Helping Programmers Navigate Code Faster with Patchworks: A Simulation Study

Austin Z. Henley, Alka Singh, Scott D. Fleming, Maria V. Luong
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
Email: {azhenley,arsingh,Scott.Fleming,mluong}@memphis.edu

Abstract—**Programmers spend considerable time navigating source code, and we recently proposed the Patchworks code editor to help address this problem. A prior preliminary study of Patchworks found that it significantly reduced programmer navigation time and navigation errors. In this paper, we expand on these findings by investigating the effect of various patch-arranging strategies in Patchworks. To evaluate these strategies, we ran a simulation study based on actual programmer navigation data. Our simulator results showed (1) that none of the strategies tested had a significant effect on programmer-navigation time, and (2) that navigating code using Patchworks, regardless of strategy, was significantly faster than using Eclipse.**

I. Introduction

Modern programmers spend a considerable portion of their time navigating from fragment to fragment of source code in their development environments. For example, one study of Java programmers showed that as much as 35% of the programmers' time was spent navigating code [5]. Another study found that 50% of programmers' time was spent foraging for information [9]. Still others have identified issues with code navigation and with "re-finding" code that slowed programmers' in their tasks [2]. Thus, our work seeks to improve the design of programming environments and to significantly reduce the time that programmers spend on code navigation.

In our previous work, we proposed a new editor design, *Patchworks*, and a preliminary user evaluation showed promising results [4]. The evaluation compared Patchworks to two existing editors, Eclipse and Code Bubbles. Participants using Patchworks navigated significantly faster than those using Eclipse, spent significantly less time arranging code than those using Code Bubbles, and made significantly fewer navigation mistakes than those using either Eclipse or Code Bubbles.

However, that work also raised two key questions that we address in this paper. One question pertained to how programmers should arrange their code as they work in Patchworks. Patchworks is based on a "ribbon of patches" idiom, depicted in Fig. 1. The user views a grid of six patches at a time, and can slide left or right along an infinite ribbon of patches. Each patch on the ribbon is a code-fragment editor (e.g., for Java classes or individual methods) that can be moved around the ribbon. This idiom allows considerable flexibility in arranging code fragments, and it is unclear what patch-arranging strategy programmers should employ to achieve the fastest navigations.

The second question was how well the results will generalize to "real world" tasks. In our prior study, participants

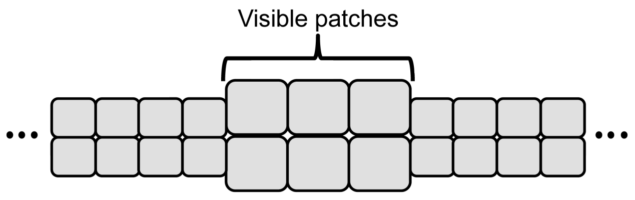

Fig. 1. The Patchworks "ribbon of patches" idiom.

performed artificial navigation tasks in which we instructed them to navigate to particular locations in the code. Thus, the editors were not evaluated with respect to navigations occurring naturally during development tasks.

To address these questions, we conducted a study in which we recorded the navigations of programmers working on their own development tasks, and then simulated users performing those same navigation sequences using various editors and code-arranging strategies. In particular, the study addressed these research questions:

RQ1: Which Patchworks patch-arranging strategy yields the fastest navigations?

RQ2: Do programmers navigate faster using Patchworks than using Eclipse?

II. The Patchworks Code Editor

Fig. 2 depicts our Patchworks code editor. The main part of the editor consists of a 3×2 grid of *patches* (Fig. 2A-2). 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. However, we defer to future work the question of what the optimal number of patches might be.

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. 2A-1) into the patch. 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.

Although the patch grid contains only six visible patches at a time, conceptually, the six are a view into a never-ending ribbon of patches, depicted in Fig. 1. 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

