desmond, Andrew Ferraiuolo, Beatrice Moissinac, and Ryan Olivieri.

Through research I have had the privilege to meet many wonderful colleagues. Thank you for the memories and companionship Titus Barik, Christian Bird, Denae Ford, Kyle Harms, Brad Myers, Michelle Ichinco, Brittany Johnson, Sandeep Kuttal, Michael Lee, Rahul Pandita, Christopher Parnin, Sruti Ragavan, Paige Rodeghero, Anita Sarma, Christopher Scaffidi, and Justin Smith. It is always a joy to reunite with each of you at conferences. I also met many great colleagues in Memphis that made living here much more enjoyable. Thank you for that Jason Braasch, Kathryn Bridson, Timothy Hnat, Maria Luong, Alka Singh, Vinay Sistla, Alistair Windsor, Hanwen Yu, and Kazi Zaman.

Words cannot express my gratitude for my close friends and family. Thank you Jeremy Boyd, Anthony Hines, and Joseph Lormand, and David Piorkowski for your company. I am deeply indebted to my parents for teaching me to work hard and do the right thing. It is through their support that I have made it this far. Thank you to my exuberant nieces and nephews for always making life interesting. Most certainly, I would like to thank Laken, for wholeheartedly supporting me in this endeavor and in what comes next.

ABSTRACT

Henley, Austin Z. Ph.D. Human-Centric Tools for Navigating Code. Major Professor: Dr. Scott D. Fleming

All software failures are fundamentally the fault of humans—the software's design was flawed. The high cost of such failures ultimately results in developers having to design, implement, and test fixes, which all take considerable time and effort, and may result in more failures. As developers work on software maintenance tasks, they must navigate enormous codebases that may comprise millions of lines of code organized across thousands of modules. However, navigating code carries with it a plethora of problems for developers. In the hopes of addressing these navigation barriers, modern code editor and development environments provide a variety of features to aid in navigation; however, they are not without their limitations. Code navigations take many forms, and in this work I focus on three key types of code navigation in modern software development: navigating the working set, navigating among versions of code, and navigating the code structure. To address the challenges of navigating code, I designed three novel software development tools, one to enhance each type of navigation. First, I designed and implemented Patchworks, a code editor interface to support developers in navigating the working set. Patchworks aims to make these more efficient by providing a fixed grid of open code fragments that developers can quickly navigate. Second, I designed and implemented Yestercode, a code editor extension to support navigating among versions of code. Yestercode does so by providing a comparison view of the current code and a previous version of the same code. Third, I designed and implemented Wandercode, a code editor extension to enable developers to efficiently navigate the structure of their code. Wandercode aims to do so by providing a visualization of the code's call graph overlayed on the code editor. My approach to designing these tools for more efficient code navigation was a *human-centric* one—that is, based on the needs of actual developers performing real software development tasks. Through user study evaluations, I found that these tools significantly improved developer productivity by reducing developers' time spent navigating and mental effort during software maintenance tasks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

You are in a maze of twisty little passages, all alike.

—William Crowther

All software failures are fundamentally the fault of humans—the software's design was flawed. These failures were estimated to have cost companies worldwide over $1.1 trillion dollars in 2016 and impacted over 4.4 billion users[1]. The high cost of such failures ultimately results in developers having to design, implement, and test fixes, which all take considerable time and effort, and may result in more failures. This is of paramount concern for both code quality and developer productivity. These complex challenges in software maintenance indicate tremendous opportunity to better support developers by enabling them to produce better code with less bugs in less time with less mental burden.

1.1 Problem: Code Navigation

As developers work on software maintenance tasks, they must navigate enormous codebases that may comprise millions of lines of code organized across thousands of modules. A key reason these developers navigate code is to seek information necessary to complete their tasks. In fact, developers have been observed spending substantial amounts

[1] https://www.tricentis.com/resource-assets/software-fail-watch-2016/

of time navigating code during development tasks. One study found that developers engaged in software maintenance tasks spent 35% of their time on the mechanics of navigating [58]. Another study found that developers engaged in debugging spent approximately 50% of their time seeking information, which involves a considerable amount of navigation [98].

However, navigating code carries with it a plethora of problems for developers. A recent study found that 50% of navigations yielded less information than developers expected, and 40% of navigations required more effort than predicted [96]. Studies have also observed developers inspecting irrelevant code [58] or getting "lost" in the code [24, 25]. From all of these findings, it is evident that developers are facing substantial barriers in efficiently navigating to code they need to successfully complete their tasks.

In the hopes of addressing these navigation barriers, modern code editor and development environments provide a variety of features to aid in navigation; however, they are not without their limitations. For example, most popular code editors provide scrolling as a means of navigating within files and tabs as a means to navigate between open files. However, developers have been observed losing their tabs [87, 106] or even closing all of them once too many are open despite still needing them [58]. Scrolling is also problematic since developers may accidentally scroll passed the location they were looking for without even realizing it. Although search features are ubiquitous in code editors, they require the developer to recall a keyword or phrase to search for, which is difficult and prone to memory failure [29, 101]. Moreover, the search results are often time consuming to process [93, 96] and often lead to irrelevant code [58, 96].

Code navigations take many forms, and in this work I focus on three key types of code navigation in modern software development: navigating the working set, navigating among versions of code, and navigating the code structure. First, developers repeatedly navigate within their *working set*—that is, the code fragments that are relevant to their task [58]. This is perhaps the most common form of navigation considering that studies found 40–60% of navigations were to previously visited code [33, 93, 94]. Second, in order for developers to understand their working sets, they may need to navigate to previous versions of code. However, there are very few tools to navigate among versions of code efficiently, and they often resort to manually reverting to a previous version of code [123]. Third, developers need to navigate the code structure to better understand the structural relationships of code, which is a necessity in order to perform maintenance tasks successfully [65]. Providing efficient means of navigating code structure has great potential in improving developer productivity [2, 33].

1.2 Approach

To address the challenges of navigating code, I designed three novel software development tools, one to enhance each type of navigation. To support developers in navigating the working set, I designed and implemented Patchworks, a code editor interface. Patchworks aims to make these more efficient by providing a fixed grid of open code fragments that developers can quickly navigate. To support navigating among versions of code, I designed and implemented Yestercode, a code editor extension. It does so by providing a comparison view of the current code and a previous version of the same code. To enable developers to efficiently navigate the structure of their code, I designed and implemented Wandercode, a code editor extension. Wandercode aims to do so by providing a visualization of the code's call graph overlayed on the code editor.

My approach to designing these tools for more efficient code navigation was a *human-centric* one—that is, based on the needs of actual developers performing real software development tasks. More specifically, my approach comprises three high-level steps: First, I base my tool designs on findings from empirical studies, often using multiple methodologies, as a form of methodological triangulation to validate the observations. Second, I apply theories from Human-Computer Interaction and Cognitive Science to my designs in an effort to ground them with existing evidence. Third, I evaluate each design by comparing it to the state of the practice with an empirical study. It was through this approach that I addressed my thesis statement:

> *Tools that support developers in efficiently navigating the working set, navigating previous versions of code, and navigating the code structure can significantly improve developers' productivity by reducing time spent navigating and mental effort during software maintenance tasks.*

1.3 Contributions

The contributions of this dissertation are as follows:

- A novel tool design, Patchworks, to support developers in efficiently navigating the working set.
- Two prototype implementations of Patchworks for Java and LabVIEW.
- Evidence from a simulation study that found simulated users could navigate more efficiently with Patchworks than Eclipse, and also uncovered potential improvements to Eclipse.
- Evidence from a preliminary user study that found participants using Patchworks navigate faster, made fewer navigation mistakes, and spent less time arranging code.
- Evidence from a industrial user study that found participants using Patchworks navigated more efficiently.
- A cohesive set of design principles for navigation affordances provided by code editors.
- Evidence from formative studies that indicate that developers often seek information from previous versions of code, but face challenges in doing so efficiently.
- A novel tool design, Yestercode, to support developers in efficiently navigating among different versions of code.

- A prototype implementation of Yestercode for LabVIEW.
- Evidence from a user study evaluation that found participants using Yestercode resulted in fewer bugs and reduced cognitive load during code-change tasks.
- A novel tool design, Wandercode, to support developers in efficiently navigating the code structure.
- A prototype implementation of Wandercode for Atom.
- Evidence from a user study evaluation that found developers using Wandercode resulted in faster task completion and reduced cognitive load during tasks to understand the structural relationships of code.

1.4 Outline

This dissertation is structured as follows. Chapter 2 provides a summary of background information and related work for navigating code. Chapter 3 presents Patchworks, a tool for navigating within working sets, along with the findings from three evaluations. Chapter 4 presents Yestercode, a tool for navigating among versions of code, along with the findings from two formative studies and an evaluation. Chapter 5 presents Wandercode, a tool for navigating the code structure, along with the findings from an evaluation. Chapter 6 contains discussion on future work. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background & Related Work

As people are walking all the time, in the same spot, a path appears.

—Lu Xun

2.1 Human-Centric Approaches to Developer Tools

Other researchers have also taken a human-centric approach to designing better software development tools. In fact, the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) publishes papers each year doing just that (see the 2017 proceedings for numerous examples [42]). To further motivate such an approach, Walenstein made the argument that tool research needs to progress from craft to evidence-based engineering in order for the field to advance, and he did so by studying the lack of cognitive support in existing tools [115]. He has not been alone in this endeavor; there have been numerous lines of work that have had success in focusing on the human aspects of software development tools that support a variety of different programming tasks. These have included tools to aid in developers recovering from interruptions [88, 86, 89, 85], to answer questions about program behavior [59, 60, 57], to answer reachability questions [66, 64], to integrate code examples from the web into the development environment [13, 12], and to backtrack to previous versions of code [122, 124, 121, 120]. A particularly relevant example is Murphy-Hill's dissertation titled *Programmer Friendly Refactoring Tools* [74]. It presents empirical findings on problems that developers have with refactoring tools, novel refactoring tools to overcome these problems, and design principles such that future tools can avoid making the same mistakes. Furthermore, there have been researchers that have studied developer tools from a more theoretical perspective to provide a foundation for future tools (e.g., [3, 49, 92]). From examining all of these works, it is promising that such an approach that focuses on human aspects can also be used to improve tools for navigating code.

2.2 Code Navigation Behaviors

A reoccurring trend in empirical studies of developers is that they navigate a lot during software maintenance tasks. One study of developers modifying an application found that they navigated, on average, 5 times per minute [33], while another study of developers debugging found that they navigated 2 times per minute [94]. Not only are navigations frequent, but they also take up a considerable amount of developers' time. In one study, developers spent about 35% of their time on the mechanics of navigating [58]. Another study found that developers spent roughly 50% of their time seeking information in the code [98].

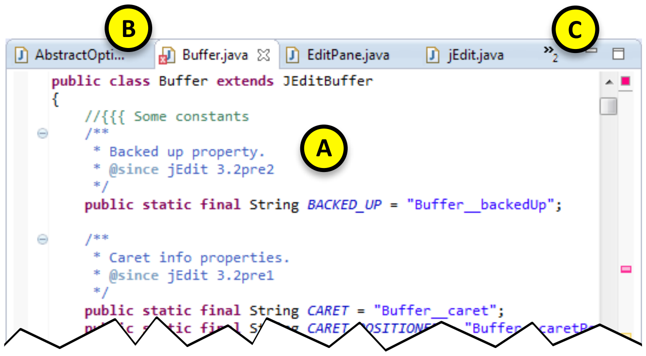
An important reason for all of these navigations is to relate information in one's working set. Ko et al.'s model of program understanding states that relating pieces of code to one another is a fundamental process for comprehension. [62]. Similarly, a series of studies on developers found that understanding the relationships between different pieces of code is necessary to make code changes [65]. Given this need for relating code, developers often navigate to code they have recently visited. In fact, three different studies of developers found that 40–60% of navigations were to already visited code [33, 93, 94].

Researchers have had success in modeling and predicting developers' navigation behavior by applying Information Foraging Theory [99] (IFT) to the domain of software engineering [38, 67, 69, 68, 95, 93, 98, 94, 96, 97, 109, 102]. In this context, IFT describes how developers seek information within an information environment that consists of information patches (e.g., code fragments), and in doing so, developers attempt to maximize the amount of information gained while minimizing the effort to obtain the information. Two notable studies of developers debugging have uncovered the types of information that developers seek [93] and the expectations of what developers think they will find while navigating code [96]. Moreover, an IFT perspective has lead to implications for tools [31] and a public Wiki to enable developers in applying design patterns [78].

2.3 Affordances for Navigating Code

At present, file-based editors are the dominant tool paradigm that developers use to create, modify, and navigate source code, and the empirical evidence above is mostly based on developers using such editors. Most modern development environments, including Eclipse, NetBeans, IntelliJ, and Visual Studio, employ file-based editors. A key characteristic of file-based editors is that for reading and editing, they present the code in units by file. For example, Fig. 1 shows an Eclipse editor with the contents of one file (*Buffer.java*) displayed. To open a file, environments typically provide features for navigating the file system and for exploring the hierarchy of code modules (e.g., Java packages, classes, and methods).

After opening code, developers can navigate the code using a few core types of affordances for navigation that are provided by code editors. To navigate among code fragments (such as methods) within a file, the main capability that file-based editors provide is vertical scrolling. As a developer opens multiple documents, they can switch between them using tabs that the user can click or cycle through using keyboard shortcuts



```
public class Buffer extends JEditBuffer
{
    //{{{ Some constants
    /**
     * Backed up property.
     * @since jEdit 3.2pre2
     */
    public static final String BACKED_UP = "Buffer__backedUp";

    /**
     * Caret info properties.
     * @since jEdit 3.2pre1
     */
    public static final String CARET = "Buffer__caret";
```

Figure 1: Eclipse's file-based editor, including (A) a vertically scrollable file, (B) visible tabs, and (C) two elided tabs.

(Fig. 1B). Code editors also provide hyperlinks based on structural properties that navigate the developer to the location. For example, in Eclipse a developer can right click a function call and select *Open Declaration* to navigate to the function definition. Lastly, there are numerous search capabilities that are provided. Fig. 2 shows Eclipse's search dialog with a variety of string search and regular expression search features. It is also common for code editors to support structural search, such as *Find All References*, which will find every reference to a specific variable, method, or class.

Given the considerable time that developers spend navigating in Eclipse [58], several qualities of traditional file-based editors may hinder efficient navigation. First, file and method listings (i.e., Visual Studio's Solution Explorer and Eclipse's Outline view) can take substantial time to look through, and even more so if the files are organized into folders. Additionally, they provide very few cues to the developer of what the file contains other than its name.

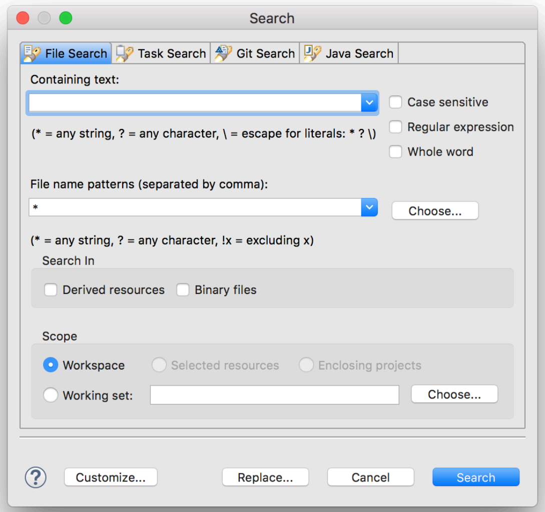
Second, because developers' working sets may contain only a subset of the methods in a file, navigating between two such methods may involve scrolling back and forth over irrelevant code. In such cases, the developer must take care not to miss the method he/she seeks as the code scrolls by. Additionally, scrolling through text has been found to disrupt a person's spatial memory [80, 81].

Third, because a developer's working set may contain methods from a variety of files, navigating between such methods may involve a combination of scrolling and tab switching. However, if the developer has too many tabs open, editors will typically begin eliding tabs, as in Fig. 1C. Moreover, revealing the elided tabs and choosing one involves additional clicking, which takes time.

Fourth, if a developer wants to navigate back and forth between two methods, it stands to reason that having both methods on screen at the same time (i.e., *juxtaposing* the methods) would be ideal; however, traditional file-based editors lack effective features for juxtaposing. For example, although Eclipse provides features for splitting the editor view so that two files can be viewed at once, developers rarely use these features [11, 62]. Moreover, prior research suggests that developers actually want to juxtapose code [9], but find doing so in file-based editors inconvenient [11, 62].

Fifth, if a developer has an idea of what to look for in the code, they may use a text search or a structural search. Performing a search is limited since it requires the developer to know what to search for, which may be the reason why these navigations account for a relatively small number of developers navigations overall [35, 58, 98]. Another problem with text searches is caused by the "vocabulary problem" [96] in source code since English terms used for code identifiers vary greatly (e.g., remove, delete, destroy, and erase are all synonyms), which increases the number of false negatives. Furthermore, the search results are time consuming to process [93], and often lead to irrelevant code [58]. To overcome this, a developer may perform a structural navigation, such as *Go To Definition*. However, such features provide weak cues as to where the navigation will lead so to use them efficiently may require the developer to already have sufficient knowledge of the program's structure.

To overcome these shortcomings, researchers have proposed tools that augment and extend these basic navigation affordances. A number of tools have extended search features by applying various techniques (e.g., Dora [45], Sando [105], ComoGen [53], and


Search

File Search | Task Search | Git Search | Java Search

Containing text:

(* = any string, ? = any character, \ = escape for literals: * ? \)

☐ Case sensitive

☐ Regular expression

☐ Whole word

File name patterns (separated by comma):

*

(* = any string, ? = any character, !x = excluding x)

Search In

☐ Derived resources ☐ Binary files

Scope

[✓ Workspace] Selected resources Enclosing projects

Working set:

Figure 2: Eclipse's search dialog, illustrating a portion of the text and regular expression search features provided.

JQuery [48]). For example, Dora utilizes natural language and lexical analysis techniques to make search more effective [45]. Code Thumbnails takes a different approach by augmenting the scrollbar with a thumbnail-like view of the source code to better support navigation within a document [23]. Flower adds hyperlinks to references in the editor that navigate the developer to other locations based on the currently selected code element [107]. However, none of these approaches have superceded the file-based editor paradigm, and developers still commonly use scrolling and tabbing features to navigate between fragments of code, and thus, experience the inefficiencies discussed above.

2.4 Human Memory of Code Navigation

There are a variety of human memory types that are relevant for navigating code. Most notably is the information that a developer is currently thinking about, contained in their *working memory* [71]. For a software maintenance task, this can be any information related to the task, such as a hypothesis about the code or even the name of the variable that the developer is looking for. While researchers have not came to a consensus on the capacity of working memory, there is evidence that humans can keep 3–5 things in their working memory at a time [21] (previously thought to be 5–9 things [71]). This capacity may seem extremely limited since developers need to retain where they are navigating, what they are looking for, where they will navigate next, where they navigated from, etc. In an attempt to offset this issue, developers may offload their memory by writing notes on paper or in a text file on their computer as they work. If developers don't utilize such a strategy, they may be faced with a memory failure when the information related to their task outgrows their working memory capacity [91].

Associative memory retains links between pieces of information [91]. An example of this for a developer is associating function names with the code that defines them. In terms of tool support, code editors attempt to leverage these associations in a variery of ways. For instance, a code editor's document tabs have labels that a developer could associate with the corresponding code. However, it may be difficult for a developer to do so since tab labels do not express much information, and the code document may contain many thousand lines of code. Tools could provide more identifiable information such that developers could build stronger associations. For example, code editors could augment tabs with a descriptive image that would allow for developers to associate the code document with.

Memories from past events are stored in our *episodic memory* [101]. This allows a developer to recall their past, such as what code element they were looking at before they went to lunch. Episodic memory is of importance while navigating since navigating can be represented as a sequence of events, and developers may need to recall a past navigation. To support this memory, tools may provide history-related features, such as a listing of recent navigations, or a timeline of code changes (e.g., Azurite [124]).

Spatial memory retains links about an object's spatial orientation and environment. In a code editor, developers may remember the order of their tabs or be able to remember that they function they need is the fourth function in the file. However, this relationship is not reliable since tab layouts may change (e.g., by closing a tab or opening another). Code

Thumbnails [23] and Code Bubbles [11] successfully leveraged developers' spatial memory to enhance navigation. To do so, Code Thumbnails provides a zoomed out view of the text that is unreadable next to the scrollbar, but is still recognizable by the shape of the text. Code Bubbles takes advantage of spatial memory by allowing the user to arrange their open code fragments on a large 2D canvas.

Recognition and *recall* are two processes of remembering information. Given a cue, recognition is the process of retrieving information from memory that has been seen before, and is generally considered to be quite efficient [29, 101]. Recall is the process of retrieving information without a cue, which has a much higher chance of failing (i.e., a person forgetting or remembering incorrectly) and requires substantial effort [29, 101]. In the context of navigating code, this means that anytime a developer needs to recall a piece of information (e.g., a method name), their memory may fail. For example, Eclipse's Outline View supports recognition by providing a list of methods in the current file, but to perform a text search for a specific method, developers need to recall the method name.

2.5 Visual Dataflow Languages

Visual dataflow programming languages are receiving renewed attention for industrial applications. At their core, these languages represent code using graphical box-and-wire diagrams, where boxes denote functions and wires denote the passing of values between functions. Although textual languages, such as Java and Python, have tended to dominate mainstream programming, a few visual dataflow languages have been able to carve out successful niches—for example, LabVIEW[1] has been used in the domain of science and engineering applications for over 30 years. The benefits of visual dataflow languages have been well documented: easier to learn [8, 15], improved liveness or immediate feedback [110], and more informative notations [37, 117]. A recent surge of visual dataflow languages have aimed to leverage these benefits for a variety of purposes:

[1] `http://www.ni.com/labview/`

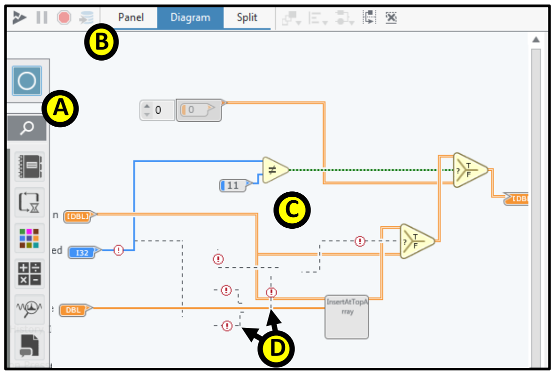


Figure 3: LabVIEW block-diagram editor. Editors have a palette (A), along with debugging and other controls (B). The pictured editor has open a block diagram (C) that is under construction, with several broken wires (e.g., D).

Google/RelativeWave's Form tool for prototyping phone apps, Filter Forge for producing image filters, RapidMiner for data analysis, and Microsoft Robotics Developer Studio for programming robots.

Instead of textual source code, visual dataflow languages are characterized by programs being made up of boxes (functions) and wires (values), as illustrated in Fig. 3-C. The execution of the program follows the dataflow via the wires. Although visual dataflow languages have a radically different syntax than textual languages, such as Java, they still provide many of the same foundational features, such as modularity.

In particular, we focus on LabVIEW, a commercial visual dataflow programming environment that is one of the most widely used visual programming languages to date [118]. In LabVIEW, programs are composed of modules, called *Virtual Instruments* (VIs). Fig. 3-C illustrates the code for a VI, represented as a *block diagram*. This VI has four inputs (box icons along the left side of the diagram) and one output (box icon at far right). It performs some conditional logic on its input (yellow triangle icons), and calls another VI (gray box; similar to subroutine). This VI is under construction, and contains several broken wires that have one or more disconnected ends and are denoted as dashed lines with red exclamation icons (Fig. 3-D).

Chapter 3

Navigating within Working Sets

Twice and thrice over, as they say, good
is it to repeat and review what is good.

—Plato

There is an apparent gap in existing tools for navigating code and the way in which developers seek information. Modern code editors provide a plethora of features to enable navigation, yet developers are still spending an inordinate amount of time navigating [58] and still not getting the information they want [96]. We argue that there are fundamental flaws in how file-based code editors are designed and the features they provide for navigating code.

To directly address the problems of file-based editors, researchers have proposed a new paradigm of *canvas-based code editors* (e.g., [11, 18, 26]). These editors aim to improve developer navigation efficiency by enabling developers to manipulate finer-grain fragments of code, as opposed to whole files, and by enabling them to arrange those fragments on a 2D canvas. Preliminary empirical evaluations of these canvas-based editors have been generally favorable: for example, one study found that developers using Code Bubbles made, on average, nearly half the navigation actions per minute as

Portions of this chapter appear in [39, 41, 43].

developers using the file-based editor, Eclipse [11]. However, opportunities to improve on the canvas-based editor concept may remain.

In particular, the 2D canvas idiom may be the source of several types of navigation inefficiency. Developers seeking code fragments on a large 2D canvas, may tend to make navigation mistakes (i.e., navigating in the wrong direction) because of the large space of possible directions to navigate. Moreover, arrangements of fragments on the canvas may result in on-screen clutter, increasingly the likelihood that a developer will fail to visually locate a fragment he/she seeks. Additionally, the 2D canvas idiom may lead developers to spend extra time and effort on arranging fragments; however, such time may not be well spent if the developer is not far enough along in the sensemaking process to choose effective arrangements, or if arrangements quickly become obsolete because the developer's information needs change rapidly (as recent research suggests [93]).

To address these concerns about canvas-based editors, we propose a new editor concept, *Patchworks*. Patchworks seeks to maintain the efficiency gains of canvas-based editors, while reducing the number of navigation mistakes that developers make and the time they spend arranging code. To achieve these goals, Patchworks employs two new interface idioms: the *patch grid* and *ribbon*. These features enable developers to edit and juxtapose code at a finer level of granularity than file-based editors, while leaving less room for navigation mistakes than canvas-based editors, and disallowing many of the time-consuming window management activities encouraged by canvas-based editors.

