Toward Principles for the Design of Navigation Affordances in Code Editors: An Empirical Investigation

Austin Z. Henley, Scott D. Fleming, Maria V. Luong
Department of Computer Science
University of Memphis
Memphis, Tennessee, USA
{azhenley, Scott.Fleming, mluong}@memphis.edu

ABSTRACT

Design principles are a key tool for creators of interactive systems; however, a cohesive set of principles has yet to emerge for the design of code editors. In this paper, we conducted a between-subjects empirical study comparing the navigation behaviors of 32 professional LabVIEW programmers using two different code-editor interfaces: the ubiquitous tabbed editor and the experimental Patchworks editor. Our analysis focused on how the programmers arranged and navigated among open information patches (i.e., code modules and program output). Key findings of our study included that Patchworks users made significantly fewer click actions per navigation, juxtaposed patches side by side significantly more, and exhibited significantly fewer navigation mistakes than tabbed-editor users. Based on these findings and more, we propose five general principles for the design of effective navigation affordances in code editors.

ACM Classification Keywords

D.2.6 Software Engineering: Programming Environments; H.5.m. Information Interfaces and Presentation (e.g., HCI): Miscellaneous.

Author Keywords

Programming environments; navigation; design principles; visual programming languages; user study.

INTRODUCTION

Design principles have proven to be a key tool for designers of interactive systems. Principles help guide the design of a system by providing a set of heuristics and best practices, often based on empirical evidence. Moreover, such principles serve to inspire new design decisions and to facilitate the comparison of competing designs. Without principles, designers are left to rely on intuition, increasing the risk of wasting time and effort relearning which designs are effective and of repeating past mistakes. Design principles have been proposed to help designers in a variety of different contexts, including the design of mixed-initiative user interfaces [17], of web pages that support information foraging [47], of affordances for debugging machine-learning systems [28], of affordances to help end-user programmers learn enough to overcome barriers [19], and of games for computer science education [30].

One actively studied domain of interactive systems that has yet to benefit from a cohesive set of design principles is code editors. Numerous code-editor designs have been proposed over the years. Early designs emphasized a windowed paradigm in which each open code file was displayed in a separate window (e.g., Visual C++ 6.0). More recent development environments have favored tabbed editor interfaces (e.g., as in Eclipse and Visual Studio). Dissatisfied with the status quo, researchers have also proposed more radical designs. For example, canvas-based editors enable programmers to arrange open code fragments on a large 2D canvas (e.g., as in Self [46], Jasper [6], Code Canvas [12], and Code Bubbles [4]). Despite all this design exploration, a set of effective design principles for code editors has yet to be codified.

In this work, we aim to create a cohesive set of design principles targeting *navigation affordances* for code editors. Code navigation is a key concern in code-editor design. Studies have shown that programming tasks typically involve large amounts of information seeking. For example, one study found that programmers spent 50% of their time foraging for information during debugging tasks [40]. As a consequence of all this information seeking, navigation takes up a significant portion of the time programmers spend on tasks. One study found that programmers spent 35% of their time on the mechanics of navigation alone [23], and another found that programmers averaged over 5 navigations per minute during programming tasks [14]. Thus, navigation affordances can have a substantial impact on a programmer's time and effort on task.

Toward our goal of design principles, we have been exploring *Patchworks* [15], an experimental code-editor design. In contrast to tabbed editors, Patchworks is based on the idiom of a sliding grid strip of code fragments. In a preliminary evaluation [15], Patchworks users navigated significantly faster and made fewer navigation mistakes than users of a tabbed editor (Eclipse). A subsequent simulation study [16] also showed that Patchworks can significantly increase the number

CHI 2017, May 6–11, 2017, Denver, CO, USA.
© 2017 ACM ISBN 978-1-4503-4655-9/17/05...$15.00.
http://dx.doi.org/10.1145/3025453.3025645

of highly efficient *on-screen navigations* (i.e., navigations to code already visible on screen) that programmers make.

Although these preliminary evaluations of Patchworks showed promise, they had key limitations. The participants were CS graduate students, leaving the question of whether the results will generalize to professional developers. Moreover, the initial evaluation involved artificial navigation tasks that enabled precise recording of navigation times, but did not explore more natural development activities, such as debugging. In the second study, navigation logs of Java programmers were used to simulate the experience of a Patchworks user; however, Patchworks' effect on navigation decisions was not tested.

