Human-Centered Approach

to Static-Analysis-Driven Developer Tools

**THE FUTURE
DEPENDS ON GOOD
HCI**

AYMAN NADEEM, GITHUB

Correctly harnessing static-analysis techniques in the design of developer tools relies on understanding human intuition. Research advances in the field continue to open up possibilities about the type of information that can be derived about a program's runtime behavior. Distilling this information into a valuable set of insights for developers depends on asking not only what is computationally feasible, but also what is intuitive and useful in the context of solving complex problems.

Several outcomes are made possible through static analysis, including but not limited to the detection of anomalies, security vulnerabilities, dead code, and other possible optimizations. Understanding how such outputs can be translated into an effective interface—one that enhances the experience of software engineering, its ergonomics, and usability—directly impacts the successful application of static-analysis research in future software development.

The tools and processes used to create software can be radically transformed by applying several techniques that fall under the broad umbrella of static program analysis. Considering the net impact software has on our world, improving developer tools can accelerate the pace of human progress, but only if done based on a thoughtful understanding of human psychology.

Given the aggregate economic, social, political, and environmental footprint of software tooling, abstractions, and language design, the opportunity cost of advancing developer-facing static-analysis applications without being grounded in human-centered methods is too high to ignore. Developer tools facilitate human decision-making. Creating effective tools is predicated on having a lucid idea of users and their needs, challenges, and blind spots.

While user behavior is typically not central to advancing static analysis, applications developed without using human-centered methods run the risk of reproducing, if not exacerbating, the limitations of current tools. Increased clarity enhances the technical rigor of code, leading to focus being shifted more toward the ideas underlying the code and the speed with which these ideas are circulated. Reducing the cognitive load imposed by tedious tools or inefficient workflows frees humans to engage in higher-order thinking and reasoning. Therefore, advances in the applications of static analysis should be aimed at supporting human decision-making.

SOFTWARE ENGINEERING HAS GROWING COMPLEXITY
Programming languages have evolved tremendously since their inception in the early 1950s and, alongside their

Such tools often introduce specific micro-optimizations to workflows in exchange for higher overall complexity.

development, an entire discipline dedicated to the creation of software has emerged. Software engineering continues to mature, not just as a practice, but also in terms of its tools, processes, state of the art, professionalism, economics, and culture. At this point, it is apparent just how complex modern software engineering has become. This complexity is the result of many factors and, more broadly, a trend in the industry that favors aggressive release cycles over a thoughtful focus on anticipating and preventing consequences to end users.

In the context of building developer-facing tools, where end users are software engineers, the rate of new tools and technologies entering the ecosystem can be high. Such tools often introduce specific micro-optimizations to workflows in exchange for higher overall complexity. The explosion of services, off-the-shelf APIs, inconsistent standards, and language boundaries that developers must work across daily has made modern development workflows untenable.

Ironically, the growing number of tools and services that attempt to assist developers often contribute to more noise, clutter, moving parts, and complexity. Worse yet, the complexity is not intellectually rich. Substantial coordination between tools is required to perform simple operations. This busywork draws from a finite pool of cognitive resources, and has been shown to result in greater decision fatigue when confronted with problems of greater substance.²⁸

The risk of complexity extends far beyond the impact it has on working conditions for software engineers. Safety-critical systems designed to operate at scale can have

unintended consequences caused by code complexity. A lawsuit filed against Toyota in 2003 exposed how an unmaintainable, untestable code base resulted in failsafe defects that caused loss of throttle control during vehicle operation, resulting in unintended acceleration.²⁷ This outcome arose because of a number of anti-patterns in the code, such as the use of 10,000 global variables, and a single point of failure. System complexity made the code not only difficult to debug, but also impossible to regulate. The National Highway and Traffic Safety Association was unable to investigate the system because of its complexity.

Litigation is evolving, however. More recently a court ruled that a defendant has the right to examine the source code used to produce evidence against the defendant.⁵ As legal ramifications grow, there is greater pressure for companies to be responsible stewards of their code.

