The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes

Austin Z. Henley
University of Memphis
Memphis, Tennessee, USA
azhenley@memphis.edu

Scott D. Fleming
University of Memphis
Memphis, Tennessee, USA
Scott.Fleming@memphis.edu

ABSTRACT

Increasingly, people are faced with navigating large information spaces, and making such navigation efficient is of paramount concern. In this paper, we focus on the problems programmers face in navigating large code bases, and propose a novel code editor, Patchworks, that addresses the problems. In particular, Patchworks leverages two new interface idioms—the patch grid and the ribbon—to help programmers navigate more quickly, make fewer navigation errors, and spend less time arranging their code. To validate Patchworks, we conducted a user study that compared Patchworks to two existing code editors: the traditional file-based editor, Eclipse, and the newer canvas-based editor, Code Bubbles. Our results showed (1) that programmers using Patchworks were able to navigate significantly faster than with Eclipse (and comparably with Code Bubbles), (2) that programmers using Patchworks made significantly fewer navigation errors than with Code Bubbles or Eclipse, and (3) that programmers using Patchworks spent significantly less time arranging their code than with Code Bubbles (and comparably with Eclipse).

Author Keywords

Integrated development environment (IDE); code editor; navigation; user study.

ACM Classification Keywords

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

INTRODUCTION

Increasingly, people are faced with navigating large information spaces. For example, foraging for information has been recognized as a key part of sensemaking in such domains as intelligence analysis [11] and end-user programming [13]. To aid people in such activities, researchers have sought methods to make navigation more efficient (e.g., [12]). In this paper, we focus specifically on the problem that programmers face in efficiently navigating source code. The navigation problem is particularly acute in programming, where modern programs may comprise millions of lines of code, organized into hundreds of thousands of interrelated modules.

CHI 2014, April 26–May 1, 2014, Toronto, Ontario, Canada.

http://dx.doi.org/10.1145/2556288.2557073

Traditionally, programmers main means of navigating code has been the *file-based code editor* (e.g., the Eclipse Java editor); however, there is growing concern about how much time programmers spend on navigation in such editors. For example, one study found that programmers spent 35 percent of their time on the mechanics of navigation [18]. Thus, a variety of approaches have been proposed to make navigation less time consuming.

One of the most common approaches is to augment a file-based editor with shortcuts that the programmer can use to instantly "jump" to locations in the code. Such approaches have explored a variety of ways to generate shortcuts, such as using structural relationships in code (e.g., [20]), programmers' natural language queries (e.g., [15]), collaborative tagging (e.g., [31]), and programmer interaction behavior (e.g., [26]). However, since these approaches support only certain types of navigations, they do not entirely free programmers from the inefficiencies of navigation in file-based editors.

To directly address the problems of file-based editors, researchers have proposed a new paradigm of *canvas-based code editors* (e.g., [3, 4, 9]). These editors aim to improve programmer navigation efficiency by enabling programmers 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 programmers using Code Bubbles made, on average, nearly half the navigation actions per minute as programmers using the file-based editor, Eclipse [3]. 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. Programmers 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 programmer will fail to visually locate a fragment he/she seeks. Additionally, the 2D canvas idiom may lead programmers to spend extra time and effort on arranging fragments; however, such time may not be well spent if the programmer is not far enough along in the sensemaking process to choose effective arrangements, or if arrangements quickly become obsolete because the programmer's information needs change rapidly (as recent research suggests [26]).

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 programmers 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 programmers 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.

To validate the benefits that Patchworks provides, we conducted an empirical evaluation that pitted Patchworks against a representative file-based editor, Eclipse, and a representative canvas-based editor, Code Bubbles. In particular, the evaluation investigated three research questions:

- RQ1: Do programmers using Patchworks navigate between fragments of code more quickly?
- RQ2: Do programmers using Patchworks make fewer navigation mistakes?
- RQ3: Do programmers using Patchworks spend less time arranging code fragments?

Our work makes several key contributions. First, we introduce a novel code-editor design for efficient code navigation, Patchworks. Second, we contribute a working prototype of Patchworks for Java programming. Third, we contribute the results of an empirical evaluation that show (1) that programmer navigation in Patchworks and in Code Bubbles was significantly faster than in Eclipse, (2) that programmers made significantly fewer navigation mistakes in Patchworks than in Eclipse or in Code Bubbles, and (3) that programmers spent significantly less time arranging code fragments in Patchworks and in Eclipse than in Code Bubbles.

