

Information Needs: Lessons for Programming Tools

Thomas D. LaToza, George Mason University

// Why is programming sometimes so frustrating and annoying and other times so fast and painless? This article surveys a few of the important lessons emerging from studies of programming and the new programming tools they motivate. //



IT OFTEN SEEMS as if nothing much ever changes with programming tools. The core experience of programming seems remarkably static: type text in a code editor, see compile errors, run the code, add log statements, and use the debugger to understand why things did not work.

Over the past few decades, however, computer scientists and psychologists

in areas such as cognitive psychology, software engineering, human-computer interaction, and computer science education have studied programming, revealing what makes it hard. Through findings from this work, new programming tools have been invented that change fundamental aspects of how a developer interacts with code.

In this article, I describe a few key findings about what programming is gleaned from the careful observation

and study of programming. Beginning with the underlying theory about the nature of human problem solving from cognitive psychology, I explore how this motivates an information needs perspective on programming and programming tools. Rather than exhaustively surveying every finding from the academic literature, I instead focus on a few particularly common and challenging developer activities—debugging, navigating concerns, understanding design rationale, and onboarding—and illustrate how one or two key findings have motivated tools to help developers work better.

The Psychology of Problem Solving

Psychologists have long examined what it is that humans do when they solve problems.¹ One key aspect of problem solving is trial and error. Problem solving is often like a maze, where humans choose which path to follow, backtracking as necessary when choices do not work as expected. Beginning with high-level goals, humans decompose these into lower-level goals.

Programming is replete with nested goals, as developers break down high-level objectives (e.g., fix this bug, implement this feature) into ever lower-level goals (e.g., determine what value this field is first initialized to). This suggests an answer to the question of “what is programming?” Programming is the act of translating a high-level change to a program into a sequence of actions with which to achieve this change.

In the best of circumstances, this translation may be routine and easy, as developers readily formulate actions that directly achieve their goals. However, programming is often dominated by moments when this is challenging—when there are barriers to achieving these goals or breakdowns where the understanding

of the impact of actions no longer matches reality.

One way to make explicit how high-level changes are decomposed into subgoals and the challenges this brings is to view each subgoal as a question. Researchers have examined the information needs of developers, identifying questions that they ask. One study examining the questions developers report to be hard to answer² found questions such as the following:

- How can I refactor this without breaking existing users?
- What does this do in this case?
- Which function or object should I pick?
- Where is this functionality implemented?

Questions often reflect specific situations that describe how code behaves, not across all executions, but when specific events occur.

Based on these insights, tools can then consider exactly how to make specific questions easier to answer. For example, consider again these questions and imagine how a tool could help. Tools might

- work to identify which methods or parameters are used externally outside a project or which behaviors are visible
- enable developers to more easily simulate the execution of a method with less setup
- allow developers to differentiate the behavior of similar methods through comparisons of execution behavior, crowdsourced content, or recorded performance data
- assist in feature location, identifying methods involved in the implementation of a specific user-facing feature.

By investigating the fit between the questions and situations that developers find challenging and the capabilities of tools in these situations, it is possible to understand the degree to which tools support developers. This might be more carefully measured by, for example, documenting how often a question occurs, how much time it takes to answer, or the consequences that occur when developers cannot answer the question or answer it incorrectly. Tools can then be evaluated for the impact that they have in these situations.

explore these in detail as representative and particularly important.

Debugging

Many factors can make debugging hard: incorrect hypotheses may arise from mistaken assumptions or misperceptions of behavior, symptoms may be separated from the root cause by a chasm involving long and complex control and data flow, and defects may involve timing or synchronization problems and be intermittent, inconsistent, or infrequent.^{4,5}

One important finding in the study of debugging is the centrality of



One important finding in the study of debugging is the centrality of following data and control flow, particularly in large and complex codebases.