In this paper, we present a new empirical study that both addresses our goal of principles as well as overcomes the limitations of prior Patchworks evaluations. In particular, we performed an observational user study that compared the ubiquitous tabbed editor design with the experimental Patchworks design to inductively build a set of design principles. To address ecological validity, participants in our new study consisted of 32 professional programmers at a large software company, and the tasks they performed were more-naturalistic program-debugging tasks. The central goals of our study were

(1) to evaluate how well Patchworks achieves its goals,
(2) to investigate other navigation-related trends not specifically addressed by Patchworks, but relevant to the design of code editors, and
(3) to create a set of design principles based on our findings.

As our subjects of study, we selected experts and tasks in a visual dataflow programming language, LabVIEW [8]. Most prior research on programmer navigation has studied textual programming languages (especially Java). A key reason for choosing a visual dataflow language was that we seek to create a universal set of editor-design principles and to avoid overfitting the principles to the idiosyncrasies of a particular programming language or paradigm. Additionally, this choice afforded us the opportunity to compare the navigation characteristics of visual language programmers to those of textual-language programmers from prior research.

Based on this work, we make the following key contributions:

- Findings of the most ecologically valid evaluation of Patchworks to date.
- Five empirically grounded principles for the design of navigation affordances in code editors based on the findings.
- The first empirical comparison of navigation characteristics of visual dataflow language (LabVIEW) programmers and textual language (Java) programmers.
- An implementation of the Patchworks editor design for the LabVIEW development environment.

BACKGROUND

Programmer Navigation

The literature on programmer navigation can help shed light on the navigation affordances programmers will find effective. As mentioned above, programmers spend a considerable portion of their time on tasks navigating [14, 23]. One reason for this navigation is that programmers engage extensively in information foraging to locate task-relevant code [23]. This process typically involves traversing relationships between fragments of code (e.g., following control-flow dependencies) [22]. Due to the size and complexity of software projects, programmers may spend considerable time inspecting irrelevant code [23, 39], or even "getting lost" in the code [10, 11].

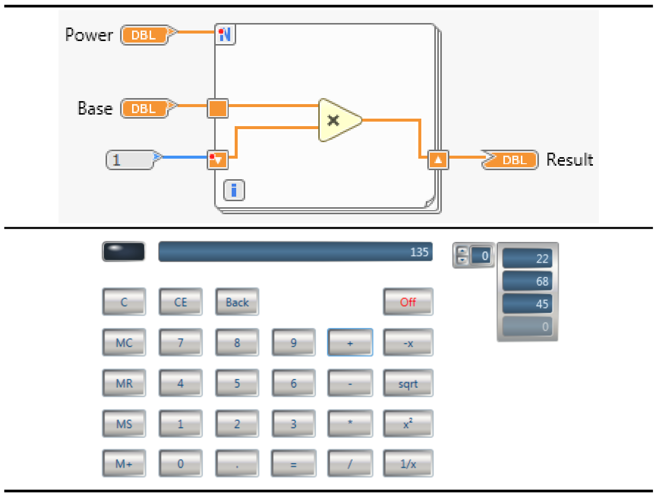

Figure 1. LabVIEW block diagram (top) and front panel (bottom).

One key pattern is that most navigations tend to revisit code the programmer recently visited. For example, studies of predictive models of programmer navigation have consistently found that one of the strongest predictors of which code method (i.e., subroutine) a programmer would click in next was how recently he/she had visited the method (i.e., more recently implies more likely) [10, 26, 29, 33, 38, 45]. Studies have also found that between 82% [16, 45] and 95% [33] of programmer navigations were to previously visited methods.

Recently, researchers have proposed novel interface designs to support revisiting. Tools such as Mylyn [21], Jasper [6], Code Bubbles [3, 4], Code Canvas [12], and Patchworks [15] enable programmers to collect and easily revisit code fragments. Other tools, such as Blaze [27] and Stacksplorer [20], provide visualizations that enable programmers to traverse code relationships. The designs of these tools range widely, and in this paper, we aim to distill key principles to help tool designers provide transformative navigation capabilities without sacrificing the essential navigation needs of programmers.

Visual Dataflow Languages

Most prior work on programmer navigation has focused on Java and Java-like programming languages, so we opted to focus our study on visual dataflow languages to mitigate the effects of Java's idiosyncrasies on our findings and to shed light on the navigation behaviors of visual dataflow programmers. Instead of textual source code, these languages are characterized by programs being made up of boxes (functions) and arrows (values), as illustrated at the top of Fig. 1. The execution of the program follows the dataflow via the arrows. Thus, visual dataflow languages have a radically different syntax than Java-like textual languages while still providing many of the same foundational features, such as modularity.