BACKGROUND

How Programmers Navigate Code

Empirical studies have found that programmers spend considerable time navigating among fragments of code. For example, one study found that programmers spent, on average, 35 percent of their time navigating between relevant code fragments [18]. One reason for this navigation is that programmers spend a lot of time seeking information in their code environment. One recent study found that programmers spent 50 percent of their time foraging for information [28]. Similarly, another study found that information "scent" was a significant predictor of where programmers would navigate [21].

One reason for all this information foraging may be that a primary challenge in debugging or adding a feature to a piece of software is identifying all the code relevant to the task, which is called the programmer's *working set* [19]. The process of identifying this code typically involves traversing relationships between fragments of code (e.g., following control flow dependencies). However, programmers may spend considerable time inspecting irrelevant code, as one study found [18], or even "getting lost" in the code, as another found [8].

In addition to exploratory navigations for finding relevant code, programmers also navigate frequently within their working sets. For example, one study of predictive models of programmer navigation found that the strongest predictor of which method a programmer would click in next was how recently he/she had visited the method (more recently implies more likely) [27]. This finding agrees with another study of models of programmers' information foraging behavior that found that the foraging model that took recency into account was the strongest predictor [22]. Thus, the programmers tended to repeatedly revisit the same code fragments.

However, these recency results may also suggest that the code sought by a programmer evolves as the task progresses. In particular, navigation decisions may be tied to the programmer's evolving information goals during information foraging. One recent study supported this idea with the finding that using only the last (one) navigation to predict where a programmer would go next produced more accurate predictions than using his/her last ten navigations [26].

As programmers navigate to revisit code fragments, their *spatial memory* may affect how efficiently they locate the fragments. Spatial memory is the ability to remember the location or orientation of objects. It has been leveraged to improve users' navigation times in graphical user interfaces [10], web sites [5], file systems [12, 16], and source code [7]. For example, Code Thumbnails [7] aids programmers in building spatial memory of source code by giving them a zoomed-out visualization of code files, laid out on a 2D canvas.

File-Based Code Editors

At present, file-based editors are the dominant tool paradigm that programmers use to create, modify, and navigate source code, and the empirical evidence above is mostly based on programmers using such editors. Most modern development environments, including Eclipse, NetBeans, and Visual Studio, employ file-based editors. In this paper, we will focus on the Eclipse[1] code editor as an exemplar of this paradigm. 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). Since the editors we will be discussing do not vary significantly in this regard, we will say little more about such features. To navigate among code fragments (such as methods) within a file, the main capability that file-based editors provide is vertical scrolling. To navigate among open files, editors typically employ tabs that the user can click or cycle through using keyboard shortcuts (Fig. 1B).

Given the considerable time that programmers spend navigating in Eclipse [18], several qualities of traditional file-based editors may hinder efficient navigation. First, because programmers' 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

[1] `http://www.eclipse.org/`

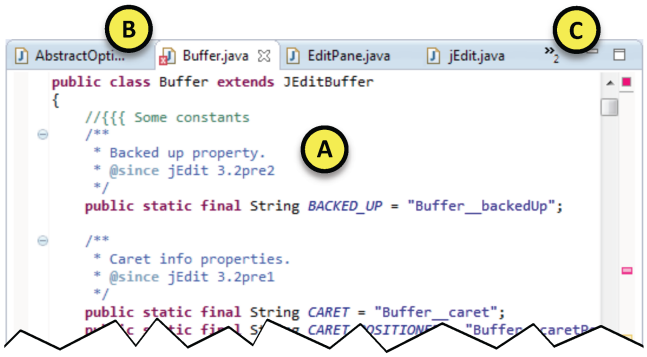

Figure 1. Eclipse's file-based editor, including (A) a vertically scrollable file, (B) visible tabs, and (C) two elided tabs.

cases, the programmer 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 [24, 25].