Developer tools can be enhanced using static-analysis applications to manage this complexity. Here are some questions to be answered: Does a program have security vulnerabilities? What is the program's worst-case execution time, and can it be further optimized? Are there any common runtime errors, such as division by zero or numeric overflow?

Static analysis provides answers to questions about the behavior of computer programs at runtime, for all possible inputs, without actually running them.

HUMAN-CENTERED DESIGN AND UNDERSTANDING PROGRAM BEHAVIOR

More information about code is useful, but what humans actually need is an interface that matches their intuition.

Consumer-facing applications have long embraced a practice of catering as closely as possible to native human intuition. To build intuitive applications, developers commonly draw on research and best practices in the fields of UX (user experience), usability, UI (user-interface) design, HCI (human-computer interaction), and cognitive science. Although these disciplines have evolved considerably over the past few decades, the dearth of design thinking is noticeable in the landscape of developer-facing tools. This is often a result of deprioritized investment in understanding how humans think, which is arguably the most important prerequisite to leveraging static analysis correctly.

Each of these disciplines informs design differently. *User experience* defines the overall perceptions, emotions, and judgments that result from an individual's interaction with a system.²³ Such perceptions are influenced by many aspects, such as how delightful it is, as well as its usability: how easily a system affords achievement of its intended purpose. *Usability* is defined as a measure of effectiveness, efficiency, and satisfaction.¹³ It is typically evaluated through a set of heuristics, such as Jakob Nielsen's 10 usability heuristics for interface design.²² *User interface* refers to anything the user may interact with, such as the visual elements, keyboard buttons, sounds, and lights. To that end, UX includes, but is not limited to, usability and UI. *Human-computer interaction* is an academic research domain that applies empirical study to better understand UX by incorporating psychology, cognitive sciences, and human factors.¹⁴ Understanding the differences between these fields ensures that relevant techniques from each

discipline are applied appropriately.

The intersection of HCI and static analysis is important because software engineering requires an understanding of program behavior. As the reach of software grows, so do standards for code to be fault-tolerant, reliable, and high quality. What *high-quality software* even means, however, remains a hotly debated topic. Given the breadth of languages, problem domains, programs, and varying degrees of complexity associated with each, the definition of *quality* can vary greatly, and so can the tools and processes thought best to achieve it. A myriad of “best practices” floods the software engineering canon to keep pace with the rapid emergence of new technologies and languages that gain adoption and often contradict one another.

Given the fluidity of *what is right* and the uncertainty of *what can go wrong*, programmers must exercise ongoing vigilance to ensure that immediate priorities are correctly identified, and to determine the best way to translate those priorities into an implementation that balances factors underlying quality, such as performance, readability, size, cost, maintainability, security, and other competing tradeoffs. To do this, programmers must understand the implications of each decision, which requires an understanding of program behavior. Understanding program behavior, however, continues to be difficult for a variety of reasons.

CONTEXT SWITCHING IS AN OBSTACLE TO UNDERSTANDING PROGRAM BEHAVIOR

One obstacle preventing developers from understanding program behavior is the large amount of context switching

required to be productive. Software engineers spend more time building mental models and checking assumptions than actually writing code, but they often have inadequate tooling to support them in this process. The information needed to make a decision can be difficult to identify, let alone access. When the programmer does know exactly what information is relevant, it is often scattered across a variety of disconnected sources that must be consulted and fused together in a way that best maps to the problem.

Context switching is known to increase error rates and slow down productivity, as it can create dangerous interruptions in a coherent flow of thoughts. Task switching has been shown to reduce performance on a primary task, resulting in higher error rates, greater time taken to complete the task, or both. Divided attention has a damaging effect on encoding memory, as well as on retrieval.²⁹ Not only does modern software engineering require significant context switching to achieve anything meaningful, but this information-finding journey is often littered with distractions and fruitless rabbit holes into which even those most seasoned in their craft can fall.