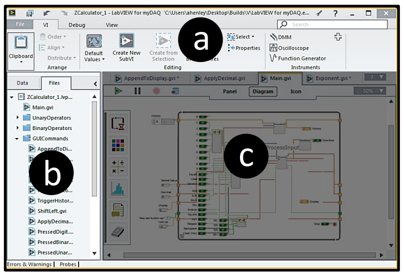

Figure 2. The LabVIEW IDE. The command ribbon (a) and the project explorer (b) were the same for all participants; however, the code editor (c) varied (tabbed vs. Patchworks).

In this paper, we focus on LabVIEW, a commercial visual dataflow programming environment that is one of the most widely used visual programming languages to date [49]. In LabVIEW, programs are composed of modules, called *Virtual Instruments* (VIs). Fig. 1 (top) illustrates the code for a VI, represented as a *block diagram*, that takes 3 inputs, performs a multiplication in a loop, then outputs the result. In LabVIEW, each block diagram is associated with a graphical user interface, called a *front panel*, as shown at the bottom of Fig. 1.

CODE EDITORS COMPARED

In our study, we compared the navigation affordances of two code-editor designs: the ubiquitous tabbed code editor and a research design, the Patchworks code editor [15]. Both editors were implemented within the LabVIEW integrated development environment (IDE) depicted in Fig. 2. The LabVIEW IDE provides features similar to those of other popular IDEs (e.g., Eclipse and Visual Studio), including a project explorer for opening VIs (Fig. 2b) as well as debugger and search tools accessible via the command ribbon (Fig. 2a).

Borrowing from information foraging theory [41], we assume a programmer's task environment to have a "patchy" structure, where a *patch* may be a code file, web page, or other interface that the programmer reads or edits as a unit. In LabVIEW, each VI is associated with two patches: its block diagram and its front panel. The programmer *navigates* by moving his/her attention from one patch to another.

All editors that we are aware of, including the two under study, maintain a set of "open" patches that the programmer has accessed (or *opened*). The tabbed and Patchworks editors differ mainly in the affordances they provide for displaying, organizing, and navigating among open VI patches, and we detail those differences below.

Tabbed Code Editor

Tabbed code editors are based on an interface paradigm similar to paper file folders, as shown in Fig. 3. Currently, they are by far the most popular code-editor interface. Tabbed editors default to a *single-patch display* in which the contents of one patch fill most of the editor's available space (Fig. 3c). Each open patch has a labeled "tab", like a paper file folder (Fig. 3a). The programmer may navigate among open patches by clicking on the tabs. If too many patches are open, some tabs are elided and made available via a drop-down menu (Fig. 3b). The programmer may reorder tabs by dragging and

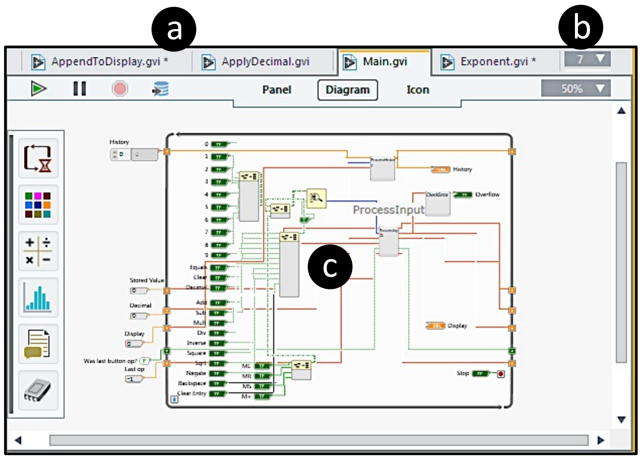

Figure 3. The tabbed code editor. Tabs (a) could be used to switch among open patches. If too many patches were open, some of the tabs were elided (b). This single-patch display (c) filled the rest of the pane with one VI patch.

