A Retrospective on How Developers Seek, Relate, and Collect Information About Code

Amy J. Ko, *University of Washington*, Brad A. Myers, *Carnegie Mellon University*, Michael Coblenz, *University of California, San Diego*, Htet Htet Aung, *HHAeXchange*

***Abstract*—In the early 2000's, software development was shifting from offline to online, and from command line to IDE. We discuss our 2004 paper examining the impact of this shift on developers' program comprehension behaviors, our motivation for the work, and it's impact on the last twenty years of empirical studies and developer tool innovations. We end with a discussion of the possible unintended impacts LLMs have on program comprehension in the coming decades.**

***Index Terms*—Software evolution, programming environments**

I. Software Engineering in 2004

The early 2000s was a time of disruption for developer tools. In some ways, it was a time of great consolidation. After decades of loosely integrated standalone command line tools, and research on highly integrated tools [10], integrated development environments (IDEs) had begun to dominate, with platforms like Eclipse and Visual Studio becoming central tools for bringing editing, testing, and debugging into unified environments. The cultural dominance of the "cowboy coder" stereotype [14], relying only on a plain text editor, a command line compiler, and minimal process, was beginning to fade, replaced by a new legitimacy of harnessing every tool available to make developers in teams more productive. The consumer internet was only a decade old, "web 2.0" of interactive, user-generated content was still nascent, and so software development was still, briefly, something that occurred mostly offline, with a centralized repository, on a developer's workstation.

It was around this time, in 2004, when the first author was interested in how developers understand and debug code. She had been reading more than two decades of research on program comprehension and the psychology of programming [2], captivated by the ways that people build mental models of software architectures, data, and control flow. But she was also reading about the more than decade of work to algorithmically model dependency graphs, and use those to interrogate software repositories to streamline debugging and understanding [30]. In this context, IDEs seemed like the perfect context in which to explore not only how developers reasoned about dependencies, but also how IDEs might support that reasoning.

Of course, we knew a lot about developers' reasoning about code already. One paper in particular was fascinating to the first author. Twenty two years earlier, Mark Weiser had written a seminal article, *Programmers use slices when debugging* [28], which spurred two decades of algorithm innovation in dependency analysis. This empirical study examined whether programmers, when debugging, construct mental models of program slices by manually analyzing control and data dependencies. In the study, participants were offered a few debugging tasks on two programs consisting of 75-150 lines of Algol-W code, consisting of a few subroutines. Crucially, these programs fit on a few pieces of paper, all of the information encoded in variable names was replaced with arbitrary single symbol names, and there were no comments. After each debugging session, participants were shown a set of relevant and irrelevant slices of the program, and through some basic statistical comparisons, found that participants were far more likely to recognize relevant slices than irrelevant ones. This suggested that they had focused their attention and mental model construction on relevant slices.

It has of course been more than 40 years since this study was done, and there are no end of critiques we might make of it now, having evolved our empirical methods substantially. Even in 2004, however, there were many things the first author found suspect:

- In the classical traditions of 20th century cognitive psychology, the study removed everything real that might influence how developers reason about code: comments, variable and function names, and other tools that support comprehension. It might be the case that developers use slices *when they have nothing else*, but what do they use when they *do* have all of these other things?
- As a human-computer interaction researcher, the first author started from the premise that user interface matters. Surely reading 1-2 pages of printed Algol code was different than reading a large, multi-file program in a modern IDE. New theories like the Hutchin's work on distributed cognition [12] showed that the mental models we build are partly a product of how we externalize cognition to the tools and structures around us. Surely IDEs changed how developers build mental models, and even what models they built.
- The way the study examined mental models, using a recall metric, rather than directly observing what developers do, seemed highly suspect. Of course, such direct observations were hard to do when someone is just reading printouts; seeing what they do in an IDE in 2004 was far more feasible.

These limitations were motivation enough to look again at how developers understand programs, but this time, more closely, and in the context of IDEs. This felt especially important as two decades of algorithmic optimization around

program slice computation had been catalyzed on a single suspect empirical study, with almost no later validation of their utility.

