Designing Affordances for Navigating Information Spaces in Code Editors

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

I. Introduction

Navigating information spaces is a fundamental yet challenging task for software developers. For example, one study found that programmers spend 35% of their time on the mechanics of navigating [7]. In another study, programmers spent 38–71% of their time foraging for information during debugging tasks [9]. This is further complicated by programmers' rapidly changing information goals [9] and mental models of the information as they navigate [6].

A potential reason for this difficulty is because there are so many information spaces in code editors. Not only do programmers want to navigate code, but they may also want to navigate historical code, analysis tool output, previous program outputs, code review comments, bug reports, etc. Although these information spaces are often supported in code editors, they usually have distinct and disjoint affordances for navigating them.

The hypothesis of my thesis is that effective development tools should support multiple spaces. For example, a programmer may want to view a previous version of the code to use as a reference as they are modifying the current version. Currently, code editors treat these two information spaces differently by providing inconsistent affordances, thus requiring even more effort for a programmer to seek information. Another example is viewing analysis tool output while performing code reviews. To do this currently, a programmer would have to leave the code editor and find a log of the analysis output or even worse, check out the version of code being reviewed from version control and rerun the analyzers. By supporting multiple information spaces, tools could drastically reduce time spent navigating, reduce cognitive load, and even reduce the number of bugs introduced.

My goal is to develop a unified set of design principles for affordances to support navigating multiple information spaces in code editors efficiently. To date, I have investigated two information spaces: recently visited source code and recently modified source code. My next step is to explore information spaces relevant to code reviews. Researching these various information spaces should allow me to extrapolate generalized design principles that cover many information spaces.

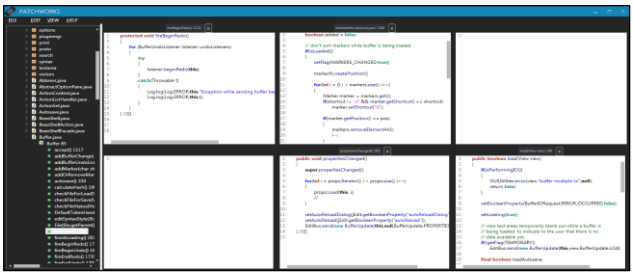

Fig. 1. The Patchworks code editor for Java. The editor consists of a fixed grid of patches that can contain code fragments of various granularities (file, class, method). Off-screen patches can be navigated to by shifting the patch strip either to the left or right.

II. Affordances for Navigating Code

As a first step, I investigated tools for code navigation. This is the most commonly supported information space in code editors. Popular code editors, such as Eclipse and Visual Studio, already provide many navigation features such as *Open Declaration*, *Find References*, and text search. Additionally, researchers have proposed extensions that further support code navigation (e.g., Code Bubbles [1] and Stacksplorer [5]) while others have studied the information needs of programmers while using existing navigation features (e.g., [7], [8], [9]).

We developed an experimental code editor design, Patchworks [3], for navigating recently visited code patches. Fig. 1 shows the initial implementation for Java code. A key feature of the design is to divide the screen into patches of fixed size and position, which can contain code fragments. This has the goal of reducing time spent managing tabs. When these patches are full, you can shift to the left or right, revealing additional patches on a virtually never-ending patch strip.

After conducting several studies, we found that using Patchworks had a number of benefits. Our initial study compared participants using Patchworks, Code Bubbles, and Eclipse to perform sequences of navigations [3]. Participants using Patchworks navigated faster, spent less time arranging their open code fragments, and made fewer navigation mistakes than other participants. Our second evaluation found that regardless of arranging strategy, Patchworks resulted in more efficient navigations [2]. For a third evaluation, we asked professional programmers to use Patchworks and a tab-based editor during debugging tasks and again found that they had significantly more efficient navigations.