Second, because a programmer'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 programmer has too many tabs open, editors will typically begin eliding tabs, as in Fig. 1C. This automatic eliding may also disrupt the programmer's spatial memory. Moreover, revealing the elided tabs and choosing one involves additional clicking, which takes time.

Third, if a programmer 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, programmers rarely use these features [3, 19]. Moreover, prior research suggests that programmers actually want to juxtapose code [1], but find doing so in file-based editors inconvenient [3, 19].

To address the inefficiencies of navigation using file-based editors, a number of approaches augment such editors by providing shortcuts (i.e., hyperlinks) to places in the code. One common approach is linking code elements in the editor to structurally related elements. For example, in Eclipse, a programmer can select a variable name in the editor, and in a single action, link to that variable's declaration. Other approaches have included searching based on natural language queries (e.g., [15]), enabling programmers to leave tags in the code and provide shortcuts to those tags (e.g., [31]), and using programmers navigation histories to infer and link to places in the code that might be relevant (e.g., [8, 26]). However, none of these approaches have superceded the file-based editor paradigm, and programmers still commonly use scrolling and tabbing features to navigate between fragments of code, and thus, experience the inefficiencies discussed above.

Canvas-Based Code Editors

To address the problems with file-based editors, researchers have proposed a new paradigm of canvas-based code editors that enable programmers to work with more fine-grain code

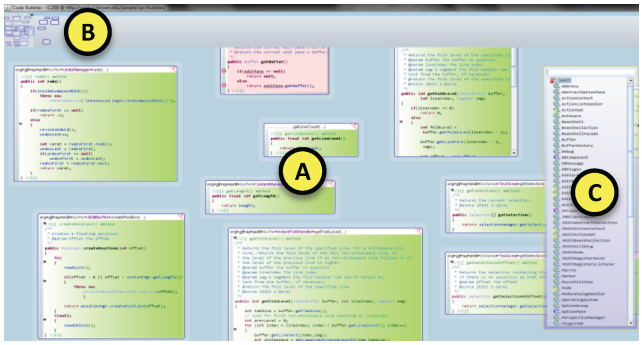

Figure 2. The Code Bubbles editor, including (A) a number of bubbles, (B) a workspace bar, and (C) a package explorer.

fragments and to arrange those fragments on a 2D canvas. This approach mitigates two key problems with file-based editors: First, it enables the programmer to focus only on relevant code fragments, rather than having to negotiate all the fragments in a file. 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 [30], JASPER [4], Code Canvas [9], and Code Bubbles [2, 3]. In the remainder of the paper, we focus on Code Bubbles as a representative example of the canvas-based paradigm.

Fig. 2 depicts the Code Bubbles editor, which enables the programmer 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. 2). The programmer 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. 2B) 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 programmer 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 [17]. Second, if programmers' goals are rapidly changing (as per reactive information foraging [22, 26]), 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 programmer 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 programmer may have to resort to visual searching, which is relatively slower than spatial memory [14]. 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 [2]. But if programmers place that much code on screen, the fragments may begin to all look alike, depriving the programmer of landmarks to spatially orient him/herself. Moreover, the programmer may be more likely to miss a sought method. Second, if the programmer 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 programmer has only two scrolling choices: up or down. However, in a 2D canvas, the programmer can pan in 360 degrees—considerably more room for navigational errors. Third, while navigating in the 2D canvas, the programmer may have difficulty identifying landmarks and orienting him/herself, which has been shown to be important for spatial memory [6], because of the potentially large number of similar-looking fragments.

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 programmer 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*.

The Patch Grid

Fig. 3 depicts our Patchworks prototype. The main part of the editor consists of a 3×2 grid of *patches* (Fig. 3B). 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) [23] 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 programmer can neither adjust the size of a patch nor the number of patches. In contrast to canvas-based editors, such as Code Bubbles, we intentionally restrict the ways in which programmers can arrange their code to discourage wasting effort on creating arrangements that provide little benefit. For example, Plumlee and Ware [29] 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 programmer 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 [16].

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. 3A) 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. 4 depicts the features of an individual patch. A patch consists of a title bar with the name of the fragment (Fig. 4A), a close button (Fig. 4B), and a code-fragment editor (Fig. 4C). The fragment name is based on the type of the fragment, so a method fragment 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.

