Improving Source Code Navigation with Patchworks

Austin Z. Henley
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
azhenley@memphis.edu

I. Introduction

Programmers spend a considerable amount of time navigating among many code fragments that may be spread across hundreds or even thousands of files. For example, one study found that programmers spent 35% of their time navigating [5]. Another study showed that 50% of programmers' time was spent foraging for information [7]. My work aims to increase programmer productivity through the design of new code editors and tools to speed up source code navigation.

There are many reasons why programmers spend so much time navigating. One possible reason is that they are seeking to identify a relatively small set of task-relevant code within the entire code base [5]. Moreover, it has been shown that programmers may spend considerable time inspecting irrelevant code [5]. Another reason programmers navigate so much is that they repeatedly visit code within this set of code. For example, one study found that the four most recently visited methods accounted for 69% of interclass navigations [6].

Currently, *file-based editors*, such as Eclipse, are the most popular paradigm, and they may be contributing to the problems associated with code navigation. They present the code by files and allow a programmer to scroll within a file or switch tabs to work on other files. Unfortunately, a programmer's working set is likely to be scattered between and within multiple code files, causing him or her to spend a lot of time scrolling and switching tabs. For example, if a programmer is working on a method at the top of a file and the bottom of a file, he or she may have to navigate back and forth, disregarding any code in between. This gets even more complicated if the programmer must frequently alternate tabs to view other files but still scroll to other fragments. One solution would be to place code side by side, as most editors allow, but programmers don't seem to use this feature [1], [5].

Researchers have sought to overcome the limitations of file-based editors with an alternative paradigm of code editors. These *canvas-based editors*, such as Code Bubbles [1] and Code Canvas [3], allow programmers to work with code fragments, such as methods, and arrange them on a 2D canvas. They enable the programmer to view only the relevant code rather than entire files and provide an easy way to juxtapose many code fragments.

Although these canvas-based editors may address some problems of file-based editors, they may also introduce new ones. A 2D space is large and may cause programmers to spend time trying to orient themselves, or the canvas may become very clutter after working for some time. This may lead to programmers investing effort in managing their code fragments; however, window management activities have been shown to be complex and time consuming [8].

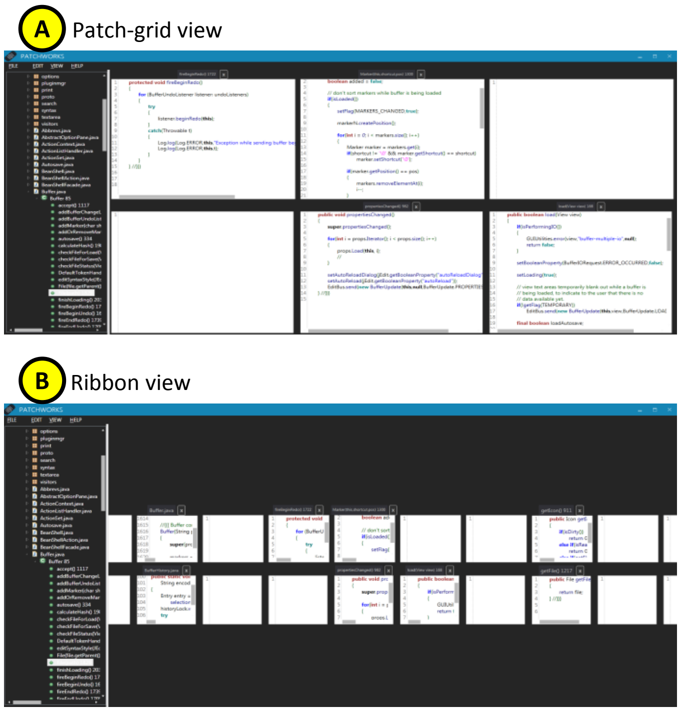

Fig. 1. The Patchworks editor, including (A) the patch-grid view and (B) the ribbon view.

To overcome the limitations of file-based and canvas-based editors, my work proposes a new tool concept known as *Patchworks* [4]. In particular, Patchworks aims to allow programmers to conveniently juxtapose code, to efficiently navigate recently visited code fragments, to significantly reduce scrolling, and to reduce the time spent arranging code.

II. Tool Concept: Patchworks

Fig. 1 shows the Patchworks prototype. The environment has six *patches* visible, which form the *patch-grid*, as shown in Fig. 1A. Each patch is an editor that can contain a code fragment in the form of a method, class, or file. To open a code fragment, the programmer drags the element from the package explorer to the patch. Fragments may be moved between patches by dragging the name above the given patch to the destination patch. If there is an existing fragment in the destination patch, the contents of the patches are swapped.

