Towards an Empirically-Based IDE: An Analysis of Code Size and Screen Space

Adam C. Short, Austin Z. Henley
University of Tennessee
Knoxville, Tennessee
{ashort11, azh}@utk.edu

***Abstract*—Integrated development environments (IDEs) are ubiquitous in software development. Despite their popularity, much of their designs are not based on empirical findings and have been virtually unchanged since their inception. More recently, researchers have proposed alternative IDE designs based on observational studies of developers and found promising benefits. However, many design decisions are still unsupported by empirical evidence. Towards designing an empirically-driven IDE, we performed an analysis on code size and screen space. First, we analyzed the size of code from 95 projects in 4 programming languages. Second, we calculated how much code can fit onscreen using various document arrangements in VS Code with common screen resolutions. We found (1) the length of code depends on the programming language, while the width of code is similar across languages and (2) the amount of code onscreen can be substantially increased by properly configuring the IDE and by using a high-resolution monitor.**

I. Introduction

Modern integrated development environments (IDEs) are ubiquitous in software development. These IDEs provide affordances for editing, debugging, and testing code in a single, configurable editor. For example, one such IDE, VS Code [10], has over 2 million monthly active users [11] while JetBrains has over 6 million developers using their family of IDEs [4]. Other popular IDEs include Eclipse [2], Visual Studio [9], Atom [1], IntelliJ IDEA [3], PyCharm [5], and Xcode [12].

However, much of the design of IDEs has been the same since their inception decades ago, and may not be based on empirical evidence [38]. For example, many IDEs still display entire code files in tabbed documents that the user can use to switch between open code documents. However, many studies have found tabs in IDEs to be problematic [15], [19], [20], [24], [29], [31], [34], [35], [37].

In an attempt to overcome the shortcomings of IDEs, researchers have proposed alternative IDE designs based on empirical studies of developers. These designs include JASPER [18], Code Bubbles [16], [17], Code Canvas [21], and Patchworks [24], [25], which each provide an alternative to the traditional tab-document metaphor used by other IDEs. These tools were based directly on findings that developers often navigate to irrelevant code or get lost in the code [19], [29], [30], [32].

Further studies in this area could have a huge impact on the design of IDEs and other developer tools. For example, tools that enable you to compare versions of code (e.g., Azurite [39] and Yestercode [23]), are dependent on displaying multiple code fragments in an effective way. Moreover, code reviewing tools (e.g., Gerrit [13], CFar [26], and Phabricator [14]) display a considerable amount of information to the developer. However, there has been little research in understanding the ideal amount of information to display without overwhelming the developer or even how much can fit on a developer's screen.

In this work, we analyze code size and screen resolution to better understand how IDEs should display and layout code. To do so, we mined 95 GitHub repositories across 4 different programming languages to calculate the length and width of code functions. Using these results, we simulated how much code can fit on screen with typical screen resolutions using a popular IDE, Visual Studio Code (VS Code). Based on the findings of our analysis we also discuss implications for the design of future IDEs.

In this work, we aim to answer the following research questions:

- RQ1: What is the typical size of code functions (length and width)?
- RQ2: How much code can fit on screen with a typical IDE and common screen resolution?

II. Related Work & Background

A. Problems with Modern IDEs

While using IDEs, developers spend considerable time and effort navigating code. For example, one study of Java developers using Eclipse found they spent approximately 35% of their time on the mechanics of navigating code [29]. Another study found that developers spent roughly 50% of their time foraging for information using Eclipse [36]. Furthermore, a recent study found that 50% of navigations yielded less information than developers expected and 40% of navigations required more effort than predicted [35].

One reason for all of this time spent navigating is that IDEs display entire code files even when a developer may only be interested in a small subset of code within that file. Because of this, developers have been observed scrolling through irrelevant code to navigate back and forth between code that is of interest [24], [29]. To circumvent the need for scrolling, developers have expressed their desire to place code

978-1-7281-0810-0/19/$31.00

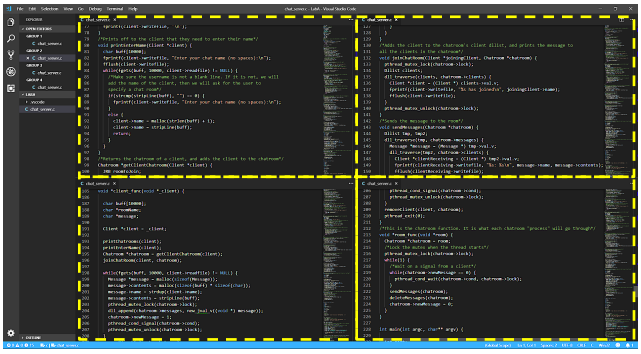

Fig. 1. The Visual Studio Code IDE with 4 code documents juxtaposed in a 2x2 grid (yellow lines for emphasis).