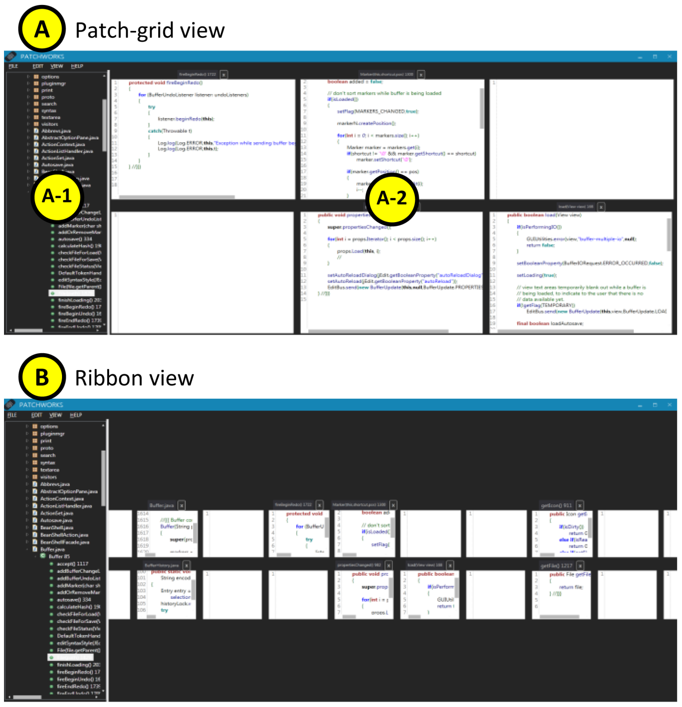

Fig. 2. The Patchworks editor, including (A) the patch-grid view, (B) a patch, and (C) the ribbon view.

ribbon. Also, the ribbon view (Fig. 2B) provides a bird's eye perspective of the ribbon.

III. Candidate Patch-Arranging Strategies

A key question of the current work is *how should programmers arrange their patches as they use Patchworks?* An important design assumption of Patchworks is that programmers will use the ribbon as a timeline, with less recently visited fragments being further back (left) and more recently visited fragments being further forward (right); thus, we frame the problem of patch arranging as deciding which patches to bring forward and when. In this work, *bringing a patch forward* entails using the ribbon view to make a new patch for the fragment adjacent to the rightmost patches on the ribbon (leaving the original patch unchanged to preserve the timeline).

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

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

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

The *Recency strategy* also tries to reduce the cost of revisits; however, unlike the above two strategies, a programmer may have difficulty performing the strategy exactly. In Recency, the programmer brings forward the current patch if it was not already on screen (i.e., it required clicking to get to), and it was among the top six most recently visited fragments. The rationale for this strategy is that studies have found recency to be a strong predictor of where a programmer will navigate [6], [8]. We chose *six* because the patch grid holds six fragments. However, it may be difficult for a programmer to recall exactly the last six methods he/she visited.

Finally, the *DOI strategy* attempts to approximate the programmer's *degree of interest* (DOI) in the current fragment to decide whether the fragment should be brought forward. It is common to estimate a person's DOI based on his/her past behavior (e.g., as in [3]). However, we compute a participant's DOI in a code fragment at a given time using his/her future navigations. Given a fragment f, for each future navigation g_i to f, we compute a weight $W(g_i) = 0.85^{d-1}$ where d is the number of navigations into the future that g_i is from the current navigation. Then, we sum all the g_i weights for f to compute the total DOI for f. In the DOI strategy, the programmer brings forward the current patch if it is among the top six greatest DOI values (and it was not already on screen). The rationale for six here is the same as for the Recency strategy. Since this strategy uses future behavior, it is the one that programmers would be least capable of performing exactly.

IV. Study Method

To address our research questions, we compared the various editors and strategies with respect to the same sequence of navigations. We first conducted a user study to collect code-navigation data (where/when they navigated), and then used that data to simulate how the developers might perform the same navigations using different editors and strategies.

We collected the navigation data from 14 graduate students (11 male, 3 female) working individually on their software projects for a graduate-level software engineering course. The projects involved developing Java EE web applications, made up of Java servlets, JSPs, and "plain old" Java classes. All participants had experience with Java and Eclipse, and on average, they had 7.5 years of programming experience ($SD = 2.9$). Although the projects varied, on average, the code bases consisted of 9344 lines of Java and JSP code spread across 84 code files. We video-recorded each participant working for 2 hours on his/her project. Participants performed "think aloud" as they worked.

Based on each participant's video data, we used qualitative coding to identify where the participant navigated. In particular, we coded each time the participant moved his/her attention from one fragment to another. We coded fragments at the granularity of methods, classes, and non-Java code files (e.g., JSP and XML). To ensure that our coding was reliable, three researchers independently coded the same 20% of the

TABLE I. PARTICIPANT NAVIGATION DATA (VALUES ROUNDED TO NEAREST WHOLE NUMBER). NAVIGATIONS PER FRAGMENT INDICATES HOW MUCH A PARTICIPANT REVISITED FRAGMENTS.

	Navigations	Files opened	Fragments visited	Navigations per fragment
Min:	50	4	15	2
Mean (SD):	157 (74)	16 (6)	27 (8)	6 (3)
Max:	303	24	39	13

data (spread across all participants), and achieved 86% inter-rater agreement (Jaccard similarity) on their codes. Then, they coded the remaining 80% independently.