3.1 Related Work: Canvas-Based Editors

To address the problems with file-based editors, researchers have proposed a new paradigm of canvas-based code editors that enable developers to work with more fine-grain code fragments and to arrange those fragments on a 2D canvas. This approach mitigates two key problems with file-based editors: First, it enables the developer to focus only on relevant code fragments, rather than having to negotiate all the fragments in a file.

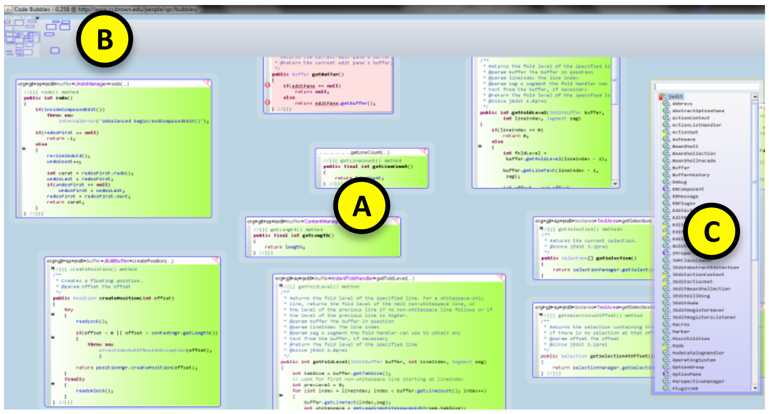


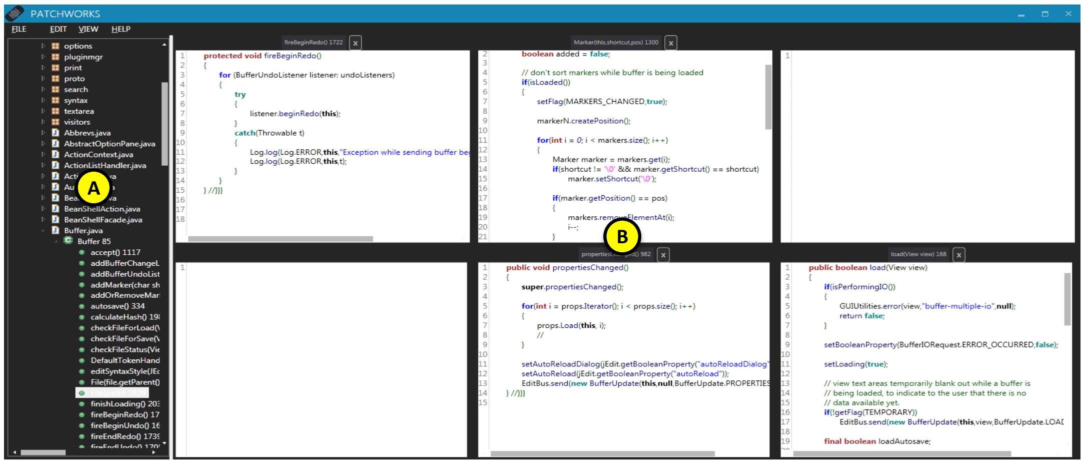
Figure 4: The Code Bubbles editor, including (A) a number of bubbles, (B) a workspace bar, and (C) a package explorer.

Second, juxtaposing code fragments on a canvas is considerably more efficient than in file-based editors. Several tools have been proposed that follow this paradigm: the Self programming system [108], JASPER [18], Code Canvas [26], and Code Bubbles [10, 11]. In the remainder of this chapter, we focus on Code Bubbles as a representative example of the canvas-based paradigm.

Fig. 4 depicts the Code Bubbles editor, which enables the developer to view and edit code fragments at the granularity of methods (as well as whole files). It presents the fragments in resizeable "bubbles" (the green and red boxes in Fig. 4). The developer can arrange the bubbles on a large 2D canvas that extends well beyond the visible area on screen. Code Bubbles also has features for grouping bubbles and for automatically drawing edges between bubbles that represent declaration relationships (neither of which are shown in the figure). Additionally, Code Bubbles has a workspace bar (Fig. 4B) that shows a bird's eye view of the canvas.

Although the canvas-based paradigm may address some problems of file-based editors, it may also introduce new ones. There are several potential problems with arranging fragments on the canvas. First, there are many more possible places to place a fragment on a 2D canvas than, for example, there are to put a tab in the tab list; however, at the time that a developer is opening code, he/she may not have sufficient understanding of the program or task to make a quick, effective decision about where to place the fragment. For example, a recent study showed that people required a rich mental model of a topic before they could effectively structure (i.e., arrange/organize) information on the topic [56]. Second, if developers' goals are rapidly changing (as per reactive information foraging [69, 93]), arrangements may be quickly rendered obsolete and not useful. Third, because it is common for fragments in canvas-based editors to automatically respace themselves, inserting a fragment near a group of fragments can cause those in the group to move in a cascading fashion. If the developer does not see how the fragments have moved, it could undermine his/her ability to leverage spatial memory for navigation.

There may also be several issues with navigating about the 2D canvas. First, if there are too many fragments on screen, the developer may have to resort to visual searching, which is relatively slower than spatial memory [44]. In fact, the creators of Code Bubbles specifically claim that it can show more code on screen than Eclipse, saying that it can fit, on average, 20 randomly selected methods from the JEdit project [10]. But if developers place that much code on screen, the fragments may begin to all look alike, depriving the developer of landmarks to spatially orient him/herself. Moreover, the developer may be more likely to miss a sought method. Second, if the developer must navigate to off-screen fragments, the large space of possible directions in which to navigate may increase the chance of navigation mistakes. Consider how this situation compares to a vertically scrolling file-based editor where the developer has only two scrolling choices: up or down. However, in a 2D canvas, the developer can pan in 360 degrees—considerably more room for navigational errors. Third, while navigating in the 2D canvas, the developer may have difficulty identifying landmarks and orienting him/herself, which has been shown to be important for spatial memory [22], because of the potentially large number of similar-looking fragments.



```
protected void fireBeginRedo()
{
	for (BufferUndoListener listener: undoListeners)
	{
		try
		{
			listener.beginRedo(this);
		}
		catch(Throwable t)
		{
			Log.log(Log.ERROR,this,"Exception while sending buffer beg
			Log.log(Log.ERROR,this,t);
		}
	}
} //}}}
```

```
	boolean added = false;

	// don't sort markers while buffer is being loaded
	if(isLoaded())
	{
		setFlag(MARKERS_CHANGED,true);

		markerN.createPosition();

		for(int i = 0; i < markers.size(); i++)
		{
			Marker marker = markers.get(i);
			if(shortcut != '\0' && marker.getShortcut() == shortcut)
				marker.setShortcut('\0');

			if(marker.getPosition() == pos)
			{
				markers.removeElementAt(i);
				i--;
			}
```

```
public void propertiesChanged()
{
	super.propertiesChanged();

	for(int i = props.Iterator(); i < props.size(); i++)
	{
		props.Load(this, i);
		//
	}

	setAutoReloadDialog(jEdit.getBooleanProperty("autoReloadDialog"
	setAutoReload(jEdit.getBooleanProperty("autoReload"));
	EditBus.send(new BufferUpdate(this,null,BufferUpdate.PROPERTIES
} //}}}
```

```
public boolean load(View view)
{
	if(isPerformingIO())
	{
		GUIUtilities.error(view,"buffer-multiple-io",null);
		return false;
	}

	setBooleanProperty(BufferIORequest.ERROR_OCCURRED,false);

	setLoading(true);

	// view text areas temporarily blank out while a buffer is
	// being loaded, to indicate to the user that there is no
	// data available yet.
	if(!getFlag(TEMPORARY))
		EditBus.send(new BufferUpdate(this,view,BufferUpdate.LOAD

	final boolean loadAutosave;
```

Figure 5: The Patchworks editor, including (A) a package explorer and (B) a 3×2 patch grid. Four of the patches contain code fragments and two are empty.

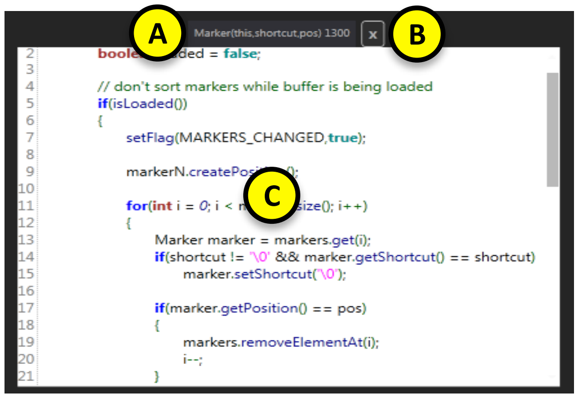
3.2 Tool Design: Patchworks

To overcome apparent issues with both traditional file-based editors and the canvas-based editors, we propose the Patchworks tool concept. Like the canvas-based editors, Patchworks allows the developer to view and edit fine-grain code fragments (methods, in particular). However, Patchworks abandons the canvas idiom, and introduces instead two new idioms: the *patch grid* and the *ribbon*.

3.2.1 The Patch Grid

Fig. 5 depicts our Patchworks prototype. The main part of the editor consists of a 3×2 grid of *patches* (Fig. 5B). Each patch is an editor that can hold code fragments at a variety of granularities, including method, class, and file. For our initial prototype, we tentatively chose for the grid to have 6 patches, a decision based on working memory capacity (7±2) [71] as well as an attempt to optimize the use of screen space while avoiding visual clutter; however, we defer to future work the question of what the optimal number of patches might be.

A key design decision was to make the grid of patches fixed: the developer can neither adjust the size of a patch nor the number of patches. In contrast to canvas-based editors,



```
boole[illegible]ded = false;

// don't sort markers while buffer is being loaded
if(isLoaded())
{
    setFlag(MARKERS_CHANGED,true);

    markerN.createPos[illegible]();

    for(int i = 0; i < m[illegible]size(); i++)
    {
        Marker marker = markers.get(i);
        if(shortcut != '\0' && marker.getShortcut() == shortcut)
            marker.setShortcut('\0');

        if(marker.getPosition() == pos)
        {
            markers.removeElementAt(i);
            i--;
        }
```

Figure 6: A patch, including (A) a title bar, (B) a close button, and (C) a code-fragment editor.

such as Code Bubbles, we intentionally restrict the ways in which developers can arrange their code to discourage wasting effort on creating arrangements that provide little benefit. For example, Plumlee and Ware [100] argue that having users manage windows adds considerable complexity due to the time and attention of sizing and positioning them. Thus, Patchworks aims to alleviate this complexity by disallowing such window management. Moreover, because the patches are spatially in six fixed positions, the developer may be less likely to miss fragments he/she is seeking and be able to scan more efficiently. Also, the patch titles always being in the same screen locations may be beneficial since it has been shown that labels enhance spatial memory [50].

Code fragments can be moved between patches in several ways. A code fragment can be opened in a patch by dragging an element from the package explorer (Fig. 5A) into the patch. The Patchworks package explorer is essentially equivalent to that of Eclipse and Code Bubbles. Fragments may be moved between patches by dragging from one patch to another. If there is an existing fragment in the destination patch, the contents of the patches are swapped.

Fig. 6 depicts the features of an individual patch. A patch consists of a title bar with the name of the fragment (Fig. 6A), a close button (Fig. 6B), and a code-fragment editor (Fig. 6C). The fragment name is based on the type of the fragment, so a method fragment

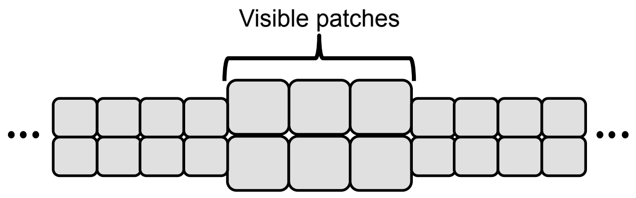


Figure 7: The conceptual ribbon of patches.

gets the name of the method plus its parameters, a class gets the name of the class, etc. The fragment editor has common code-editor features, such as syntax highlighting and code folding. Clicking the close button removes the fragment from the patch, leaving the patch empty.

3.2.2 The Ribbon

Although the patch grid contains only six visible patches at a time, conceptually, the six patches constitute a view into a never-ending ribbon of patches. Fig. 7 depicts the ribbon concept. The visible patch grid can be shifted left or right along the ribbon via keyboard shortcuts or menu items. Patchworks animates left/right shifts to convey to the developer the feeling of moving along the ribbon.

A key design decision was to make the ribbon extend out along one dimension, as opposed to, say, a 2D grid. By restricting the ribbon to one dimension, the developer can slide in only two directions along the ribbon, left or right. In contrast to a 2D canvas, this considerably reduces the room for the developer to make a navigation mistake (e.g., by navigating in the wrong direction). This idea is consistent with prior evidence comparing 2D and 3D interfaces. For example, one study of both physical and virtual 2D and 3D systems found that the participants' ability to locate information deteriorated as dimensionality increased, and that the participants reported the higher dimension systems to be more cluttered and less efficient [19].

Similar to Code Bubble's workspace bar, Patchworks has a *ribbon view* (Fig. 8) that presents a bird's eye view of the ribbon. The developer can use the ribbon view to adjust

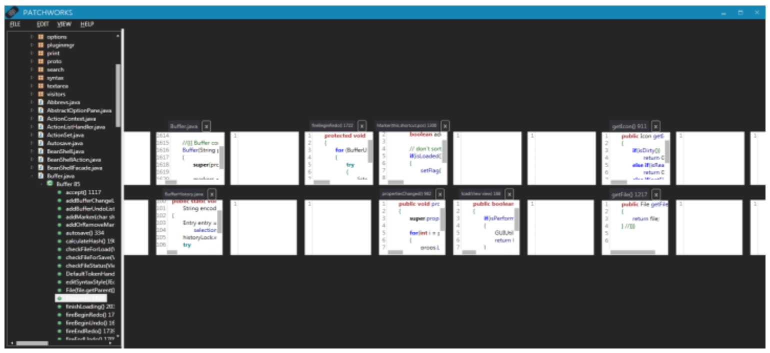


Figure 8: Patchworks' ribbon view, which provides a bird's eye view of the ribbon.

the visible patch grid. In contrast to Code Bubbles, the ribbon view provides information about each patch, and the developer can manipulate patches in the ribbon. Each fragment in the ribbon view includes a name (in a title bar), a thumbnail preview of its contents, and a close button. The developer can use the ribbon view to add, move, swap, and close patches (via the same interactions as in the patch grid).

3.3 Evaluation: Simulation Study

As an initial evaluation of the Patchworks design, we conducted a study in which we simulated users navigating code with Patchworks, using different strategies, compared to Eclipse. Using this approach, we sought to answer two questions:

- Which strategy of arranging patches in Patchworks yields the fastest navigations?
- Do developers navigate faster using Patchworks than using Eclipse?

3.3.1 Candidate Patch-Arranging Strategies

A key question of the current work is *how should developers arrange their patches as they use Patchworks?* The patch and ribbon features offer a wealth of possible ways to arrange code fragments, so what advice should we give to a developer using Patchworks?

An important design assumption of Patchworks is that developers will use the ribbon as a timeline, with less recently visited fragments being further back (left) and more recently visited fragments being further forward (right); thus, we frame the problem of patch arranging as deciding which patches to bring forward and when. For purposes of our evaluation, *bringing a patch forward* entails using the ribbon view to make a new patch for the fragment adjacent to the rightmost patches on the ribbon (leaving the original patch unchanged to preserve the timeline).

We considered four possible strategies for deciding if/when to bring a patch forward, and we define them below. To facilitate automated simulation of the strategies, we defined them formally. A key concern in choosing these strategies was the extent to which a developer would be capable of performing the strategy as defined. We selected two simplistic strategies that a developer would likely be capable of doing, and two that approximate complex internal developer behavior, but that might be difficult for a developer to perform exactly.

In the *Never strategy*, the developer never brings any patches forward. This simple strategy serves as a baseline because it covers the case where a developer essentially does no arranging of code.

In the *Distance strategy*, the developer brings forward the current patch if he/she shifted more than three columns on the ribbon to get to the patch. The rationale for *three* columns is that the patch grid is three columns wide. The difficulty of performing this strategy should be low: the developer needs only to count how many shifts (up to four) it takes to get to a patch. We hypothesize that this strategy will improve upon the Never strategy by bringing distal code forward, thus reducing the navigation cost of revisiting that code.

The *Recency strategy* also tries to reduce the cost of revisits; however, unlike the above two strategies, a developer may have difficulty performing the strategy exactly. In Recency, the developer brings forward the current patch if it was not already on screen (i.e., it required clicking to get to), and it was among the top six most recently visited fragments. The rationale for this strategy is that studies have found recency to be a strong predictor of where a developer will navigate [87, 95]. We chose *six* because the patch grid holds six fragments. However, it may be difficult for a developer to recall exactly the last six methods he/she visited.

Finally, the *DOI strategy* attempts to approximate the developer's *degree of interest* (DOI) in the current fragment to decide whether the fragment should be brought forward. It is common to estimate a person's DOI based on his/her past behavior (e.g., as in [27]). However, we compute a participant's DOI in a code fragment at a given time using his/her future navigations. Given a fragment f, for each future navigation g_i to f, we compute a weight $W(g_i) = 0.85^{d-1}$ where d is the number of navigations into the future that g_i is from the current navigation. Then, we sum all the g_i weights for f to compute the total DOI for f. In the DOI strategy, the developer brings forward the current patch if it is among the top six greatest DOI values (and it was not already on screen). The rationale for six here is the same as for the Recency strategy. Since this strategy uses future behavior, it is the one that developers would be least capable of performing exactly.

3.3.2 Method

To address our research questions, we first conducted a study of CS graduate students engaged in software development tasks using Eclipse. We collected the participants' code-navigation data (where/when they navigated), and then used that data to simulate how the developers might perform the same navigations using Patchworks and following the various patch-arranging strategies described in Section 3.3.1.

3.3.2.1 Navigation-Data Collection

Our participants consisted of 14 graduate students (11 male, 3 female) enrolled in a graduate-level software engineering course. They reported, on average, 7.5 years of programming experience ($SD = 2.9$) and 2.25 years of professional programming experience ($SD = 2.6$). All participants had experience with Java and Eclipse.

We observed each participant working on his/her software project for the course. The projects involved developing a Java EE web application (made up of Java servlets, JSPs, and “plain old” Java classes) over roughly 8 weeks. The participants did this work as part of a development team. The exact tasks on which each participant worked varied, depending on the particular needs of his/her project at the time of the study. On average, the project code bases consisted of 9344 lines of Java and JSP code spread across 84 code files.

Each participant took part individually in one 135-minute session in which we observed him/her working. The participants were first trained on how to do think-aloud for 15 minutes, and then were given 2 hours to work on their projects. They could work on any project-related task they wanted, and had full access to the internet and their phones, including for purposes of communicating with their team. As raw data, we collected screen-capture video, audio of participant utterances, and video of the participants themselves (from the waist up).

Based on each participant’s session data, we used qualitative coding to identify where the participant navigated throughout the session. In particular, we coded each time the participant moved his/her attention from one fragment to another. We also recorded whether the destination was already on screen. We coded fragments at the granularity of methods, classes, and non-Java code files (e.g., JSP and XML). Our coding rules were designed to record only deliberate, intentional navigations, and not navigations that were artifacts of the Eclipse interface.

To ensure that our coding was reliable, we used a standard inter-rater agreement exercise. Three researchers performed the coding. They each independently coded the same 20% of the data (spread across all participants), and achieved 86% agreement (Jaccard similarity) on their codes. Then, they coded the remaining 80% independently.

3.3.2.2 Simulating Navigation in Patchworks

To gain insight into how Patchworks users would fair given the same sequence of navigations each participant made, we built a simulator that uses the navigation data to create alternative scenarios of interaction. The simulator simulates both users of Eclipse and Patchworks, and the simulated users can be given different usage strategies, such as the ones in Section 3.3.1. To simulate Eclipse users, the simulator goes through the navigation data step by step, opening/closing tabs, scrolling through files, and switching tabs (even when elided), as dictated by the input and chosen usage strategy. Similarly, for Patchworks, it simulates a user opening patches, dragging and dropping patches, and shifting the ribbon.

To cope with the complexity of simulating these complex interfaces, our simulator makes several simplifying assumptions. First, we assume that Eclipse displays up to 8 tabs at a time, eliding the rest (see Fig. 1C). In reality, the number varied from 6 to 8. For the purposes of comparing Eclipse and Patchworks, our choice of 8 was conservative because it requires extra clicks to switch to an elided tab, and thus, having more tabs can only reduce navigation times. Second, we assume that Eclipse users know exactly where in a file they need to scroll, and click the scroll bar to jump there (without visual scanning). Again, this assumption is conservative, because it minimizes the cost associated with scrolling. We make a similar assumption regarding Patchworks users: we assume that they know where exactly on the ribbon their destination is. However, unlike our one-click scrolling assumption in Eclipse, simulated Patchworks users click once for each column shift of the ribbon. Third, we assume that users use the mouse rather than keyboard shortcuts for navigation actions. However, this assumption likely favors Eclipse because in our previous study [39], all participants used the mouse in Eclipse (slower), whereas they all used keyboard shortcuts in Patchworks (faster).

3.3.2.3 Evaluation Metrics and Statistical Analyses

Based on the simulated scenarios, we computed two metrics for comparing strategies and tools: the number of simulated navigations to patches that were already on screen and the Keystroke-Level Model (KLM) cost of each navigation in seconds. KLM [16] is a technique for estimating interaction times by breaking tasks into low-level operations. Although these metrics are not entirely orthogonal, each offers important insights. KLM addresses the time cost of the user interactions (e.g., clicks) that produce each navigation. However, KLM, being a model, ignores many details of the real world. Thus, we also included the simpler on-screen navigations metric. To build confidence that our results are valid, we looked for triangulation among these two metrics.

In this work, we follow the example of others who have used KLM to model programming tasks (e.g., [5]), and make some common assumptions. We used the original KLM values for mouse clicks (0.1 for down or up, 0.2 for both), keystrokes (0.28), and positioning the mouse (1.10) [16]. We also used the *mental operator* (1.35), the time required to think or perform eye movements. For simplicity, we ignored the system response time, as others have [4].

Regarding our statistical methods, we followed the standard practice of testing our data for normality to decide whether parametric or non-parametric tests were appropriate. Based on the Shapiro-Wilk test, we could not reject for any of our data the null hypothesis that the data were normally distributed ($p > 0.05$ for each treatment). Thus, we used parametric statistical tests, such as analysis of variance (ANOVA).

3.3.2.4 Limitations

There were several potential limitations to the generalizability of our findings that should be addressed in future work. Our participants were all graduate students, so it is an open question whether seasoned professional developers would produce the same results. The tasks may also not be representative; however, we tried to enhance their realism in several ways: (1) by having them involve industrial-strength development platforms and frameworks, (2) by having participants work on a team project, and (3) by having the participants work on whatever tasks naturally arose in their projects. Finally, our simulation made some simplifying assumptions, such as omitting certain aspects of human cognition; however, the KLM times were based on a well-validated cognitive model.

With regard to whether the differences that can be observed between Patchworks and Eclipse with this method, we went to considerable lengths to be conservative in our

Table 1: Participant navigation data (values rounded to nearest whole number). Recall that fragments are methods, classes, or non-Java code files, such as JSP and XML. Navigations per fragment indicates how much a participant revisited fragments.

	Navigations	Files opened	Fragments visited	Navigations per fragment
Min:	50	4	16	2
Mean (SD):	195 (89)	16 (6)	28 (8)	7 (4)
Max:	308	25	39	18

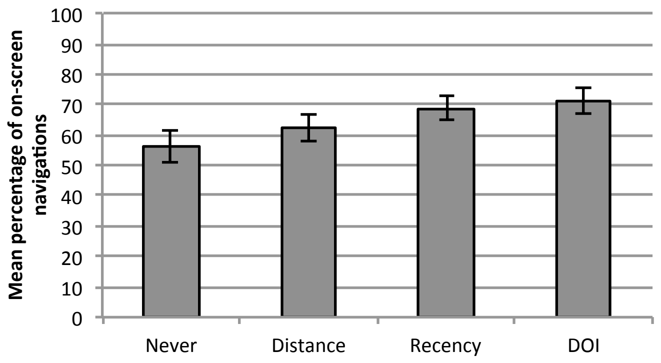


Figure 9: Aggregate patch-arranging strategy results: Percentage of simulated navigations to fragments already on screen (bigger is better; $n = 14$). Whiskers denote standard error.

comparison. Moreover, we hypothesize that Patchworks further improves upon Eclipse in ways that can not captured by this study. A design goal of the ribbon is to support spatial, episodic, and associative memories when developers arrange their patches, unlike tabs in file-based editors which are spatially unstable, often being closed when too many are open, and not strongly associated with their content [90]. Patchworks also supports working at method granularity, which is not supported by Eclipse, but may have an effect on performance.

3.3.3 Results

In this section, we present the simulator results for each research question. Table 1 summarizes our participants' navigation data, which we used as input to the simulator.

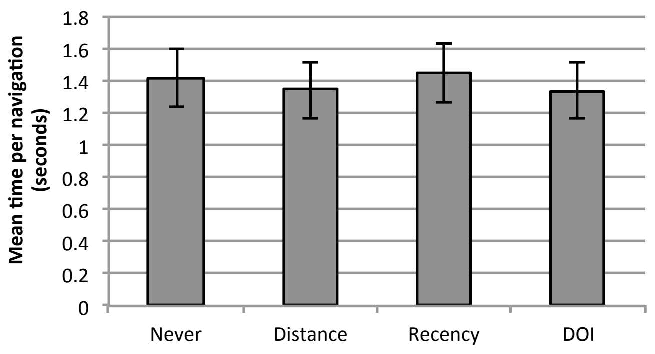


Figure 10: Aggregate patch-arranging strategy results: KLM times (smaller is better; $n = 14$). Whiskers denote standard error.