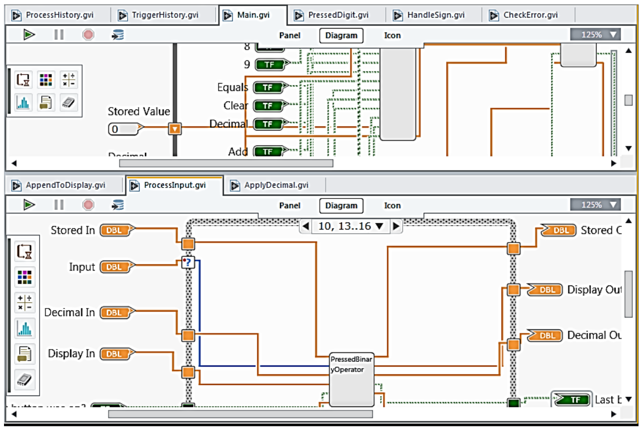

Figure 4. Tabbed editor manipulated into a multi-patch display, allowing two VI patches to be placed side by side.

dropping, and may close a patch by clicking a button on its tab. The programmer may also manipulate a tabbed editor into a *multi-patch display* (in which two patches are placed side by side) by splitting the editor pane into two panes, and dragging and dropping tabs between the panes, as illustrated in Fig. 4.

Patchworks Code Editor

The Patchworks code editor is an experimental design that aims to ease navigation among open patches. In this work, 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. 5, the Patchworks editor's main display consists of a *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

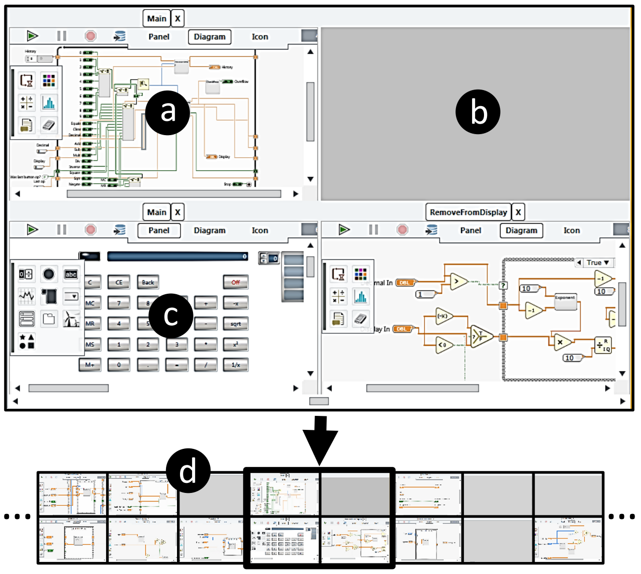

Figure 5. The Patchworks code editor. This multi-patch display divides the editor pane into a patch grid with four patch cells. Each patch cell may contain a block diagram (a) or front panel (c) for a VI, or it may be empty (b). The displayed patch grid is actually a view into a larger patch strip (d), and the programmer can navigate by sliding the view left or right along the patch strip.

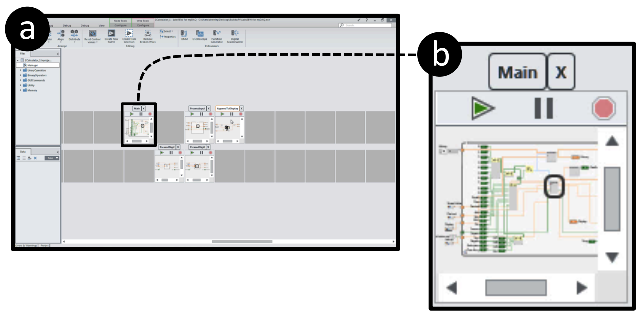

Figure 6. Patchworks' bird's eye view (a) enables the user to zoom out, and survey and organize his/her open patches. It represents each patch with a thumbnail (b).

for Java [15] 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.

To enable navigation beyond the patches visible on screen, the patch grid is actually a view into a larger *patch strip*, illustrated by Fig. 5d. The programmer navigates among patches by sliding the view left or right along the patch strip using a keyboard shortcut or horizontal scrollbar. This 1-dimensional patch strip aims to reduce navigation mistakes by simplifying the space of possible navigation actions (slide left or right) and to facilitate efficient visual scanning of open patches. Additionally, 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. 6. The programmer can rearrange patches from the bird's eye view via the same dragging and dropping interactions as in the patch-grid view, and can zoom back into the patch-grid view by clicking on a selected patch.

Patchworks also 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.

METHOD

To address our research goals, we conducted an observational laboratory study of professional LabVIEW programmers 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.

Participants

Our participants consisted of 32 professional programmers 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.

Code Base and Tasks

For our study, the participants worked on a calculator application, illustrated in Fig. 1 (bottom), written in LabVIEW, consisting of 34 VIs. The application was based on a publicly available sample application [7] 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.

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.

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 [16], 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.

RESULTS

To address our research goals, we collected and analyzed over 21 hours of video of our 32 professional programmers 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 programmers' navigation traits with those of Java programmers in prior studies.