There exists a virtually infinite number of patches offscreen, to the left and right, on what is known as the *ribbon*.

The visible grid can be shifted left or right using a keyboard shortcut or menu item. These actions are supplemented with animations to convey the act of moving along the ribbon. Also, the programmer can zoom out using the ribbon view, as shown in Fig. 1B.

Two key design decisions were to make the grid of patches fixed and to make the ribbon extend in one dimension. In contrast to file-based editors, the fixed grid makes juxtaposition always enabled, and unlike canvas-based editors, it reduces tinkering with the interface and restricts the ways in which programmers can arrange their code. Programmers need not cleanup or organize the ribbon, they just need to shift the ribbon, thus taking the patches out of sight but saving time on cleanup and keeping them intact in case they are ever needed again. The ribbon is one dimensional so that when seeking a code fragment, it is either to the left or right, unlike canvas-based editors that use a large 2D canvas where a programmer has more opportunity to navigate in the wrong direction.

III. Results of Preliminary Evaluations

I first evaluated Patchworks with a small user study involving 15 participants [4]. The study compared Patchworks with Eclipse, a representative file-based editor, and Code Bubbles, a representative canvas-based editor. The participants opened and arranged a set of related code fragments from a large open source project, and I then asked them to navigate to specific methods, while timing them. The results showed that 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.

As a second evaluation, I performed a simulation study to better understand how programmers should use the ribbon to arrange their code as they work on a development task. I did this by building a simulator, given navigation data as input, that evaluates different strategies of patch-arranging using the keystroke-level model. The simulations showed that there was no significant difference between different patch-arranging strategies. Additionally, Patchworks significantly outperformed Eclipse, regardless of the strategy used. These results suggest that programmers can use Patchworks in a number of different ways while still benefiting from the ribbon.

IV. Future Work

I would like to get feedback from the consortium on what direction my research with Patchworks should go next. In particular, I am considering two paths.

One path is to extend an existing IDE, such as Eclipse, with Patchworks and integrate existing features, such as the debugger. Doing so will enable me to further study and evaluate the design in a realistic environment. Furthermore, since Eclipse is a tool commonly used by professional programmers and provides them access to their usual features and plugins, it will enhance the ecological validity. However, this route has a large initial investment of time to implement the tool (I estimate 9 months) during which I may not be producing publishable results. Moreover, Eclipse is a changing platform which may lead to considerable effort to maintain the plugin. Ideally, I would let programmers download the tool and use it for their own projects for an extended period of time, but getting participants to do so would be difficult. Alternatively, running a lab study with the tool could provide valuable results but will detract from the realism that this feature-complete tool was built for.

The other path I am considering is to build a more novel tool that is more convenient to collect data and explore the design, but is less like a professional programming environment. For example, building a web IDE with basic editing capabilities and a testing environment could be completed within a month using open source frameworks and libraries. This approach makes it easy to distribute and collect data by placing it on the web and collecting data from many users. To attract participants, the IDE could be marketed as a learning environment for undergraduate students that are interested in learning a new programming language (e.g., Ruby on Rails) that is not typically taught in formal classes. Although not as realistic of a study, it should provide a wealth of usage data and feedback without a huge investment to develop the tool.

There are a number of other questions I would like to look into, regardless of which path I pursue to implement the tool. Since the programmers I have observed use the ribbon as a timeline, it could be beneficial to annotate the ribbon with timestamps and provide a way to jump back to specific times. While Patchworks has been designed for navigating recently visited code, it could be interesting to provide features for exploring code. For example, other researchers have investigated the impact of sharing navigation data [2], which could lend itself to the ribbon concept by showing team member's ribbons in the ribbon view. Moreover, integrating a recommender system based on information foraging theory [7] could enable programmers to quickly navigate recently visited code while also finding relevant code that he or she had not yet explored. Investigating these options with Patchworks could provide insight on how software development tools can be enhanced to improve code navigation.

References

[1] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: A working set-based interface for code understanding and maintenance," in *Proc. CHI*, 2010, pp. 2503–2512.

[2] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proc. VL/HCC*, 2005, pp. 241–248.

[3] R. DeLine and K. Rowan, "Code Canvas: Zooming towards better development environments," in *Proc. ICSE*, 2010, pp. 207–210.

[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. 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.

[8] M. D. Plumlee and C. Ware, "Zooming versus multiple window interfaces: Cognitive costs of visual comparisons," *ACM Trans. Comput.-Hum. Interact.*, vol. 13, no. 2, pp. 179–209, 2006.