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

Based on the simulated scenarios, we computed two metrics for comparing strategies and tools: the number of simulated navigations to patches that were already on screen and the Keystroke-Level Model (KLM) cost of each navigation in seconds. KLM [1] is a technique for estimating interaction times by breaking tasks into low-level operations. Although these metrics are not entirely orthogonal, each offers important insights. KLM addresses the time cost of the user interactions (e.g., clicks) that produce each navigation. However, KLM, being a model, ignores many details of the real world. Thus, we also included the simpler on-screen navigations metric. To build confidence that our results are valid, we looked for triangulation among these two metrics.

V. RESULTS

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

Regarding RQ1, as Figs. 3 and 4 show, the simulator results across patch-arranging strategies were fairly similar. Indeed, Kruskal-Wallis tests showed no statistically significant difference among the strategies in the number of navigations simulated programmers made to patches already on screen, and no significant difference in the KLM times yielded by simulated programmers.

Regarding RQ2, Figs. 3 and 4 also show that the simulated Patchworks users had a substantially greater percentage of navigations to patches already on screen and substantially lower KLM times than simulated users of Eclipse. For purposes of statistics, we compared simulated Patchworks users using the Distance patch-arranging strategy to simulated Eclipse users (however, our results were the same no matter which strategy we chose). Wilcoxon signed-rank tests showed that simulated Patchworks users had significantly more navigations to fragments already on screen ($Z = 3.30, p < 0.001$), and had significantly lower KLM times than simulated users of Eclipse ($Z = -3.30, p < 0.001$).

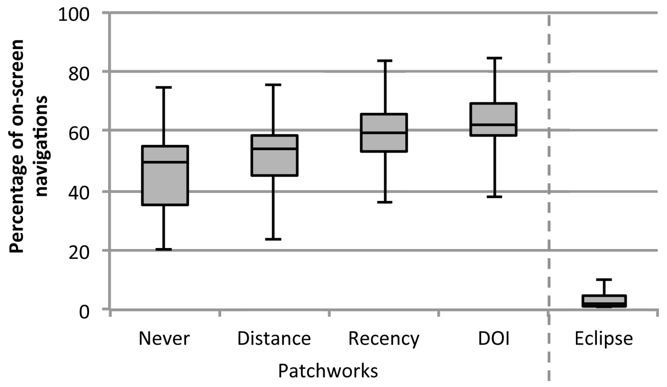

Fig. 3. Aggregate results: Percentage of simulated navigations to fragments already on screen (bigger is better; $n = 14$).

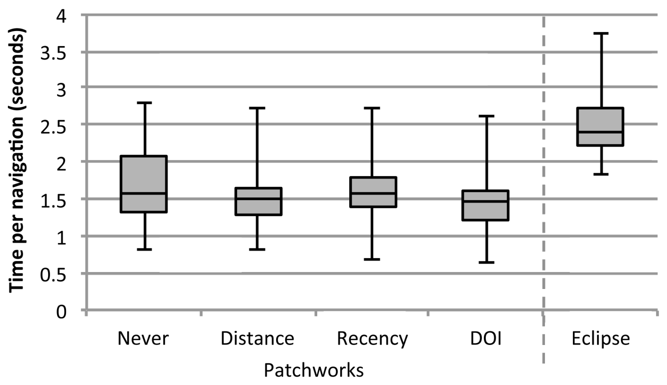

Fig. 4. Aggregate results: KLM times (smaller is better; $n = 14$).

VI. DISCUSSION

A. Patchworks: Robust to Different Strategies

Based on our RQ1 results, it generally mattered little which patch-arranging strategy was chosen. No strategy showed a statistically significant improvement over any of the others, and the magnitude of the differences between the strategies were relatively small. For example, the best-performing strategy (DOI, the future-peeking strategy) improved upon the worst (Never, the no-arranging strategy) by only 16 percentage points for mean on-screen navigations and by only a 10% speedup in mean KLM times. This result suggests that Patchworks users need not be overly concerned about what strategy they use, as long as they treat the ribbon as a timeline.

Although, in general, strategy had little effect, for a few participants' navigation sequences, certain strategies worked far better or worse than the others (see Figs. 5 and 6). For example, participants P14 and P7 make an interesting contrast. For P14's navigations, the Never strategy outperformed the others, whereas for P7's, that strategy performed worse than the others. P14's curious result can be explained because he did not revisit patches very often. In fact, he had the fewest navigations per fragment of any participant. Because P14 did little revisiting, there were fewer opportunities to navigate to patches already on screen, and the cost of moving patches forward in anticipation of revisits did not outweigh the cost of simply shifting the ribbon. P7's result can also be explained because she made many back-and-forth navigations

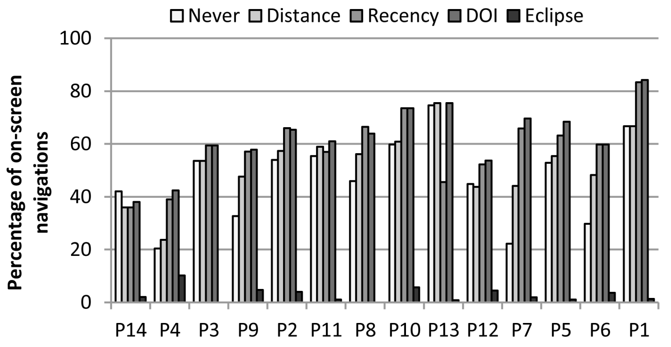

Fig. 5. Per-participant results: Percentage of simulated navigations to fragments already on screen (bigger is better). Participants sorted from fewest navigations per fragment to greatest.