Patchworks Evaluation Results

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. 7 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

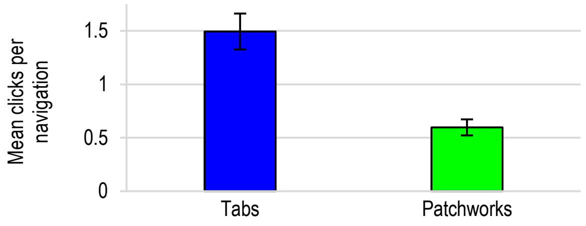

Figure 7. Patchworks participants performed significantly fewer click actions per navigation than did Tabs participants (smaller bars are better). Whiskers denote standard error.

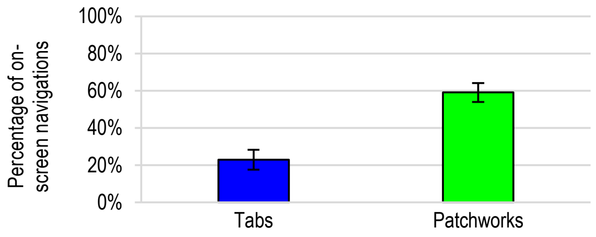

Figure 8. Patchworks participants made significantly more on-screen navigations than did Tabs participants. Whiskers denote standard error.

during their tasks. In contrast, only 10 of 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. 8 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: $M = 1.80$, $SD = 0.58$; 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

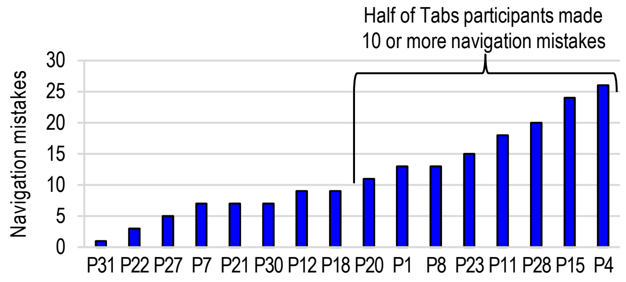

Figure 9. Tabs participants made numerous navigation mistakes in which they clicked on a tab other than the one they intended. All Tabs participants made at least one such mistake, and half of Tabs participants made 10 or more.

navigation mistakes in which they were seeking a particular patch and clicked the wrong tab to get there. As Fig. 9 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.

Trends across Treatments

In addition to evaluating Patchworks' design goals, we checked for three key trends in how programmers 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

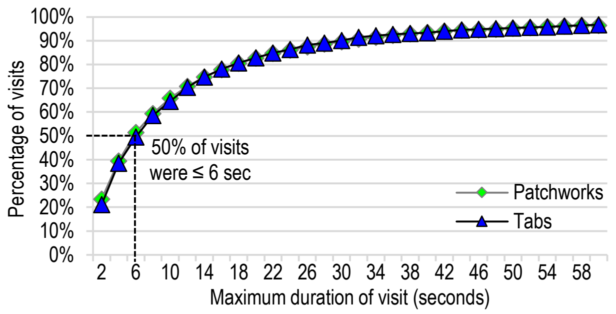

Figure 10. Participants often visited patches for very short intervals, which is consistent with quick scanning. In fact, half of visits lasted 6 seconds or less.

of patches. One key indicator of this behavior was that most of the visits to patches that the participants made were quick. As Fig. 10 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 [13], 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. 11 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 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 programmers 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

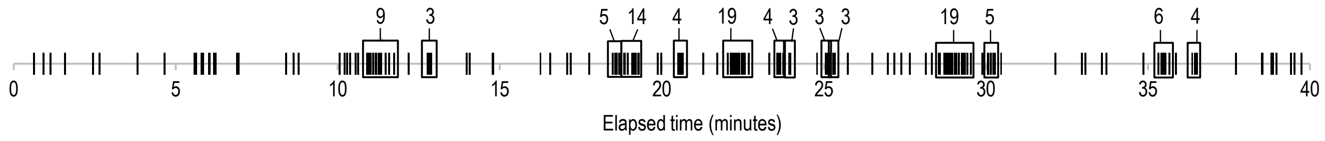

Figure 11. Navigation timeline for P17 with DBSCAN clusters highlighted and cluster size labeled.