3.3.3.1 RQ1 Results: Patchworks Patch-Arranging Strategies

As Figs. 9 and 10 show, the simulator results across patch-arranging strategies were fairly similar. In terms of navigations to on-screen patches, ANOVA showed that there was a significant difference between strategies ($F(3,39) = 9.15$, $p < 0.01$). However, a pairwise comparison (with Bonferroni correction) showed a significant difference only between the future-peeking DOI strategy and two other strategies: the baseline Never strategy ($p = 0.02$) and Distance strategy ($p < 0.01$). Similarly, ANOVA showed no significant difference between strategies in their KLM times.

Looking at the individual participant results in Figs. 11 and 12, certain strategies stood out as performing noticeably worse or better for some participants. For example, Recency performed particularly poorly for P13. We discuss some possible reasons for these anomalies in Section 6.

3.3.3.2 RQ2 Results: Multi-Row Tabs in Eclipse

As Fig. 13 shows, a significant developer-navigation speedup can be achieved in Eclipse by allowing more editor tabs and never closing/reopening tabs. Comparing (1) an 8-tab limit with tab closing/reopening, (2) an 8-tab limit without tab closing, and (3) multi-row tabs (no tab limit) without tab closing, ANOVA showed a significant difference in KLM times ($F(2,26) = 14.57$, $p < 0.01$). A pairwise comparison (with Bonferroni

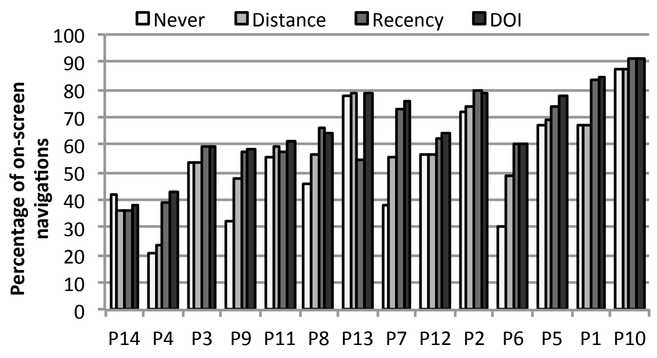


Figure 11: Per-participant patch-arranging strategy results: Percentage of simulated navigations to fragments already on screen (bigger is better). Participants sorted from fewest navigations per fragment to greatest.

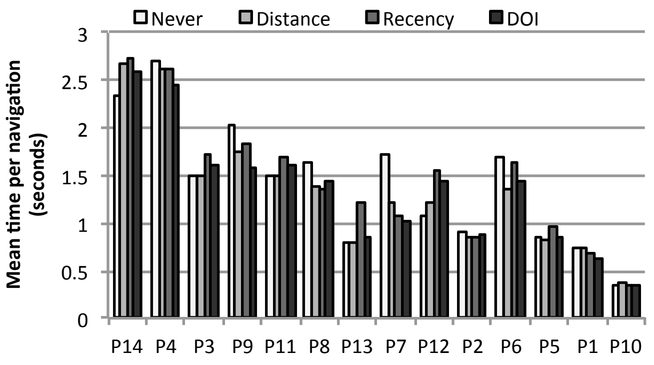


Figure 12: Per-participant patch-arranging strategy results: KLM times (smaller is better). Participants sorted from fewest navigations per fragment to greatest.

correction) showed that KLM times for multi-row tabs without closing were significantly lower (faster) than either of the 8-tab limit treatments.

3.3.3.3 RQ3 Results: Patchworks Versus Eclipse

As Figs. 14 and 15 show, the simulated Patchworks users had a significantly greater percentage of navigations to patches already onscreen and significantly lower KLM times than simulated users of either version of Eclipse (normal or multi-row tabs). In these results, we had our simulated Patchworks users use the Distance patch-arranging strategy, because it was the simplest strategy that actually involved bringing patches forward. However, these results were the same no matter which strategy we chose. For the on-screen navigation results, ANOVA shows a significant difference between Patchworks

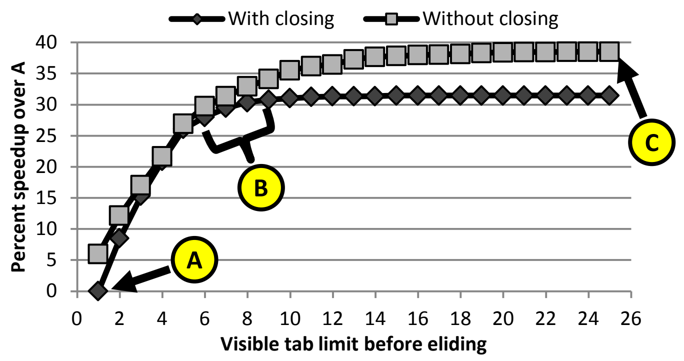


Figure 13: KLM comparison of Eclipse editors that allow different numbers of visible tabs before eliding. Label A denotes the baseline (only 1 visible tab and tab closing/reopening included); B denotes the range of visible tabs that participants saw (6–8); and C denotes the case of multi-row tabs (no eliding) that are never closed.

and the two versions of Eclipse ($F(2,26) = 41.6$, $p < 0.01$). A pairwise comparison (with Bonferroni correction) showed that the simulated Patchworks users had significantly more on-screen navigations than simulated users of both plain Eclipse ($p < 0.01$) and multi-row tabs Eclipse ($p < 0.01$). Similarly, for KLM times, ANOVA showed a significant difference between Patchworks and the Eclipses ($F(2,26) = 62.2$, $p < 0.01$), and a pairwise comparison (with Bonferroni correction) showed that the KLM times for simulated Patchworks users were significantly lower than simulated users of both plain Eclipse ($p = 0.01$) and multi-row tabs Eclipse ($p = 0.01$).

Turning to the per-participant results in Figs. 16 and 17, one anomalous case stood out: P4, the only simulated user to perform better using Eclipse than Patchworks. In Section 3.3.4, we discuss the possible reasons for this anomaly.

3.3.4 Discussion

3.3.4.1 Patchworks: Robust to Different Strategies

Based on our RQ1 results, it generally mattered little which patch-arranging strategy was chosen. Only the future-peeking DOI strategy showed a statistically significantly improvement over any of the others, and the magnitude of the difference was relatively small. For example, the best performing strategy improved upon the worst by only 15

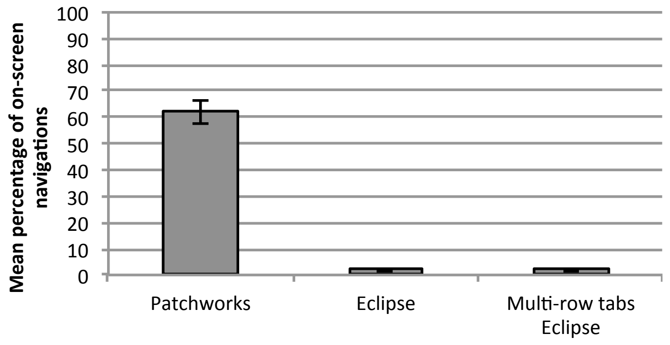


Figure 14: Aggregate Patchworks versus Eclipse results: Percentage of simulated navigations to fragments already on screen (bigger is better; $n = 14$). Whiskers denote standard error.

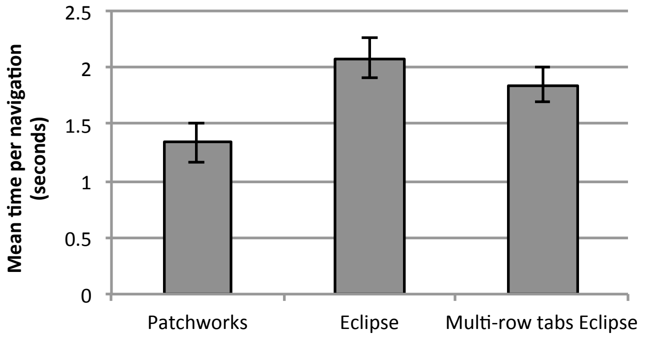


Figure 15: Aggregate Patchworks versus Eclipse results: KLM times (smaller is better; $n = 14$). Whiskers denote standard error.

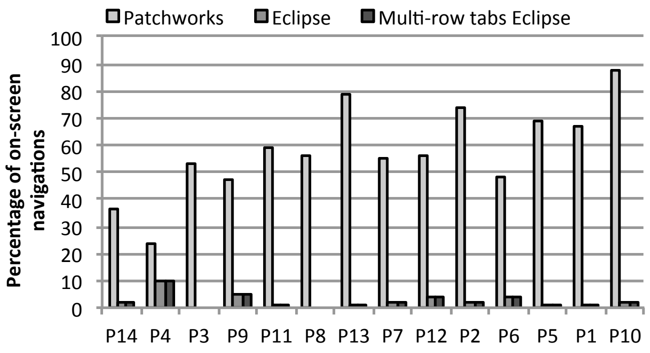


Figure 16: Per-participant Patchworks versus Eclipse results: Percentage of simulated navigations to fragments already on screen (bigger is better). Participants sorted from fewest navigations per fragment to greatest.

percentage points for on-screen navigations and by only an 8% speedup in KLM time. This result suggests that Patchworks users need not be overly concerned about what strategy they use, as long as they treat the ribbon as a timeline.

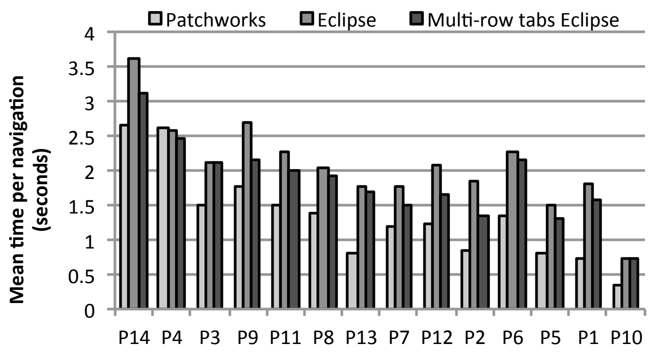


Figure 17: Per-participant Patchworks versus Eclipse results: KLM times (smaller is better). Participants sorted from fewest navigations per fragment to greatest.

Although, in general, strategy had little effect, for a few participants certain strategies worked far better or worse than the others (see Figs. 11 and 12). For example, participants P14 and P7 make an interesting contrast, because P14 was the only participant for whom the Never strategy outperformed the others, whereas for P7, that strategy showed the greatest decline in performance relative to the others. P14's curious result can be explained because he did not revisit patches very often. In fact, he had the fewest navigations per fragment of any participant. Because, P14 was not revisiting, there were fewer opportunities to navigate to patches already on screen, and the cost of moving patches forward in anticipation of revisits did not outweigh the cost of simply shifting the ribbon. P7's result can also be explained because she made many back-and-forth navigations between fragments that were initially placed at the extreme ends of the ribbon. In her case, bringing forward patches made a considerable difference in the cost of these navigations.

For P13, the Recency strategy performed noticeably worse than the others. Due to an accident of his navigation sequence, the Recency strategy made many long, costly shifts along the ribbon to far-away patches, and when it finally brought the far-away patches forward, it turned out that they were no longer needed, leading to more wasted time.

Lastly, P6 had a curious contradiction for the Distance strategy. The Distance strategy had fewer on-screen navigations than the other non-baseline strategies, Recency and DOI, but counterintuitively, also had the best KLM time of the three. For this participant, Recency and DOI performed far more rearrangements, and thus, required more time compared to the short shifts that Distance performed.

3.3.4.2 Eclipse: Time to Adopt Multi-Row Tabs?

Our RQ2 simulator results, showing that multi-row tabs were significantly faster than eliding editor tabs, suggest that developers might benefit from using a plugin that provides multi-row tabs. In fact, there are already signs of latent demand for such tab features: several questions about how to get multi-row tabs have appeared on the popular Stack Overflow website, and received over 45,000 views cumulatively.[1] Furthermore, the Eclipse bug tracker contains an open feature request from 2004 for multi-row and vertical tab support, with active discussion as late as 2018.[2]

However, there may be drawbacks to multi-row tabs that our simulation study could not detect. One such drawback may arise because with each new row of tabs added, the layout of the rest of the editor changes, generally shrinking the amount of code that is visible. Having less visible code may bother some developers. Furthermore, changing the layout may also be disruptive to developers' spatial memory.

3.3.4.3 Patchworks: Faster than the Fastest Eclipse

Based on our RQ3 results, Patchworks significantly outperformed both plain Eclipse and the multi-row tabs version. Although Patchworks' KLM times were strong, where it really shone was in increasing the number of navigations to fragments already on screen. Given the limited viewing space in the Eclipse editor, a user is lucky if there are even three or four total fragments visible at a time. In contrast, with Patchworks, there are always six slots where fragments can go.

[1]See `http://stackoverflow.com/q/3059690/938695`, `http://stackoverflow.com/q/8297293/938695`, and `http://stackoverflow.com/q/18137114/938695`.

[2]See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=58945`.

Also clear from the per-participant results (see Figs. 15 and 16) is that Patchworks does a particularly effective job of supporting revisits. For example, it is no accident that P10, the participant with the greatest navigations per fragment, has the greatest percentage of navigations to on-screen patches (nearly 90%) and the lowest KLM times of any participant. Similarly, the only participant for which Patchworks had worse KLM times than Eclipse (albeit by a small amount) is P4, who also happened to have among the fewest navigations per fragment.

3.4 Evaluation: Preliminary User Study

As a next step towards evaluating the Patchworks design, we conducted a controlled laboratory study of developers using Patchworks, Eclipse, and Code Bubbles to perform sequences of navigations. This approach allows us to better evaluate Patchworks by using human participants. We sought to answer three research questions:

- Do developers using Patchworks navigate between fragments of code more quickly?
- Do developers using Patchworks make fewer navigation mistakes?
- Do developers using Patchworks spend less time arranging code fragments?

3.4.1 Method

To address our three research questions, we conducted a controlled study that compared our Patchworks editor to a traditional file-based editor, Eclipse, and a canvas-based editor, Code Bubbles. Thus, there were three treatments, one for each editor. To test a treatment, participants used the editor to perform tasks on a Java code base that involved opening and arranging methods (i.e., code fragments), and navigating to methods. Due to time constraints, each participant received only two of the treatments, randomly assigned, but balanced so that each treatment was tested equally. Since each participant tested two editors, there were two task sets, one for each editor. We blocked for two potential confounds: treatment order (due to learning effects) and task set. In addition to addressing our main RQs, we also assessed the participants' subjective opinions of the editors.

3.4.1.1 Participants

Our participants consisted of 15 university students (12 males, 3 females; 14 graduate, 1 undergraduate). They reported, on average, 4.9 (*SD* = 2.6) years of programming experience. All participants reported having at least 2 years of Java programming experience and some experience with Eclipse. None of the participants were familiar with Code Bubbles or Patchworks.

3.4.1.2 Subject Code Base

Each of the two task sets involved opening, arranging, and navigating code from the JEdit open-source text editor. We chose JEdit because it is a real-world software project, comprising 5876 Java methods and 98,718 lines of code. Each task set involved a different group of 30 methods. To enhance ecological validity, the methods in each group were all relevant to a particular concern. One of the concerns pertained to JEdit's autosave feature and the other pertained to text folding. We generated the groups using Suade [116], an automated software engineering tool for mapping methods to concerns.

3.4.1.3 Task Sets

Each task set had the following format, differing only in the concern code involved. First, the participant performed a method-opening and code-arranging task. He/she was given the list of 30 methods related to the concern, along with the reason why each method was included, and instructed to open all the methods and arrange them as he/she saw fit. The participant was also told that he/she could take as much time as he/she wanted to do this arranging. We did not reveal to the participant that we were timing this task. Next, the participant performed a series of 10 navigation tasks. For each navigation task, the participant was shown a card with the name of a method (as well as its package and class), and instructed to navigate to the method as quickly as possible. We asked participants not to use code-search features, because such features do not vary significantly among the editors, and were not the focus of this evaluation. The participant had to successfully complete each navigation task before beginning the next one, and we informed the participant that we were timing these tasks.

3.4.1.4 Procedure

Each participant's session took roughly 60 minutes. First, the participant filled out a background questionnaire, and was given an overview of the session format. Next, the participant performed the two task sets described above, each using a different tool. Before starting each task set, the participant was briefly introduced to the features of the tool he/she would be using. Last, the participant filled out a questionnaire regarding his/her opinions of the editors. If the participant used Patchworks, he/she also answered a set of Likert-scale questions regarding his/her opinion of Patchworks. In addition to the questionnaire responses and recorded times, the collected data comprised audio and video of each participant's task performance, including screen-capture video.

3.4.1.5 Analysis Method

Statistical Tests Following standard practice, we tested the distributions of our data for normality to decide whether parametric or non-parametric statistical tests were appropriate. For RQ1 (navigation time) and RQ2 (number of mistakes), the Shapiro-Wilk test showed that for each treatment, the recorded times and counts were most likely not normally distributed ($p < 0.01$ for each), so we used the non-parametric Kruskal-Wallis rank sum test for those data. On the other hand, for RQ3 (arranging time), based on Shapiro-Wilk, we could not reject the null hypothesis that any of the recorded times were normally distributed ($p > 0.05$ for each treatment), so we used the parametric analysis of variance (ANOVA) test for those data.

Our clustering by participant might have violated the data independence assumption of our statistical tests; however, we found near 0 correlation between participant and navigation time (Spearman's $r_s = 0.06$), between participant and mistakes (Spearman's $r_s = -0.06$), and between participant and opening time (Pearson's $r = -0.03$). Thus, we conclude that participant had no significant effect on our response variables.

Navigation-Mistake Coding To identify participants' navigation mistakes during their navigation tasks, we used the following objective criteria. We coded a navigation action as a mistake if the action moved the participant's view farther away from the target method. In our coding, we merged sequences of repeated mistakes into a single mistake. In Eclipse, scrolling in the wrong direction or selecting the wrong file tab constituted a navigation mistake. In Patchworks, shifting the view along the ribbon in the wrong direction constituted a navigation mistake.

In Code Bubbles, identifying mistakes was complicated by the 2D canvas. If the target bubble was *in view* (i.e., the majority of the bubble and its entire name were visible on screen), we coded a mistake if a navigation action caused the bubble to go *out of view* (i.e., majority off screen or name obscured). Fig. 18 illustrates how we coded mistakes if the target bubble was out of view. In such cases, we coded a mistake if the participant's navigation action caused the center of his/her view to move in any of the directions labeled *mistake directions*. To eliminate small unintentional moves and other artifacts of using a mouse, we counted only moves that resulted in bringing an out-of-view bubble (not necessarily the target) into view.

3.4.1.6 Limitations

Our study has several limitations that should be addressed in future work. First, from an ecological standpoint, the tasks that the participants performed were artificial, and were chosen to enable us to focus on our specific research questions. However, as a way to enhance ecological validity, we had them work with code from a real-world software project (JEdit), and we selected code fragments that were related to a shared concern. An open question that we will address in future work is how working with code over longer

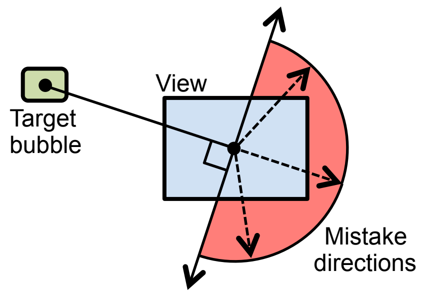


Figure 18: Method for coding navigation mistakes in Code Bubbles when the participant's target bubble is out of view.

periods of time affects developer navigation (e.g., because the developer has more time to develop spatial memory). Second, all the participants in our study were students (mostly graduate), and it is an open question whether our results would hold for seasoned professional developers. Third, reactivity effects may have influenced participant performance/responses; however, we were careful not to tell participants which tool was ours or what comparisons we were making. Although participants certainly knew that Eclipse was not our tool, none were familiar with Code Bubbles or Patchworks. Fourth, the participants were using Code Bubbles and Patchworks for the first time, and it is an open question how experts with those tools might perform. However, it is encouraging that, despite being unfamiliar, participants navigated significantly faster with those tools than with the more familiar Eclipse.

Additionally, we identified *misses*, that is, navigation mistakes in which participants had the target method in view, but failed to spot it. To count as a miss, the majority of the target method had to be in view, and the participant had to perform a navigation action that moved the method out of view.

3.4.2 Results

3.4.2.1 RQ1 Results: Navigation Time

As Fig. 19 shows, the mean navigation times for participants using Code Bubbles and Patchworks were considerably lower than for those using Eclipse. In fact, participants'

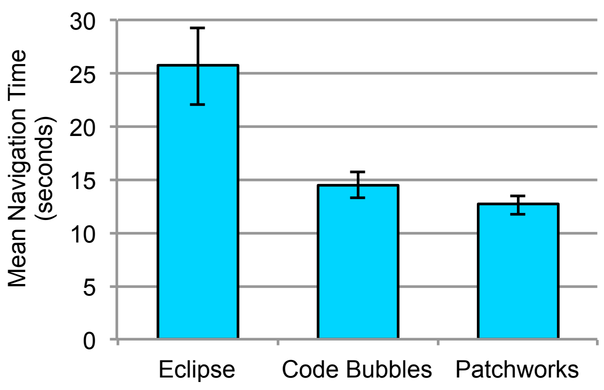


Figure 19: Participants' mean navigation times in Eclipse ($n = 99$), Code Bubbles ($n = 98$), and Patchworks ($n = 99$). Whiskers denote standard error.

navigations in Eclipse took, on average, roughly double the time they did in Patchworks. Indeed, the results of a Kruskal-Wallis test revealed a significant effect of editor on navigation time ($\chi^2(2) = 12.1$, $p < 0.01$). Furthermore, a pairwise multiple-comparisons test showed that participants' navigation times with Eclipse were significantly greater than with Code Bubbles ($p < 0.05$) and with Patchworks ($p < 0.05$).

3.4.2.2 RQ2 Results: Navigation Mistakes

As Fig. 20 shows, participants made considerably fewer navigation mistakes using Patchworks than using Eclipse or Code Bubbles: Patchworks users make less than half the number of mistakes that Eclipse users and Code Bubbles users made. The results of a Kruskal-Wallis test showed a significant effect of editor on number of mistakes ($\chi^2(2) = 8.56$, $p = 0.01$). Looking at just the mistakes in which developers failed to see a code fragment that was on screen (see the inner miss-mistake bars in Fig. 20), the effect of Patchworks was even stronger: Patchworks users missed code fragments one quarter or less of the number of times that they missed a fragment in Code Bubbles and Eclipse. Again, a Kruskal-Wallis test revealed a significant effect of editor on miss mistakes ($\chi^2(2) = 6.22$, $p = 0.04$).

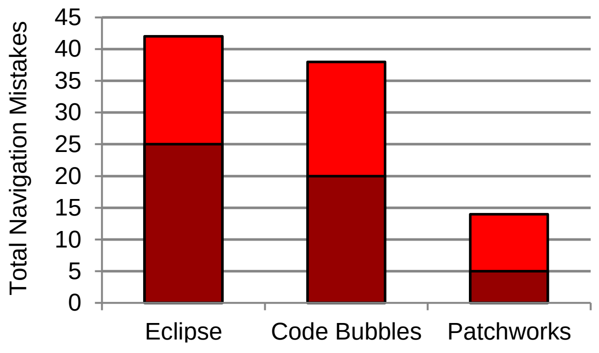


Figure 20: The number of navigation mistakes that participants made while navigating (out of 99 navigation tasks for Eclipse and Patchworks, and 98 for Code Bubbles). Inner bars show the subset of mistakes that were misses.

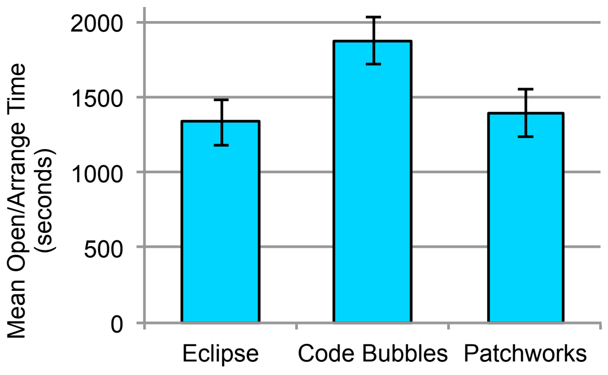


Figure 21: Mean time that participants took to open and arrange their code. Whiskers indicate standard error. $n = 10$ for each tool.

3.4.2.3 RQ3: Time to Open/Arrange Code

As Fig. 21 shows, participants using Code Bubbles spent considerably more time opening and arranging their code than they did using Patchworks or Eclipse: for example, on average, Code Bubbles users took over 8 minutes longer than Patchworks users (who took 24 min on average). Indeed, an ANOVA test revealed a significant effect of editor on time ($F(2,27) = 3.67$, $p = 0.04$). Delving deeper into the model, the regression coefficients, with Code Bubbles as intercept, showed a significant difference between Code Bubbles and both Eclipse ($p = 0.02$) and Patchworks ($p = 0.04$).

Table 2: Results of Patchworks opinion questionnaire (on a 7-point likert scale from 7 = most favorable to 1 = least favorable).

Question	Min	Mean (Std. Dev.)	Max
Easy/hard to learn?	6	6.63 (0.50)	7
Easy/hard to use?	5	6.27 (0.64)	7
Features help/hinder?	4	5.63 (0.92)	7
Like/dislike?	4	6.09 (0.83)	7
Would you use Patchworks?	5	6.27 (0.90)	7

3.4.2.4 Participants' Subjective Opinions

As Table 2 shows, all participants reported positive opinions of Patchworks. Recall that each participant who used Patchworks completed a Likert-style opinion questionnaire (5 questions, each covering a different aspect of Patchworks). Based on their responses, participants found Patchworks particularly easy to learn and use, and they would use the patch grid and ribbon features if available in their IDE of choice.