What questions are the most important for programming tools to support? Studies offer many answers. A study of questions developers report to be hard to answer found that understanding design rationale, understanding implementations, and debugging are the most common challenges.² A study of the low-level actions developers take while programming found that developers spend the most time navigating code (35%), more than reading (20%) or editing (20%) code. Of course, developers face a variety of other challenges in specific situations, such as onboarding onto a new project.³ While the information needs literature suggests a wide variety of other questions, I

following data and control flow, particularly in large and complex codebases. Studies have characterized how developers debug as the process of information foraging, choosing which of the many call relationships between methods to navigate to find necessary information.⁶ Developers report hard-to-answer questions, such as the following:

- In what situations or user scenarios is this called?
- How do calls flow across process boundaries?
- What is the original source of these data?
- How did this runtime state occur?²

Through observing the moment-to-moment behavior of developers, it is possible to explain what makes this hard. Developers ask reachability questions and search forward or backward across control flow for statements matching search criteria.⁵ For example, to understand what a test is doing that is different from normal app behavior, developers attempt to compare the execution behavior in both cases, identifying statements that executed differently in each case. This is often difficult: developers can spend tens of minutes answering a single

their search and let their programming environment identify relevant information in the execution trace. In Reacher, developers can, beginning with a statement in the code, invoke an upstream or downstream search, generating a set of all statements that executed before or in response to the current statement.⁸ From this, developers can search, entering keywords to match identifiers in method calls or field reads and writes. This then generates a visualization that explains the control flow between the method a developer is currently viewing to each

Developers also navigate to understand the use and behavior of methods they call. Modern IDEs, such as Eclipse and IntelliJ, support easily browsing to the definition of a method, seeing its callers, or even following these paths several layers deep in a tree view. However, developers still struggle and can sometimes become disoriented.⁸ Even small overhead switching between files can slow developers down or, worse yet, cause them to miss information and insert defects.

One solution is for the IDE to explicitly represent a task context, the task-relevant methods and relationships. In Mylar (now Mylyn), developers are offered a filtered element browser, listing only elements that belong to the task context.¹⁰ When switching between elements within the task context, developers need not remember where functionality was located, drill down through long and complex package structures, and find where they need to navigate to. Developers can instead simply switch between the relevant elements they have recently interacted with or that have been inferred to be relevant. This enables developers to spend less time navigating and more time editing code. Mylyn is now available as part of the Eclipse IDE.

A more radical approach might be to completely rethink how code is displayed to the developer. Rather than requiring developers to hierarchically browse through directories, packages, files, and classes to find relevant code, an IDE might instead simply show only the code that is relevant. The presentation need not even be in the traditional, linear listing of a file. Instead, the IDE might directly show code on a 2D canvas, including only task-relevant methods, depicting control and data flow

A better solution might be to let developers directly express their search and let their programming environment identify relevant information in the execution trace.

reachability question, may get lost and disoriented, and might erroneously make assumptions, causing bugs.

One way to better support debugging might be to simply let developers debug backward, stepping forward rather than back. In situations where developers want to, for example, trace the original source of data, this can help, as developers can more easily follow data backward without having to constantly set breakpoints and re-run the code or guess and insert many log statements. Research tools have long envisioned this,⁷ and this feature can now be found in some commercial tools, such as Visual Studio's Time Travel Debugging.

However, a better solution might be to let developers directly express

of these related methods and statements. Through this, developers can complete programming tasks considerably more quickly and successfully.

Navigating Concerns

Surprisingly, detailed examination of developers' time has shown that they can spend 35% of their time simply navigating, as they iterate through search results, navigate between indirect dependencies, and recover task contexts.⁹ One reason is that concerns are often scattered, and working to implement a specific feature will require gathering information and editing code in many files. More than 90% of the changes to the Mozilla and Eclipse projects involved multiple files.¹⁰