To consider the impact of context switching, refer to table 1, which shows the amalgamation of the digital and physical tools that are used, sometimes simultaneously, to facilitate software development.

At the end of this information-finding journey, even if the programmer is somehow successful at maintaining a coherent train of thought while jumping across several contexts, the information retrieved may itself be incorrect or insufficient. This process repeats until enough information is available to build a reasonable

TABLE 1: **TOOLS USED TO FACILITATE SOFTWARE DEVELOPMENT**

TOOL	EXAMPLE USAGE
Terminal	Separate windows and tabs for REPLs, git processes, database consoles, server processes, error logs, etc.
Integrated Development Environment (IDE)	Several windows and tabs for relevant repositories
Web browser	Multiple tabs featuring developer docs, researching unfamiliar errors on Stackoverflow.com, GitHub.com (Issues, PRs, documentation, projects, blobs, diffs, gists), reading up on language features, frameworks and concepts, etc.
GUI	GitHub Desktop, Azure Data Studio, GraphiQL, etc.
Chat application	Team channels and direct message chats
Paper and pen	To take notes, quickly draw out diagrams, triage errors seen in terminal, create to-do lists
Whiteboard	To explain and share concepts between teammates

understanding of what is going on and what must be done next. Lacking adequate tools, however, the programmer may use this information to make decisions that are improperly evaluated. Complex code paths, unreliable validation systems, flaky tests, and tedious processes leave much to be desired when the programmer just wants to visualize control flow or assess impact across dependencies.

Poor interfaces supporting the creation of a mental model also widen the gap between the source code produced and the labor required to generate that result. Visibility of state and control flow are additionally reduced over problems that span platforms and languages, as each language is supported by its own set of tools. While LSP (Language Server Protocol) is one attempt to provide

a lingua franca for communicating between language compilers, few editors support integrations that unlock LSP's cross-platform capabilities, because of the required overhead.

This problem of scattered or insufficient information is exacerbated in the absence of documentation. Documenting everything is often too onerous, so it becomes an activity secondary to writing code. In this situation, justifications for decisions remain unclear. Newcomers to a codebase amass information less efficiently than they could if learning was better facilitated by existing tools.

Context switching has increased, especially as high-frequency collaboration has become normalized. While asynchronous communication tools are designed to help connect workers, they are often used synchronously for realtime conversations. This has led to workplace cultures where visible activity is often conflated with productivity. A constant stream of notifications not only creates an artificial sense of urgency, but is damaging to the deep focus demanded by programming. Research consistently shows that individuals experienced significantly more stress, anxiety, and irritability when tasks were interrupted.²¹ Studies also show that after being interrupted, workers don't resume their primary task 41 percent of the time.²⁴ On average, trying to resume a primary task after an interruption took 25 minutes longer.¹⁹ Correctness and security guarantees naturally suffer when the cognitive load of context switching is high.

WELL-DESIGNED ABSTRACTIONS IMPROVE HCI

The goal of any well-designed system, not just software, is to ensure that relevant knowledge is available at the appropriate time and place. Throughout its history, computer science has addressed this objective through the design and implementation of abstractions. Abstractions help distribute vast amounts of information into subsystems, where each constituent subsystem has a dedicated purpose. Layers of abstractions fill the chasm between human and machine, with each layer translating conceptual ideas into ones and zeroes.

Well-designed abstractions and languages can help elucidate program behavior. Abstractions provide focus at the layer most relevant to the problem domain, hiding information that is assumed to be unimportant. Unfortunately, identifying what is important is just as controversial as the topics of “quality” and “best practices.” Because of this, the design of abstractions is often more an art than a science. Creating the right boundaries and expressing appropriate information at each layer is critical to providing the most useful information that precisely befits the context. Abstractions can also backfire. They can be too opaque, and in an effort to raise the signal of what is most important, they end up hiding too much.