When asked which of the tools they liked better and why, participants also favored Patchworks most often. They expressed preferring Patchworks 2:1 over Eclipse and 2:1 over Code Bubbles. Although they also preferred Code Bubbles over Eclipse, the margin was smaller at 3:2.

3.4.3 Discussion

3.4.3.1 Summary of Results

Table 3 summarizes the results of our evaluation, and as the table shows, Patchworks matched or exceeded Eclipse and Code Bubbles on each of the evaluation criteria. As a result, Patchworks performed better overall (i.e., across all criteria) than the other editors. In particular, Patchworks was able to maintain the navigation speed gains of the canvas-based approach, Code Bubbles, but like the file-based Eclipse editor, Patchworks

Table 3: Summary of results. A check indicates statistical evidence, and a tilde indicates evidence, but on a sample not amenable to statistics. Two checks in the same row indicate comparable results (i.e. a tie).

Result	Eclipse	Code Bubbles	Patchworks
Fastest navigation		✔	✔
Fewest mistakes			✔
Least arranging	✔		✔
Most preferred			~

users spent less time arranging their code fragments. Moreover, Patchworks surpassed the other editors in helping developers avoid navigation mistakes.

In the remainder of this section, we discuss these results with respect to our qualitative observations and the participants' comments on the questionnaire, and close with a discussion of our study's limitations.

3.4.3.2 Eclipse: Excessive Scrolling Leads to Missed Methods

Based on our qualitative observations, participants' long navigation times in Eclipse appeared largely due to time spent scrolling through files full of irrelevant code (as opposed to switching tabs), and failing to see the methods they wanted among that code. For example, P13 was searching for the method `updateStructureHighlight` within a file, *TextArea.java*, which was over 5500 lines long. In general, participants had no trouble navigating to the correct file: only two participants (P1 and P6) chose the wrong tab while navigating. P13 was no exception, and he rapidly navigated to the correct file tab; however, the editor had been previously scrolled to the middle of the file, and the target method was not in view. He began slowly scrolling toward the top of the file, scanning over methods as he went. But after passing a few methods, perhaps growing impatient, he suddenly began scrolling much more rapidly. He quickly arrived at the top of the file, having passed the target method without seeing it. He then began scrolling back down the file, this time more slowly; however, this slow pace did not last long. Again, he sped up, passing the target method a second time without seeing it, until finally arriving at the bottom of the file. He repeated this process several times before finally spotting his target. In total, he spent nearly 2 minutes scrolling, and passed the method without seeing it 5 times.

Because users of Code Bubbles and Patchworks were able to open code at the granularity of methods, they had the advantage of never having to scroll through code to reach their navigation targets. They had to search only through the 30 methods related to their current concern, which they navigated among by panning the view around the canvas in Code Bubbles, and sliding the view along the ribbon in Patchworks. Furthermore, Eclipse typically displayed only one or two methods at a time on screen, whereas Code Bubbles and Patchworks displayed substantially more methods, increasing the chances that the participant's target was already in view. One final possible advantage Code Bubbles and Patchworks users had was that the method names were clearly presented in bubble or patch title bars, as opposed to being embedded in blocks of text, as in the Eclipse editor.

3.4.3.3 Code Bubbles: More Organized, but Misses Still Common

Participants using Code Bubbles clearly invested more into the arrangement of their code fragments than Patchworks or Eclipse users. For example, when P3 performed the opening/arranging task using Code Bubbles, he first arranged his bubbles into four groups. However, apparently disatisfied with this arrangement, he spent an additional 16 minutes and 13 seconds reorganizing them into six groups. In total, this involved 270 bubble-moving actions. Similarly, P12 realized he was spending considerable time arranging his bubbles, and made several apologetic remarks: "sorry this is taking longer [compared to Patchworks]." He later commented about Code Bubbles being tougher to use than Patchworks: "the second one [Code Bubbles] took more time for me because the dragging and dropping those things [bubbles] around…"

In contrast, participants spent little time arranging the patches in Patchworks. In fact, only two participants moved a patch after he/she initially placed it. We observed that most participants placed the code along the ribbon in one direction. When asked what their strategy was for arranging the patches, several participants (P4, P7, P10, and P15) said they placed them sequentially along the ribbon. Other participants (P5, P11, P14) said they didn't have a strategy but P11 clarified that a strategy wasn't needed because of the ribbon view. Regardless of the strategy used, it was common for participants to leave empty columns of patches between groups, possibly as landmarks or separators.

Despite how much time Code Bubbles users spent arranging their bubbles, they still made numerous navigation errors (significantly more than Patchworks). A common mistake was for Code Bubbles users to revisit bubbles they had already seen while searching for a particular method. For example, P2 arranged his bubbles into four columns, grouped by class and function. On one of the navigation tasks, he started with the viewing area at the top of his third column. The target method, `requestFocus`, was on screen, but he failed to notice it. Instead, he panned to the bottom of the column slowly, and then back to the top. Still not seeing the method, he again panned downwards, but only halfway down the column, before going back to the top and finally noticing the method. P13 had a similar incident while performing a navigation. He had arranged his bubbles in a large square-like shape grouped by class. He started searching for the method `read` from the bottom left corner of the canvas and then panned around in a clockwise fashion, following the perimeter of his bubble cluster. Eventually, he reached the part of the cluster he had started from. He then panned around the perimeter of his cluster two more times, only this time going counter-clockwise. Finally, he spotted the method after having missed it a total of three times.

Perhaps, for the above reasons, participants made several critical comments regarding Code Bubbles. For instance, P9 preferred Code Bubble to Eclipse, but added the caveat that Code Bubbles is better only when the user is “cautious.” On the other hand, P14 described Code Bubbles as “impressive,” but still favored Eclipse.

3.4.3.4 Participant Feedback on Patchworks

In addition to the participants’ positive Likert scores regarding Patchworks, a number of their questionnaire responses got at “why” they liked it. For instance, Participant P15 said that, compared to Eclipse, in Patchworks “It is much easier to navigate.” Similarly, P13 described Patchworks as a “good way to organize methods and work switching between them,” and P3 commented that Patchworks “definitely gives me an extra opportunity for quick navigation and for placing code side by side for easy viewing.” P5 described Patchworks as “very fluid,” and added “Eclipse is nice when only focusing on one file at a time (as I usually am doing), but Patchworks seems like a great tool for comparing and following code that may not be organized in a way that makes sense to me (ex: if I were viewing code someone else had written).” In comparing Patchworks and Code Bubbles, Participant P10 even indicated that the patch grid and ribbon features indeed overcame issues with canvas-based editors: “because it’s easy to navigate through the grid rather than a canvas.” These comments are encouraging and lend credence to the idea that Patchworks had successfully achieved its design objectives.

Participants also offered critical feedback and suggestions for improving Patchworks. Participant P4 wanted more lines of text in a patch, and P5 wanted more “zoom levels” for viewing the ribbon. P15 reported having difficulty finding the method names of patches. (This information was presented in a label at the top of each patch, but the font may have been too small for him to see.) P11 reported feeling “clumsy” using Patchworks, and preferred Eclipse because it was “more familiar.” She also expressed wanting search features like Eclipse has, and support for multiple monitors (although she did not give specifics). Most of these concerns seem straightforward to address, and we will consider them in future work.

3.5 Tool Design: Patchworks for LabVIEW

To understand how the Patchworks design generalizes to other development environments, we applied it the LabVIEW visual programming environment. Based on our findings from the simulation study and preliminary user study, and to better suit the LabVIEW environment, we altered the Patchworks design to better suit LabVIEW. In this version of Patchworks, we focus on four key subgoals of the design:

- Reduce the cost of navigation (i.e., clicks per navigation).
- Facilitate the juxtaposition of patches (i.e., placing patches side by side on screen).
- Reduce navigation mistakes (i.e., accidentally navigating to the wrong patch).
- Enable efficient toggling between multi-patch and single-patch views.

As illustrated in Fig. 22, the Patchworks for LabVIEW editor's main display consists of the *patch grid* with four fixed *patch cells*. Each patch cell may contain a VI block diagram or front panel. The choice of four patch cells was based on the typical size of a VI and the study-monitor size. (In contrast, the prior Patchworks editor for Java [39] had six patch cells.) This multi-patch display aims to facilitate juxtaposition of patches by emphasizing the presentation of more than one patch on screen at a time, while the fixed grid layout aims to reduce tedious window management. Patches can be efficiently moved and swapped between cells via dragging and dropping.

A new feature is that Patchworks for LabVIEW enables toggling a patch to a "blowup" view that takes up the entire editor space. To toggle into or out of the blowup view, the programmer double-clicks the selected patch's title label. In this way, Patchworks aims to enable efficient switching between multi-patch and single-patch display. Similar to Patchworks for Java, the Patchworks editor enables toggling to a bird's eye view of the patch strip that shows a thumbnail of each open patch, as illustrated in Fig. 23. The developer can rearrange patches from the bird's eye view via the same dragging and

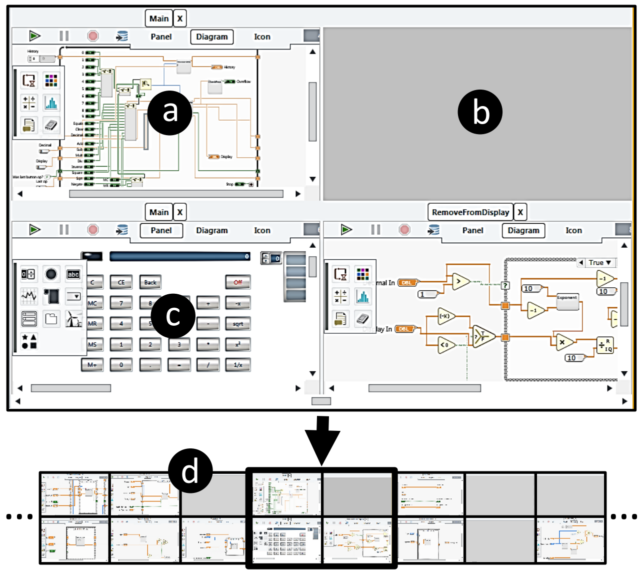


Figure 22: The Patchworks code editor for LabVIEW. This multi-patch display divides the editor pane into a patch grid with four patch cells. Each patch cell may contain a block diagram (a) or front panel (c) for a VI, or it may be empty (b). The displayed patch grid is actually a view into a larger patch strip (d), and the programmer can navigate by sliding the view left or right along the patch strip.

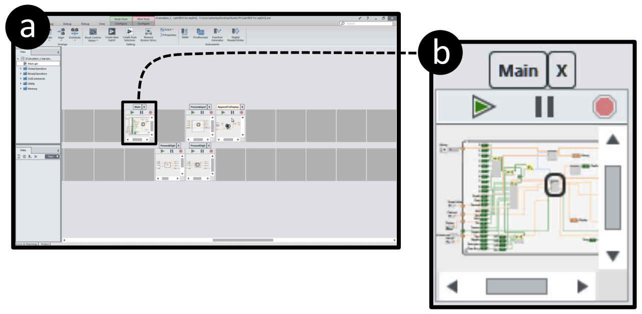


Figure 23: Patchworks for LabVIEW's bird's eye view (a) enables the user to zoom out, and survey and organize his/her open patches. It represents each patch with a thumbnail (b).

dropping interactions as in the patch-grid view, and can zoom back into the patch-grid view by clicking on a selected patch.

3.6 Evaluation: Industrial User Study

Our third evaluation of Patchworks entailed applying it to a visual language environment. While extending the LabVIEW development environment with the Patchworks interface, I saw the need to make two modifications to the Patchworks design. I added a blow-up view that allows developers to focus on one piece of code. The other change was the number of patches in the patch grid, which I reduced to 4 due to the large size of LabVIEW code. Having a more generalized design that has been applied to a different development environment, our next step was to run a more ecologically valid lab study. The goals of this study were to:

- Evaluate the performance of developers using Patchworks.
- Investigate other navigation trends.
- Create a set of design principles for code editors.

3.6.1 Method

To address our research goals, we conducted an observational laboratory study of professional LabVIEW developers engaged in debugging. We divided the participants into two treatments, each using a different editor. The *Tabs Group* used an editor based on the ubiquitous tabbed interface to perform debugging tasks, whereas the *Patchworks Group* used the Patchworks editor.

3.6.1.1 Participants

Our participants consisted of 32 professional developers from a large technology company (27 males, 5 females). All held bachelor's degrees in engineering, and five also held master's degrees. On average, they had 1.6 years of professional programming experience (*SD* = 1.0) and 2.1 years of LabVIEW experience (*SD* = 1.8). They all reported using LabVIEW on a daily basis for their jobs.

3.6.1.2 Code Base and Tasks

For our study, the participants worked on a calculator application written in LabVIEW, consisting of 34 VIs. The application was based on a publicly available sample application [20] written in an old version of LabVIEW, and we ported this sample application to the current version of LabVIEW. We also seeded the application with five bugs that we actually encountered during the porting process.

As tasks, each participant worked on fixing the five seeded bugs. The tasks were ordered from least to most difficult. For example, Task 1 required understanding mainly one VI, whereas Task 4 required understanding interactions between several. All participants did the tasks in the same order, and had to complete the current task before beginning the next.

3.6.1.3 Procedure

Prior to their study sessions, participants were randomly assigned to the Tabs and Patchworks Treatment Groups such that there were 16 participants in each group. Each session took roughly 1 hour. First, the participant filled out a background questionnaire. Next, the participant viewed a short presentation on the code base and the editor they would be using (tabbed or Patchworks, depending on treatment). The participant then practiced with the editor and code base for 10 minutes, answering questions about the source code. The participant then worked for 40 minutes on the debugging tasks. To better understand where participants placed their attention, we asked them to "think aloud" as they worked. At the end of the session, the participants took part in a semi-structured interview in which they discussed any issues they had and their thoughts on the editor. We recorded screen-capture video and audio of participant utterances throughout the session.

3.6.1.4 Qualitative Analysis

As our main analysis, we used qualitative coding methods to identify where the participants navigated—that is, on which patches they placed their attention—throughout each of their sessions. Following the coding rules from a prior study [43], we coded each time (accurate to a second) a participant moved his/her attention from one patch to another. Additionally, we coded navigation mistakes based on participants' utterances after a navigation, indicating if a navigation did not result in their intended destination.

To ensure the reliability of our analysis results, we applied a standard inter-rater reliability method in which two researchers independently code the same 20% of the data, and must achieve at least 80% agreement using the Jaccard index. When the researchers have reached this level of agreement, they may split up the remaining data to be coded individually. The two coders for our study achieved 88.9% agreement for 20% of the video data, and individually coded the remaining data.

3.6.1.5 Limitations

Our study had several limitations to generalizability common to laboratory studies of developers. For example, there is a question as to how well the sample of developers and tasks in our study represent our target populations. Our participants were unfamiliar with the code base and were given a relatively short amount of time (less than 1 hour) to work; however, they were all professional developers, increasing the likelihood that they were representative of expert developers of visual dataflow languages. The code base was also relatively small; however, it was based on an open source example, and the seeded bugs were ones we actually encountered in practice. Finally, we studied only two editor designs, leaving the comparison of other designs for future work.

3.6.2 Results

To address our research goals, we collected and analyzed over 21 hours of video of our 32 professional developers engaged in debugging, half using a tabbed editor and half using Patchworks. On average, participants completed 2.63 of the 5 tasks ($SD = 0.71$), with each completed task taking an average of 9.55 minutes ($SD = 7.24$). Although there was no significant difference in the rate of success or overall task time between the treatment groups, they did exhibit key differences with respect to navigation—the focus of our investigation.

In this section, we first report the results of our comparative evaluation of Patchworks' navigation affordances. Next, we describe empirical trends across our treatments that are relevant to the design of navigation affordances and that are not specifically addressed by Patchworks. We close the section with a comparison of our LabVIEW developers' navigation traits with those of Java developers in prior studies.

3.6.2.1 Results: Patchworks Evaluation

Using our analysis data, we evaluated the extent to which Patchworks met each of its four main design goals.

Reducing Navigation Cost To evaluate the goal of reducing navigation cost, we randomly sampled 10 navigations from each participant and counted the number of click actions the participant performed in making that navigation. Such actions mainly included tab clicks in the tabbed editor and patch-strip shift actions in Patchworks.

As Fig. 24 shows, navigations in Patchworks were considerably more efficient, requiring fewer clicks, than navigations in the tabbed editor. On average, Patchworks participants made less than half (40%) of the click actions that Tabs participants made per navigation. This difference was statistically significant (Mann–Whitney: $U = 22.5$, $Z = 3.96$, $p < 0.0001$).

Facilitating Patch Juxtaposition Consistent with Patchworks' goal of facilitating juxtaposition, Patchworks participants juxtaposed patches more than Tabs participants. Every Patchworks participant juxtaposed patches during their tasks. In contrast, only 10 of

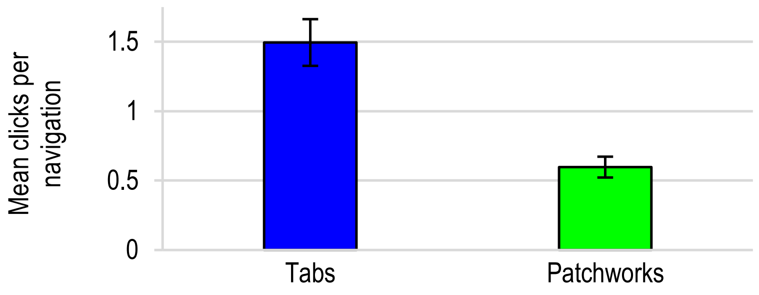


Figure 24: Patchworks participants performed significantly fewer click actions per navigation than did Tabs participants (smaller bars are better). Whiskers denote standard error.

the Tabs participants (63%) juxtaposed patches. This difference was statistically significant (Fisher's exact test: $p < 0.01$).

Patchworks participants also rearranged their juxtaposed patches more than Tabs participants. In fact, only one Tabs participant, P15, rearranged his juxtaposed patches. In contrast, 11 of the 16 Patchworks participants had at least one episode of rearranging patches on the patch strip. Of these 11 participants, they dragged and dropped a patch from one grid cell to another an average of 4.9 times ($SD = 3.6$). Comparing just the Tabs and Patchworks participants who juxtaposed at all, significantly more of the Patchworks participants rearranged their juxtaposed patches than did the Tabs participants (Fisher's exact test: $p < 0.01$).

Following from Patchworks participants' greater tendency to juxtapose patches, they also made significantly more on-screen navigations than Tabs participants (Mann–Whitney: $U = 29$, $Z = 3.71$, $p < 0.0002$). Fig. 25 illustrates this difference. On-screen navigations are highly efficient because they require moving only one's eyes (no clicking). In one extreme case, Patchworks Participant P13 made over 80% of his navigations to patches already on screen.

The Patchworks participants' high rate of on-screen navigations had a considerable influence on their lower average cost of navigation; however, it was not the only contributor. Even if on-screen navigations were excluded from our cost analysis, Patchworks participants still made significantly fewer clicks per navigation (Tabs:

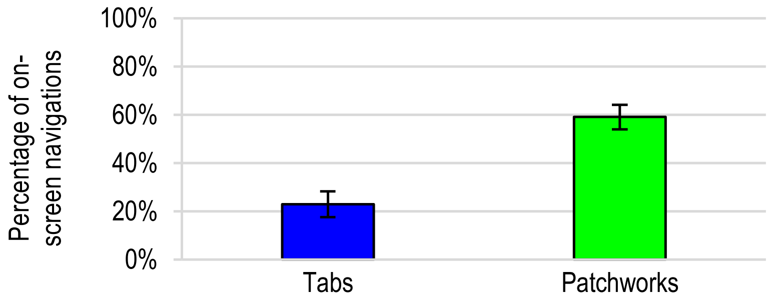


Figure 25: Patchworks participants made significantly more on-screen navigations than did Tabs participants. Whiskers denote standard error.

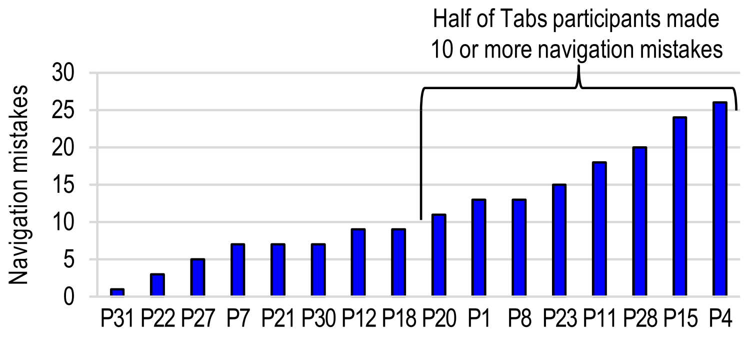


Figure 26: Tabs participants made numerous navigation mistakes in which they clicked on a tab other than the one they intended. All Tabs participants made at least one such mistake, and half of Tabs participants made 10 or more.

$M = 1.80$, $SD = 0.58$; Patchworks: $M = 1.36$, $SD = 0.23$; Mann–Whitney: $U = 53.5$, $Z = 2.79$, $p < 0.003$).

Reducing Navigation Mistakes Another reason that Patchworks participants made fewer clicks per navigation was that they made fewer navigation mistakes. Based on our qualitative analysis, no Patchworks participant ever indicated mistakenly navigating to a patch other than the one he/she intended. In contrast, Tabs participants made many navigation mistakes in which they were seeking a particular patch and clicked the wrong tab to get there. As Fig. 26 shows, all Tabs participants made at least one navigation mistake, and on average, Tabs participants made 11.75 mistakes per session ($SD = 7.28$)—roughly one every 3 minutes. Because no Patchworks participant indicated making a navigation mistake, and every Tabs one did, the statistical difference between the treatments was highly significant (Fisher's exact test: $p = 0$).

Enabling Efficient Blowup View To evaluate the goal of enabling efficient toggling between single-patch and multi-patch displays, we counted uses of Patchworks' blowup-patch feature to see whether participants used it consistently. We also checked whether the blowup view tended to complement the multi-patch view (as opposed to replacing it). In this evaluation, there was no comparison data, as the tabbed editor lacked an analogous feature.

A strong majority of Patchworks participants made use of the blowup view. All but three Patchworks participants toggled at least once, and nearly one third of all Patchworks participants blew up patches over 10 times. Moreover, the participants who used the feature did so an average of 15.5 times ($SD = 17.5$)—more than once for every 3 minutes.

For Patchworks participants, the blowup view complemented the multi-patch grid well. On average, participants did not stay in the blowup view very long during each use ($M = 33.5$ seconds, $SD = 20.9$), preferring to use it in combination with the patch grid. As a result, all Patchworks participants, except one, spent less than 50% of their sessions in the blowup view.

3.6.2.2 Results: Trends across Treatments

In addition to evaluating Patchworks' design goals, we checked for three key trends in how developers navigate and manage open patches. None of these trends were specifically addressed by Patchworks' design goals; however, they all pose relevant considerations in the design of navigation affordances for code editors.

Rapid Patch Scanning One trend we tested for was rapid scanning of patches. In particular, we wanted to see if participants tended to make their visits to patches short, and if they tended to make series of short visits indicative of scanning patches.

Based on our navigation data, all participants, regardless of treatment group, did a considerable amount of rapid scanning of patches. One key indicator of this behavior was

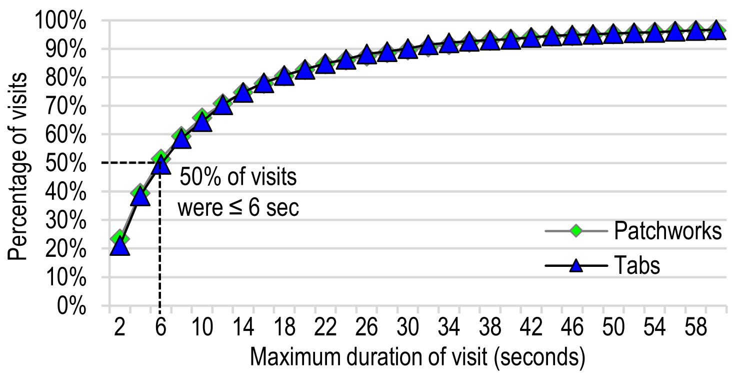


Figure 27: Participants often visited patches for very short intervals, which is consistent with quick scanning. In fact, half of visits lasted 6 seconds or less.

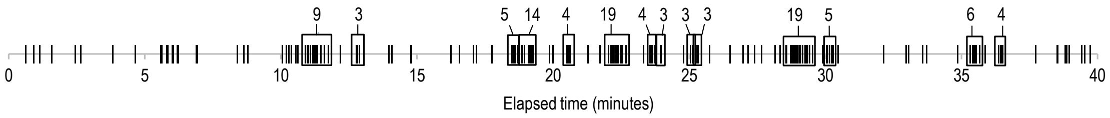


Figure 28: Navigation timeline for P17 with DBSCAN clusters highlighted and cluster size labeled.

that most of the visits to patches that the participants made were quick. As Fig. 27 shows, half of the visits that participants made to patches lasted 6 seconds or less.

To see if participants were making these short navigations in clusters, we applied DBSCAN [30], a density-based clustering algorithm commonly applied to time-series data. We allowed for up to 5 seconds between navigations and a minimum of 3 navigations to be clustered.