II. The study

The study we designed [16] was actually part of one of the first author's classes. We had a broad semester-long project, and the first author had teamed up with classmate (and now colleague) James Fogarty to run a study that jointly examined two things: the first author's question about the influence of IDEs on program comprehension and Fogarty's question on how developers mediate interruptions. We designed a simple study in which developers were given a small ∼500 line Java program that implemented a stroke-based painting application with the Swing user interface toolkit. They received five tasks, fixing a scroll bar rendering bug, a defect in the color selector tool, unreliable undo behavior, adding a line drawing feature, and a stroke thickness feature. They used the Eclipse IDE, with a full set of standard plugins for searching, dependency analysis. They were also interrupted occasionally, to meet Fogarty's goals, though we viewed this as an authentic form of interruption from colleagues.

To analyze developer behavior at a highly granular level, we screen-recorded everything, allowing us to retrospectively examine all of their actions. With the help of then undergraduate Michael Coblenz and masters student Htet Htet Aung, we translated these screen recordings into a detailed log of observable developer actions, ranging from reading and editing code to navigating static and dynamic dependencies, searching for text, manual testing, reading documentation, changing source files, and more. We then analyzed these action sequences to examine how developers orchestrate program understanding using IDEs.

While the results were a bit of a "fishing expedition" of every possible empirical observation that seemed notable, there were a few key findings that we believe had lasting value:

- In contrast to Weiser's empirically-derived concept of a slice, we found that developers were more likely to build out a "working set" of potentially relevant code. This sometimes included things that approximated slices, but not complete slices, or even precise slices, but rather possible subgraphs of program dependencies that might be worth further investigation, or areas of implementation that might be important to implementing a feature. This more process-oriented, fuzzy view of code relevance was in direct contrast to the more unambiguous, rigid definition of a program slice, building upon other new theories at the time, such as information foraging [22].
- Whereas Weiser's study considered the final mental models that developers had built, our study examined the role of navigation and exploration in constructing models. Developers navigated direct dependencies, just as the Weiser study posited, but they also navigated *indirect* (or transitive) dependencies. And they did all of this through a combination of searching, scrolling, definition jumping, and reading, all to construct a working set of relevant context in order to localize a defect or plan a feature. A notable navigation that we found *poorly* supported by the IDE was returning *back* to previously visited locations.
- Given the same problem, developers vary in what they find relevant, the order in which they find it relevant, and the amount of reading, re-reading, and re-navigation they do, but this variation is mostly shaped by early choices in a task about what to attend to. In the study, this variation largely explained variation in productivity and success. This suggested that developer productivity was not necessarily an intrinsic quality, but more like the ant in Herb Simon's seminal thought experiment; just like the route an ant on a beach takes through the sand is heavily influenced by the hills, valleys, and rocks on the way, developers' program comprehension appeared shaped by the code and tools in their environment [27].

We think the study was also notable methodologically: rather than relying on careful experiment designs and macro-scale observed outcomes, this was one of the few empirical studies in software engineering to watch developers at the micro-scale to understand how low-level navigation tasks end up shaping their high-level success. Another notable paper that did this, on the same topic, was Robillard et al.'s fantastic *How effective developers investigate source code* [23], published just before our work, and found quite similar trends, but notably also found that methodical strategies for understanding were more successful than opportunistic ones.

III. Impact

We are strong believers that no one paper is responsible for progress or impact in research. Our study, as much as it has been cited and built upon, itself built upon decades of research in software engineering, HCI, and cognitive psychology. And any impact our work had also part of broader trends, in both research and practice at the time. What we list here, then, are shifts and changes in the world of developer tools that have some trace of our ideas, even though they far more shaped by others.

One area of impact was on tools that built upon the notion of a "working set." *Code Bubbles* [3] and *Jasper* [8], for example, were directly inspired by this paper, as were the subsequent efforts at Microsoft Research on the *Debugger Canvas*[1]. Comments like the ones below suggest suggest something about the paper's findings, and the tools built upon them, resonates closely with some developers' program understanding needs.

> "Awesome Debugging Tool for VS 2010, speeds up debugging time considerably."
>
> "I must say this is awesome feature. I wish this was available to Premium users. It sucks that I cannot use this at work."
>
> "Biggest innovation in debugging. Please port to newer VS or make it open-source."

Other navigation features in IDEs like "back" buttons, do not have explicit provenance back to our work, but are at least validation of the discovery that developers' navigation of code is a central activity that needs support.