“The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.”
— Edsger W. Dijkstra, “The Humble Programmer” 1972 ACM Turing Lecture

The tensions underlying abstraction design are similar

to those associated with using static analysis to build developer tools. It is important to surface an appropriate level of detail. While certain contexts demand more granular precision, greater detail is not always conducive to navigating and inspecting vast amounts of code more easily. On one hand, users may be inundated with overwhelming metrics that can paralyze decision-making or guide attention toward less meaningful data that leads to yak shaving and micro-optimizations. On the other hand, there is the risk of concealing important information or introducing automation that yields undesirable outcomes. The tension between detail and generality is unique to each problem and must be negotiated as such.

WHY IS HCI DEPRIORITIZED?

While it is difficult to pinpoint why exactly human-centered methods are given less importance in the design of developer-facing tools, the following speculations explore cultural aspects that may play a role.

Skewing toward compilers and programming languages

Academics focusing on static analysis tend to be compiler researchers and programming language theorists whose interests and expertise slant toward studying language capabilities, expressive algebras, interpreters, and algorithmic design. Because of their specialization, discussion of usability and HCI is often limited. Furthermore, ideas on static analysis have existed in computer science literature for a lot longer than the time researchers have had to explore their limits in modern systems. Therefore, there is a disconnect between

research focusing on what insights can be extracted from code and the UX challenges such insights could mitigate in the wild.

While an increasing amount of active research is concerned with using HCI to help people better understand code, this research is still mostly disconnected from static-analysis applications.¹⁰ Research in task analysis, while capable of characterizing procedural knowledge as a way to measure the efficacy of various interfaces, also remains disconnected from the implementation of static-analysis techniques in developer tools.

Engineers who are less likely to be design savvy

Unlike front-end engineers who may have more expertise in design, or product engineers who are skilled in building a business case justifying their work, engineers concerned with bringing static-analysis tools to market typically come from corners of academia and tend to be hired into roles more narrowly focused on algorithm design, scale, and infrastructure. Given their expertise, they may not prioritize a design process that draws on empirical user research or usability to validate their assumptions, leading to less intuitive implementations.

Because of their specialization, they also prioritize technical correctness over communicating the business value proposition to leaders and stakeholders, communication that could lead to more organizational support and design resources. A functioning body of code in the form of a proof of concept is pointless unless its value to users and the business is clearly communicated. Even if significant engineering investment is poured into

Standards
emerge
organically
when they
are favored
by market forces.

a prototype, skepticism around the usefulness of such work grows, and such projects may become orphaned by a lack of external support necessary to carry them into production. Since most static-analysis techniques are also costly and difficult to scale, the justification for this work dwindles.

Lack of consistency and standards

Modern software engineering offers unique debugging challenges that stem from a lack of consistency and standards. While open-source software was historically blamed for the lack of standardization, proprietary software contributes just as much to inconsistencies. This is because standards emerge organically when they are favored by market forces, not through systematic development of protocols by standards committees.²

In a competitive market, this means there is more incentive for companies to release whatever is most economically viable, even if the resulting developer experience combines a mishmash of tools that are increasingly disconnected. It is also difficult to achieve consistency, given the inherent tension between widespread standardization and domain specificity. The diversity of application areas, along with the pace of new developments in hardware technology, make it difficult and impractical to develop an algorithmic Esperanto, as different abstractions are meant to serve special-purpose problems.¹¹

Software engineering that is all about the code

Software engineering requires a myriad of noncoding

tasks, yet the production of code itself is given more importance than intermediate artifacts required to support a high-quality product. The single-minded obsession with producing code can create a false sense of productivity and mask suboptimal approaches within a local maximum. As a result, auxiliary exercises that support more nuanced thinking and better software development, such as product planning, project management, documentation, or human-centered design, are done poorly, if at all.