The DBSCAN results showed that every participant engaged in quickly scanning across multiple patches. To illustrate, Fig. 28 visualizes the DBSCAN results for Participant P17, with his 14 clusters of quick navigations spread throughout his session. Overall, 44% of all navigations that participants made were part of a quick-navigation cluster. On average, participants had 17.3 such clusters ($SD = 8.6$) during their 40-minute sessions. No participant had fewer than 4 clusters, and one participant had 34.

Cleaning Up Open Patches Another trend that we checked for was patch-cleanup behavior—the tendency to let open patches accumulate and then engage in bulk closing of

Table 4: The textual-language programmer navigation data sets we compared with our visual dataflow language programmer data.

Data Set	Participants	Task Type	Task Time	Language	Size of Code
Fritz et al. FSE'14	5 professionals, 4 CS students, 3 CS faculty	Feature modification	75 min	Java	14–53k LOC
Henley et al. VL/HCC'14	14 CS students	Feature addition	120 min	Java	3–17k LOC
Piorkowski et al. CHI'12	11 professionals	Debugging	70 min	Java	97k LOC
Piorkowski et al. ICSME'15	11 CS students	Debugging	30 min	Java	99k LOC
This Study	32 professionals (16 Tabs + 16 Patchworks)	Debugging	40 min	LabVIEW	34 VIs

the patches. To this end, we logged each time a participant closed a patch, and to detect bulk closures, we again used DBSCAN to identify clusters (10 seconds between closes, minimum of 3 closes per cluster).

Based on our data, nearly all participants engaged in cleanup, closing open patches throughout their sessions. The Tabs and Patchworks participants closed patches at similar rates: Tabs Participants averaged 14 closes per session ($SD = 6.8$) and Patchworks Participants averaged 16.1 ($SD = 8.9$). Our DBSCAN results showed that 63% of closes were part of clusters, and participants had an average of 3.4 clusters per session ($SD = 2.2$). The average number of closes in a cluster was 4.1 ($SD = 1.8$), which accounted for 69% of the open patches on average ($SD = 22$).

Navigations to Program-Output Patches A final trend we tested for was the extent to which developers navigate to program-output patches. Our LabVIEW-based editors differ from those of other popular IDEs in that they present the output (i.e., front panel) of the program under development in patches similar to the code patches. We included navigations to these program-output patches in our navigation coding, and analyzed the frequency of those navigations relative to the code ones.

Based on our data, all participants, regardless of treatment, made a considerable number of navigations to program-output patches. On average, Tabs Participants made 29.2% of their navigations to program-output patches ($SD = 7.5$) and Patchworks participants made 26.2% ($SD = 7.9$).

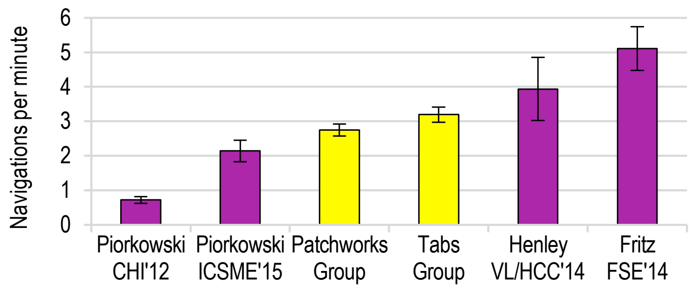


Figure 29: Our visual dataflow language developers (light bars) navigated at rates similar to prior textual language developers (dark bars). Whiskers denote standard error.

3.6.2.3 Results: Visual Dataflow versus Textual Language Developers

To compare the navigation behavior of textual language developers and visual dataflow language developers, we compared our data with results reported by four prior studies (i.e., [33, 43, 93, 94]). We focused our comparison on two key navigations traits: (1) how much the developers navigated and (2) how much the developers revisited patches. In the prior studies, the patches among which developers navigated were Java methods. For our comparison analysis, we operationalized patches as VI block diagrams, which are analogous to methods. Table 4 describes each of the prior textual-language studies.

As Figs. 29 and 30 show, the visual dataflow language developers exhibited similar navigation traits to the textual language developers from prior studies. The rates of navigation for our participants were squarely in the middle of the range of navigation rates exhibited by textual language developers from the literature (Fig. 29). Furthermore, our visual dataflow participants also revisited patches frequently—even more so than prior textual language developers (Fig. 30).

3.6.3 Discussion

3.6.4 Implications for Design: Principles

Toward our goal of design principles for code editors, we built upon our quantitative results with additional qualitative analyses of our think-aloud data. In particular, we mapped from quantitative data points to participant episodes, and examined those episodes

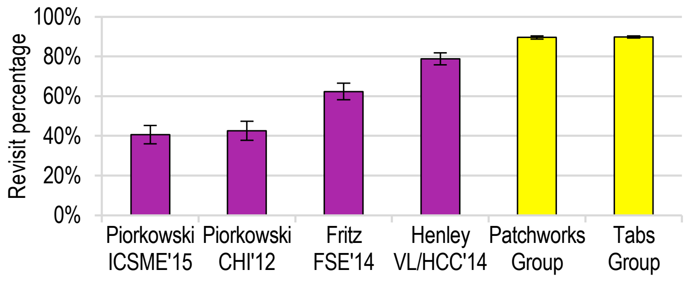


Figure 30: Like prior textual language developers (dark bars), our visual dataflow language developers (light bars) revisited patches frequently. Whiskers denote standard error.

Table 5: Candidate design principles for code editors.

Name	Definition
Juxtaposition Principle	Enable efficient (re)arranging of multiple open patches side by side on screen at a time.
Signpost Principle	Thumbnails/summaries/labels of open patches must provide sufficient information to quickly make effective navigation decisions.
Blowup Principle	Enable efficient toggling of an open patch within a multi-patch display to an enlarged, possibly single-patch, display.
Cleanup Principle	Enable efficient closing of open patches that are not currently relevant and causing clutter.
Heterogeneity Principle	Apply the above principles to all types of frequently visited patches (not only code).

in detail to help explain and expand upon our results. Informed by these quantitative and qualitative data, we inductively propose five principles, summarized in Table 5.

3.6.4.1 The Juxtaposition Principle

The Juxtaposition Principle emphasizes enabling developers to efficiently (re)arrange multiple patches side by side on screen. Although both the Patchworks and tabbed editors support juxtaposing patches, the Patchworks editor better satisfies the Juxtaposition Principle's efficiency aspect. Patchworks' fixed grid of patch cells meant that juxtaposing was simply a matter of opening patches, each in its own cell. In contrast, the tabbed editor required the developer to tediously split, position, and resize windows. As our results show, every Patchworks participant juxtaposed patches during their tasks, whereas six of the Tabs participants never juxtaposed at all.

The benefit of Patchworks' compliance with the Juxtaposition Principle was made clear by how many more navigations its users made to patches already on screen than did the tabbed-editor users (recall Fig. 25). Such on-screen navigations are highly efficient as they require only moving one's eyes.

The lack of juxtaposing by Tabs participants was not for lack of desire to do so, however:

> P15: "One useful feature might be to have the block diagram tile here [beside a front panel]... I want to see both at once."
>
> P31: Wanted to "look at a bunch of windows next to each other."

Moreover, all Tabs participants who did juxtapose expressed difficulty in doing so:

> P20: "It is difficult to have multiple VIs [patches] open at the same time and be looking at them at the same time."
>
> P21: "I don't have very many options in terms of how I can arrange these [patches]."

For example, P31 struggled for nearly 2 minutes with juxtaposing before arriving at a satisfactory configuration.

The ease of *rearranging* patches in Patchworks paid off for Patchworks participants by enabling them to set up long series of these on-screen navigations. For example, Patchworks Participant P9 rearranged his patches so that he could watch four VIs simultaneously while debugging. With this arrangement, he made 36 consecutive on-screen navigations. P3 also rearranged his patches effectively: he had three episodes of rearranging that yielded 25, 15, and 14 consecutive on-screen navigations, respectively. P5 clearly expressed his appreciation for rearranging in the Patchworks editor:

> P5: "The carousel [Patchworks] is nice because you have a lot of different options [to arrange]."

In lieu of juxtaposing, Tabs participants resorted to less-efficient clicking back and forth between tabs. For example, P7 made 164 of his 187 navigations in back-and-forth sequences. Another participant, P15, made a sequence of 27 navigations in less than 3 minutes flipping between the same two tabs. These navigation actions could have been eliminated if the patches were juxtaposed on screen.

3.6.4.2 The Signpost Principle

The Signpost Principle emphasizes the importance of having small, concise representations or summaries of open patches that provide sufficient information to quickly make effective navigation decisions. Tabs were the main form of such representations used in the tabbed editor. In contrast, Patchworks used code thumbnails to summarize the contents of VIs in the bird's eye view (recall Fig. 23b).

In many circumstances, tabs did not convey sufficient information for participants to make effective navigation decisions, and as a consequence Tabs participants made numerous navigation mistakes (recall Fig. 26). The problems with tabs were further reinforced in numerous episodes by Tabs participants. For example, P4 had two episodes of "losing" tabs.

P4: "Where did I put it? ... I lost it."

P4: "Way too many things open at this point ... Where did it go? ... Nope, I lost it. Now I can't go back because I don't know where I was."

Tabs Participant P18 further expounded on the difficulty of having too many tabs:

P18: "If I have too many tabs, same issue if I have too many windows, it makes it really small and you can't see the name."

In contrast, Patchworks participants expressed positive impressions of the thumbnail patches in the bird's eye view.

P25: "You can zoom out, and you can pan out to see what all you actually have open, so that's really helpful, so it resolves the getting-lost-in-a-thousand-VIs thing."

P2: "I do like that you can zoom out and see everything that you are looking at... Definitely helps."

Two Patchworks participants even drew direct comparisons between the tabbed and the Patchworks designs:

> P5: "Hopefully you name your VIs well so that you can re-navigate [using tabs], but I think with the carousel [Patchworks] it would definitely be an improvement once you got used to it"

> P6: "This is excellent. I like this a lot better than having to go through the list of VIs you have open… It is definitely easier to navigate."

3.6.4.3 The Blowup Principle

The Blowup Principle emphasizes providing a developer the ability to efficiently toggle a patch in a multi-patch display into an enlarged "blowup" view. Only the Patchworks editor contained a feature for toggling to a blowup view. Thus, the empirical support for this principle came entirely from the Patchworks participants.

As our empirical results showed, Patchworks participants made substantial use of the blowup-patch feature in tandem with the multi-patch display of the patch grid. For example, P26 toggled into the blowup view 40 times during his session, yet he spent only 26% of his session in this view. Likewise, another participant, P2, commented explicitly on liking to toggle back and forth between the patch grid and blowup view:

> P2: "Easy to zoom in, zoom out, bring it to be full sized [referring to the blowup view] … It is nice to go back and forth [toggles and untoggles the blowup view repeatedly]."

One popular strategy was to use the blowup view for large, complex VIs. For instance, P26 repeatedly toggled the complex ProcessInput VI into blowup mode, while using the multi-patch display to visit smaller VIs. Similarly, P14 also used the blowup view when she needed to understand complex VIs. She toggled the complex Main VI into a blowup view several times during her session:

> P14: "I'm going to make it 100% of my screen so I can read into it… I'm trying to figure out what is going on in this Main VI."

3.6.4.4 The Cleanup Principle

The Cleanup Principle emphasizes helping developers to efficiently close open patches that are cluttering the workspace and are not currently relevant to what the developer is working on. Implicit in this principle is the idea that relevant patches should not be closed along with the irrelevant ones. Both the tabbed and Patchworks editors had only rudimentary features for closing individual open patches (clicking a button on the tab or patch-cell label, respectively). Thus, for all our participants, regardless of treatment, cleaning up open patches meant closing each selected patch individually.

As our results showed, participants tended to let open patches pile up, and then perform cleanup on groups of patches. For example, P4 became overwhelmed by the number of open patches, and began closing patches:

P4: "Too many things open right now. I just need to clean it up a bit."

The importance of closing only irrelevant patches was made clear by episodes where participants inadvertently closed relevant ones. If participants closed all of their patches, they almost always began reopening a subset of the patches; however, they often had difficulty finding the right patches to reopen. For example, Tabs Participant P20 had started cleaning up, rapidly closing tabs, but then realized he had accidentally closed the one he wanted:

P20: "I think I closed the VI by accident."

He then re-opened four of the patches he had just closed in a tedious search for the relevant patch. Thus, helping developers to perform cleanup *without closing relevant patches* was an important consideration in the Cleanup Principle.

3.6.4.5 The Heterogeneity Principle

The Heterogeneity Principle emphasizes facilitating developer navigation among a variety of patch types—beyond only code. In our study, both the tabbed and Patchworks editors treated program output (LabVIEW front panels) as first-class patches similar to the source code (block diagrams). Regardless of the editor used, participants made over a quarter of their navigations to those non-code patches.

A debugging strategy employed by all the Patchworks participants but one was to juxtapose code patches with program-output patches to observe the effect of code changes on runtime values. For example, P9 juxtaposed three block diagrams (code patches) and one front panel (program-output patch). He then repeatedly interacted with the front panel and watched the dataflow propagate through each of the block diagrams in an attempt to determine the cause of an incorrect value. Other Patchworks participants discussed similar strategies:

P13: "I placed them side by side to see the front panel and diagram, mostly for the Main VI, to see where inputs were going through."

P16: "I usually had the Main front panel and diagram up, and one or two other diagrams up."

Interestingly, this juxtaposing approach was popular among Tabs participants as well. Although Tab participants rarely juxtaposed, every one of the ten who did, did so to place a code patch alongside a program-output patch. For example, at first, P28 had his tabbed editor in the default single-patch display. He repeatedly switched back and forth between a block diagram (code) tab and a front-panel (program output) tab to observe how values were affected. He repeated this tedious process for nearly 6 minutes before becoming frustrated and reconfiguring his editor to juxtapose the two patches, enabling him to more efficiently navigate between them. Such empirical observations clearly motivate the Heterogeneity Principle's emphasis on non-code patches.

3.6.5 Triangulation with Prior Research

To enhance the credibility and validity of our candidate principles, we triangulated with prior research results. Although the principles were inspired by our empirical study, support for each one could also be found scattered throughout the literature.

Regarding the Juxtaposition Principle, there is support for the idea that developers want to juxtapose code and doing so enhances their ability to navigate. For example, one study found that developers expressed wanting to juxtapose code patches [9]; however, similar to our Tabs participants, they rarely do because of the tediousness of juxtaposing patches in tabbed editors, such as Eclipse [9, 11, 62]. In our previous study comparing editors with single-patch and multi-patch displays (Eclipse versus Code Bubbles and Patchworks), participants who used the multi-patch editors navigated significantly faster than those who used the single-patch editor—despite being experienced with the single-patch editor and unfamiliar with the multi-patch ones [39].

Furthermore, the importance of being able to rearrange patches efficiently in multi-patch displays has also been well motivated. For example, Plumlee and Ware [100] have argued convincingly about the inefficiencies of window management. In particular, they found that sizing and positioning windows on screen required users to devote inordinate amounts of time and attention. Efficiency of rearranging patches is of particular importance to developers, because studies have found that their goals change rapidly during programming tasks [69, 93], thus, suggesting that they will need to rearrange often. However, in modern tabbed code editors, such rearranging is tedious, and as a consequence, developers rarely do so [11, 62].

Prior research on recognition versus recall has clear implications for the Signpost Principle. It is well known that, in accessing a memory, recognition based on cues in the environment requires humans to do less mental processing than does recalling without such environmental support (e.g., [1]). The Signpost Principle is concerned with being able to quickly remember sufficient information about patches to make effective navigation decisions. Thus, it emphasizes providing developers with effective opportunities for recognition. Consistent with our study results, prior works have critiqued tabbed editors for the lack of recognition support offered by tabs. For example, they have found tabs to not be representative of their associated patches, and thus, disruptive to associative memory [86, 91]. In fact, one study showed that developers frequently had to check the content of patches associated with tabs because they could not infer the content from the labels on the tabs [106]. To overcome these issues, thumbnails, similar to those provided by the bird's eye view in the Patchworks editor, have been investigated. For example, thumbnails have been found to effectively support information retrieval by end users [103] and to help developers navigate large code files and code bases [23]. Recently, such thumbnails have been incorporated into several popular code editors, including Visual Studio and Sublime Text.

In contrast to the above principles, we are unaware of any existing research related to the Blowup Principle; however, blowup-patch features have recently begun to appear in popular code editors. For example, both Eclipse and Visual Studio allow developers to toggle editor patches to a blowup view. However, these editors are still based on single-patch tabbed editors. In their cases, the main benefit of the blowup view is to overcome the limited screen space afforded to the editor due to a multitude of other panels present in the development environment (e.g., package explorer, console output, outline view, etc.). In fact, the tabbed version of LabVIEW also had many views; however, its designers did not include a blowup feature for editor patches, suggesting that they could have benefited from the Blowup Principle.

Prior research has confirmed the Cleanup Principle, and in particular, the importance of helping developers clean up only irrelevant code. For example, studies have shown that developers often close all of their open patches to proceed from a clean slate [25, 90]. Many modern code editors provide "close all" features; however, wiping all tabs has also been found to be problematic, because developers actually want to keep some of the patches open. For example, one study found that developers wasted an average of 60 seconds to reopen the patches they wanted after closing too many of them [58]. Others have similarly reported on the costliness of recovering such state after suspending a task due to the loss of contextual cues [47]. However, despite this evidence, tools have yet to provide effective support for selective cleanup.

Support for the Heterogeneity Principle from prior research has mainly focused on the importance of facilitating navigation between code and program output. A primary benefit of enabling such navigation is that it potentially shortens a developers' feedback loop of running the application, interacting with it, and observing the runtime behavior. For example, the Whyline programming environment allows a developer to ask questions about a program's output and provides direct links to the relevant code [61]. Another tool, the Theseus programming environment, annotates JavaScript source code with runtime information, thus, combining program code and output into a single patch [70]. Still other tools have provided navigational links between code and other types of artifacts, such as emails and bug reports [113] and related code examples from the web [13]. Such tools further illustrate the potential benefits of considering heterogenous patch types in the design of code editors.

Chapter 4

Navigating Versions of Code

They always say time changes things, but you actually have to change them yourself.

—Andy Warhol

To understand one's working set, the developer may need to see information from previous versions of the code. For example, during a code-change task, the developer may need to refer back to the original code to remember how the code worked prior to the current changes. This can be illustrated by a developer making a complex refactoring to their code that involves removing several variable declarations across multiple functions. Doing so would cause errors at all of the locations that those variables are referenced, and thus the developer may need to see the original declarations in order to finish the change, such as the original data type or initial value. Researchers have began studying this behavior of navigating to previous versions of code with one recent study of developers that found that the developers needed to view previous versions to get more context about how the code evolved [109]. Not only are these navigations beneficial and sometimes neccessary for a developer to make a change, but they happen frequently as well. In fact, a

Portions of this chapter appear in [40].

study found that the developers navigated to a previous version of their code once every six minutes, on average [123].

However, code editors provide few affordances for navigating to previous versions of code. The most commonly used feature is undo and redo, which allows the developer to revert to a previous state of the code. Despite the undo feature being ubiquitous, it is problematic since it means the developer can not view both versions simultaneously. Furthermore, the developer loses the current version of code when using undo, unless they redo all of the actions back to that state. Another solution is to use version control systems, which often have comparison views. Again, despite version control systems being standard tools in software development, they come with a number of limitations. For example, they only allow for comparing the code to versions that were explicitly checked-in, and not any intermediate versions.

In particular, visual programming environments have received little or no tool support for developers to navigate to the information that they may need during code changes. The tediousness and error-proneness of making code changes, such as refactorings, has been well recognized in textual programming environments [55, 75, 76]. Textual programming environments support code changes in various ways, including autocomplete (e.g., Calcite [73] and Graphite [82]), automated refactoring (e.g., ReBA [28] and BeneFactor [34]), and code smell detection (e.g., PMD and DETEX [72]). Although a few researchers have begun investigating code-change support for visual programming environments (e.g., SDPA [17]), the work is still in the early stages, and has yet to see widespread adoption in practice. In our own formative interviews of LabVIEW developers (Section 4.1.1), many developers confirmed that making code changes was currently difficult and problematic.

In a formative user study of LabVIEW developers engaged in refactoring (Section 4.1.2), we found that the developers encountered considerable challenges retaining information from recent versions of code during *rewiring*. In visual dataflow languages, rewiring may involve changing existing wires to connect to different box input/outputs, or introducing new wires into existing code. The ability to rewire effectively is of critical importance in visual dataflow languages, because nearly every code change requires some manipulation of wires. However, during rewiring activities, participants consistently forgot what wires represented and introduced bugs by mixing up wires.

To address these problems with needing information from previous versions of code during code-change tasks, we propose *Yestercode*, a novel tool concept for visual dataflow programming environments (Section 4.2). Two key goals of Yestercode are to enable developers to efficiently access past versions of their code, and to juxtapose a reference copy of an older version with their current code by viewing them side by side. Toward achieving these goals, we made several key design decisions, including transparent automated version logging, tight code-editor integration (e.g., copy and pasting capability from reference-copy to code editor), and annotations to aid comprehension of the differences between the reference copy and current code.

In this chapter, we present an initial prototype of Yestercode for LabVIEW as well as report the findings from our formative studies and a user-study evaluation using the prototype (Section 4.3.1). In particular, our user study involved 14 professional LabVIEW developers, and addressed the following research questions:

- RQ1: Do developers using Yestercode introduce fewer bugs during change tasks?
- RQ2: Does using Yestercode affect the time it takes developers to complete change tasks (for better or for worse)?
- RQ3: Do developers using Yestercode have lower cognitive load during change tasks?
- RQ4: Do developers have favorable opinions of Yestercode?

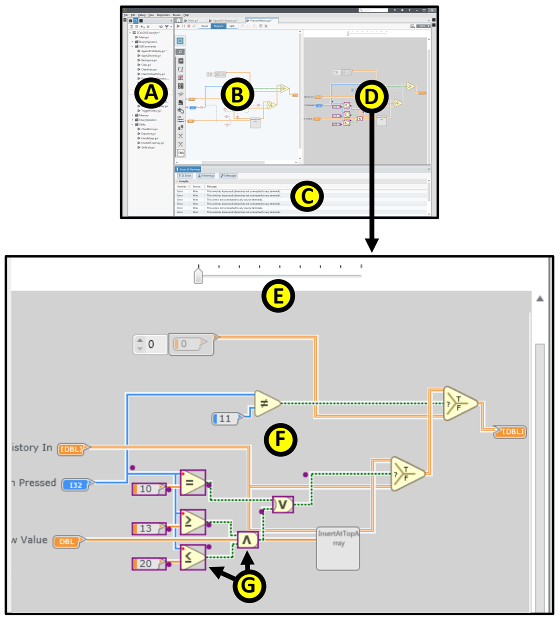


Figure 31: Yestercode-extended LabVIEW development environment. The standard LabVIEW features include a project explorer (A), a code editor (B), and an error listing (C). Yestercode extends the development environment with an additional view (D) that allows navigating the version history (E), which displays a previous version of the code (F) with annotations showing the difference from the current version (e.g., G).

4.1 Formative Investigations

4.1.1 User Interviews

As a first step in understanding the barriers that visual dataflow developers face, we conducted a series of formative interviews. In particular, we held 54 semi-structured interviews with engineers and managers at National Instruments (the maker of LabVIEW). Each interview lasted between 15 and 30 minutes. Our participants consisted of 36 Application Engineers (who provide support to LabVIEW customers), 7 Software Engineers, 4 Hardware Engineers, 4 Group Managers, and 3 UX Engineers. We guided the discussion with questions regarding problems they have with the LabVIEW IDE, problems customers run into, and potential tool support to address these problems.

A key trend from the interviews was that participants found modifying existing LabVIEW code a challenge. Participants often said that either they or customers will go to great lengths to avoid modifying existing code. For example, they often “start from a blank slate” and rewrite code rather than directly modifying code they had previously written. One manager even said that, for her team, modifying code was a “million dollar problem,” because several of her engineers waste effort each day rewriting similar but different programs, and then ultimately “throw away the code.”

When we prompted participants about *why* modifying existing code was a challenge, their responses provided only a few insights. Most often, they offered vague answers, like that it was “just easier” to rewrite code. A few participants said that it was difficult to remember what specific code does, and that it was too time consuming and tedious to make substantial changes. Three engineers cited the general lack of explicit textual identifiers in visual dataflow code as making it particularly difficult to recall what wires “mean”.

4.1.2 User Study

To better understand why developers said that it is easier to rewrite code rather than modify it in LabVIEW, we conducted an exploratory user study of LabVIEW developers engaged in change tasks. The study involved 6 participants: 3 graduate students with 1–2.5 years of LabVIEW experience, 1 Hardware Engineer with 4 years of LabVIEW experience, and 2 Software Engineers with 16 years of LabVIEW experience each. Each participant took part in a 60-minute programming session in which he/she made improvements to an existing LabVIEW application that we provided. To get each participant started, we gave him/her three change tasks. Once the participant completed those tasks, we asked him/her to make additional improvements as he/she saw fit. If the participant became stuck, we suggested additional change tasks. At the end of the session, the participant took part in a 30-minute semi-structured interview in which we played back a video of his/her task performance, and asked questions about his/her goals, barriers, and strategies.

A key trend in our user study was that participants exhibited difficulty rewiring components correctly. Many participants made comments about how tedious rewiring a portion of code was. For example, one participant began rewiring, and quickly exclaimed “This is going to take all day!” Rewiring was so difficult that all 6 participants accidentally introduced at least one bug into the program while rewiring. Moreover, 4 of the 6 participants started but then abandoned a code change that they determined was too difficult.

To cope with the challenges of rewiring code, participants commonly used one of three code-change strategies. One strategy was to use the editor’s Undo/Redo commands in quick succession to quickly look at older versions of the code being rewired and then return to the current version. Another strategy was, prior to rewiring, to copy and paste the code into the same editor window so as to keep the copy visible as a reference while they rewired the original code. The final strategy was to take screenshots of the code prior to modifying it, and then to use the screenshot as a reference while rewiring. Interestingly, all three of these strategies shared the common theme of enabling the developers to refer to older versions of the code as they performed rewirings.