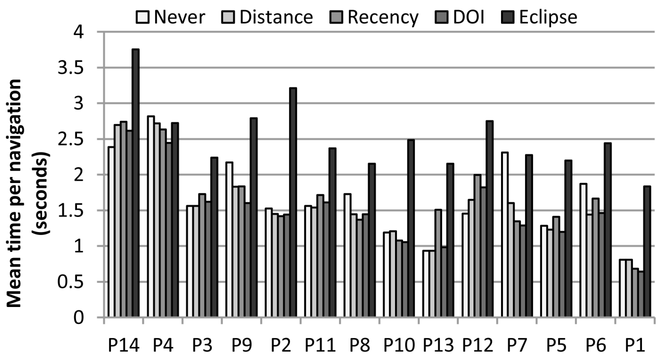

Fig. 6. Per-participant results: KLM times (smaller is better). Participants sorted from fewest navigations per fragment to greatest.

between fragments that were initially placed at the extreme ends of the ribbon. In her case, bringing forward patches made a considerable difference in reducing the cost of these navigations.

As another atypical example, for P13's navigation sequence, the Recency strategy performed noticeably worse than the others. In his case, the Recency strategy made many long, costly shifts along the ribbon to far-away patches, and when the strategy finally brought the far-away patches forward, it turned out that they were no longer needed, leading to more wasted time.

B. Patchworks: Faster than Eclipse Regardless of Strategy

Based on our RQ2 results, Patchworks significantly outperformed Eclipse. Although Patchworks' KLM times were strong, where it really shone was in increasing the number of navigations to fragments already on screen. Given the limited viewing space in the Eclipse editor, a user is lucky if there are three or four fragments visible at a time. In contrast, Patchworks always displays six patches where fragments can go.

Also clear from the per-participant results (Figs. 5 and 6) is that Patchworks does a particularly effective job of supporting revisits. For example, it was no accident that P1's navigation sequence, the one with the greatest navigations per fragment, yielded among the greatest percentage of navigations to on-screen patches (67–84 percent, depending on strategy) and the lowest KLM times. Similarly, the only navigation sequence for which Patchworks (Distance strategy) had worse KLM times than Eclipse (albeit by a small amount) was P4's, who had among the fewest navigations per fragment.

VII. Conclusion

In this paper, we have presented a simulation study of the Patchworks and Eclipse code editors. The study evaluated strategies for arranging code fragments in Patchworks, and compared Patchworks and Eclipse. Our main measures were the number of navigations that simulated users made to code that was already on screen, and the cost of navigation based on KLM. Key findings included the following: (RQ1) There was little difference among the patch-arranging strategies and (RQ2) Patchworks had significantly more on-screen navigations and significantly lower cost per navigation than Eclipse, regardless of patch-arranging strategy.

These results raise several interesting possibilities for future work. One question is how programmers might use Patchworks over days, months, or years. For example, augmenting the ribbon with timestamps may allow programmers to find what they were working on a few hours ago or even a few weeks ago. Another idea is to enable collaborative sharing of ribbons, which may lead to interesting uses, such as maintaining task-relevant working sets between team members over periods of time or helping in onboarding new developers. One final idea is applying information foraging theory (as in [7]) to design a recommender system that predicts the fragments a programmers will visit and automatically assists in arranging those fragments on the ribbon. In conclusion, exploring these ideas will provide further insights into the design of effective tools for the next generation of programming environments.

Acknowledgment

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

[2] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," in *Proc. ACM Symp. Software Visualization* (SOFTVIS '05), 2005, pp. 183–192.

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

[4] A. Z. Henley and S. D. Fleming, "The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *Proc. CHI*, 2014, to appear.

[5] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks," in *Proc. ICSE*, 2005, pp. 126–135.

[6] C. Parnin and C. Gorg, "Building usage contexts during program comprehension," in *Proc. 14th IEEE Int'l Conf. Program Comprehension* (ICPC '06), 2006, pp. 13–22.

[7] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart, "Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers," in *Proc. CHI*, 2012, pp. 1471–1480.

[8] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *Proc. VL/HCC*, 2011, pp. 109–116.

[9] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Proc. CHI*, 2013, pp. 3063–3072.