CodeRibbon: More Efficient Workspace Management and Navigation for Mainstream Development Environments

Benjamin P. Klein and Austin Z. Henley
University of Tennessee
Knoxville, Tennessee
bklein3@vols.utk.edu, azh@utk.edu

***Abstract*—Developers spend considerable time navigating and managing open code documents in their development environment. Researchers have proposed novel interfaces to address the problems of workspace management, such as the Patchworks and Code Bubbles code editors, which replace the traditional tabbed document interface. However, these interfaces are not available in mainstream development environments despite their promising laboratory results. In this paper, we demonstrate CodeRibbon, a user interface for more efficient workspace management and navigation that is publicly available as a code editor plugin. CodeRibbon provides a virtually endless ribbon of code documents that are arranged in an adjustable grid for efficient juxtaposition and navigation with minimal document management. Our implementation is open source and currently in development as plugins for Atom and VS Code. Since prior research on these interfaces has been limited to laboratory studies, we aim to collect usage data from a longitudinal field study involving professional developers engaged in real-world tasks. This work should provide a better understanding of how tools can support developers in efficiently managing their development environments.**

Demonstration video: https://youtu.be/m5wQ87ItVGg

***Index Terms*—code editor, development environment, workspace management, code navigation**

I. Introduction

Developers spend considerable time and effort navigating and managing open code documents in their development environment. One notable study found that developers spend 35% of their time on the mechanics of navigating [11], such as scrolling or managing tabs. Despite developers spending time on these navigation activities, they often can not find the information they are looking for [13], [14]. The design of development environments needs to better support developers in managing and navigating their workspace.

Researchers have proposed novel user interfaces to address the problems of workspace management that replace the traditional tabbed document interface. Code Bubbles [2], [3] provides a large two-dimensional canvas for opening and arranging code documents and Patchworks [8], [9] provides a ribbon that extends off the screen to the left and right with a fixed grid of code documents. However, these interfaces have not been adopted by mainstream development environments although they both had promising laboratory results.

In this paper, we demonstrate CodeRibbon, a user interface for more efficient workspace management and navigation that is publicly available as a code editor plugin. CodeRibbon provides a virtually endless "ribbon" of code documents that are arranged in an adjustable grid for efficient juxtaposition and navigation with minimal document management. Currently, our implementation is open source and in development as plugins[1] for Atom and VS Code, two popular code editors.

To better understand how workspace management can be improved for developers, we plan to perform a longitudinal field study involving professional developers engaged in real-world tasks. By integrating into existing development environments, developers can continue to use their other tooling and adopt CodeRibbon with less friction. This approach is to overcome the limitations of prior studies on related tools that were in lab settings with artificial tasks and different development tools than typically used by the participants.

II. Background & Related Work

Tabbed document interfaces afford users to open multiple documents, switch between them using tabs, scroll within a single document, and view one or more documents at a given time by tiling the finite workspace. Popular code editors that employ this interface include Visual Studio, Eclipse, IntelliJ products, XCode, VS Code, and Atom. Despite tabbed documents being the most common paradigm found in code editors, numerous studies have identified their limitations. For example, a study found that developers spent 35% of their time on the mechanics of navigating code documents, such as scrolling and tab switching. Furthermore, developers often can not find the tab they are looking for [9], [12], [16] or accidentally close it [11]. Similarly, researchers have identified tabs to be problematic in web browsers as well [4], [7]. Much of these problems could be solved if tabbed documents could be arranged side-by-side more efficiently, which developers want to do [1], [9], but they report that it is often too tedious [3], [9], [11].