documents side by side [15], [24], [25]. However, it requires tedious window management-like actions to do so and in one study a participant compared it to a "jigsaw puzzle" [15]. In fact, four studies of developer behavior found that 40-90% of navigations are to recently visited code [22], [25], [33], [34], which could make side by side arrangements of code particularly beneficial.

Another potential issue with IDEs are the document tabs, which provide a means of switching between code files. Numerous studies have observed developers "losing" tabs during development tasks where the developer is unable to locate a specific open code document [19], [24], [25], [29], [31], [37]. In some situations, the developers just closed all of the open documents, resulting in even more time taken to get back to relevant code. Tabs may be easy to lose due to their name not being representative of their contents (i.e., the filename may not be descriptive of the contained code) or because they are spatially unstable (i.e., the tab's location on screen changes when another tab is opened or closed).

B. Empirically-Based IDE Designs

To address the problems with traditional IDEs, researchers have proposed new design concepts based on empirical findings. For example, JASPER [18], Code Bubbles [16], [17], Code Canvas [21], and Patchworks [24], [27] change the typical way that code is displayed on screen. These editors are characterized by features that allow efficiently arranging multiple pieces of code onscreen without tabs.

Code Bubbles enables developers to open code into "bubbles" on a large two-dimensional canvas and arrange them freely. These bubbles may contain a function, a class, or an entire code file. The canvas is far larger than the visible screen area, so the developer can move the viewable area in both dimensions. Patchworks takes a different approach, by providing a fixed grid of "patches", which may also contain a function, class, or entire file. Once the grid is full, it can be shifted to the left or right on a virtually endless ribbon to reveal more patches.

III. Study Methodologies

To address our research questions, we analyzed code repositories from GitHub and then simulated different code arrangements in an IDE. First, we mined 95 repositories from GitHub to measure the size of code for different programming languages. For our second question, we measured the space available for code on a popular text editor and determined how much code could fit for 9 different document arrangements.

A. Mining GitHub Repositories

We selected the 25 most popular GitHub repositories for C++, Java, JavaScript, and Python, as of March 12th, 2019, for a total of 100 repositories [1]. These programming languages were picked due to their popularity [8]. We removed 5 repositories from our analyses since they did not include any code (e.g., one was a list of libraries and resources outside of the repository). We analyzed 524,547 code functions from the remaining 95 repositories. For each function, we calculated the length of each function in lines and the width in characters. The width is the max number of characters on a single line within the function. Additionally, we excluded any function that contained over 1,000 characters on a single line. These were "minified" JavaScript files, which contain no newlines or whitespace, and are not meant to be read or modified by developers.

B. Typical Screen Resolution

To know the typical screen resolutions of monitors that developers use, we contacted 7 professional software developers from two large technology companies. We asked what their company's standard practice was for providing monitors to newly hired developers in terms of the screen size and resolution. One company provides developers the option of either two 24-inch monitors or one 30-inch. The other company gives developers two 27-inch monitors. The resolutions of these displays are 1920x1080 or 2560x1440. Similarly, Stack Overflow's 2018 Developer Survey reported that 51% of developers use two monitors and 32% use one monitor (though the resolutions are not reported) [6].

C. Simulating Development Environments

Using the results from both III-A and III-B, we simulated how much code can fit on screen at one time. We performed this simulation using VS Code with the default settings. VS Code is the most popular IDE as reported by developers on the 2018 and 2019 Stack Overflow Developer Surveys [6], [7], with 51% of the 87,317 respondents using VS Code. Using two common screen resolutions, we determined how many functions of code can fit on the screen, for each of the 4 programming languages, using the average width and height described in the previous subsection.