[1] https://marketplace.visualstudio.com/items?itemName=DebuggerCanvasTeam.DebuggerCanvas

Despite 20 years of improvements since 2004, programmers still spend a large amount of time navigating. In a larger-scale study with professionals, Xia et al. [29] found that in an automated analysis of two-week-long recordings that developers spent about 24% of their time navigating in code. Other researchers have contributed complementary findings that, combined with our results, have generated insights about tool design. For example, Sillito et al. [26] studied *questions* programmers ask while programming, which complements our analysis of *actions*. It remains to be seen whether the advent of large language models (LLMs) may reduce the fraction of time spent navigating. Even traditional autocomplete systems reduce the amount of time spent reading documentation, which is one of the causes of navigation activities [15].

Our results showing how much time developers spend on specific activities, including browsing, reading, and writing, have been cited to motivate research on helping developers with program comprehension tasks. Moreno et al. [20], for example, proposed summarization tools for code, whereas Zhang et al. proposed summarization tools for unit tests. Our results showing how much time developers spend navigating has been cited to argue that reducing the number of files may improve efficacy [25].

Researchers have used our results to provide background regarding debugging [1], [19] and feature location [7]. Zhang et al. [33] leveraged our finding that developers use contextual information to develop automated program repair techniques. Myers and Stylos used our results to show the importance of names in APIs [21], since we found that guessing possible names of identifiers and searching for them was a key technique participants used to find relevant code.

Leveraging observations of human developers has provided useful insights regarding how to make tools as effective as possible. SequencR is a programming tool that is designed to mimic human bug fixing methods [5]. RepairAgent is an LLM-based program repair tool that, unlike other LLM-based tools, considers how *humans* write code rather than only considering the most effective algorithms [4]. Turning to debuggers, Chiş et al. [6] developed a framework for moldable debuggers, arguing on the basis of our paper that domain-specific interfaces are needed for debugging particular kinds of software, since programmers need different information in different contexts. Robillard and Manggala [24] designed a code recommendation tool, ConcernDetector, which suggests code that might be relevant to a developer's current task.

Later studies of *code search*, summarized by Di Grazia and Pradel [9], further characterized the kinds of searches developers do. For example, developers conduct both free-form queries and structured queries (such as by providing code with holes). According to Liu et al. [18], these studies led to 318 publications regarding new code search tools between 2007 and 2020, the span of years that the paper analyzed. Half of those tools pertained to text-based code search; other categories included searching for code clones and API usage examples.

Our paper's observations about *context* have been extended and corroborated in other works. Fritz et al. [11] analyzed change contexts, finding (among other observations) that different developers construct significantly different contexts from each other even when conducting the same tasks. However, 73% of the elements of contexts were structurally connected with each other. Consistent with our findings, developers often start constructing their contexts with textual searches: 9 of 12 participants conducted a search within the first eight steps.

IV. Future work

As notable as all of this work is, it is clear that software development, and programming more broadly, is in another period of disruption. We appear to be at the end of 20 years of the web transforming how developers get answers to questions about languages, APIs, frameworks, and more. Now, all of that user-generated content from the web has enabled the training of LLMs that can often synthesize reasonably correct solutions to many parts of programs, summaries of program behavior, and even detect defects. While it is not yet the revolution that many have claimed — results are far too inconsistent and often grossly incorrect to fully automate anything — it does change the interfaces through which developers are making sense of code. Emerging studies (e.g., [32]) have already begun to investigate these trends, often using the same process lens as our 2004 work, examining how LLMs are changing how developers go about writing and understanding code.

And yet, much remains the same. One of the key challenges surfacing in these recent studies of LLMs are that no matter how good they get, they are wrong often enough that they put even greater pressure on program comprehension, to detect and overcome all of its failures [13]. If LLMs become an indispensable tool for development, and the need to comprehend its output becomes central, it may be that the long history of studies on program comprehension will become even more important in shaping data-driven developer tools.

References

[1] Alaboudi, A., LaToza, T.D. (2023) What constitutes debugging? An exploratory study of debugging episodes. Empirical Software Engingeering 28(117).

[2] Blackwell, A. F., Petre, M., & Church, L. (2019). Fifty years of the psychology of programming. *International Journal of Human-Computer Studies*, 131, 52-63.

[3] Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., & LaViola Jr, J. J. (2010, April). Code bubbles: a working set-based interface for code understanding and maintenance. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2503-2512).