When there is little to no upfront system design, the architecture is less likely to have clean interfaces that separate functionality and allow for easier debugging. With less organization that structures code into dedicated components, engineers are required to maintain a mental model of a broader system, which increases the risk of human error. When design and planning tasks are performed, they are often not assigned as much importance as writing code and are given less dedication and time. When low-quality outcomes reflect the reduced priority given to noncoding tasks, it only feeds the industry's confirmation bias that all work outside of code is futile "bikeshedding." It is no surprise that tools that could support noncoding software engineering tasks are stunted, given the stigma involved in engaging with them.

Speed and iteration outweigh quality and strategic thinking

When the most important success metric is shipping speed, quality takes a back seat and shortcuts are rewarded. Deploying to production by any means necessary is valued

more than engaging in any preliminary due diligence that could save teams months, if not years, of time and effort. In his book *High- Output Management*, Andy Grove describes this phenomenon as the “activity over output” problem, where busywork is conflated with productive outcomes.⁹ In their paper “Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice,” Joel Brandt et al. describe how iteration can result in shortsighted choices. When iterating on the code matters more than anything, studies show that documentation, system architecture, and code reuse suffer.⁴

While quick iteration, fast delivery, and tightened development cycles can offer tremendous insight into user behavior and carve an incremental path toward progress, insufficient design thinking applied up front can also impair progress. The myopic focus can result in more bugs, greater technical debt, and limited thought given to internal developer experience. Iteration also imposes limits and traps a system’s evolution within a local maximum. An analogy for this local maximum can be seen in how iteration impacts outcomes in John Conway’s solitaire game of “life.” The initial state determines how patterns evolve across future generations.⁸ While tired slogans such as “Move fast and break things” are being phased out with a wider acceptance of how reductionist phrases can create dangerous oversights, a significant ideological shift is required to value the type of long-term focus and thoughtfulness that will one day trickle into higher priority being given to developer experience.

HCI equated with a superficial coat of paint

HCI and its subfields are incorrectly conflated with surface-level visual details, such as iconography, colors, and fonts. Of course, HCI is far more than that; it is concerned with everything a user encounters and perceives. HCI defines what information is presented; how it is structured; its meaning, usefulness, and timing; and what responses to it are available and when. HCI also encompasses techniques to investigate the effectiveness of an implementation, which may include how learnable it is, how much slower or faster a particular task becomes as a result of a given implementation, and what errors users make and how frequently they encounter them.

Glorification of hacker culture

With several of the industry's trailblazing companies attributing their success to "disruptive" practices that went against conventional bureaucracy, a strong disdain for process and structure emerged. While abandoning some forms of rigidity served companies well, seeing organizational processes as antithetical to creativity created different problems. Startups not only challenged workplace norms, but also demonstrated that traditional education mattered less. Reduced emphasis on credentialing is positive and allows talent to be drawn from a wider pool than was possible before. A disregard for education, however, also widens the gap between companies and academic institutions.

Over time, this has resulted in greater ignorance of computer science research and reduced connection to an existing academic lineage of ideas. Solutions

This results in developer tools being built as a series of micro-adjustments and reactions rather than holistic systems.

to longstanding problems are often half-baked and disconnected from decades of research. This results in developer tools being built as a series of micro-adjustments and reactions rather than holistic systems. When E. F. Codd suggested the model for relational design, his proposal was fortified with solid theoretical underpinnings.⁶ Modern development practices offer few examples demonstrating a principled application of computer science research.

Gatekeeping culture

Several codebases rely on oral tradition, making knowledge transfer difficult. As a result, participation in a given system is more challenging. Poor organizational dynamics may amplify this phenomenon when engineers have an incentive to maintain complexity as a means of job security. When no one else is capable of scrutinizing a given system, it is less likely to be questioned, and those who do possess an intimate familiarity with it can use it to become indispensable. Working in systems that require engineers to hold more information in their heads is seen as a badge of honor rather than a design failure.