[1]https://utk-se.github.io/CodeRibbon/

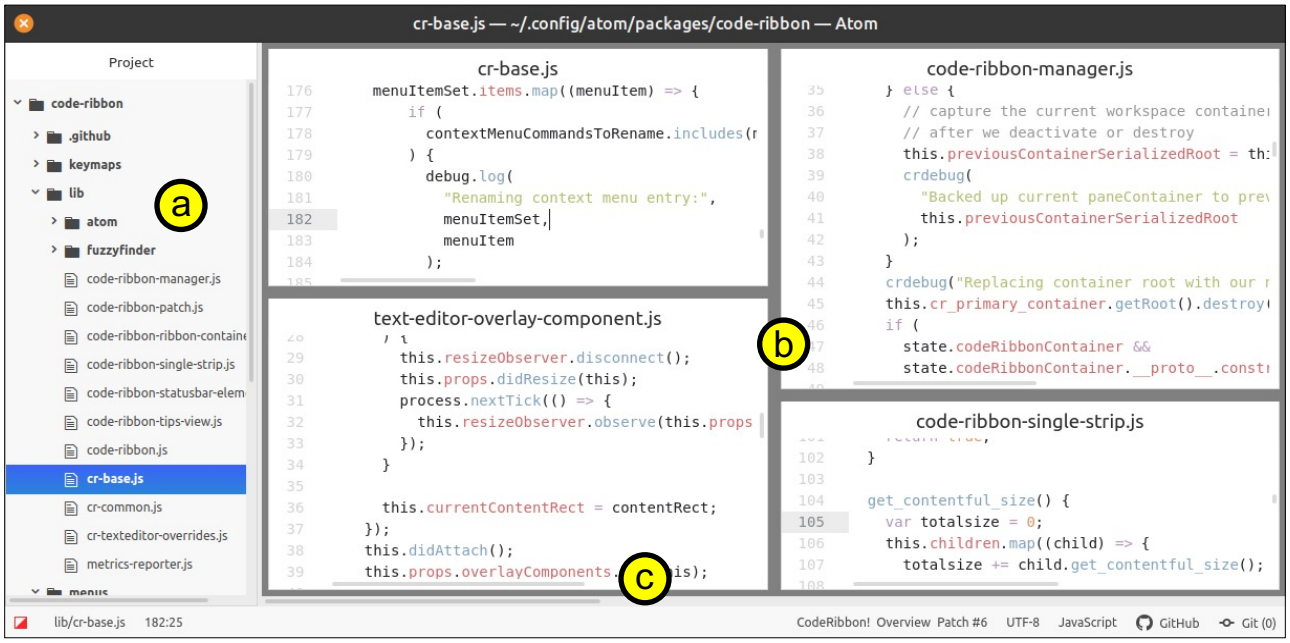

Fig. 1. The CodeRibbon interface as an Atom plugin composed of (a) Atom's standard file listing, (b) CodeRibbon's grid of code documents showing a resized 2x2 configuration, and (c) there are many more documents to the right that are off screen, that can be navigated to using keyboard shortcuts or a horizontal scrollbar. Navigating to these off-screen documents works like an animated carousel.

Researchers proposed Code Bubbles [2], [3] that provides a interface involving *bubbles* of code fragments that can be arranged on a large two-dimensional canvas. Laboratory studies found that developers using Code Bubbles navigate faster than when using traditional code editors [2], [3], [8]. However, the open two-dimensional space may provide too much freedom, with one study finding that developers spent considerable time tinkering with the arrangements of their open code documents. Although the paradigm has been applied by other researchers (i.e., Code Canvas [6] and Debugger Canvas [5]), this paradigm has likely not been made available in existing mainstream development environments due to the complexity in implementing it.

In an effort to overcome the limitations of Code Bubbles while leveraging the benefits, researchers proposed Patchworks [8], [10]. It provides a one-dimensional *ribbon* interface that extends past the left and right edges of the screen, with code documents arranged on a grid. To reduce window management activities, the grid cannot be adjusted or resized. In a controlled study comparing Patchworks, Code Bubbles, and Eclipse, participants navigated faster, spent less time arranging the code documents, and made fewer navigation mistakes [8]. Despite these benefits, Patchworks is not available in mainstream development environments and has not been validated using realistic programming tasks.

III. Tool: CodeRibbon

We designed CodeRibbon (see Fig. 1) to address the workspace management and navigation issues that developers continue to face. It uses the *ribbon* interface introduced by Patchworks [8] that enables efficient navigation to groups of open code documents. In contrast to Patchworks, we added a configurable grid based on the freedom of Code Bubbles [2] that allows the user to change the number of visible code documents (e.g., 2x2 or 2x1; compare Figs. 1 and 3). Additionally, we incorporated more recent design principles about code editors regarding document management [9], [15]. Both plugins are implemented in JavaScript with the Atom plugin available[2] and ready for everyday use, while the VS Code plugin requires more development to circumvent the restrictive nature of VS Code's plugin API. The features are described in the remainder of this section.

A. Navigating the Ribbon

Navigation within the visible section of the ribbon initially works similarly to that of editors that tile their workspaces and defaults to a grid of empty code documents. Once more code documents are needed than the visible grid can hold, the user can shift the ribbon to the left or right, revealing more empty documents. Users can click to focus any code document or use keyboard shortcuts to shift focus in any cardinal direction. Controls that would normally change the active tab now move focus along code documents within the ribbon, e.g. Ctrl-Tab moves the focus to the next code document in the column and then to the next column.

When a document is opened from a tree view in conventional editors it creates a tab, however in CodeRibbon newly opened code is placed into an empty code document within the ribbon. These documents are automatically added to the end of the ribbon (on the right) such that a user will never run out of available space for documents. While CodeRibbon has multiple strategies available for determining where to open new documents, the user can also directly drag files onto

[2]https://utk-se.github.io/CodeRibbon/

Fig. 2. CodeRibbon's overview mode that shows a zoomed out view of the ribbon, and enables users to easily close, rearrange, and navigate.