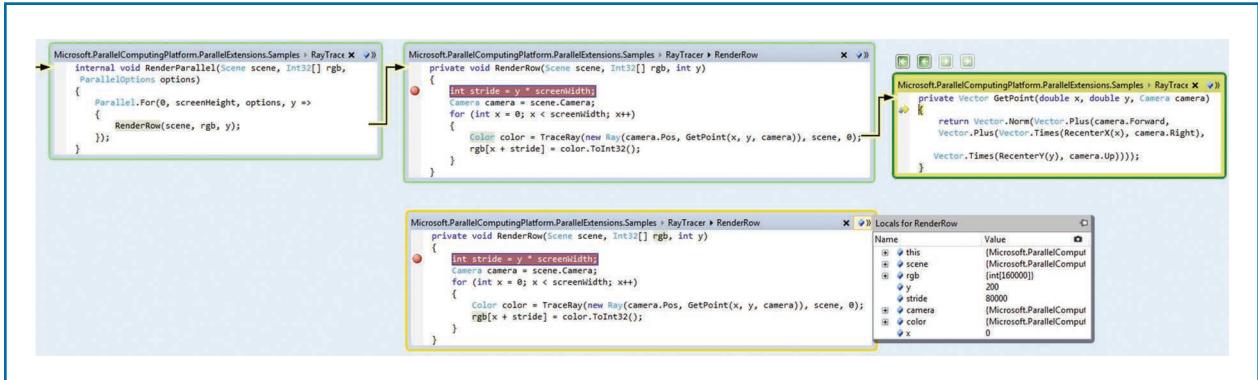


FIGURE 1. The Debugger Canvas visually arranges methods and debug windows on a 2D canvas. For example, when stepping through multithreaded code, each thread gets its own color, the currently executing method is highlighted in yellow, and each bubble has its own debugger window. (Source: Rob Deline; used with permission.)

connections and even overlaying debugging information.¹¹ Task contexts might even be saved for later use, enabling easier task resumption or sharing with teammates. Debugger Canvas is an add-in for Visual Studio that offers such a canvas-centered experience (Figure 1). Feedback from users revealed many times when it offered substantial value, such as when working with long and complex control flow, in a large unfamiliar codebase, or reasoning about dependency injection. Practical usability and performance issues sometimes limited its use.

Design Rationale

You're looking at some really strange code that seems to defy explanation: a method call is being made to a getter function, but the return value of the getter is ignored. Can you just remove it? Or is there some reason this code is here?

Developers working in large and complex codebases are constantly bombarded with questions about rationale. Some of the most frequently reported hard-to-answer questions² and most serious problems developers report facing¹² include the following:

- Why wasn't it done this other way?
- Was this intentional, accidental, or a hack?
- What is the policy for doing this?

active documentation, design rules are made explicit and checked against the code.¹³ Developers can view design rules for the current file in a panel to the right of their code (Figure 2). When editing code, developers

Even small overhead switching between files can slow developers down or, worse yet, cause them to miss information and insert defects.

This is largely because design rules vanish, failing to be documented or updated and remaining tacit in developers' heads.

IDEs might better support developers in working with design rationale by making it more explicit and better connected to the code. Rather than leave it in documents disconnected from the code or in comments disconnected from all of its uses, design rules might instead be bidirectionally connected to the code. In

receive immediate feedback, with violated design rules highlighted in red. Developers can see text describing the rationale for the design rule. To determine how to write code the right way, developers can navigate to positive examples of code snippets that follow the rule. This enables developers to work faster and more successfully.