The Challenge of UID

The underlying technical challenges affect UID (user interface design). A cross-platform world full of rapidly changing APIs makes it difficult to implement helpful graphical user interfaces. The design of CLIs (command-line interfaces) presents its own challenges, given the constant tension between designing according to legacy systems and standards such as Linux that developers are already

familiar with—even if it means persisting the UI failures of established systems—versus building anew, which reduces cognitive overhead but forces users to learn something new.

Without thoughtful design that balances consistency and usability, there is also a risk of imposing reductionist methods that increase barriers. An example of this includes a proposal for an object-oriented UI, which reverses the idea of human-centered design by applying a computational framework to a graphical representation.¹⁷ Even if the object-oriented framework is based on a skeuomorphism fad that represents information in terms of objects akin to the objects humans interact with in their day-to-day lives, it is removed from human experience by a few degrees. Not all problems are object oriented. This reproduces the problem, which is taking a framework and fitting it around human cognition rather than being intuitive to mental processes.

Deeply complex creatures

Understanding our own neuroses is a difficult undertaking in and of itself, regardless of application domain. To do so for an activity as complex and varied as programming, while acknowledging human psychological diversity, is even harder.

HUMAN-CENTERED DATA

There is an ocean of evidence for the importance of human-centered design. The framing and presentation of complex information impact how quickly it is understood and how effectively problems are solved. These ideas have been long accepted in the field of data visualization,

where the focus is on driving graphical representations of statistics that best map to how humans intuitively think.

A well-known historical example demonstrating the power of data visualization involves the work of Florence Nightingale. During the Crimean War in 1854, Nightingale was a lead nurse in a military field hospital, investigating unusually high death rates. She demonstrated that the sources of death were poor sanitation, bad drainage, and contaminated water. What was most remarkable about her work was not the data collection or statistical analysis but the way she presented her findings. She developed the type of data visualization now known commonly as a pie chart, or polar area diagram, illustrating sources of patient mortality published in the report, *The Royal Commission on India* (1858–1863). Traditional statistical reports would have been less likely to be understood by influential leaders or busy politicians who could act on the information, but because this diagram effectively communicated dense data in an easily consumable format, Nightingale’s work motivated changes in sanitation that resulted in the death rate descending from 42 percent to 2 percent (see figure 1).¹⁵

This example demonstrates that more information is not necessarily always enough. Even if static analysis presented the “facts” of code, it would not improve engineers’ ability to act on said facts. To deliver information about software systems most effectively, the dynamics of interactions between humans and code must be well understood. Doing so requires deepening the understanding of cognition and memory, and identifying how information already available is made opaque through poor design choices.

FIGURE 1: 1858 DIAGRAM BY FLORENCE NIGHTINGALE.

Cognition is defined in the Oxford Online Dictionary as “the mental action or process of acquiring knowledge through thought, experience, and the senses” (<https://en.oxforddictionaries.com/definition/cognition>). Cognitive processes include seeing, listening, reading, writing, remembering, learning, and decision-making. Memory refers to the process used by the human brain to encode, store, and retrieve information and is required for logical reasoning and linguistic relationships to develop. Since there are different theories on how human memory works, such as the dual theory of memory, and several programmers have encountered the limitations of short-

term memory,¹ the question needs to be asked: How might theories of cognition and memory be understood and integrated into the design of interactions that support software development?

Tooling makes tacit knowledge explicit

Learning new programming languages and gaining familiarity with new tools or codebases usually involves negotiating a steep learning curve. The effort required to understand concepts and become productive diminishes with time and practice. Exposure to a problem, and the experience that accompanies practice, are critical in programming-related tasks because of the invisible tacit knowledge that is rarely surfaced through tools.

Tacit knowledge refers to the intuition, muscle memory, and insights needed to support programming tasks that are embedded so deeply in the mind through experience that they rarely surface to the consciousness. This means that seasoned programmers may know *what* works, but not *why*, and are unable to translate this wisdom to others. In contrast, *explicit knowledge* is information that is consciously recognized and translated into guidelines. Codified best practices are an example of explicit knowledge.