Fig. 3. CodeRibbon illustrating a 2x1 grid configuration, in contrast to the 2x2 grid in Fig. 1.

documents to open them or use a fuzzy search to open files from their project within an empty document.

Once the ribbon is populated with opened code documents the user is able to move those documents by dragging one over another, swapping them in place. Conventional tiling methods such as "splitting" a single view into two are instead replaced with functions that create additional documents within a column, or create additional columns within the ribbon. This enables users to efficiently place code documents side-by-side horizontally or vertically.

B. Dynamic Ribbon Layout

To address the problems caused by the existing flexibility of window management in modern operating systems the exact number of columns displayed from the ribbon is dynamically calculated. In the case where a developer splits their monitor vertically between the editor and another window, the editor is often not wide enough to display the same number of columns as when the editor takes up the whole workspace. In a less common environment developers may use a workspace that is extremely wide, for example on 32:9 widescreen monitors, and in these cases additional columns are displayed in order to fit as much code on screen as possible without losing readability.

In order to determine how many columns should optimally be shown at various editor window widths, we used the average code width and line length. It's been shown that a large variety of languages obey an approximate line length guide [15] of 80 characters. CodeRibbon's default behaviour will continue to show columns so long as each column maintains visibility of the code up to the user's preferred line length setting. This dynamic layout approach is key to allowing users the flexibility to organize their editor window alongside other information sources, whereas tabbed document interfaces are usually not able to change a user's existing layout automatically when the window is resized.

C. Overview Mode

Shown in Fig. 2, CodeRibbon's overview mode provides a bird's eye-like visualization of your open code documents that enables developers to quickly scan what documents are open. In practice this mode is implemented by zooming out from the ribbon and increasing the space between each document to provide for better visual separation between individual documents. This mode aims to improve the time it takes for users to identify documents within the ribbon, along with allowing users to move along the ribbon in a coarser and less detail-oriented view. Once in overview mode the same amount of scrolling moves past more Patches and the user can more easily move the focus of the editor with standard directional keybindings.

D. Ribbon Movement

Since studies have found developers spend a lot of time navigating their code, we focused on how users move about their ribbon. Most operations that shift focus along the ribbon are designed in such a way to minimize disruption to spatial memory and to make it less visually taxing to follow movements. In order to minimize the chance of losing a mental position along the ribbon and reduce saccadic eye movements, smooth scrolling and animated transitions are used. In the most extreme location changes such as opening a file which is placed at the tail end of the ribbon, a smooth scroll is used to ensure the user is able to get a feel for the distance which was traveled along the ribbon to get to the tail end.

Transitioning into overview is also animated fluidly for this same reason, if we were to transition instantly we would risk visual jerk forcing users to mentally find their position again. Even though the documents still display the same content in overview mode, the text itself is often too small to read, making the transition even more important since users have less information with which to reorient themselves. Additionally, documents are given extra padding in overview mode to make the ribbon's structure and layout more visible, and the active document is highlighted.

E. Increases in Flexibility

In Patchworks, the user was not able to configure the layout of the editor, while CodeRibbon continues to restrict the fundamental layout of columns along the ribbon, users are granted flexibility within each column to account for more varied lengths of code. Most notably the layout is not confined to the initial grid, but instead is confined to a series of columns which contain documents stacked vertically. Each of these columns have the same flexibility as many split-view editors, allowing a longer file to take up the entire height of the workspace, while shorter length files can share the column

with other small files instead of taking up the entire height of an originally-sized document.

In order to lower the barrier to entry for existing users of editors that allow *splitting* a document pane into two, CodeRibbon maintains similar functionality to that of the operations that split existing views. In many scenarios a user desires to open a single file next to itself in order to view two distinct portions of that file, in a tabbed document editor with tiling that normally means splitting the single open file right or left, which creates a vertically divided workspace of two views. In CodeRibbon, instead of splitting an existing view in two halves vertically an additional column is inserted along the ribbon at that location to give a similar experience.

IV. Preliminary Results

During initial development we had over 800 downloads and a few dozen users provided feedback on their experience. The majority of this feedback has been taken into consideration and changes to CodeRibbon have already been made, with some features still under development. In addition to the feedback of other users, the development of CodeRibbon took place almost entirely while using CodeRibbon itself by one of the authors, which was a primary method in which bugs were found.

Examples of what features were requested include the ability to split code documents, column creation and removal, and the zoom functionality. Additional features such as the ability to drag and manage the columns of the ribbon in the overview mode are planned to be implemented from user suggestions. Furthermore, we had planned to only release the plugin as an Atom plugin, but the project has received dozens of requests for it to be ported to VS Code, which we are actively working on. Another user liked the CodeRibbon design so much, that they have began implementing their own version for a different editor. Surprisingly, one user reported that they switched from their current development environment to Atom with CodeRibbon in order to use it.