Table 1. 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
[14] Fritz et al. FSE'14	5 professionals, 4 CS students, 3 CS faculty	Feature modification	75 min	Java	14–53k LOC
[16] Henley et al. VL/HCC'14	14 CS students	Feature addition	120 min	Java	3–17k LOC
[36] Piorkowski et al. CHI'12	11 professionals	Debugging	70 min	Java	97k LOC
[37] 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

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% ($SD = 7.9$).

Visual Dataflow versus Textual Language Programmers

To compare the navigation behavior of textual language programmers and visual dataflow language programmers, we compared our data with results reported by four prior studies (i.e., [14, 16, 36, 37]). We focused our comparison on two key navigations traits: (1) how much the programmers navigated and (2) how much the programmers revisited patches. In the prior studies, the patches among which programmers navigated were Java methods. For our comparison analysis, we operationalized patches as VI block diagrams, which are analogous to methods. Table 1 describes each of the prior textual-language studies.

As Figs. 12 and 13 show, the visual dataflow language programmers exhibited similar navigation traits to the textual language programmers 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 programmers from the literature (Fig. 12). Furthermore, our visual dataflow participants also revisited patches frequently—even more so than prior textual language programmers (Fig. 13).

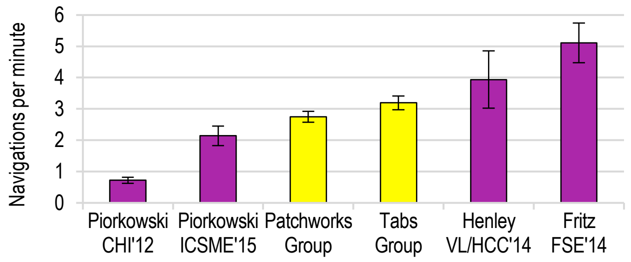

Figure 12. Our visual dataflow language programmers (light bars) navigated at rates similar to prior textual language programmers (dark bars). Whiskers denote standard error.

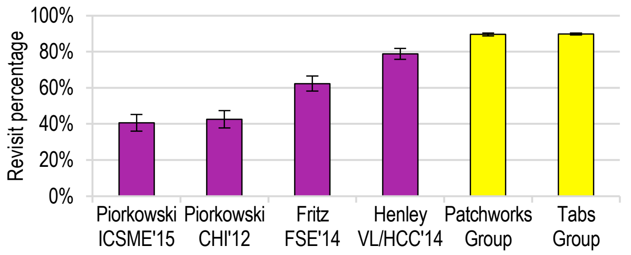

Figure 13. Like prior textual language programmers (dark bars), our visual dataflow language programmers (light bars) revisited patches frequently. Whiskers denote standard error.

DISCUSSION

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 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 2.

Table 2. 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).

The Juxtaposition Principle

The Juxtaposition Principle emphasizes enabling programmers 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 programmer 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. 8). 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.

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. 6b).

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. 9). 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."

The Blowup Principle

The Blowup Principle emphasizes providing a programmer 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."

The Cleanup Principle

The Cleanup Principle emphasizes helping programmers to efficiently close open patches that are cluttering the workspace and are not currently relevant to what the programmer 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 programmers to perform cleanup *without closing relevant patches* was an important consideration in the Cleanup Principle.

The Heterogeneity Principle

The Heterogeneity Principle emphasizes facilitating programmer 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.

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 programmers want to juxtapose code and doing so enhances their ability to navigate. For example, one study found that programmers expressed wanting to juxtapose code patches [2]; however, similar to our Tabs participants, they rarely do because of the tediousness of juxtaposing patches in tabbed editors, such as Eclipse [2, 4, 25]. 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 [15].

Furthermore, the importance of being able to rearrange patches efficiently in multi-patch displays has also been well motivated. For example, Plumlee and Ware [42] 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 programmers, because studies have found that their goals change rapidly during programming tasks [29, 36], thus, suggesting that they will need to rearrange often. However, in modern tabbed code editors, such rearranging is tedious, and as a consequence, programmers rarely do so [4, 25].

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 programmers 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 [32, 35]. In fact, one study showed that programmers frequently had to check the content of patches associated with tabs because they could not infer the content from the labels on the tabs [44]. 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 [43] and to help programmers navigate large code files and code bases [9]. 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 programmers 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 programmers clean up only irrelevant code. For example, studies have shown that programmers often close all of their open patches to proceed from a clean slate [11, 34]. Many modern code editors provide "close all" features; however, wiping all tabs has also been found to be problematic, because programmers actually want to keep some of the patches open. For example, one study found that programmers wasted an average of 60 seconds to reopen the patches they wanted after closing too many of them [23]. Others have similarly reported on the costliness of recovering such state after suspending a task due to the loss of contextual cues [18]. 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 programmers' feedback loop of running the application, interacting with it, and observing the runtime behavior. For example, the Whyline programming environment allows a programmer to ask questions about a program's output and provides direct links to the relevant code [24]. Another tool, the Theseus programming environment, annotates JavaScript source code with runtime information, thus, combining program code and output into a single patch [31]. Still other tools have provided navigational links between code and other types of artifacts, such as emails and bug reports [48] and related code examples from the web [5]. Such tools further illustrate the potential benefits of considering heterogenous patch types in the design of code editors.