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. 5 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 programmer 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 programmer 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 programmer 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 [5].

Similar to Code Bubble's workspace bar, Patchworks has a *ribbon view* (Fig. 6) that presents a bird's eye view of the ribbon. The programmer can use the ribbon view to adjust the visible patch grid. In contrast to Code Bubbles, the ribbon view provides information about each patch, and the programmer 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 programmer can use the ribbon view to add, move, swap, and close patches (via the same interactions as in the patch grid).

EVALUATION 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

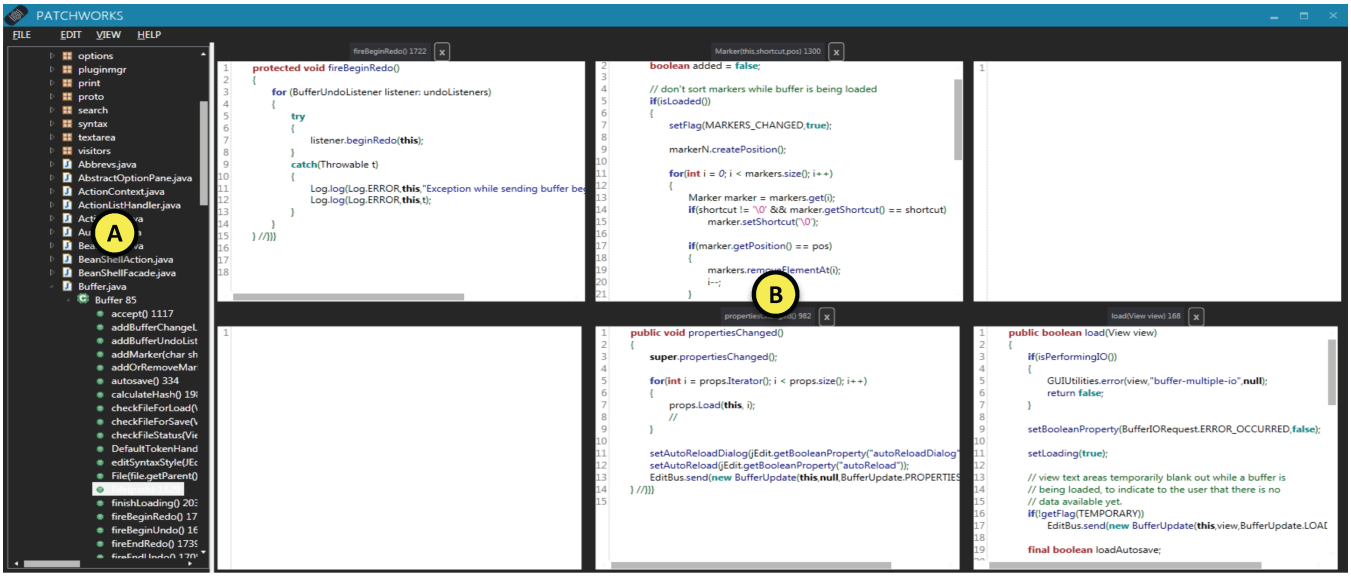

Figure 3. The Patchworks editor, including (A) a package explorer and (B) a 3×2 patch grid. Four of the patches contain code fragments and two are empty.

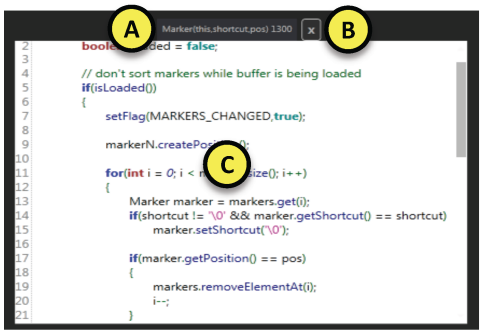

Figure 4. A patch, including (A) a title bar, (B) a close button, and (C) a code-fragment editor.

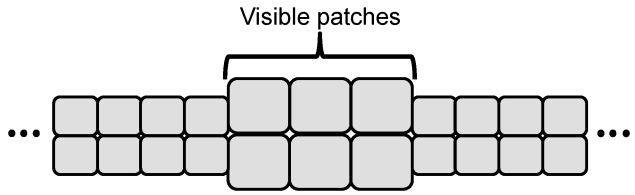

Figure 5. The conceptual ribbon of patches.