4.2 Tool Design: Yestercode

Based on the code-change barriers and coping strategies revealed by our formative investigation, we designed the Yestercode tool for visual dataflow programming environments. In particular, Yestercode aims to help with rewiring changes by enabling the developer to efficiently refer to a prior version of the code while making changes. We hypothesize that by having a readily available reference version of the code, the developer will exert less mental effort recalling the meaning of the wires and other elements, and thus, will make fewer mistakes resulting from memory failures. Fig. 31 illustrates our Yestercode design (instantiated as an extension to the LabVIEW IDE). We arrived at this design via an iterative process of collecting and incorporating feedback from professional LabVIEW developers for our evolving design.

As a result of our design process, we identified five key principles for the design of Yestercode. In particular, our design should...

- Transparently record the version history of a block diagram as the developer edits it.
- Enable efficient navigation of the version history to recover reference versions of the code.
- Enable juxtaposition of the current block diagram with an older reference version of that diagram.
- Provide visual cues in the older reference version of the block diagram to call out differences with the current version.
- Provide tight integration between the reference-version view and the code editor, for example, such that code in the reference version may be copied and pasted into the editor.

In the remainder of this section, we highlight the features of our Yestercode design that satisfy these principles.

4.2.1 Transparent Recording of Version History

As a developer edits a VI in the code editor (Fig. 31-B, Yestercode automatically records the changes made. To keep the number changes recorded manageable, Yestercode does not record those that concern only the code element's spatial location (e.g., dragging a node to a different location onscreen). This automated recording is transparent to the developer in that it is completely automatic, requiring no explicit instructions from the user.

Based on our formative user study, the design decision to record versions transparently was preferable to having the developer explicitly initiate the saving of versions. Participants in the user study often did not realize they wanted an older version of the code until they were well into making a change and became stuck. Thus, forcing developers to explicitly save older versions would introduce a problem of *premature commitment* in which they must make a decision (whether or not to save a version) before they are able to decide (whether they will want to refer to that version) [36].

4.2.2 Efficient Navigation of Version History

To enable efficient navigation of the recorded versions, Yestercode provides a history slider (Fig. 31-E). Sliding the shuttle to the left, steps back in time through the version history. To assist the developer in finding the appropriate version, Yestercode displays the currently selected version just below the slider (Fig. 31-F). To choose an older version of the code to use as the reference, the developer needs simply to slide the shuttle to the appropriate version, and leave it there.

4.2.3 Juxtaposition of Current and Older Versions

To enable the developer to place an older reference version of the code side by side with the current version, Yestercode splits the editor pane to situate a reference-version view (Fig. 31-D) directly next to the code editor (Fig. 31-B). This juxtaposition of editor and reference version enables the developer to quickly refer back and forth between them, a behavior common to participants in our formative user study. For times when the developer does not need the reference version, the view can be collapsed and hidden.

4.2.4 Visual Cues Denoting Version Differences

To facilitate comparing the features of the reference version of the code with the version in the editor, Yestercode provides visual cues in the reference-version view to indicate boxes and wires that have been removed or updated in the current version (e.g., Fig. 31-G). In particular, a purple border around a box or a purple dot on a wire indicates that the box/wire has been deleted or modified with respect to the current version in the editor. Although Yestercode provides no explicit cues to call out elements that have been added in the current version, based on our experience, it is often the case that existing elements are changed to incorporate new elements, providing cues that implicitly call out the additions.

4.2.5 Tight Integration with Editor

To provide tight integration with the code editor, the Yestercode reference-version view is designed to be an editor extension (in our LabVIEW case, sharing the same graphical pane as the editor). In our prototype, tight editor integration offered several benefits. Aside from being read-only, the appearance of and user interactions with the reference-version view were identical to those of the code editor. Chief among these interactions was the ability to copy and paste code from the reference-version view to the code editor. Additionally, other editor features were available in the reference-version view, such as contextual help provided by hovering the mouse pointer over elements in the block diagram, and selection and highlighting of elements in the block diagram to aid reading.

4.3 Evaluation: User Study

To evaluate Yestercode, we conducted a lab study of LabVIEW developers engaged in code-change tasks. It was of within-subjects design, were each participant received the standard LabVIEW IDE and the Yestercode-extended LabVIEW IDE. We sought to answer the following research questions:

- Do developers using Yestercode introduce fewer bugs?
- Does using Yestercode affect the time it takes developers to complete tasks?
- Do developers using Yestercode have lower cognitive load?
- Do developers have favorable opinions of Yestercode?

4.3.1 Method

4.3.1.1 Participants

Our participants consisted of 14 professional LabVIEW developers (11 male, 3 female) from a large technology company. They reported, on average, 4.36 years of programming experience ($SD = 1.74$) and 1.98 years of LabVIEW experience ($SD = 1.89$). All participants programmed in LabVIEW as part of their daily work.

4.3.1.2 Code Base and Change Tasks

For our study, the participants performed code-change tasks on a calculator application written in LabVIEW and composed of 34 VIs. The application was based on a publicly available sample application[1] written in an old version of LabVIEW. We ported this sample application to the current version of LabVIEW, and modified it to follow the official LabVIEW development guidelines [79]. The code was fully functional and did not contain any known bugs.

Each participant performed six code-change tasks on the calculator application. Each task was based on one of Fowler's refactorings [32]. The tasks were divided into two sequences of three (one sequence for each treatment). For each three-task sequence, the first task required an *Inline Method* refactoring, the second required an *Extract Method* refactoring, and the third required an *Introduce Parameter Object* refactoring. In our LabVIEW context, the Inline Method tasks (IM1 and IM2) each involved replacing a call to a VI with the contents the VI itself; the Extract Method tasks (EM1 and EM2) each involved pulling out part of a VI and making the part its own VI; and the Introduce Parameter Object tasks (IPO1 and IPO2) each involved replacing a set of multiple wires coming out of a block with a single wire that bundles the output values together.

[1] `http://www.ni.com/example/30779/en/`

4.3.1.3 Procedure

Each participant took part in an individual session that lasted approximately 1 hour. All participants began the session by filling out a background questionnaire and receiving an introduction to the calculator-application code. Next, each participant completed a first 3-task sequence with one treatment and a second 3-task sequence with the other treatment. Seven randomly selected participants used the control IDE first, and the other seven used the Yestercode-extended IDE first. Each 3-task sequence began with an introduction to the IDE that the participant would be using for that sequence. Next, the participant performed the 3-task sequence in order, completing each task before beginning the next. We asked each participant to "think aloud" as he/she worked. Once the participant had finished each task, he/she completed a cognitive-load questionnaire (details below). At the end of the session, all participants completed an opinion questionnaire regarding Yestercode, and took part in a semi-structured interview in which they discussed any issues that they had while working.

4.3.1.4 Data Collection

The data collected for the study comprised screen-capture video and audio of the participants as well as their questionnaire responses. For the cognitive load questionnaire, we used a well-validated instrument based on Cognitive Load Theory [83]. The instrument features a 7-point Likert question that measures a person's overall cognitive load during a task. Prior work showed that the instrument does not interfere with task performance, is sensitive to small differences in workload, and is reliable [84]. Moreover, the instrument has been shown to highly correlate with more-complex self-reporting instruments (e.g., NASA TLX) [119] as well as with physiological sensors (e.g., heart rate) [84].

4.3.1.5 Limitations

Our user study has several limitations inherent to laboratory studies of developers. First, our sample of visual dataflow developers and the tasks in our study may not

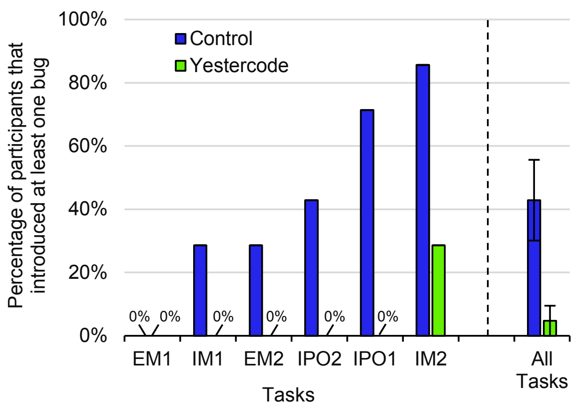


Figure 32: Yestercode users introduced substantially fewer bugs than control users (smaller bars are better). Whiskers denote standard error.

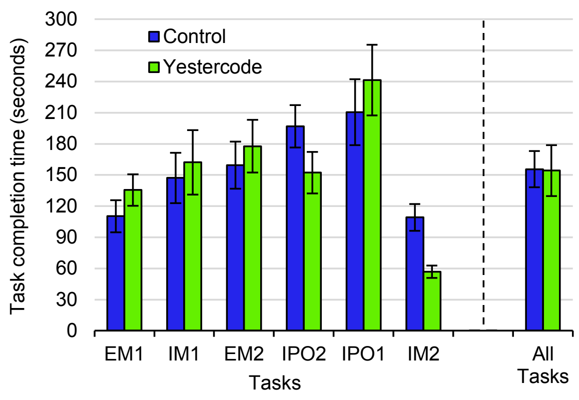


Figure 33: Overall, Yestercode users and control users did not differ in time taken to complete tasks. Whiskers denote standard error.

represent all such developers; however, we recruited professionals from a large tech company to increase the likelihood that they are representative of other professional developers. Second, the code base was small, but we based it on an open source project in an effort to make it more representative of actual projects. Third, the tasks, although based on common refactorings from the literature, were relatively short, and may not be representative of more complex tasks. Fourth, reactivity effects (e.g., participants consciously or unconsciously trying to please the researchers) may have occurred; however, we tried to minimize them by presenting the two versions of LabVIEW as possible design alternatives for a new release, and did not disclose that the Yestercode version was our invention. Finally, order effects of the tool and tasks may have affected our observations, but we counterbalanced the tool order to control for this effect with respect to our statistical tests.

4.3.2 Results

All 14 of our LabVIEW developers completed all six code-change tasks. Thus, for each task, seven participants completed the task using the control IDE and seven using the Yestercode-extended IDE.

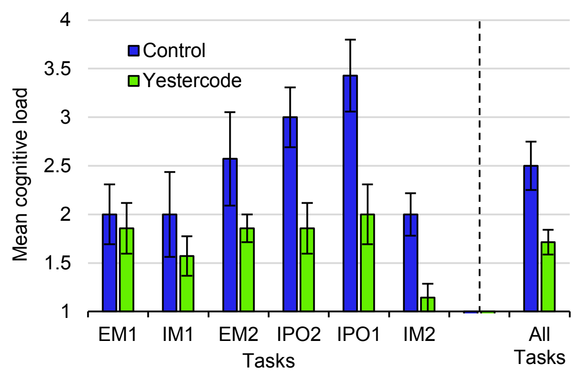


Figure 34: Yestercode users reported substantially lower cognitive load than control users (smaller bars are better). Whiskers denote standard error.

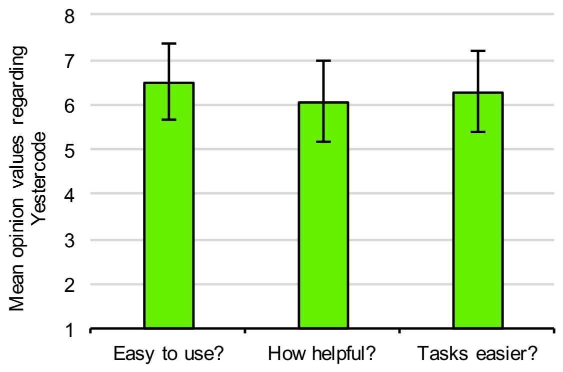


Figure 35: Participant responses were highly favorable on the Yestercode opinion questionnaire (Likert scale; 1 is negative, 7 is positive, 4 is neutral). Whiskers denote standard error.

4.3.2.1 RQ1 Results: Bugs Introduced

As Fig. 32 shows, participants using the Yestercode-extended IDE introduced considerably fewer bugs than those using the control IDE. In fact, Yestercode users did not introduce any bugs, except for one task (IM2). In contrast, control users introduced bugs during every task, but one (EM1). Indeed, the results of a Mann–Whitney *U* test revealed that Yestercode users introduced significantly fewer bugs than control users ($U = 31.5$, $Z = 2.3$, $p < 0.05$).

4.3.2.2 RQ2 Results: Time on Task

As Fig. 33 shows, Yestercode users completed tasks in similar amounts of time to that of control users. The task time differed by less than 10% for three of the tasks between the treatments. However, control users took 92% longer than Yestercode users on IM2, which was also the task with the highest rate of bugs. Averaging across all tasks, the time taken to complete the tasks did not differ significantly between the control and Yestercode treatments.

4.3.2.3 RQ3 Results: Cognitive Load

As Fig. 34 shows, Yestercode users reported considerably lower cognitive loads than control users. In fact, Yestercode users reported a lower cognitive load than control users for every task, on average. For the task that participants introduced the most bugs, IM2, control users reported a cognitive load that was 7 times higher than Yestercode users, on average. The results of a Mann–Whitney U test showed that Yestercode users reported a significantly lower cognitive load than control users ($U = 34.5$, $Z = 2.7$, $p < 0.01$).

4.3.2.4 RQ4 Results: Opinions of the Participants

To assess participants' opinions of Yestercode, each participant responded to the following questions at the end of the session (7-point Likert scale; "this tool" referred to Yestercode):

- How difficult or easy was this tool to use?
- How helpful was this tool?
- The tasks with this tool were more difficult or easier?

As Fig. 35 shows, participants reported highly positive opinions of Yestercode on the opinion questionnaire. None of the 14 participants responded negatively to any question, and only two participants gave neutral responses (one regarding ease of use and the other regarding whether Yestercode made the tasks easier). Moreover, all participants reported finding Yestercode helpful to some degree.

4.3.3 Discussion

Overall, the quantitative results of our Yestercode evaluation were highly favorable. Yestercode significantly reduced users perceived cognitive load (Section 4.3.2.3). Moreover, Yestercode users took roughly the same amount of time on tasks as control users (Section 4.3.2.2), but introduced significantly fewer bugs in their solutions (Section 4.3.2.1). Finally, the participants generally scored Yestercode highly on the opinion questionnaire, with no one giving a negative score on any question (Section 4.3.2.4). To delve deeper into our results, we analyzed our data for qualitative evidence that helps explain our outcomes.

4.3.3.1 When participants got stuck, they turned to Yestercode

Yestercode often helped participants when they got stuck by saving them from having to back out of their changes by undoing. For example, while working on Task IPO1, P4 deleted a group of wires that he was going to rewire. However, having done so, he realized that he no longer knew what they were supposed to be wired to. That is when he turned to Yestercode:

> P4: "Oh! I have history! So I deleted all the wires inside the True Case. So instead of Ctrl-Zing, I am going to look at the history."

Similarly, P9 discussed the importance of having Yestercode to help when getting stuck:

> P9: "That is the hardest thing when you're refactoring. You delete a bunch of stuff, then you're, like, where did all of it go?"

Other participants echoed P4's sentiment that Yestercode was preferable to using Undo to get back lost information:

> P3: "With the history [Yestercode], it was great because I could just click through and I didn't even have to do all those undos."

> P6: "When I had the history feature, I never had to undo anything."

4.3.3.2 Yestercode reduced effort and tedium

One possible reason that participants expressed preferring Yestercode to using Undo was that Undo was tedious and labor intensive by comparison. For example, when P12 got stuck while using the control IDE, he first looked for a feature like Yestercode. He could not remember where a particular wire went, so he asked the researcher running the session if the control IDE had any capabilities similar to those of Yestercode. After being instructed it did not, he resorted to tediously performing over 15 undo actions to see where the wire originally went. He then clicked Redo another 15 times until he returned to his latest version.

Other participants also complained of having to perform time-consuming and tedious activities in the control IDE to access and manage older versions of the code. For example, P8 expressed his frustration about having to undo his changes:

> P8: "No, I don't want to go back to the way it was. I just want to *see* the way it was! I don't want to undo all of my changes just to see it."

Similarly, P10 and P13 described their tedious processes of switching between tabs to see code so that they could perform code changes:

> P10: "This was harder to do because I didn't have the history tool [Yestercode]. I had to go back and forth a lot."

> P13: "I usually use copy paste when I'm making a change or I'll flip back and forth between two different tabs if I'm moving code."

P2 also expressed preferring the juxtaposition that Yestercode enabled over switching between tabs:

> P2: "I think the way it is built, it is over here. It isn't intrusive. You can make it smaller. It is very accessible. It isn't like I have to open up another tab."

To avoid switching between tabs and performing many undo/redo actions, some users of the control IDE took screenshots of the code to use as a reference:

> P9: "I use the screenshot tool in Windows to do that for me, so I will take a snapshot, then I will move the window over and edit while I look at it."

> P12: "For that first task without history [Yestercode], I had to do a lot of undo and redo just to check. For the next task, I took a screenshot instead."

However, as P7 pointed out, having Yestercode's reference-version view reduced much of the window management that would need to be performed to juxtapose a screenshot:

P7: “With the history [Yestercode], I don’t even have to move anything around. I can just see where it goes… It was way easier with this [Yestercode].”

4.3.3.3 Participants leveraged Yestercode for verification

One possible reason that Yestercode users produced significantly fewer bugs in their task outcomes was that a number of them used Yestercode to verify their code changes, often at the end of their tasks. One such verification episode was especially fruitful for P10. She believed that she had completed the task but wanted to verify it:

P10: “I could look at this to make sure they’re all wired correctly. This is where this comes in handy, for sure.”

By performing this verification, she found that several of her wires were incorrect. She then deleted all of the wires, and quickly redid them one by one, checking Yestercode before each change. Similarly, several other participants commented on the utility of Yestercode for verifying correctness:

P12: “This is a good use of the history [Yestercode] where I can go back and make sure something didn’t get disconnected. Did all of these wires go to the right place?”

P2: “Even when you don’t mess up, it is good for validation.”

This tendency to use Yestercode for verification may explain why Yestercode users had similar task times as users of the control IDE. Because Yestercode made it convenient to verify changes, the Yestercode users spent additional time doing so, and also caught more bugs in the process. A similar phenomenon was observed by Burg et al. for their tool, Timelapse [14]: participants used the tool repeatedly, which was viewed as beneficial, yet their task time stayed the same as participants without the tool.

It remains an open question, however, whether Yestercode users had fewer bugs in their task outcomes mainly because of this verification behavior or mainly because they created fewer bugs to begin with. For example, although using Yestercode for verification may have helped some users spot bugs, such use did not guarantee success. There was one instance where a Yestercode participant introduced a bug that went unnoticed, even after using Yestercode to verify his change. In particular, P3 introduced a bug during Task IM2. After performing a code change, he spent nearly a minute reviewing his code using Yestercode. However, he failed to notice that he had accidentally swapped two wires. In sum, although verifying code changes with Yestercode could not catch all bugs, it demonstrably enabled participants to catch some, contributing (at least in part) to the Yestercode users' lower bug-introduction rate.

4.3.3.4 Participants missed Yestercode when it was gone

Participants often expressed wishing that they had Yestercode on tasks for which they were assigned the control IDE. For example, while using the control IDE, P2 became confused while performing a rewiring during Task IPO2:

> P2: "This is going to be a mess."

As a result, he undid all his work and laboriously repeated it. Later, when he was introduced to the features of Yestercode, he lamented the time he could have saved earlier if he had had the tool:

> P2: "In the last one [task], I was deleting wires and couldn't remember what they were. I had to undo everything."

Other participants also made comments on how helpful Yestercode would have been on tasks they were made to perform with the control IDE:

> P7: "This [Yestercode] would have been helpful on the first tasks."

> P5: "On the first three tasks, I was completely deleting everything and it would have been nice to just look to my right [at Yestercode]."

> P8: "This one [later task] was more difficult, because I didn't have the history window [Yestercode]. On the other one [earlier task], I knew I could just go back and rely on that. With this one, I had to plan everything, or else I would get jammed."

P13: “This [Yestercode] would have been helpful on the previous exercises to see where the wires were and where they go.”

4.4 Related Work

In prior work, developers have needed access to older versions of code for a variety of reasons. Researchers have studied developers in the contexts of backtracking to a previous version of code [123], aborting a refactoring [111], and exploring the design space or variations of a program [63, 109]. Our motivation differs from that of these works in that our purpose in storing and retrieving older versions is to provide the developer with a reference while making changes to the code.

The two most prevalent technologies for storing and retrieving older versions of code are version control systems and undo features. Version control systems (VCSs) are widely used in professional software development for managing versions of source code and facilitating collaborative development. For example, Git is one such VCS that has been growing in popularity, with one website reporting that there are 266,781 public Git repositories[2]. However, most VCSs require users to explicitly perform commit actions when they want to save a revision of their code. As mentioned in Section 4.2, in our context of code-change support, this user interaction introduces a premature commitment issue [36]: the developer must know ahead of time which versions he/she will need later, and thus, which to versions to save using their VCS.

Using undo features are also common for backtracking to a previous version of code. This invaluable feature is common in many applications and allows a user to change his/her code file by a single step in a linear fashion [7]. A longitudinal study found that developers backtracked more than 10 times per hour using undo features or manually changing the code [123]. Even our formative user study participants would often use undo to get back to a previous reference version of their code. However, in doing so, they would

[2]https://www.openhub.net/repositories/compare

temporarily give up their current version of the code in order to view a previous version, and they ran the risk of losing their current version permanently if they accidentally made a change after performing undo. Researchers have proposed a number of approaches to address this problem, including ones based on a tree-based history (e.g., [114]) and selective undo (e.g., GINA [6], Amulet [77], and Azurite [124]); however, none of these approaches were specifically designed to address the issue of providing a reference version to compare with the current version.

Beyond getting back to a previous version of code, developers often want to compare two versions side-by-side. The UNIX *diff* utility is the seminal tool for comparing two text files. Its features have been incorporated into mainstream code editors, such as Eclipse and Visual Studio, to allow the comparison of two versions of code, often with text highlighting to indicate inserted or deleted text. Furthermore, researchers have applied diff to programming concepts other than text, such as LSdiff, which identifies systematic code changes by analyzing code elements and their structural dependencies [54]. Indeed, Yestercode's annotations for denoting differences between the reference block diagram and current one were inspired by diff.

More recently, researchers have combined undo support with a comparison view. Azurite is an Eclipse plug-in for textual-code editors that allows a developer to selectively undo a region of code to a previous state, and provides a comparison feature that shows a before and after version of the relevant code [124]. Additionally, Azurite transparently logs the code edits and shows a timeline visualization. While this tool fully supports developers trying to backtrack, it may not effectively support a developer who is attempting to perform a new change while referencing a previous version. For example, unlike Yestercode, the comparison view is in an interface separate from the code editor, and as a consequence, the developer cannot make edits to the current version while juxtaposing it with the older version.

Chapter 5

Navigating the Code Structure

If you don't know where you are going,
any road will get you there.

—Lewis Carroll

A particularly efficient form of navigation is *structured navigations*—that is, navigating along the structural relationships of code. One reason that these navigations are of particular interest to developers is that developers must understand the structural relationships of their code to accomplish their tasks successfully [62, 65]. Studies have found that these structured navigations are more efficient than other forms of navigation and lead to higher success in code-change tasks [33, 104].

Researchers have proposed tools to enhance navigating the code structure by providing a call graph visualization. Such a visualization displays the call relationships between methods (e.g., which methods get called by a specific method). Prodet [2] and Reacher [66] are such tools, that provide a call graph visualization in a pane below the code editor, and allow developers to click on a node in the graph to navigate to that location. Another notable tool is Stacksplorer [51], which displays two lists of methods next to the code editor, one on each side, which represents a portion of the call graph.

However, these tools may have their own barriers that prevent developers from using them efficiently. The call graph visualizations can be quite large which may get in the way of the other tools they are using (e.g., the code editor). In fact, a study of developers using Prodet found that they rarely used its “map view” feature, possibly because of how much screen space it required [2]. Additionally, tools like Prodet [2], Reacher [66], and Stacksplorer [51] put the information into panes that are docked around the code editor, but one study already revealed that developers struggle with managing tab panes in their development environment [9]. These visualizations may also overwhelm developers with too much information, which could hinder the efficiency of such a tool. For example, Reacher annotates the graph with icons to represent the type of call, whether it is called in a conditional, or if it is called in a loop. Although this information may be useful to a developer, it may only be used sparingly so it may hinder comprehension if it is always present.

To address these potential problems with call graph tools and to provide efficient means to navigate the code structure, we designed Wandercode. Key design goals of Wandercode include presenting the visualization as an overlay on top of the code editor and only providing minimal contextual information with the graph. Additionally, Wandercode provides affordances that enable the developer to view more of the graph when needed as well as to enable the developer to explore the graph.