Limitations
Our study had several limitations to generalizability common to laboratory studies of programmers. For example, there is a question as to how well the sample of programmers 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 programmers, increasing the likelihood that they were representative of expert programmers 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.

CONCLUSION
In this paper, we have presented an empirical study comparing the navigation behaviors of professional LabVIEW programmers using two different code-editor interfaces: the ubiquitous tabbed editor and the experimental Patchworks editor. Key findings of our study included the following:

- Patchworks users performed significantly fewer clicks per navigation than users of the tabbed editor.
- Patchworks users juxtaposed patches significantly more than users of the tabbed editor, and as a consequence, also made more highly efficient on-screen navigations.
- Patchworks users exhibited significantly fewer navigation mistakes than users of the tabbed editor.
- All programmers, regardless of editor, tended to let open patches pile up and then clean them up by closing them en masse; however, following such closings, they often had difficulty locating patches they had inadvertently closed.
- All programmers, regardless of editor, frequently navigated between code and program-output patches.
- Like textual language programmers in prior studies, the visual dataflow language programmers in our study exhibited high rates of navigation and frequently revisited patches.

Based on these findings, we proposed five principles for the design of effective navigation affordances in code editors.

Moving forward, we hope that the principles we proposed will help serve as an invaluable resource and foundation for the designers of code editors. There continues to be considerable enthusiasm for the design of new navigation affordances for code editors (e.g., [4, 12, 15, 20, 27]). However, prior to this work, a cohesive set of guidance on the design of these complex, multifaceted interfaces has been sorely lacking. Thus, our proposed principles help fill an important gap that should enable designers to explore imaginative new features while ensuring that the essential navigation needs of programmers continue to be met effectively.

ACKNOWLEDGMENTS
We give special thanks to Andrew Dove for his counsel on all things LabVIEW. This material is based upon work supported by National Instruments and by the National Science Foundation under Grant No. 1302117. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Instruments or of the National Science Foundation.

REFERENCES
1. John R. Anderson and Gordon H. Bower. 1972. Recognition and retrieval processes in free recall. *Psychol. Rev.* 79, 2 (March 1972), 97–123.
2. Andrew. Bragdon. 2009. *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.

3. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 2010a. 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)*. ACM, New York, NY, USA, 455–464.

4. Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 2010b. 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)*. ACM, New York, NY, USA, 2503–2512.

5. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: Integrating Web search into the development environments. In *Proc. 28th Int'l Conf. on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 513–522.

6. Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. 2006. JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange (ETX '06)*. ACM, New York, NY, USA, 65–69.

7. National Instruments Corporation. 2008. LabVIEW Calculator. http://www.ni.com/example/30779/ Accessed: 2017-01-06.

8. National Instruments Corporation. 2017. LabVIEW System Design Software. http://www.ni.com/labview/ Accessed: 2017-01-06.

9. Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. 2006. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *Proceedings of the Visual Languages and Human-Centric Computing (VL/HCC '06)*. IEEE Computer Society, Washington, DC, USA, 11–18.

10. Robert DeLine, Mary Czerwinski, and George Robertson. 2005. Easing Program Comprehension by Sharing Navigation Data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '05)*. IEEE Computer Society, Washington, DC, USA, 241–248.

11. Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. 2005. Towards Understanding Programs Through Wear-based Filtering. In *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis '05)*. ACM, New York, NY, USA, 183–192.

12. Robert DeLine and Kael Rowan. 2010. Code Canvas: Zooming Towards Better Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 207–210.

13. Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Proc. 2nd Int'l Conf. Knowledge Discovery and Data Mining (KDD '96)*. 226–231.

14. Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. 2014. Developers' Code Context Models for Change Tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 7–18.

15. Austin Z. Henley and Scott D. Fleming. 2014. 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)*. ACM, New York, NY, USA, 2511–2520.

16. Austin Z. Henley, A. Singh, Scott D. Fleming, and Maria V. Luong. 2014. 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)*. 77–80.