We simulated several different arrangements of code documents in VS Code in an effort to explore possible configurations that a developer might use. In particular, we used 9 arrangements, starting with one document onscreen, two documents side by side, and continued this pattern up to a 3x3 grid of code documents. Most IDEs support these arrangements by dragging a tab document across the screen,

[1] https://github.com/utk-se/CodeSize

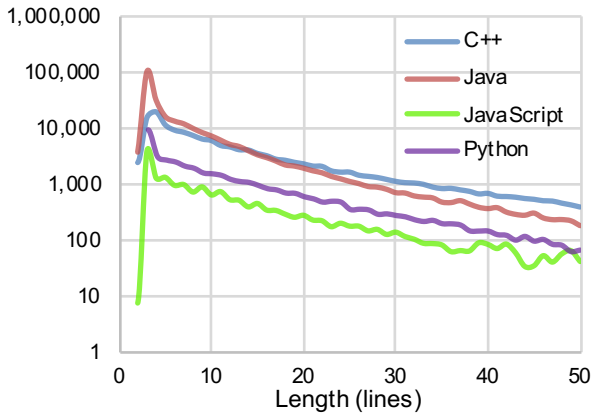

Fig. 2. Frequency distributions of lengths of code functions per language.

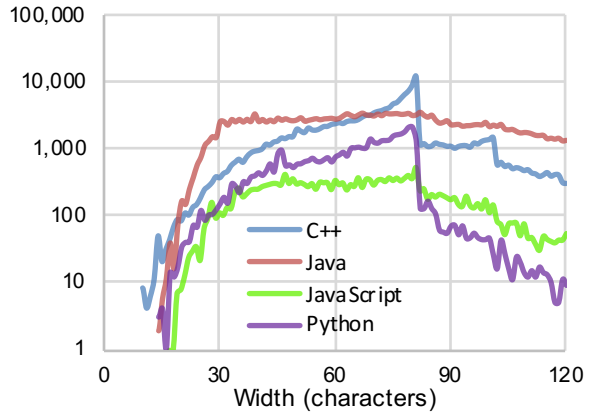

Fig. 3. Frequency distributions of widths of code functions per language.

TABLE I
NUMBER OF CODE FUNCTIONS VISIBLE ONSCREEN WITH A 1920X1080 MONITOR (HIGHLIGHTING INDICATES HIGHTEST/LOWEST VALUES).

Code arrangement (columns x rows)	C++	Java	JS	Python
1x1	2.57	5.64	2.31	3.21
1x2	2.46	5.41	2.21	3.07
1x3	2.31	5.07	2.08	2.88
2x1	5.14	11.28	4.62	6.42
2x2	4.92	10.82	4.42	6.14
2x3	2.62	10.14	4.16	5.76
3x1	5.57	13.48	5.84	7.40
3x2	5.52	12.94	5.61	7.11
3x3	5.18	12.13	5.26	6.66

which will split the adjacent editor in half. You can repeat this process to juxtapose more documents. For example, Fig. 1 shows VS Code with a 2x2 arrangement of code documents.

TABLE II
NUMBER OF CODE FUNCTIONS VISIBLE ONSCREEN WITH A 2560X1440 MONITOR (HIGHLIGHTING INDICATES HIGHTEST/LOWEST VALUES).

Code arrangement (columns x rows)	C++	Java	JS	Python
1x1	3.90	8.57	3.51	4.87
1x2	3.80	8.34	3.41	4.74
1x3	3.69	8.12	3.32	4.62
2x1	7.80	17.14	7.02	9.74
2x2	7.60	16.68	6.82	9.48
2x3	7.38	16.24	6.64	9.24
3x1	11.70	25.71	10.53	14.61
3x2	11.40	25.02	10.23	14.22
3x3	11.07	24.36	9.96	13.86

IV. Evaluation Results

A. RQ1 Results: Size of Code

Recall that we calculated the length and width of 524,547 functions from popular code projects on GitHub. Fig. 2 shows the frequency distribution of function lengths for each of the 4 programming languages that we investigated. The overall distributions appear similar, with a large number of functions being around 5 lines of code, regardless of language, although the proportions do differ. In fact, the average length of functions per programming language is considerably different: the average for Java is 8.87 lines ($SD = 2.80$), C++ is 19.48 lines ($SD = 6.29$), Python is 15.60 lines ($SD = 6.86$), and JavaScript is 21.68 lines ($SD = 9.90$).

Fig. 3 shows the frequency distributions for function width per language. Recall that we measured a function's width by the max number of characters on a single line within the function. The figure shows that after 80 characters, all languages with the exception of Java, have a steep drop off. The average widths for Java is 70.27 characters ($SD = 11.68$), C++ is 74.96 characters ($SD = 14.90$), Python is 72.75 ($SD = 14.26$), and JavaScript is 66.32 characters ($SD = 14.63$) . Overall, the languages have an average function width of 71.01 characters ($SD = 3.70$).