[4] Islem Bouzenia, Premkumar Devanbu, Michael Pradel (2025). RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. International Conference on Software Engineering 2025.

[5] Z. Chen, S. Kommrusch, M. Tufano, L. -N. Pouchet, D. Poshyvanyk and M. Monperrus (2021). SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering*, 47(9), 1943-1959.

[6] Chiş, A., Gîrba, T., Nierstrasz, O. (2014). The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In Software Language Engineering (SLE).

[7] Dit, B., Revelle, M., Gethers, M., & Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. Journal of software: Evolution and Process, 25(1), 53-95.

[8] Coblenz, M. J., Ko, A. J., & Myers, B. A. (2006, October). JASPER: an Eclipse plug-in to facilitate software maintenance tasks. In Proceedings of the 2006 *OOPSLA workshop on eclipse technology eXchange* (pp. 65-69).

[9] Luca Di Grazia and Michael Pradel (2023). Code Search: A Survey of Techniques for Finding Code. ACM Computing Surveys. 55(11), Article 220.
[10] Engels, G., Lewerentz, C., Nagl, M., Schäfer, W., & Schürr, A. (1992). Building integrated software development environments. Part I: tool specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(2), 135-167.
[11] Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich (2014). Developers' code context models for change tasks. Foundations of Software Engineering.
[12] Hutchins, E. (2000). Distributed cognition. International Encyclopedia of the Social and Behavioral Sciences. *Elsevier Science*, 138, 1-10.
[13] Imai, S. (2022, May). Is github copilot a substitute for human pair-programming? an empirical study. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (pp. 319-321).
[14] Janes, A. A., & Succi, G. (2013). The dark side of agile software development: First results. In *Selected topics to the User Conference on Software Quality, Test and Innovation* 2013: ASQT 2013; September 19th-20th, 2013, Graz, Austria. Oesterreichische Computer Gesellschaft.
[15] Jiang, S., & Coblenz, M. (2024). An Analysis of the Costs and Benefits of Autocomplete in IDEs. Proceedings of the ACM on Software Engineering (FSE), 1284-1306
[16] Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12), 971-987.
[17] Jacob Krüger, Thorsten Berger, Thomas Leich (2018). Features and How to Find Them: A Survey of Manual Feature Location. In Software Engineering for Variability Intensive Systems: Foundations and Applications.
[18] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy (2021). Opportunities and Challenges in Code Search Tools. ACM Computing Surveys 54(9), Article 196.
[19] Marco Manca, Fabio Paternò, Carmen Santoro, and Luca Corcella (2019). Supporting end-user debugging of trigger-action rules for IoT applications, International Journal of Human-Computer Studies, 123, 56–69.
[20] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock and K. Vijay-Shanker, Automatic generation of natural language summaries for Java classes. ICPC 2013.
[21] Myers, Brad A., and Jeffrey Stylos. "Improving API usability." Communications of the ACM 59(6) (2016), 62–69.
[22] Pirolli, P., & Card, S. (1999). Information foraging. *Psychological Review*, 106(4), 643.
[23] Robillard, M. P., Coelho, W., & Murphy, G. C. (2004). How effective developers investigate source code: An exploratory study. IEEE Transactions on software engineering, 30(12), 889-903.
[24] M. P. Robillard and P. Manggala (2008). Reusing Program Investigation Knowledge for Code Understanding. IEEE International Conference on Program Comprehension.
[25] Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. 2017. Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names (2017). TOSEM 26(2).
[26] J. Sillito, G. C. Murphy and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34(4), 434-451.
[27] Simon, H. A. (1988). *The science of design: Creating the artificial*.
[28] Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, 25(7), 446-452.
[29] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan and S. Li. (2018) Measuring Program Comprehension: A Large-Scale Field Study with Professionals." in IEEE Transactions on Software Engineering, 44(10) 951–976
[30] Xu, B., Qian, J., Zhang, X., Wu, Z., & Chen, L. (2005). A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes, 30(2), 1-36.
[31] Zhang, Benwen, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. ASE 2016.
[32] Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). Practices and challenges of using github copilot: An empirical study. arXiv preprint arXiv:2303.08733.
[33] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen (2024). A Survey of Learning-based Automated Program Repair. ACM Trans. Softw. Eng. Methodol. 33(2).