17. Eric Horvitz. 1999. Principles of Mixed-initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 159–166.

18. Shamsi T. Iqbal and Eric Horvitz. 2007. 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)*. ACM, New York, NY, USA, 677–686.

19. W. Jernigan, A. Horvath, M. Lee, M. Burnett, T. Cuilty, S. Kuttal, A. Peters, I. Kwan, F. Bahmani, and A. Ko. 2015. A principled evaluation for a principled Idea Garden. In *Proc. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*. 235–243.

20. Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. 2011. 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)*. ACM, New York, NY, USA, 217–224.

21. Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '06)*. ACM, New York, NY, USA, 1–11.

22. Katja Kevic, Braden M. Walters, Timothy R. Shaffer, Bonita Sharif, David C. Shepherd, and Thomas Fritz. 2015. Tracing Software Developers' Eyes and Interactions for Change Tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 202–213.

23. Andrew J. Ko, Htet Aung, and Brad A. Myers. 2005. 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)*. ACM, New York, NY, USA, 126–135.

24. Andrew J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1569–1578.

25. Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987.

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

27. Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2012. Blaze: Supporting Two-phased Call Graph Navigation in Source Code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems (CHI EA '12)*. ACM, New York, NY, USA, 2195–2200.

28. Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces (IUI '15)*. ACM, New York, NY, USA, 126–137.

29. Joseph Lawrance, Margaret Burnett, Rachel Bellamy, Christopher Bogart, and Calvin Swart. 2010. Reactive Information Foraging for Evolving Goals. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 25–34.

30. Michael J. Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, Sheridan Long, Margaret Burnett, and Andrew J. Ko. 2014. Principles of a debugging-first puzzle game for computing education. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*. 57–64.

31. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490.

32. Chris Parnin and Robert DeLine. 2010. Evaluating Cues for Resuming Interrupted Programming Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 93–102.

33. Chris Parnin and Carsten Görg. 2006. Building Usage Contexts During Program Comprehension. In *Proc. 14th IEEE Int'l Conf. Program Comprehension (ICPC '06)*. 13–22.

34. Chris Parnin, Carsten Görg, and Spencer Rugaber. 2010. CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 15–24.

35. Chris Parnin and Spencer Rugaber. 2012. Programmer information needs after memory failure. In *Proc. 20th IEEE Int'l Conf. Program Comprehension (ICPC '12)*. 123–132.

36. David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. 2012. 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)*. ACM, New York, NY, USA, 1471–1480.

37. David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. 2015. 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)*. 11–20.

38. David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret Burnett, and Rachel Bellamy. 2011. 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)*. 109–116.

39. David Piorkowski, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. 2016. 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)*. ACM, New York, NY, USA, 97–108.

40. David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. 2013. The Whats and Hows of Programmers' Foraging Diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, 3063–3072.

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

42. Matthew D. Plumlee and Colin Ware. 2006. Zooming Versus Multiple Window Interfaces: Cognitive Costs of Visual Comparisons. *ACM Trans. Comput.-Hum. Interact.* 13, 2 (June 2006), 179–209.

43. George Robertson, Mary Czerwinski, Kevin Larson, Daniel C. Robbins, David Thiel, and Maarten van Dantzich. 1998. Data Mountain: Using Spatial Memory for Document Management. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '98)*. ACM, New York, NY, USA, 153–162.

44. Janice Singer, Robert Elves, and Margaret-Anne Storey. 2005. NavTracks: Supporting Navigation in Software Maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, Washington, DC, USA, 325–334.

45. Alka Singh, Austin Z. Henley, Scott D. Fleming, and Maria V. Luong. 2016. An Empirical Evaluation of Models of Programmer Navigation. In *IEEE Int'l Conference on Software Maintenance and Evolution (ICSME '16)*. 9–19.

46. Randall B. Smith, John Maloney, and David Ungar. 1995. 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)*. ACM, New York, NY, USA, 47–60.

47. Jared M. Spool, Christine Perfetti, and David Brittan. 2004. *Designing for the Scent of Information*. User Interface Engineering, Middleton, MA.

48. Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. 2005. Hipikat: A Project Memory for Software Development. *IEEE Trans. Softw. Eng.* 31, 6 (2005), 446–465.

49. Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. 2006. Evidence in Favor of Visual Representation for the Dataflow Paradigm: An Experiment Testing LabVIEW's Comprehensibility. *Int. J. Hum.-Comput. Stud.* 64, 4 (April 2006), 281–303.