978-1-5090-0252-8/16/$31.00

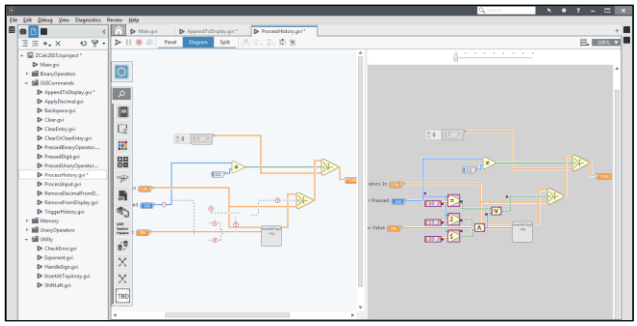

Fig. 2. The Yestercode extension to the LabVIEW IDE. Yestercode provides an additional view that allows the user to navigate the current VI's version history and that displays the code of the selected older version with visual cues that annotate the differences with the current version.

Based on the findings of these studies, we proposed 5 design principles to further improve code editor designs. These principles are: facilitating efficient patch juxtaposition, enabling efficient toggling between single-patch and multi-patch displays, providing informative thumbnails/labels of open patches, enabling efficient closing of open patches, and applying these principles to all types of open patches (not just code). By comparing and contrasting a variety of editor designs, we were able to discover improvements to both traditional tab-based editors (e.g., Visual Studio) as well as more experimental editors (e.g., Patchworks).

III. Affordances for Navigating Historical Code

Another information space that has recently been recognized as important is historical versions of code. For example, Yoon and Myers found that programmers often want to revert portions of their code to a previous version [11]. Researchers have also been interested in how end user programmers navigate among variations of programs (e.g., [10]).

As part of my exploration, we designed a tool, Yestercode [4], that allows a programmer to navigate through recent versions of code and compare it to the most recent version. As shown in Fig. 2, the editor is split into two documents, the current version of the code and a previous version. A slider allows the programmer to quickly navigate through the history of the code. This feature is designed so that programmers can view previous versions without any upfront effort (e.g., committing changes to a version control system). Additionally, it is built into the editor itself, so that the programmer can continue editing the current version while also interacting with the previous version.

In an evaluation of professional programmers using Yestercode, we found several benefits. First, participants using Yestercode introduced significantly fewer bugs than the control participants. Second, Yestercode did not impact the time taken to complete tasks. Third, Yestercode participants reported significantly lower cognitive load while performing the tasks.

Based on these results, we developed 5 design principles that Yestercode followed. The tool should transparently record the version history without any explicit actions by the programmer to do so. It should also allow for juxtaposition of two versions while enabling the programmer to efficiently navigate the version history. Furthermore, the tool should annotate the version differences with visual cues while also being tightly integrated into the code editor, allowing actions such as copy-and-pasting from a previous version.

IV. Proposed Work

For the next piece of my dissertation, I will investigate the combination of information spaces relevant to code reviewing. In particular, I will explore how adding additional information, such as analysis tool output, improves code reviewing.

To carry this work forward, I will be interning at Microsoft Research. This will enable me to study the information needs of professional programmers during code reviews. I'll then extend a code reviewing tool to better support their needs and evaluate the design with a user study.

As a final effort of my work, I plan to take these sets of design principles and form a generalized set that can be applied to other information spaces. In the current state of the field, tool developers have to rely on their intuition or just mimic other systems, which may lead to unsuitable design decisions. These principles will assist the designers of information navigation tools by providing a coherent framework of heuristics based on empirical evidence.

References

[1] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proc. ICSE*, 2010, pp. 455–464.

[2] A. Z. Henley, A. Singh, S. D. Fleming, and M. V. Luong, "Helping programmers navigate code faster with patchworks: A simulation study," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, July 2014, pp. 77–80.

[3] A. Z. Henley and S. D. Fleming, "The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, 2014, pp. 2511–2520.

[4] A. Z. Henley and S. D. Fleming, "Yestercode: Improving code-change support in visual dataflow programming environments," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2016, to appear.

[5] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: call graph navigation helps increasing code maintenance efficiency," in *Proc. UIST*, 2011, pp. 217–224.

[6] A. Kittur, A. M. Peters, A. Diriye, T. Telang, and M. R. Bove, "Costs and benefits of structured information foraging," in *Proc. CHI*, 2013, pp. 2989–2998.

[7] 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.

[8] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett, "Foraging and navigations, fundamentally: Developers' predictions of value and cost," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. New York, NY, USA: ACM, 2016.

[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 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, 2013, pp. 3063–3072.

[10] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, "Foraging among an overabundance of similar variants," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, 2016, pp. 3509–3521.

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