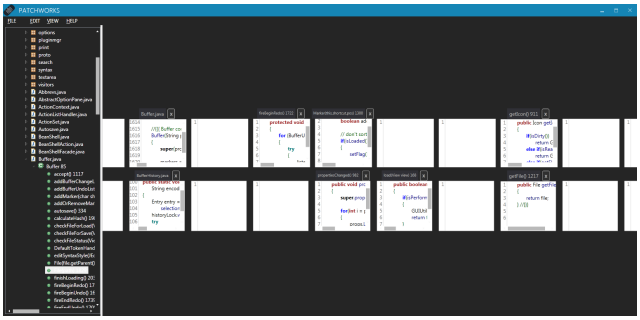

Figure 6. Patchworks' ribbon view, which provides a bird's eye view of the ribbon.

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.

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.

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 [32], an automated software engineering tool for mapping methods to concerns.

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.

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.

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.

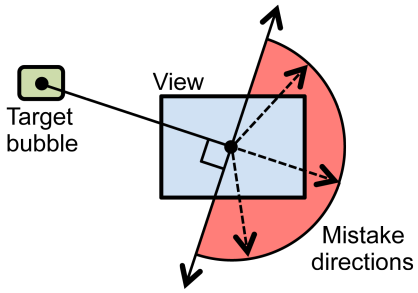

Figure 7. Method for coding navigation mistakes in Code Bubbles when the participant's target bubble is out of view.

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. 7 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.

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.

RESULTS

RQ1 Results: Navigation Time

As Fig. 8 shows, the mean navigation times for participants using Code Bubbles and Patchworks were considerably lower than for those using Eclipse. In fact, participants' 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 ($p < 0.05$).

RQ2 Results: Navigation Mistakes

As Fig. 9 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 programmers failed to see a code fragment that was on screen (see the

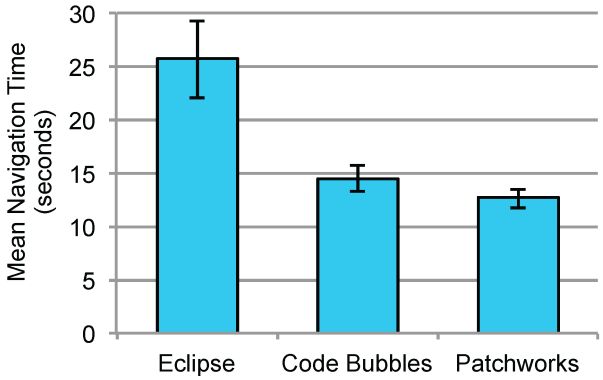

Figure 8. Participants' mean navigation times in Eclipse ($n = 99$), Code Bubbles ($n = 98$), and Patchworks ($n = 99$). Whiskers denote standard error.

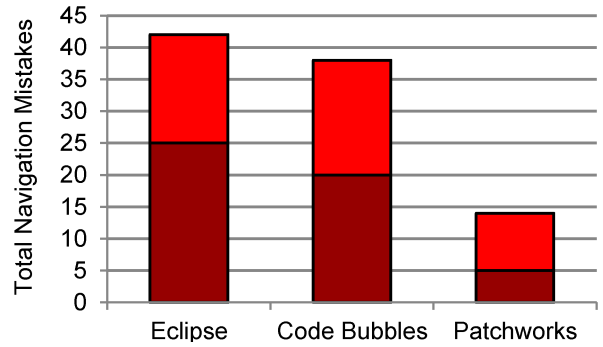

Figure 9. The number of navigation mistakes that participants made while navigating (out of 99 navigation tasks for Eclipse and Patchworks, and 98 for Code Bubbles). Inner bars show the subset of mistakes that were misses.

inner miss-mistake bars in Fig. 9), 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 ($\chi^2(2) = 6.22, p = 0.04$).

RQ3: Time to Open/Arrange Code

As Fig. 10 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 ($p = 0.04$).

Participants' Subjective Opinions

As Table 1 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.