Onboarding

You see a really interesting open source project to build a new multitrack audio

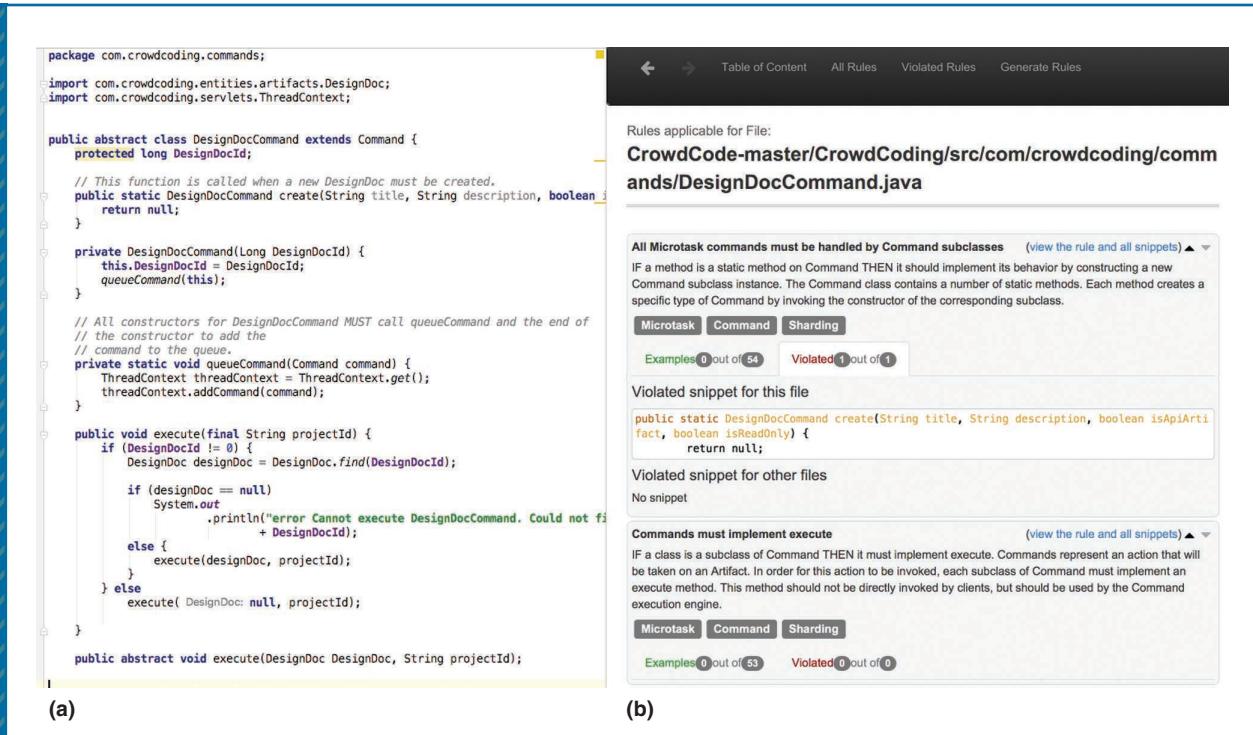


FIGURE 2. Active documentation integrates design rules into the IDE, offering explanations of related design rules and immediate feedback when rules are violated. For example, when (a) editing the file `DesignDocCommand` in the code editor, (b) the active documentation identifies design rules that apply to this file, including examples of rules satisfied and violated by code in the file.

editor and recorder. Having always wanted to work on an audio project, you decide you'd like to contribute. But what do you need to do to get started?

Software engineering researchers have studied the barriers that developers face when joining new projects in traditional organizations and in open source projects. In open source projects, these barriers can include

- identifying appropriate contacts and receiving feedback
- identifying tasks and artifacts
- understanding the project structure, complex code, and setting up a workspace
- outdated, unclear documentation
- learning project practices.³

Together, these barriers can take days or more to overcome. Knowing that

these barriers exist, developers may be dissuaded from contributing to a project. This diminishes the developers available to contribute to open source projects, further stressing busy contributors in responding to requests from users.

Programming tools have envisioned ways to reduce these barriers. One first step is to provide a more preconfigured programming environment, reducing the need to download and configure a new tool, set up dependencies, and ensure building works correctly. Commercial tools have already begun to solve this issue, offering cloud-hosted programming environments that are preconfigured and ready to code. Tools such as Codesandbox, available at <https://codesandbox.io/>, offer an online code editor, complete with pre-built projects.

But could programming environments go further to reduce onboarding barriers? A substantial problem is just how much knowledge about a project a developer must have to make a meaningful contribution. Developers must find where in the code to start and identify related methods as well as read this code to understand how it works. What if the IDE could instead simply create a small, self-contained task where a developer could decide to contribute to a software project in 20 or 30 min?