In this chapter, we present an initial prototype of Wandercode as well as report the findings from a user-study evaluation using the prototype. In particular, our user study involved 8 participants answering comprehension questions regarding structural relationships of code. We compared Wandercode to another tool that presents recommendations of relevant code in a list, based on the call graph. Our study addressed the following research questions:

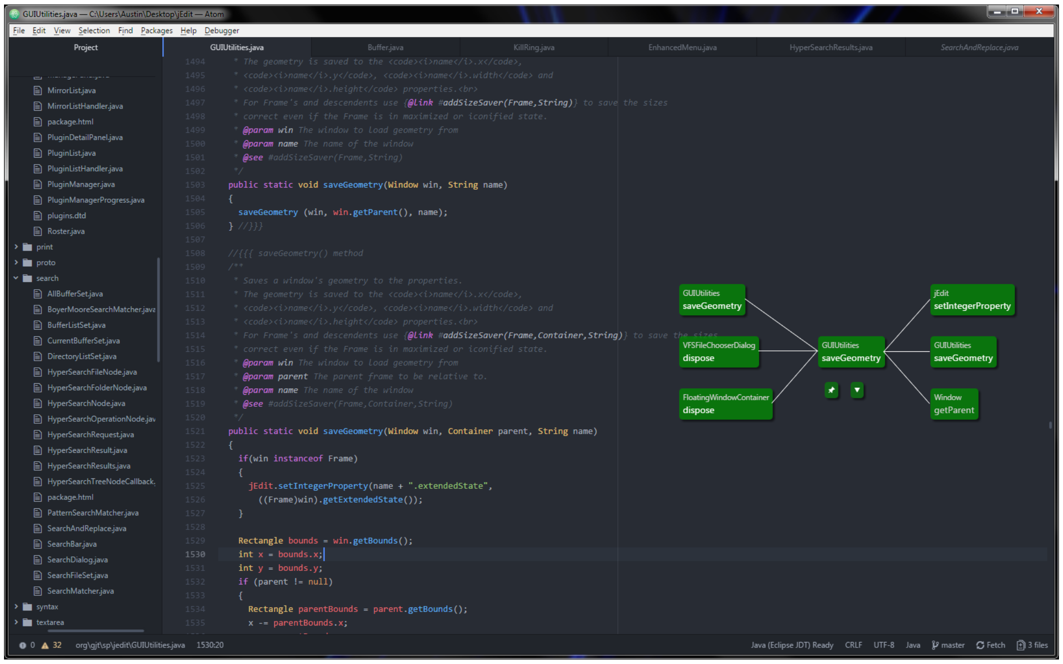
- RQ1: Does using Wandercode reduce the time developers take to answer questions regarding the structure of the code?

- RQ2: Do developers using Wandercode have lower cognitive load while seeking information about the code structure?
- RQ3: Do developers have favorable opinions of Wandercode?

5.1 Related Work: Recommendation Systems for Code Locations

Another approach is to provide recommendations of relevant code to the developer with the ability to quickly navigate to the location. The predominant means of doing so is to provide a listing of methods that when clicked, take the developer to that location. Some tools use structural relationships to generate recommendations (e.g., Strathcona [46] and Suade [116]), others use navigation history (e.g., Team Tracks [24], Navtracks [106]), and many use a combination of techniques (e.g., Mylar [52], Piorkowski et al.'s PFIS recommender [93], and Hipikat [112]). Other tools augment their recommendations with a visualization, such as portions of a call graph (e.g., Rearcher [66] and Prodet [2]). Even though these recommendation systems have been popular for researchers to design, there has been little adoption in industry.

Even though these recommendation systems are a promising start in aiding developers in navigating to relevant code, they may have a number of limitations that prevent them from seeing widespread adoption. First, these tools provide little to no justification to the developer as to why they are recommending particular locations (e.g., Mylar provides only a score value and Piorkowski et al.'s recommender shows relevant keywords). Second, the tools provide long lists which may overwhelm the developer. Third, the tools constantly update the list so developers can not systematically navigate to each recommendation in the list.



```
 * The geometry is saved to the <code><i>name</i>.x</code>,
 * <code><i>name</i>.y</code>, <code><i>name</i>.width</code> and
 * <code><i>name</i>.height</code> properties.<br>
 * For Frame's and descendents use {@link #addSizeSaver(Frame,String)} to save the sizes
 * correct even if the Frame is in maximized or iconified state.
 * @param win The window to load geometry from
 * @param name The name of the window
 * @see #addSizeSaver(Frame,String)
 */
public static void saveGeometry(Window win, String name)
{
  saveGeometry (win, win.getParent(), name);
} //}}}

//{{{ saveGeometry() method
/**
 * Saves a window's geometry to the properties.
 * The geometry is saved to the <code><i>name</i>.x</code>,
 * <code><i>name</i>.y</code>, <code><i>name</i>.width</code> and
 * <code><i>name</i>.height</code> properties.<br>
 * For Frame's and descendents use {@link #addSizeSaver(Frame,Container,String)} to save the sizes
 * correct even if the Frame is in maximized or iconified state.
 * @param win The window to load geometry from
 * @param parent The parent frame to be relative to.
 * @param name The name of the window
 * @see #addSizeSaver(Frame,Container,String)
 */
public static void saveGeometry(Window win, Container parent, String name)
{
  if(win instanceof Frame)
  {
    jEdit.setIntegerProperty(name + ".extendedState",
      ((Frame)win).getExtendedState());
  }

  Rectangle bounds = win.getBounds();
  int x = bounds.x;
  int y = bounds.y;
  if (parent != null)
  {
    Rectangle parentBounds = parent.getBounds();
    x -= parentBounds.x;
```

Figure 36: Atom with Wandercode's call graph visualization displayed on the right (green boxes and white lines). It is displaying three callers (left portion of the graph) and three callees (right portion of the graph). The current method is drawn in the center of the graph.

5.2 Tool Design: Wandercode

Based on the proposed designs of other call graph-based tools, we designed Wandercode. Our design seeks to maintain the benefits that previous designs have gained while also overcoming some of the potential problems that they may have. These problems include taking up too much screen space for the visualization and overloading the developer with information. Our implementation is shown in Fig. 36 as an extension to the open source code editor Atom.

Through the development of Wandercode, we identified four key principles for the design. In particular, our design should…

- Overlay the call graph visualization onto the code editor.
- Prevent information overload by reducing the information displayed.
- Provide more nodes of the call graph as needed.
- Enable exploring of the call graph.

- Present more relevant calls first.

In the remainder of this section, we highlight the features of our Wandercode design that satisfy these principles.

5.2.1 Overlay the Editor with the Call Graph

The call graph visualization provided by Wandercode is overlayed on top of the code editor, as shown in Fig. 36, when a developer places the text cursor in the body of a method. In particular, Wandercode does this to address issues with screen space and window management in development environments. By providing the visualization on top of the editor, the developer does not need to make a deliberate choice between what they would want to view, either more of the editor, more of the call graph, or more of some other view in their environment. This approach of using an overlaid interface has been utilized by other development tools, such as ReSharper's "Go To Everything" search feature.

5.2.2 Prevent Information Overload

Wandercode aims to prevent information overload by only displaying minimal information regarding each method call. Each node of the graph only displays the method name and class name, and is displayed either on the left or right of the current method, indicating whether it is a *caller* or a *callee* of the current method. In our design of Wandercode, we limited the number of callers and callees to 3 nodes, so as to not overload the developer's working memory. Additionally, subsequent calls of the same method are merged together (i.e., calling the same method multiple times will only be shown once). Other tools provide numerous types of additional information.

5.2.3 Provide More Information as Needed

If a developer needs more information than the call graph is showing, there are two options. One option is to click the node of interest in the call graph to navigate to that

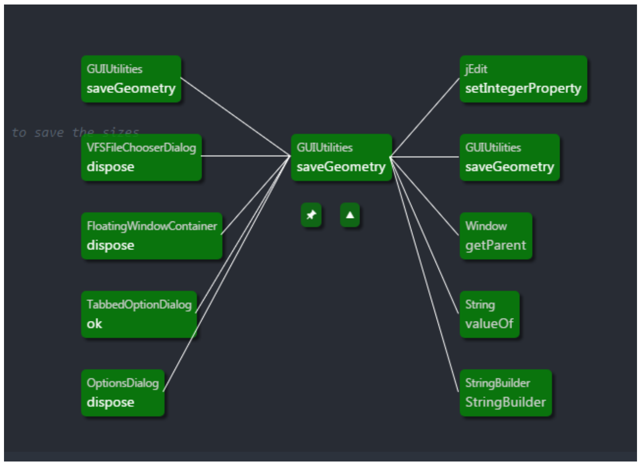


Figure 37: Close-up of Wandercode's call graph visualization. The graph has been "expanded" to display more method calls. Each node displays the class name and method. Below the center node are two buttons: pin and expand/collapse.

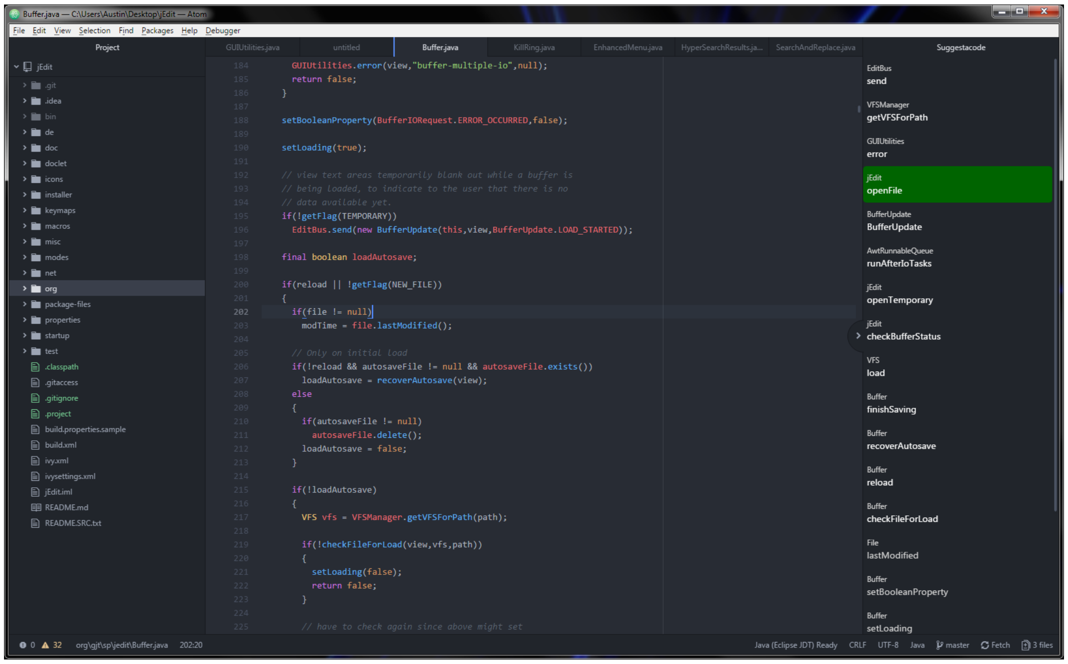
location in the code editor. The other option is to click Wandercode's expand button, revealing additional callers and callees. Fig. 37 shows an already expanded graph. Our initial design choice was to limit the total number of callees/callers to a total of 10 due to screen size constraints. Clicking the button again will hide the additional nodes from the call graph.

5.2.4 Enable Exploring the Call Graph

To enable efficient navigation of the call graph, Wandercode provides a button to "pin" the graph, which locks the current graph and prevents it from updating. This feature is to allow the developer to navigate to callers and callees without the graph updating and resulting in the developer losing their place in the graph.

5.2.5 Present More Relevant Calls First

Wandercode uses a recommendation system to choose which methods to display in the graph and to rank them based on their relevance. The recommendation criteria is based on the method's popularity throughout the entire codebase, meaning that methods that are called more often will be sorted higher. Additionally, methods that have their source code contained in the current project are prioritized over methods contained in libraries. Other


```
  GUIUtilities.error(view,"buffer-multiple-io",null);
  return false;
}

setBooleanProperty(BufferIORequest.ERROR_OCCURRED,false);

setLoading(true);

// view text areas temporarily blank out while a buffer is
// being loaded, to indicate to the user that there is no
// data available yet.
if(!getFlag(TEMPORARY))
  EditBus.send(new BufferUpdate(this,view,BufferUpdate.LOAD_STARTED));

final boolean loadAutosave;

if(reload || !getFlag(NEW_FILE))
{
  if(file != null)
    modTime = file.lastModified();

  // Only on initial load
  if(!reload && autosaveFile != null && autosaveFile.exists())
    loadAutosave = recoverAutosave(view);
  else
  {
    if(autosaveFile != null)
      autosaveFile.delete();
    loadAutosave = false;
  }

  if(!loadAutosave)
  {
    VFS vfs = VFSManager.getVFSForPath(path);

    if(!checkFileForLoad(view,vfs,path))
    {
      setLoading(false);
      return false;
    }

    // have to check again since above might set
```

Figure 38: Atom with the control extension displaying a list of recommendations on the right. The list is based on the call graph and when clicked will open that code method in the editor.

tools that recommend code fragments that may be of interest to developers have been based on a variety of criteria, and often combine several, such as lexical and structural similarity (e.g., Mylar [52], Piorkowski et al.'s PFIS recommender [93], Strathcona [46], Suade [116], and Hipikat [112]). However, we designed Wandercode such that developers will be able to know *why* the recommendations are being made, which may not be possible with a complex recommendation system.

5.3 Evaluation: User Study

To evaluate Wandercode, we conducted a lab study of developers tasked with answering questions regarding the structure of code. It was of within-subjects design, where each participant used the Atom code editor in conjunction with the two treatments: the Wandercode plugin and a control. The control was a plugin that provides a list of recommendations of relevant code based on the call graph. For the design of this tool, we modeled it primarily after existing list-based recommendation systems (e.g., Mylar [52] and Navtracks [106]), as seen in Fig. 38. We sought to answer the following research questions:

- RQ1: Does using Wandercode reduce the time developers take to answer questions regarding the structure of the code?
- RQ2: Do developers using Wandercode have lower cognitive load while seeking information about the code structure?
- RQ3: Do developers have favorable opinions of Wandercode?

5.3.1 Method

5.3.1.1 Participants

Our participants consisted of 10 Java developers (7 male, 3 female), consisting of 8 graduate students, 1 undergraduate student, and 1 professional developer. They reported, on average, 5.3 years of programming experience ($SD = 2.45$) and 2.3 years of Java programming experience ($SD = 0.67$). All participants reported having experience with Eclipse and Sublime.

5.3.1.2 Comprehension Tasks

For our study, the participants answered questions regarding the code structure of a large open source Java project, jEdit. Our study design took inspiration from two prior studies on the effects on a call graph-based tool [66, 51], specifically the tasks of answering questions regarding the code's structure [66]. Each participant performed eight tasks involving answering a specific question about the relationships between code methods and classes. The question was provided to them on paper along with a starting point in the jEdit project and rationale as to why they might need to know this information. An example of a question is, "Imagine you need to modify the parameters of the method setWholeWord(). However, doing so would cause errors in any method that calls setWholeWord(). Which methods would need to be fixed? (There are two.)" They were allowed to use any navigation feature available to them in Atom. If they did not complete the task within 10 minutes, we asked them to continue to the next task (this only occurred once). The tasks were divided in to two sequences of four (one sequence for each treatment). For each four-task sequence, there were two “upstream” questions and two “downstream” questions, referring to the direction in which the answer was in the call graph.

5.3.1.3 Procedure

Each participant took part in an individual session that lasted approximately 1 hour. All participants began the session by filling out a background questionnaire and receiving an introduction to the Atom code editor and jEdit codebase. We demonstrated how Atom’s code navigation features, such as searching within a file, searching within a project, and finding all references of a code element, Next, each participant completed the first 4-task sequence with one treatment and the second 4-task sequence with the other treatment. Five randomly selected participants used the control extension first, and the other five used the Wandercode extension first. Each 4-task sequence began with an introduction to the extension that the participant would be using for that sequence. Next, the participant performed the 4-task sequence in order, completing each task before beginning the next. We asked each participant to “think aloud” as he/she worked. Once the participant had finished each task, he/she completed a cognitive-load questionnaire (details below). At the end of each 4-task sequence, the participant completed a usability questionnaire regarding the extension they used. After both 4-task sequences were completed, all participants took part in a semi-structured interview in which they discussed their experiences with using the two extensions.

5.3.1.4 Data Collection

The data collected for the study comprised screen-capture video and audio of the participants as well as their questionnaire responses. For the cognitive load questionnaire,

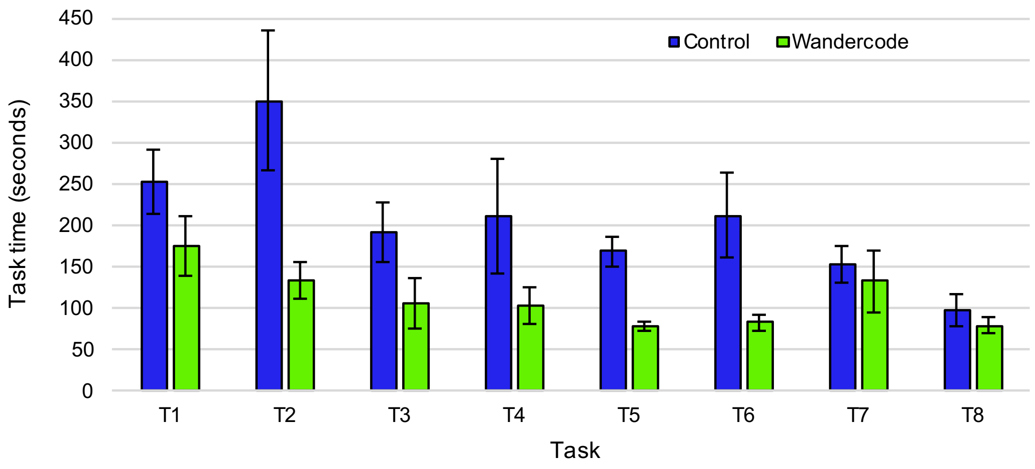


Figure 39: Wandercode users completed tasks significantly faster than control users (smaller bars are better). Whiskers denote standard error.

we used a well-validated instrument based on Cognitive Load Theory [83]. The instrument features a 7-point Likert question that measures a person's overall cognitive load during a task. Prior work showed that the instrument does not interfere with task performance, is sensitive to small differences in workload, and is reliable [84]. Moreover, the instrument has been shown to highly correlate with more-complex self-reporting instruments (e.g., NASA TLX) [119] as well as with physiological sensors (e.g., heart rate) [84].

5.3.1.5 Limitations

Our user study has several limitations inherent to laboratory studies of developers. First, our sample of developers, consisting of mostly students, may not represent all such developers. Second, the tasks are artificial questions that may not be ecologically valid in a broader software development context; however, we based them on prior studies of relevant tools. Third, reactivity effects (e.g., participants consciously or unconsciously trying to please the researchers) may have occurred; however, we tried to minimize them by presenting the two tools as extensions to Atom, and did not disclose that they were our invention. Finally, order effects of the tool and tasks may have affected our observations, but we counterbalanced the tool order to control for this effect with respect to our statistical tests.

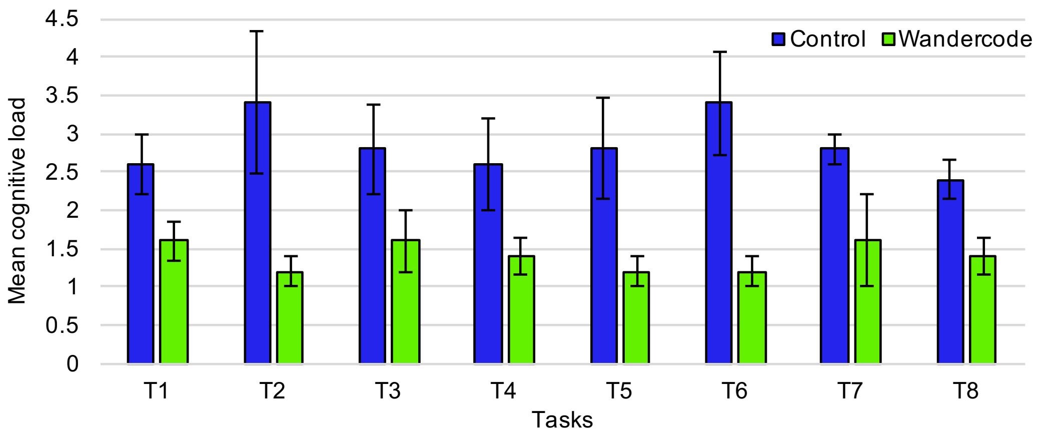


Figure 40: Wandercode users reported significantly lower cognitive load than control users (smaller bars are better). Whiskers denote standard error.

5.3.2 Results

5.3.2.1 RQ1 Results: Time on Task

As Fig. 39 shows, participants using Wandercode tool completed tasks considerably faster bugs than those using the control tool. In fact, Wandercode users completed every task faster, on average. Indeed, the results of a Mann–Whitney U test revealed that Wandercode users completed tasks significantly faster than control users ($U = 7$, $Z = 2.57$, $p < 0.05$).

5.3.2.2 RQ2 Results: Cognitive Load

As Fig. 40 shows, Wandercode users reported considerably lower cognitive loads than control users. In fact, Wandercode users reported a lower cognitive load than control users for every task, on average. The results of a Mann–Whitney U test showed that Wandercode users reported a significantly lower cognitive load than control users ($U = 0$, $Z = 3.3$, $p < 0.01$).

5.3.2.3 RQ3 Results: Opinions of the Participants

To assess participants' opinions of the two tools, each participant responded to the following questions at the end of the task sequences (7-point Likert scale):

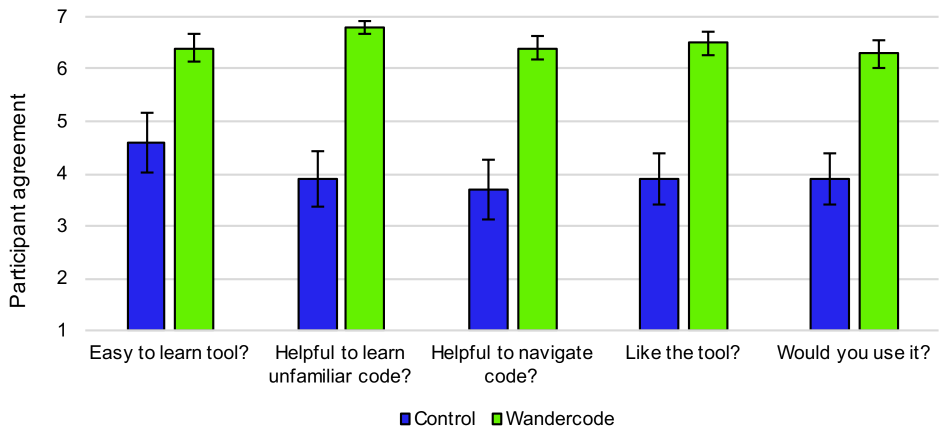


Figure 41: Participants reported significantly higher opinions of Wandercode than of the control tool (larger bars are better). Whiskers denote standard error.

- How difficult/easy was this tool to learn?
- How unhelpful/helpful do you think this tool is for learning about unfamiliar code?
- How unhelpful/helpful do you think this tool is for navigating code?
- How much did you dislike/like using this tool?
- If this tool was made available to you, how often would you use it for your coding tasks?

As Fig. 41 shows, users reported significantly higher opinions of Wandercode than the control tool. In fact, every participant reported a higher or equivalent score for Wandercode than they did for the control. The results of a Mann–Whitney U test showed that participants reported significantly higher opinions of Wandercode than the control tool ($U = 0$, $Z = 3.74$, $p < 0.001$). Additionally, when asked which tool they preferred, 9 of the 10 participants said Wandercode, and one participant said "it depends" on the situation.

5.4 Discussion

Participants were highly favorable of Wandercode, and expressed it throughout their session.

There were also a number of improvements that participants suggested at the end of the study session for Wandercode. The most common suggestion was to add arrows (rather than just lines) to the call graph to portray the direction of which method is being called. Participants also requested the ability to view all of the callees and callers, not just a subset of them. Additionally, one participant asked to view Wandercode's call graph only when he hovers over a method.

Although the participants preferred Wandercode, many of them also liked using Suggestacode. One notable means of improving Suggestacode according to the participants is to separate the callees from the callers. By doing so, Suggestacode would display the same structural information as Wandercode does but in list form. When comparing the two tools, participants did indicate that they liked Suggestacode since it takes up very little screen space.

Chapter 6

Future Work

It never gets easier, you just go faster.

—Greg LeMond

Our tool designs and study results have led the way to an abundance of potential future works. Through our human-centric approach, we were able to collect feedback about potential improvements from our participants by observing them use our tools and by interviewing them. Our participants were often eager to describe the problems they have with their current tools, how our tools could help them, and how our tools could help them even more if they were improved with specific features. Additionally, we were overwhelmed by the number of suggestions from reviewers and attendees at conferences on how our tools could better help them with their programming tasks.

We have studied each our tools in isolation, but could they better aid developers by being combined into a single environment? For example, the Patchworks grid could be used to contain Yestercode's reference view and the timeline slider to choose the version of code could be added to the patch's label. Wandercode could be used to suggest which code fragments should be open in a given patch by analyzing the call graph of surrounding patches containing code. These features would also incorporate well with Patchwork's blowup view, which would allow developers to focus on the reference version of code or the structurally relevant code fragments.

One major step we would like to take is to run field studies to understand how Patchworks, Yestercode, and Wandercode help developers in more realistic tasks for longer periods of time. Although our laboratory studies have yielded promising results, it is still an open question how these designs will perform in a more ecologically valid setting. To do so, we could release plugins of each of our tools (e.g., for Visual Studio or Atom) that are freely available to the public. If the users opt-in, we would log how they interact with our tools to understand how they are using each feature. Furthermore, we would follow up with users with a survey to get rich qualitative data about their experiences.

One of the most popular questions we have received for our tools is: "how does this extend to multiple monitors?" Considering it is so prevalent for software developers to have 2 monitors or more, it is a legitimate concern to wonder how our tools will make the most of such an environment. For example, it is an open question whether Patchworks' ribbon should extend across multiple monitors or each monitor should display its own ribbon. Yestercode's view of the previous version of code could be displayed on another monitor to free up space where the developer is actively editing code. In regards to Wandercode, a feature could be added that enables the developer to expand the call graph visualization onto a second monitor (similar to Prodet's map view [2]).