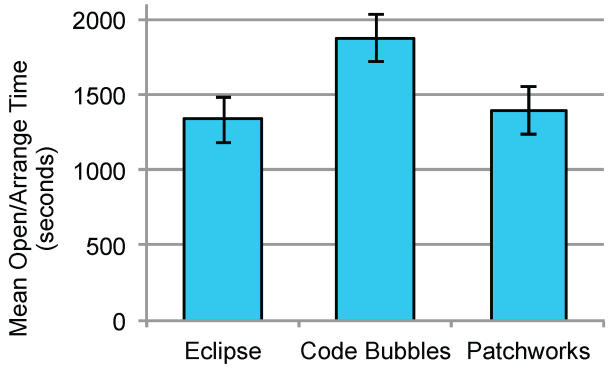

Figure 10. Mean time that participants took to open and arrange their code. Whiskers indicate standard error. $n = 10$ for each tool.

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

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

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.

DISCUSSION

Summary of Results

Table 2 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 users spent less time arranging their code fragments. Moreover, Patchworks surpassed the other editors in helping programmers 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.

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

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

Table 2. 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).

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.

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 counterclockwise. 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.

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.

Limitations

Our study had 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 periods of time affects programmer navigation (e.g., because the programmer 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.

CONCLUSION

In this paper, we have presented the novel Patchworks code-editor, with its two new interface idioms: the patch grid and the ribbon. Our aim was to help programmers navigate to fragments of code more quickly and with fewer navigation mistakes, and to help programmers spend less time arranging code fragments. The results of our user study show promising evidence that Patchworks fulfills these goals:

RQ1 (navigation time): Programmers using Patchworks navigated significantly faster than those using Eclipse (and comparably to those using Code Bubbles).

RQ2 (navigation mistakes): Programmers using Patchworks made significantly fewer mistakes than those using Eclipse and those using Code Bubbles.

RQ3 (arranging code): Programmers using Patchworks spent significantly less time arranging code than those using Code Bubbles (and a comparable amount of time to those using Eclipse).

These findings suggest several promising directions for future research. One open question is how programmers might best arrange fragments in the ribbon. For example, Participant P15 said that his strategy was to lay the fragments out sequentially, in the order he opened them. Scaling this strategy across time and tasks, a programmer could continually build as-needed clusters of patches in one direction along the ribbon, allowing obsolete clusters to accumulate as he/she goes. Having the patch clusters ordered by when they were accessed potentially enables the programmer to leverage a combination of temporal and spatial memory to locate patches. Building on this idea, the ribbon might be augmented with time-sequence information or even manual tagging. Another open question is how to support multiple monitors. The number of patches in the patch grid might simply be expanded to fill the available space, or alternatively, multiple patch grids might be provided, each with an independent view of the ribbon. In conclusion, answering these questions will represent a substantial step toward delivering programmers more fluid and less frustrating code navigation, and helping them regain the time they now lose to the tedious mechanics of navigation.

ACKNOWLEDGMENTS

We thank Dale Bowman Armstrong for her counsel on statistical methods. This material is based upon work supported by the National Science Foundation (NSF) 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 the NSF.

REFERENCES

1. Bragdon, A. Creating simultaneous views of source code in contemporary IDEs using tab panes and MDI child windows: A pilot study. Tech. Rep. CS-09-09, Brown Univ., 2009.
2. Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, Jr., J. J. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.*, ICSE '10 (2010), 455–464.
3. Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, Jr., J. J. Code Bubbles: A working set-based interface for code understanding and maintenance. In *Proc. CHI '10* (2010), 2503–2512.
4. Coblenz, M. J., Ko, A. J., and Myers, B. A. JASPER: An Eclipse plug-in to facilitate software maintenance tasks. In *Proc. 2006 OOPSLA Workshop Eclipse Technology eXchange*, ETX '06, ACM (2006), 65–69.