In microtask programming, developers complete self-contained microtasks, such as implementing a few lines within a function or listing a set of test cases for the description of a function.¹⁴ Rather than view or understand the whole codebase, developers instead work with an

individual artifact, such as a function or a test. The IDE tracks the state of each artifact, identifying what work needs to be done next and automatically generating microtasks as necessary. Using microtask programming, developers can onboard onto a new software project and submit a code contribution in less than 15 min. These contributions can then be combined by the environment back into working code.

Behavioral science offers a lens with which to understand programming as the act of translating high-level changes to programs into actions. By observing the moment-to-moment activity of developers, it is possible to describe what makes programming hard by identifying specific questions that are hard to answer. Tools can change how to program by offering new and easier ways to answer these questions. However, both the tasks and contexts where programming occurs are diverse, requiring many approaches. As a developer, it is important to be a knowledgeable consumer of programming tools, being aware of just what challenges are most important in their context and what tools may be available to meet these challenges. 

References

1. H. A. Simon, *The Sciences of the Artificial*, 3rd ed. Cambridge, MA: MIT Press, 1996.
2. T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in *Proc. Workshop Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010, pp. 1–6. doi: 10.1145/1937117.1937125.
3. I. Steinmacher, M. A. Graciotto Silva, M. A. Gerosa, and D. F. Redmiles,



ABOUT THE AUTHOR



THOMAS D. LATOZA is an assistant professor of computer science at George Mason University. His research interests include studying how humans interact with code and designing new ways to build software. Further information about him can be found at <https://cs.gmu.edu/~tlatoza/>. Contact him at tlatoza@gmu.edu.

“A systematic literature review on the barriers faced by newcomers to open source software projects,” *Inf. Softw. Technol.*, vol. 59, pp. 67–85, Mar. 2015. doi: 10.1016/j.infsof.2014.11.001.

4. M. Eisenstadt, “My hairiest bug war stories,” *Commun. ACM*, vol. 40, no. 4, pp. 30–37, Apr. 1997. doi: 10.1145/248448.248456.
5. T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in *Proc. Int. Conf. Software Engineering (ICSE)*, 2010, pp. 185–194. doi: 10.1145/1806799.1806829.
6. J. Lawrence, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, “How programmers debug, revisited: An information foraging theory perspective,” *Trans. Softw. Eng.*, vol. 39, no. 2, pp. 197–215, 2013. doi: 10.1109/TSE.2010.111.
7. H. Lieberman and C. Fry, “Bridging the gulf between code and behavior in programming,” in *Proc. Conf. Human Factors Computing Systems (CHI)*, 1995, pp. 480–486. doi: 10.1145/223904.223969.
8. T. D. LaToza and B. A. Myers, “Visualizing call graphs,” in *Proc. Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 117–124. doi: 10.1109/VLHCC.2011.6070388.
9. 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,” *Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006. doi: 10.1109/TSE.2006.116.
10. M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proc. Symp. Foundations Software Engineering (FSE)*, 2006, pp. 1–11. doi: 10.1145/1181775.1181777.
11. R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger canvas: Industrial experience with the code bubbles paradigm,” in *Proc. Int. Conf. Software Engineering (ICSE)*, 2012, pp. 1064–1073. doi: 10.1109/ICSE.2012.6227113.
12. T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: A study of developer work habits,” in *Proc. Int. Conf. Software Engineering (ICSE)*, 2006, pp. 492–501. doi: 10.1145/1134285.1134355.
13. S. Mehrpour, T. D. LaToza, and R. K. Kindi, “Active documentation: Helping developers follow design decisions,” in *Proc. Visual Languages and Human-Centric Computing (VL/HCC)*, 2019, pp. 2–11. doi: 10.1109/VLHCC.2019.8818816.
14. T. D. LaToza, A. Di Lecce, F. Ricci, W. B. Towne, and A. van der Hoek, “Microtask programming,” *Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1106–1124, 2019. doi: 10.1109/TSE.2018.2823327.