B. RQ2 Results: Code on Screen

Using the results from RQ1, we simulated how many functions can be opened in VS Code using different arrangements (e.g., two documents side by side). Table I and Table II show how many functions can be visible onscreen for 1920x1080 and 2560x1440, respectively. The results vary considerably given the arrangement of onscreen. For example, with C++ you can see 2.57 to 5.57 functions with a 1920x1080 resolution monitor. With 2560x1440, the range for C++ is 3.90 to 11.70 functions. For most of these arrangements, the limitation is vertical space. However, for 1920x1080 using 3 columns of documents requires more horizontal space.

V. Discussion

The results indicate that the length of code depends on the programming language, while the width is similar across programming languages. The results also suggest that juxtaposing multiple panes would increase the amount of code able to be viewed simultaneously, though an optimal amount of code differs by screen resolution and programming language.

A. Code Size

The average length varies considerably from language to language. Java had the shortest average length of 8.87 lines, which we believe is due to it being an explicitly object-oriented programming language, and it have many small functions called getters and setters. Additionally, developers often follow the philosophy that functions are responsible for only one thing. Fig. 2 shows that most languages have an increase of functions of length 3 to 4 lines; however, Java has many more functions in this range, which gives evidence of more, smaller functions.

All four of the languages we investigated tend to have an average width of 71 characters, with little deviation. A possible rationale for this is that terminals have historically had 80 character limitations (and before that, punch cards had 80 columns). Some code editors will suggest breaking a line into multiple lines after crossing this threshold.

B. How much code can fit on screen?

The results of RQ2 depends both on the screen resolution as well as what programming language is used. For 2560x1440 monitors, the optimal way to view code is with 3 columns and 1 row of code documents. This allowed for the most code to completely fit on the screen without the developer having to scroll horizontally. Adding additional rows, such as a 3x2 or 3x3 configuration, would require a minor amount of vertical scrolling. For all layouts with more than 3 columns on 2560x1440 monitors, the developer would need to scroll horizontally to be able to see the full function. Similarly, all layouts with more than 2 columns on 1920x1080 monitors the developer would need to scroll horizontally see all the code. Additionally, this all depends on functions being separated without additional whitespace or comments (e.g., Javadoc).

An important assumption that these results depend on is that relevant functions are adjacent to one another within the code file. If this assumption does not hold, then this number of onscreen functions is drastically overestimated. To overcome this, the developer would need to open additional documents and arrange them appropriately.

C. Implications for Design

The results of our analyses reveal several implications for the design of IDEs. For example, it further motivates the need for IDEs such as Code Bubbles [17], Code Canvas [21], and Patchworks [24]. However, our results could lead to features in these IDEs that allow developers to efficiently arrange the onscreen code documents. For example, Code Bubbles could align bubbles into a grid and Patchworks could provide arrangement configurations other than 3x2.

Another feature for IDEs could be to automatically configure the document panes based on the developer's behavior, the programming language, and the available screen space. For example, if the developer frequently revisits a small set of code locations (as they often do [22], [25], [29], [31]), then the IDE would suggest placing those code documents side by side and would do so automatically with a single click. The IDE would also be able to take into account the specific size of those functions, as well as other panes that the developer is using for their task. In fact, Johnson et al. declared the need for *bespoke tools*, that adapt to an individual developer's needs [28].

D. Limitations

Our analyses have several limitations that should be addressed in future work. First, we only analyzed code from 95 public repositories using 4 different languages. It is possible that these projects do not make for a representative sample. Second, we only simulated opening and arranging the code documents using one code editor, VS Code, and chose the default settings. Changing the settings, such as what panes are visible or the font size, would have a considerable effect on the results. Third, we only considered two possible screen resolutions. However, in an attempt to mitigate each of these limitations, we selected the repositories, languages, screen resolutions, and editor all based on popularity.

VI. Conclusion

In this paper, we analyzed the size of over 500,000 code functions and simulated opening code with various arrangements to better understand how IDEs should display and layout code. The analyses yielded the following findings:

- RQ1 (size of code): The length of code depends on the programming language that is used, while the width of code is similar across languages.
- RQ2 (code on screen): The amount of code on screen can be substantially increased by properly configuring the IDE and by using a higher resolution monitor.

We hope that the findings of this empirical investigation are transformative to the future of IDE designs and ultimately lead to more productive developers. The future work of this direction include further studies on developers' habits of arranging code onscreen and designing an IDE that reactively adjusts the arrangement of code documents based on the developer's behavior, screen resolution, and programming language. This work should be another substantial step towards an empirically-based integrated development environment.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1850027.