5. Cockburn, A., and McKenzie, B. Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *Proc. CHI '02* (2002), 203–210.
6. Darken, R. P., and Sibert, J. L. Wayfinding strategies and behaviors in large virtual worlds. In *Proc. CHI '96* (1996), 142–149.
7. DeLine, R., Czerwinski, M., Meyers, B., Venolia, G., Drucker, S., and Robertson, G. Code Thumbnails: Using spatial memory to navigate source code. In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, VL/HCC '06 (2006), 11–18.
8. DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. Towards understanding programs through wear-based filtering. In *Proc. 2005 ACM Symp. Software Visualization*, SOFTVIS '05 (2005), 183–192.
9. DeLine, R., and Rowan, K. Code Canvas: Zooming towards better development environments. In *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.*, ICSE '10 (2010), 207–210.
10. Ehret, B. D. Learning where to look: Location learning in graphical user interfaces. In *Proc. CHI '02* (2002), 211–218.
11. Evans, B., and Card, S. Augmented information assimilation: Social and algorithmic Web aids for the information long tail. In *Proc. CHI '08* (2008), 989–998.
12. Fitchett, S., Cockburn, A., and Gutwin, C. Improving navigation-based file retrieval. In *Proc. CHI '13* (2013), 2329–2338.
13. Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., and Kwan, I. End-user debugging strategies: A sensemaking perspective. *ACM Trans. Comput.-Hum. Interact. 19*, 1 (May 2012), 5:1–5:28.
14. Hick, W. E. On the rate of gain of information. *Quarterly Journal of Experimental Psychology 4*, 1 (1952), 11–26.
15. Hill, E., Pollock, L., and Vijay-Shanker, K. Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. 22nd IEEE/ACM Int. Conf. Automated Software Eng.*, ASE '07 (2007), 14–23.
16. Jones, W. P., and Dumais, S. T. The spatial metaphor for user interfaces: Experimental tests of reference by location versus name. *ACM Trans. Inf. Syst. 4*, 1 (Jan. 1986), 42–63.
17. Kittur, A., Peters, A. M., Diriye, A., Telang, T., and Bove, M. R. Costs and benefits of structured information foraging. In *Proc. CHI '13* (2013), 2989–2998.
18. Ko, A. J., Aung, H., and Myers, B. A. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proc. 27th Int'l Conf. Software Engineering*, ICSE '05 (2005), 126–135.
19. Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. 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.
20. Krämer, J.-P., Kurz, J., Karrer, T., and Borchers, J. Blaze: Supporting two-phased call graph navigation in source code. In *Ext. Abstracts CHI '12* (2012), 2195–2200.
21. Lawrance, J., Bellamy, R., and Burnett, M. Scents in programs: Does information foraging theory apply to program maintenance? In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, VL/HCC '07 (2007), 15–22.
22. Lawrance, J., Burnett, M., Bellamy, R., Bogart, C., and Swart, C. Reactive information foraging for evolving goals. In *Proc. CHI '10* (2010), 25–34.
23. Miller, G. A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev. 63*, 2 (1956), 81–97.
24. O'Hara, K., and Sellen, A. A comparison of reading paper and on-line documents. In *Proc. CHI '97* (1997), 335–342.
25. O'Hara, K., Sellen, A., and Bentley, R. Supporting memory for spatial location while reading from small displays. In *Ext. Abstracts CHI '99* (1999), 220–221.
26. Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R., and Swart, C. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proc. CHI '12* (2012), 1471–1480.
27. Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., Burnett, M., and Bellamy, R. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, VL/HCC '11 (2011), 109–116.
28. Piorkowski, D. J., Fleming, S. D., Kwan, I., Burnett, M. M., Scaffidi, C., Bellamy, R. K., and Jordahl, J. The whats and hows of programmers' foraging diets. In *Proc. CHI '13* (2013), 3063–3072.
29. Plumlee, M. D., and Ware, C. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ACM Trans. Comput.-Hum. Interact. 13*, 2 (June 2006), 179–209.
30. Smith, R. B., Maloney, J., and Ungar, D. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proc. 10th Annu. Conf. Object-Oriented Programming Syst., Languages, and Applicat.*, OOPSLA '95 (1995), 47–60.
31. Storey, M.-A., Cheng, L.-T., Bull, I., and Rigby, P. Shared waypoints and social tagging to support collaboration in software development. In *Proc. Conf. Computer Supported Cooperative Work*, CSCW '06, ACM (2006), 195–198.
32. Warr, F. W., and Robillard, M. P. Suade: Topology-based searches for software investigation. In *Proc. 29th Int'l Conf. Software Eng.*, ICSE '07 (2007), 780–783.