There also exist a number of features to support team and social aspects of software development that could enhance our tools. Participants have suggested adding the ability to share the contents of their Patchworks ribbon with other developers. Doing so could act as a form of documentation (i.e., these code fragments are relevant to this task) while also improving productivity by letting developers resume from this state. Similarly, developers might need to view the intermediate steps that resulted in a specific version of code written by another developer, which Yestercode could provide. In an effort to recommend more relevant code, Wandercode could take into account team members' navigation history when ranking the recommendations.

In particular to Patchworks, the optimal number of patches onscreen is not well known. To better understand this, we plan to analyze the typical size of code fragments from various programming languages while also considering monitor size. Moreover, the ribbon is ripe for augmentation. The ribbon view could have buttons to display recommendations of relevant code, documentation, examples, bug report information, notes from the developer, and even runtime information.

Yestercould could also be further studied; in particular, we would like to understand how well the tool works in a textual programming environment. Participants also suggested that the timeline could be improved with a better visualization. We could do this by incorporating features from the Azurite [124] tool which portrays how long an edit took and what type of edit it is. Additionally, to make the timeline easier to read and navigate, Yestercode could merge sequences of minor edits or filter changes that are often not interesting. For example, a sequence of actions that add comments to the code could be merged into one.

Our user study evaluation of Wandercode lead to a variety of interactions designs that should be further studied. Most notably, we would like to investigate the impact of tools overlaying information onto the code editor (as opposed to in a separate pane). One approach to this could be a study that presents several UI prototypes to users, where each prototype has made a single change to the design (e.g., a vertical versus horizontal graph orientation). It is an open question as to how many methods should be presented in the call graph, and exactly what information should be provided with them. Several participants also wanted to view more of the call graph at times, not just the subset that is recommended by Wandercode. Although the recommendations appear to be very helpful, there may be times when a view of the entire graph is beneficial. To implement these features, we could provide affordances that Reacher [66] and Prodet [2] provide.

Additionally, we did not study the best interaction design for when to show/hide Wandercode's visualization.

Chapter 7

Conclusions

There must be some kind of way out of here.

—Bob Dylan

Despite code navigation being fundamental to any software maintenance task, it is notoriously problematic. These barriers cause developers to spend an inordinate amount of time and effort just navigating their code in an attempt to complete their tasks. A major contributor to these barriers is that tools for navigating code have consequential flaws.

My approach towards a potential solution to this problem is to design tools based on the needs of developers. In particular, I base my tool designs on findings from empirical studies, including laboratory studies and interviews. In an effort further improve the designs, I leverage existing theories, such as Information Foraging Theory, that explain the behavior and needs of developers while navigating code. To validate that my design helps developers overcome their navigation problems, I evaluate my tools in rigorous laboratory studies.

Since developers spend a tremendous amount of time revisiting a small set of code, I designed the Patchworks code editor to make those navigations more efficient. By making the editor consist of a fixed grid of open code fragments, developers can greatly reduce the amount of time they spend managing tabs and scrolling through files. To evaluate Patchworks, I conducted three studies. First, a simulation study found that developers should navigate faster with Patchworks than a traditional file-based code editor, regardless to how they use it. Second, a preliminary user study comparing Patchworks to two other code editors, found that Patchworks users navigated faster, made fewer navigation mistakes, and spent less time arranging their code. Third, a user study comparing Patchworks to a traditional code editor found that Patchworks enabled more efficient navigation, while also finding a number of improvements that could be made to other code editors.

To enable developers to efficiently navigate among versions of their code, I designed Yestercode. My formative investigations found that developers face considerable challenges when trying to get information from past versions of code. Yestercode aims to address this by automatically recording each change to the code and providing a comparison view of a selected version of code with the current version. A user study found that it decreased the number of bugs introduced and decreased developers' cognitive load during code-change tasks.

Given that code is so highly connected, I designed Wandercode, a tool that provides a visualization based on a program's call graph. Wandercode aims to make navigations more efficient by providing recommendations on where a developer should navigate next while also presenting structural information about the code. An initial user study found that developers using Wandercode completed tasks significantly faster, it reduced their cognitive load, they found it easy to use, and the developers were highly favorable of the tool.

The results of my studies have emphasized the chasm in software development tools for navigating code efficiently. It is my hope that the findings contained in this dissertation are a monumental step in recognizing the dire need for better software development tools and in designing tools for more efficient code navigation. I am optimistic that tool builders in the future will be able to design more human-centric tools to aid software developers in their arduous software maintenance tasks.

Appendix A

Initial Patchworks Study Materials

A.1 IRB Approval Letter

IRB Approval2732

Institutional Review Board <irb@memphis.edu> Mon, Jul 1, 2013 at 11:28 AM
To: "Austin Zachary Henley (azhenley)" <azhenley@memphis.edu>
Cc: "Scott Fleming (sdflming)" <sdflming@memphis.edu>

Hello,

The University of Memphis Institutional Review Board, FWA00006815, has reviewed and approved your submission in accordance with all applicable statuses and regulations as well as ethical principles.

PI NAME: Austin Henley
CO-PI:
PROJECT TITLE: Usability Evaluation of Patchworks Code Editor
FACULTY ADVISOR NAME (if applicable): Scott Fleming

IRB ID: #2732
APPROVAL DATE: 6/30/2013
EXPIRATION DATE:
LEVEL OF REVIEW: Expedited

Please Note: Modifications do not extend the expiration of the original approval

Approval of this project is given with the following obligations:

1. If this IRB approval has an expiration date, an approved renewal must be in effect to continue the project prior to that date. If approval is not obtained, the human consent form(s) and recruiting material(s) are no longer valid and any research activities involving human subjects must stop.

2. When the project is finished or terminated, a completion form must be completed and sent to the board.

3. No change may be made in the approved protocol without prior board approval, whether the approved protocol was reviewed at the Exempt, Exedited or Full Board level.

4. Exempt approval are considered to have no expiration date and no further review is necessary unless the protocol needs modification.

Approval of this project is given with the following special obligations:

Thank you,

Ronnie Priest, PhD

Institutional Review Board Chair

The University of Memphis.

Note: Review outcomes will be communicated to the email address on file. This email should be considered an official communication from the UM IRB. Consent Forms are no longer being stamped as well. Please contact the IRB at IRB@memphis.edu if a letter on IRB letterhead is required.

A.2 Background Questionnaire

Background Questionnaire

1. Age range:
   Choose an item.

2. Education (highest degree or level completed):
   Choose an item.

3. Current field of study or profession:
   Choose an item.

4. How many years of…

   a. … programming experience: Click here to enter text.

   b. … professional programming experience: Click here to enter text.

   c. … Java programming experience: Click here to enter text.

5. What are your primary programming languages?

6. What are your primary code editing tools (for example, Eclipse or Visual Studio)?

7. How often do you place code side-by-side with these tools?

A.3 Poststudy Questionnaire

Post-Session Questionnaire

How easy/difficult to learn were the patch and ribbon features?
Choose an item.

How easy/difficult to use these features?
Choose an item.

How do you think these features will affect the amount of time coding tasks take?
Choose an item.

How much did you like/dislike the patch and ribbon features?
Choose an item.

If the patch and ribbon features were available in your code editor of preference, would you use them?
Choose an item.

What did you think of the patches and ribbon?

Did you have any difficulties figuring out how to use the patches and ribbon?

Please write any comments you may have here:

Appendix B

Preliminary Refactoring Study Materials

B.1 IRB Approval Letter

Gmail by Google

Austin Henley <azh321@gmail.com>

IRB Approval 3530

1 message

Institutional Review Board <irb@memphis.edu> Wed, Jan 21, 2015 at 8:41 AM
To: "Austin Zachary Henley (azhenley)" <azhenley@memphis.edu>, "Scott Fleming (sdflming)" <sdflming@memphis.edu>

Hello,

The University of Memphis Institutional Review Board, FWA00006815, has reviewed and approved your submission in accordance with all applicable statuses and regulations as well as ethical principles.

PI NAME: Austin Henley
CO-PI: Kazi Zaman, Maria Luong, Alka Singh
PROJECT TITLE: Understanding LabVIEW Programmers
FACULTY ADVISOR NAME (if applicable): Scott Fleming

IRB ID: #3530
APPROVAL DATE: 1/16/2015
EXPIRATION DATE: 1/16/2016
LEVEL OF REVIEW: Expedited

Please Note: Modifications do not extend the expiration of the original approval

Approval of this project is given with the following obligations:

1. If this IRB approval has an expiration date, an approved renewal must be in effect to continue the project prior to that date. If approval is not obtained, the human consent form(s) and recruiting material(s) are no longer valid and any research activities involving human subjects must stop.

2. When the project is finished or terminated, a completion form must be completed and sent to the board.

3. No change may be made in the approved protocol without prior board approval, whether the approved protocol was reviewed at the Exempt, Exedited or Full Board level.

4. Exempt approval are considered to have no expiration date and no further review is necessary unless the protocol needs modification.

Approval of this project is given with the following special obligations:

Thank you,

James P. Whelan, Ph.D.

Institutional Review Board Chair

The University of Memphis.

Note: Review outcomes will be communicated to the email address on file. This email should be considered an official communication from the UM IRB. Consent Forms are no longer being stamped as well. Please contact the IRB at IRB@memphis.edu if a letter on IRB letterhead is required.

B.2 Background Questionnaire

Background Questionnaire

1. Gender:
2. Age:
3. Education (highest degree completed):
4. Field(s) of study or major:
5. Current job title:
6. Is English your primary language? If not, then what is your primary language?
7. How many years of…
    a. Programming experience:
    b. Professional programming experience:
    c. LabVIEW experience:
    d. What other programming languages do you have experience with?

Appendix C

Yestercode Study Materials

C.1 Background Questionnaire

Background Questionnaire

1. Gender:
2. Age:
3. Education (highest degree completed):
4. Field(s) of study or major:
5. Current job title:
6. Is English your primary language? If not, then what is your primary language?
7. How many years of…
    a. Professional experience:
    b. Programming experience:
    c. LabVIEW experience:
8. What other programming languages do you have experience with?

C.2 Tool Questionnaire

Tool Questionnaire

How difficult or easy was this tool to use? (1-7)

1	2	3	4	5	6	7
Very Hard						Very Easy

How helpful was this tool? (1-7)

1	2	3	4	5	6	7
Very Unhelpful						Very Helpful

The tasks with this tool were more difficult or easier? (1-7)

1	2	3	4	5	6	7
Much Harder						Much Easier

Appendix D

Wandercode Study

D.1 IRB Approval Letter

Gmail **Austin Henley <azh321@gmail.com>**

PRO-FY2018-608 - Initial: Approval - Expedited
1 message

Institutional Review Board <irb@memphis.edu> Tue, May 29, 2018 at 8:38 AM
To: "Austin Zachary Henley (azhenley)" <azhenley@memphis.edu>, "Scott Fleming (sdflming)" <Scott.Fleming@memphis.edu>

Institutional Review Board
Office of Sponsored Programs
University of Memphis
315 Admin Bldg
Memphis, TN 38152-3370

May 29, 2018

PI Name: Austin Henley
Co-Investigators:
Advisor and/or Co-PI: Scott Fleming
Submission Type: Initial
Title: Usability Evaluation of Code Navigation Tools
IRB ID **:** **#**PRO-FY2018-608

Expedited Approval: May 25, 2018
Expiration: May 25, 2019

Approval of this project is given with the following obligations:

1. This IRB approval has an expiration date, an approved renewal must be in effect to continue the project prior to that date. If approval is not obtained, the human consent form(s) and recruiting material(s) are no longer valid and any research activities involving human subjects must stop.

2. When the project is finished or terminated, a completion form must be submitted.

3. No change may be made in the approved protocol without prior board approval.

Thank you,
James P. Whelan, Ph.D.
Institutional Review Board Chair
The University of Memphis.

D.2 Background Questionnaire

Background Questionnaire

Leave any question blank if you would prefer to not disclose the answer.

1. Email address:
2. University of Memphis UID (this is needed to process the gift card):
3. Gender:
4. Age:
5. Education (highest degree completed):
6. Field(s) of study or major:
7. Current job title:
8. Is English your primary language? If not, then what is your primary language?
9. How many years of…
    a. Professional experience:
    b. Programming experience:
    c. Java experience:
10. What other programming languages do you have experience with?
11. What code editors do you have experience with (e.g., Eclipse, Visual Studio, Sublime)?

D.3 Tool Questionnaire

Tool Questionnaire

How difficult/easy was this tool to learn? (1-7)

1	2	3	4	5	6	7
Very Difficult						Very Easy

How unhelpful/helpful do you think this tool is for learning about unfamiliar code? (1-7)

1	2	3	4	5	6	7
Very Unhelpful						Very Helpful

How unhelpful/helpful do you think this tool is for navigating code? (1-7)

1	2	3	4	5	6	7
Very Unhelpful						Very Helpful

How much did you dislike/like using this tool? (1-7)

1	2	3	4	5	6	7
Disliked						Liked

If this tool was made available to you, how often would you use it for your coding tasks?

1	2	3	4	5	6	7
Never						Always

References

[1] J. R. Anderson and G. H. Bower. Recognition and retrieval processes in free recall. *Psychol. Rev.*, 79(2):97–123, Mar. 1972.

[2] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Braunlich, and T. Fritz. A field study on fostering structural navigation with prodet. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 229–238, May 2015.

[3] T. Barik. *Error Messages as Rational Reconstructions*. PhD thesis, North Carolina State University, 2018.

[4] R. Bellamy, B. John, and S. Kogan. Deploying CogTool: Integrating quantitative usability assessment into real-world software development. In *Proc. ICSE*, pages 691–700, 2011.

[5] R. Bellamy, B. John, J. Richards, and J. Thomas. Using CogTool to model programming tasks. In *Evaluation and Usability of Programming Languages and Tools* (PLATEAU '10), pages 1:1–1:6. ACM, 2010.

[6] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3):269–294, Sept. 1994.

[7] T. Berlage and A. Genau. A framework for shared applications with a replicated architecture. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST '93)*, pages 249–257. ACM, 1993.

[8] A. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? In *Proceedings of the IEEE Symposium on Visual Languages (VL '96)*, pages 240–246, 1996.

[9] A. Bragdon. Creating simultaneous views of source code in contemporary IDEs using tab panes and MDI child windows: A pilot study. Technical Report CS-09-09, Brown Univ., 2009.

[10] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the*

32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[11] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[12] J. Brandt. *Example-Centric Programming: Integrating Web Search into the Development Process*. PhD thesis, Stanford University, 2010.

[13] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating Web search into the development environments. In *Proc. 28th Int'l Conf. on Human Factors in Computing Systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[14] B. Burg, A. J. Ko, and M. D. Ernst. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*, pages 259–268. ACM, 2015.

[15] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, Mar. 1995.

[16] S. K. Card, A. Newell, and T. P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., 1983.

[17] C. Chambers and C. Scaffidi. Smell-driven performance analysis for end-user programmers. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 159–166, Sept 2013.

[18] M. J. Coblenz, A. J. Ko, and B. A. Myers. Jasper: An eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, ETX '06, pages 65–69, New York, NY, USA, 2006. ACM.

[19] A. Cockburn and B. McKenzie. Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *Proc. CHI*, pages 203–210, 2002.

[20] N. I. Corporation. LabVIEW Calculator, 2008. Accessed: 2017-01-06.

[21] N. Cowan. Metatheory of storage capacity limits. *Behavioral and Brain Sciences*, 24(1):154–176, 2001.

[22] R. P. Darken and J. L. Sibert. Wayfinding strategies and behaviors in large virtual worlds. In *Proc. CHI*, pages 142–149, 1996.

[23] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code Thumbnails: Using spatial memory to navigate source code. In *Proceedings of the Visual Languages and Human-Centric Computing*, VL/HCC '06, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society.

[24] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.

[25] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 183–192, New York, NY, USA, 2005. ACM.

[26] R. DeLine and K. Rowan. Code Canvas: Zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 207–210, New York, NY, USA, 2010. ACM.

[27] T. d'Entremont and M.-A. Storey. Using a degree of interest model to facilitate ontology navigation. In *Proc. VL/HCC*, pages 127–131, 2009.

[28] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 441–450. ACM, 2008.

[29] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[30] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. 2nd Int'l Conf. Knowledge Discovery and Data Mining*, number 34 in KDD '96, pages 226–231, 1996.

[31] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans. Softw. Eng. Methodol.*, 22(2):14:1–14:41, Mar. 2013.

[32] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[33] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 7–18, New York, NY, USA, 2014. ACM.

[34] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 211–221. IEEE Press, 2012.

[35] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill. How developers use multi-recommendation system in local code search. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 69–76, July 2014.

[36] T. R. G. Green. Cognitive dimensions of notations. In *People and Computers V*, pages 443–460. Cambridge University Press, 1989.

[37] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[38] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Trans. Comput.-Hum. Interact.*, 19(1):5:1–5:28, May 2012.

[39] A. Z. Henley and S. D. Fleming. The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2511–2520, New York, NY, USA, 2014. ACM.

[40] A. Z. Henley and S. D. Fleming. Yestercode: Improving code-change support in visual dataflow programming environments. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 106–114, Sept 2016.

[41] A. Z. Henley, S. D. Fleming, and M. V. Luong. Toward principles for the design of navigation affordances in code editors: An empirical investigation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5690–5702, New York, NY, USA, 2017. ACM.

[42] A. Z. Henley, P. Rogers, and A. Sarma, editors. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[43] A. Z. Henley, A. Singh, S. D. Fleming, and M. V. Luong. Helping programmers navigate code faster with Patchworks: A simulation study. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '14, pages 77–80, July 2014.

[44] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4(1):11–26, 1952.

[45] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng. (ASE*, pages 14–23, 2007.

[46] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, May 2005.

[47] S. T. Iqbal and E. Horvitz. Disruption and recovery of computing tasks: Field study, analysis, and directions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 677–686, New York, NY, USA, 2007. ACM.

[48] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 178–187, New York, NY, USA, 2003. ACM.

[49] B. Johnson. *A Tool (Mis)communication Theory and Adaptive Approach for Supporting Developer Tool Use*. PhD thesis, North Carolina State University, 2017.

[50] W. P. Jones and S. T. Dumais. The spatial metaphor for user interfaces: Experimental tests of reference by location versus name. *ACM Trans. Inf. Syst.*, 4(1):42–63, 1986.

[51] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.

[52] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proc. 4th Int'l Conf. Aspect-oriented Software Development (AOSD)*, pages 159–168. ACM, 2005.

[53] K. Kevic, T. Fritz, and D. C. Shepherd. Comogen: An approach to locate relevant task context by combining search and navigation. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 61–70, Sept 2014.

[54] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 309–319. IEEE Computer Society, 2009.

[55] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, pages 50:1–50:11. ACM, 2012.

[56] A. Kittur, A. M. Peters, A. Diriye, T. Telang, and M. R. Bove. Costs and benefits of structured information foraging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2989–2998, New York, NY, USA, 2013. ACM.

[57] A. Ko. *Asking and Answering Questions about the Causes of Software Behavior*. PhD thesis, Carnegie Mellon University, 2008.

[58] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 126–135, New York, NY, USA, 2005. ACM.

[59] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[60] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[61] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[62] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, Dec. 2006.

[63] S. K. Kuttal, A. Sarma, and G. Rothermel. On the benefits of providing versioning support for end users: An empirical study. *ACM Trans. Comput.-Hum. Interact.*, 21(2):9:1–9:43, Feb. 2014.

[64] T. LaToza. *Answering Reachability Questions*. PhD thesis, Carnegie Mellon University, 2012.

[65] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194. ACM, 2010.

[66] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124. IEEE, 2011.

[67] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proc. CHI*, pages 1323–1332, 2008.

[68] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Softw. Eng.*, 39(2):197–215, Feb. 2013.

[69] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart. Reactive information foraging for evolving goals. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 25–34, New York, NY, USA, 2010. ACM.

[70] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM.

[71] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev.*, 63(2):81–97, 1956.

[72] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[73] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '10)*, pages 15–22, 2010.

[74] E. Murphy-Hill. *Programmer Friendly Refactoring Tools*. PhD thesis, Portland State University, 2009.

[75] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE '08)*, pages 421–430. IEEE, 2008.

[76] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 287–297. IEEE Computer Society, 2009.

[77] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.

[78] T. Nabi, K. M. D. Sweeney, S. Lichlyter, D. Piorkowski, C. Scaffidi, M. Burnett, and S. D. Fleming. Putting information foraging theory to work: Community-based design patterns for programming tools. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 129–133, Sept 2016.

[79] National Instruments. *LabVIEW Development Guidelines*, Apr. 2003. Part Number 321393D-01.

[80] K. O'Hara and A. Sellen. A comparison of reading paper and on-line documents. In *Proc. CHI*, pages 335–342, 1997.

[81] K. O'Hara, A. Sellen, and R. Bentley. Supporting memory for spatial location while reading from small displays. In *CHI Ext. Abstracts*, pages 220–221, 1999.

[82] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 859–869. IEEE Press, 2012.

[83] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. Van Gerven. Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist*, 38(1):63–71, 2003.

[84] F. G. Paas, J. J. Van Merriënboer, and J. J. Adam. Measurement of cognitive load in instructional research. *Perceptual and Motor Skills*, 79(1):419–430, 1994.

[85] C. Parnin. *Supporting Interrupted Programming Tasks with Memory-Based Aids*. PhD thesis, Georgia Institute of Technology, 2014.

[86] C. Parnin and R. DeLine. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 93–102, New York, NY, USA, 2010. ACM.

[87] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *Proc. 14th IEEE Int'l Conf. Program Comprehension*, ICPC '06, pages 13–22, 2006.

[88] C. Parnin, C. Gorg, and S. Rugaber. Taskboard: Tracking pertinent task artifacts and plans. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 317–318, May 2009.

[89] C. Parnin, C. Görg, and S. Rugaber. Codepad: Interactive spaces for maintaining concentration in programming environments. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 15–24, New York, NY, USA, 2010. ACM.

[90] C. Parnin, C. Görg, and S. Rugaber. Codepad: Interactive spaces for maintaining concentration in programming environments. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 15–24, New York, NY, USA, 2010. ACM.

[91] C. Parnin and S. Rugaber. Programmer information needs after memory failure. In *Proc. 20th IEEE Int'l Conf. Program Comprehension*, ICPC '12, pages 123–132, 2012.

[92] D. Piorkowski. *Information Foraging Theory as a Unifying Foundation for Software Engineering Research: Connecting the Dots*. PhD thesis, Oregon State University, 2016.

[93] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, CHI '12, pages 1471–1480, New York, NY, USA, 2012. ACM.

[94] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? How production bias affects developers' information foraging during debugging. In *31st IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 11–20, 2015.

[95] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 109–116, Sept. 2011.

[96] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett. Foraging and navigations, fundamentally: Developers' predictions of value and cost. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 97–108, New York, NY, USA, 2016. ACM.

[97] D. Piorkowski, S. Penney, A. Z. Henley, M. Pistoia, M. Burnett, O. Tripp, and P. Ferrara. Foraging goes mobile: Foraging while debugging on mobile devices. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–17, Oct 2017.

[98] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3063–3072. ACM, 2013.

[99] P. Pirolli and S. Card. Information foraging in information access environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 51–58, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[100] M. D. Plumlee and C. Ware. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ACM Trans. Comput.-Hum. Interact.*, 13(2):179–209, June 2006.

[101] G. A. Radvansky. *Human memory*. Routledge, 2017.

[102] S. S. Ragavan, B. Pandya, D. Piorkowski, C. Hill, S. K. Kuttal, A. Sarma, and M. Burnett. Pfis-v: Modeling foraging behavior in the presence of variants. In

Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, CHI '17, pages 6232–6244, New York, NY, USA, 2017. ACM.

[103] G. Robertson, M. Czerwinski, K. Larson, D. C. Robbins, D. Thiel, and M. van Dantzich. Data Mountain: Using spatial memory for document management. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, UIST '98, pages 153–162, New York, NY, USA, 1998. ACM.

[104] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

[105] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 15:1–15:2, New York, NY, USA, 2012. ACM.

[106] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[107] J. Smith, C. Brown, and E. Murphy-Hill. Flower: Navigating program flow in the ide. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 19–23, Oct 2017.

[108] R. B. Smith, J. Maloney, and D. Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 47–60, New York, NY, USA, 1995. ACM.

[109] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3509–3521, New York, NY, USA, 2016. ACM.

[110] S. L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[111] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 233–243. IEEE Press, 2012.

[112] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th Int'l Conf. on Software Engineering (ICSE '03)*, pages 408–418, 2003.

[113] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.

[114] J. S. Vitter. US&R: A new framework for redoing. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE '84)*, pages 168–176. ACM, 1984.

[115] A. Walenstein. *Cognitive support in software engineering tools: A distributed cognition framework*. PhD thesis, Simon Fraser University, 2002.

[116] F. W. Warr and M. P. Robillard. Suade: Topology-based searches for software investigation. In *Proc. ICSE*, pages 780–783, 2007.

[117] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.

[118] K. N. Whitley, L. R. Novick, and D. Fisher. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility. *Int. J. Hum.-Comput. Stud.*, 64(4):281–303, Apr. 2006.

[119] D. Windell and E. Wiebe. Measuring cognitive load in multimedia instruction: A comparison of two instruments. *Annual meeting of the American Educational Research Association*, 2007.

[120] Y. Yoon. *Backtracking Support in Code Editing*. PhD thesis, Carnegie Mellon University, 2015.

[121] Y. Yoon and B. A. Myers. Semantic zooming of code change history. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 95–99, Oct 2015.

[122] Y. Yoon, B. A. Myers, and S. Koo. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 119–126, Sept 2013.

[123] Y. S. Yoon and B. A. Myers. A longitudinal study of programmers' backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 101–108, July 2014.

[124] Y. S. Yoon and B. A. Myers. Supporting selective undo in a code editor. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 223–233, Piscataway, NJ, USA, 2015. IEEE Press.