REFERENCES

[1] "Atom," https://atom.io/, accessed: 2019-05-04.

[2] "Eclipse," https://www.eclipse.org/, accessed: 2019-05-04.

[3] "Intellij idea," https://www.jetbrains.com/idea/, accessed: 2019-05-04.

[4] "Jetbrains 2018 annual report," https://www.jetbrains.com/annualreport/2018/tools/, accessed: 2019-05-04.

[5] "Pycharm," https://www.jetbrains.com/pycharm/, accessed: 2019-05-04.

[6] "Stack overflow developer survey 2018," https://insights.stackoverflow.com/survey/2018, accessed: 2019-05-04.

[7] "Stack overflow developer survey 2019," https://insights.stackoverflow.com/survey/2019, accessed: 2019-05-04.

[8] "Tiobe index for may 2019," https://www.tiobe.com/tiobe-index/, accessed: 2019-05-04.

[9] "Visual studio," https://visualstudio.microsoft.com/, accessed: 2019-05-04.

[10] "Visual studio code," https://code.visualstudio.com/, accessed: 2019-05-04.

[11] "Visual studio code at connect 2017," https://code.visualstudio.com/blogs/2017/11/16/connect, accessed: 2019-05-04.

[12] "Xcode," https://developer.apple.com/xcode/, accessed: 2019-05-04.

[13] *Gerrit*, Accessed Jan 2019, https://www.gerritcodereview.com/.

[14] *Phabricator*, Accessed Jan 2019, https://www.phacility.com/.

[15] 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.

[16] 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. 32nd ACM/IEEE Int'l Conf. Software Engineering - Volume 1*, ser. ICSE '10, 2010, pp. 455–464.

[17] 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. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '10, 2010, pp. 2503–2512.

[18] M. J. Coblenz, A. J. Ko, and B. A. Myers, "Jasper: An eclipse plug-in to facilitate software maintenance tasks," in *Proc. 2006 OOPSLA Workshop on Eclipse Technology eXchange*, ser. ETX '06, 2006, pp. 65–69.

[19] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proc. 2005 IEEE Symp. Visual Languages and Human-Centric Computing*, ser. VL/HCC '05, 2005, pp. 241–248.

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

[21] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proc. 32nd ACM/IEEE Int'l Conf. Software Engineering - Volume 2*, ser. ICSE '10, 2010, pp. 207–210.

[22] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 7–18.

[23] A. Z. Henley and S. D. Fleming, "Yestercode: Improving code-change support in visual dataflow programming environments," in *2016 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '16)*, Sept 2016, pp. 106–114.

[24] A. Z. Henley and S. D. Fleming, "The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '14, 2014, pp. 2511–2520.

[25] 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.

[26] A. Z. Henley, K. Muçlu, M. Christakis, S. D. Fleming, and C. Bird, "Cfar: A tool to increase communication, productivity, and review quality in collaborative code reviews," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 157:1–157:13. [Online]. Available: http://doi.acm.org/10.1145/3173574.3173731

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

[28] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman, "Bespoke tools: Adapted to the concepts developers know," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 878–881. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803197

[29] 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. 27th Int'l Conf. Software Engineering*, ser. ICSE '05, 2005, pp. 126–135.

[30] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[31] 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.

[32] C. Parnin, C. Görg, and S. Rugaber, "Codepad: Interactive spaces for maintaining concentration in programming environments," in *Proc. 5th Int'l Symp. Software Visualization*, ser. SOFTVIS '10, 2010, pp. 15–24.

[33] 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. ACM SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '12, 2012, pp. 1471–1480.

[34] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, "To fix or to learn? how production bias affects developers' information foraging during debugging," in *31st IEEE Int'l Conf. Software Maintenance and Evolution*, ser. ICSME '15, 2015.

[35] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950302

[36] 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. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '13, 2013, pp. 3063–3072.

[37] J. Singer, R. Elves, and M. D. Storey, "Navtracks: Supporting navigation in software maintenance," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, 2005, pp. 325–334. [Online]. Available: https://doi.org/10.1109/ICSM.2005.66

[38] A. Walenstein, "Cognitive support in software engineering tools: A distributed cognition framework," Ph.D. dissertation, Simon Fraser University, 2002.

[39] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proc. 37th Int'l Conf. Software Engineering (ICSE '15)*, 2015, pp. 223–233.