After a long period of development on CodeRibbon for Atom we began to notice patterns in how users made use of their ribbon and organized documents. While we do not yet have quantitative data to separate these usage paradigms, we saw a clear divergence away from our primary hypothesized usage paradigm: the ribbon as a timeline. Using the ribbon as a timeline was the hypothesized pattern for Patchworks [8], where the ribbon grows from the tail end naturally as the user opens additional files to edit or view. In this usage pattern the user would move backward along the ribbon in order to retrace their code or mental path to their current working set which lives near the tail end of the ribbon. Additionally, this pattern enables users to open all new documents to the right without ever closing unneeded documents.

In other usage patterns users would consistently maintain a smaller total ribbon size, and instead of growing quickly rightwards they instead will rearrange their current working set to match the approximate layout of the code they were working with. Often this layout would resemble a top-down exploratory approach to understanding a larger software, where the beginning of the ribbon is more abstract than code opened rightwards, which is more concrete in most cases. This paradigm closely resembles how canvas-based editors (e.g., Code Bubbles [2]) were used to arrange code segments by relation. With the addition of the ability to create and close columns anywhere within the ribbon these users were more likely to keep their working set in the middle of the ribbon. This appears to be a result of those users opening new documents on both the left and right of their current working set.

V. Proposed Field Study

In contrast to the Patchworks and Code Bubbles studies, which were limited to controlled environments or tasks, our goal is to gather data from users in their natural working environments over several months. Once the study commences we will gather anonymous usage metrics from CodeRibbon users that have accepted an opt-in prompt to contribute to this research. This collection will happen in the background with little to no effect on the user. We will release the logging plugin separately, such that other researchers can use it and to collect baseline data about how developers manage their tabs in Atom and VS Code without CodeRibbon.

Primary metrics to be collected include: open and close document events, document rearrangement or resizing events, broad editing events, and ribbon movements. For privacy, each code document opened will be given a unique identifier for the session and no individual keystrokes or identifiers will be recorded. Analyzing these results can be used to synthesize new design guidelines for code editors and to evaluate the effectiveness of CodeRibbon in an ecologically valid setting.

VI. Conclusion

In this paper, we have presented the CodeRibbon interface for more efficient workspace management and navigation in mainstream development environments. The design builds on the prior novel interfaces, Patchworks and Code Bubbles, and is aimed to overcome the limitations of tabbed document interfaces with a ribbon interface with a configurable grid of code documents. The grid provides efficient juxtaposition that can be shifted left or right, revealing more code documents. Our design is implemented as Atom and VS Code plugins such that CodeRibbon can be integrated with developers existing workflows and development environments.

Given the positive feedback of our initial plugin release, we aim to gather insights from the software engineering research community. In the future, we will perform a longitudinal field study to understand how developers use interfaces like CodeRibbon for realistic development tasks over long periods of time. Doing so will shed light on how mainstream development environments can better support developers in managing and navigating their workspaces.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grant Nos. 1850027 and 2008408.

References

[1] A. Bragdon, "Creating simultaneous views of source code in contemporary IDEs using tab panes and MDI child windows: A pilot study," Brown Univ., Tech. Rep. CS-09-09, 2009.

[2] 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 *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 455–464.

[3] 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 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512.

[4] J. C. Chang, N. Hahn, Y. Kim, J. Coupland, B. Breneisen, H. S. Kim, J. Hwong, and A. Kittur, "When the tab comes due:challenges in the cost structure of browser tab usage," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445585

[5] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: industrial experience with the code bubbles paradigm," in *Proc. ICSE*, 2012, pp. 1064–1073.

[6] R. DeLine and K. Rowan, "Code Canvas: Zooming towards better development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 207–210.

[7] N. Hahn, J. C. Chang, and A. Kittur, "Bento browser: Complex mobile search without tabs," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3173825

[8] 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*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2511–2520.

[9] A. Z. Henley, S. D. Fleming, and M. V. Luong, "Toward principles for the design of navigation affordances in code editors: An empirical investigation," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: ACM, 2017, pp. 5690–5702. [Online]. Available: http://doi.acm.org/10.1145/3025453.3025645

[10] A. Z. Henley, A. Singh, S. D. Fleming, and M. V. Luong, "Helping programmers navigate code faster with Patchworks: A simulation study," in *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '14, July 2014, pp. 77–80.

[11] 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 *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 126–135.

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

[13] 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*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 97–108.

[14] 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*, ser. CHI '13. ACM, 2013, pp. 3063–3072.

[15] A. C. Short and A. Z. Henley, "Towards an empirically-based ide: An analysis of code size and screen space," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2019, pp. 199–203.

[16] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting navigation in software maintenance," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 325–334.