Developer tools can do more to facilitate programmer behavior by making tacit knowledge more explicit. One must ask: What are best practices of good software engineering, and how can such behavioral outcomes be designed into the tooling? Usability heuristics are one approach to bridging this gap.¹²

“Computers are good at following instructions, but not at reading your mind.” — Donald Knuth, “The Textbook,” 1984

One way to make tacit knowledge explicit is by surfacing data that is available but hidden from users. This includes information required by the compilation or interpretation steps of most languages. Raising this information using HCI can elucidate a lot of ambiguity sustained by oversights in the design of many tools and languages, such as the design of error messages and front-end CLIs. Error messages often require users to infer a lot more than they should have to. Becoming proficient at diagnosing errors and reacting to them appropriately impacts how productive you are in a codebase, but most of this skill is built over time through a lot of practice.

Rather than requiring developers to sift manually through verbose logs to connect sources of errors to solutions and to learn to pattern-match over time, usability principles can be applied in order to boost the signal-to-noise ratio and improve the debugging experience. There are several examples of active projects in this design space. Projects aimed at improving developer experience include rethinking the terminal user experience in Elm,⁷ Lambdabot being used to provide instructional guidance for Haskell,¹⁸ or Agsy, an external program used to help search for programs in the language Agda that satisfy a specification.³

Choice of programming language also has a tremendous impact on the formation of ideas. Several studies demonstrate a strong relationship between natural spoken languages and thought patterns.²⁰ Psycholinguistics is a

discipline concerned with how language is acquired, used, and comprehended, and the insights from this field apply to spoken and computer languages. Similarly, the expressive power of a given computer programming language can shape what programmers accomplish, and how. Language choice influences thought processes, and determines what information you have and the granularity of your control when operating over this information.¹⁶

While many factors steer debates around the relative superiority of various programming languages, most language communities have active projects aimed at improving user interaction in one way or another. Typed languages are often touted as advantageous for their compile-time guarantees and for being capable of providing more clarity through hints and documentation available within the IDE.

Types by themselves are not enough, however. They go only as far as the user experience that takes advantage of the information they offer. How type information is effectively used and communicated may reduce the need for complex static analyses. Comparing the differences in experience between Haskell's error messages and those provided by Rust or Elm demonstrates a difference in user experience, not type system. There are also examples of tools used to visualize execution information about dynamic languages, such as Python Tutor²⁵ or the Rust Analyzer, which provides insights about Rust code and enables editor capabilities such as autocompletion and jump to definition, and provides semantic information relevant to the state.²⁶

THE FUTURE DEPENDS ON GOOD HCI

For a relatively recent development in human history, software has had a staggering impact. As this impact broadens, the risk of human error intensifies. In this world, complex and opaque systems simply do not scale. A human-centered approach for evolving tools and practices is essential to ensuring that software is scaled safely and securely. Greater insight into human behavior can improve reliability, widen the pool of contributors, and increase the quality of results.

Static analysis can unveil significant information about program behavior, but the goal of deriving this information should not be to accumulate hairsplitting detail. Ensuring the relevance and effectiveness of information and processes necessitates an acute awareness of human behavior and an acknowledgment of the existing challenges in developer workflows.

HCI can help direct static-analysis techniques into developer-facing systems that structure information and embody relationships in representations that closely mirror a programmer's thought. All good tools extend your capabilities, and developer tools can be thought of as no less than a prosthesis for the human mind.

Just as communication has been essential to the evolutionary survival of our species, the survival of great software depends on programming languages that support, rather than inhibit, communicating, reasoning, and abstract thinking. It has long been acknowledged that programming facilitates communication, not just between humans and machines, but among humans. Using a human-

centered framework to leverage static-analysis techniques can smooth the jagged, disconnected fragmentation of existing developer workflows into a more coherent, seamless experience.

References

1. Altmann, E. M. 2001. Near-term memory in programming: a simulation-based analysis. *International Journal of Human-Computer Studies* 54(2), 189–210; <https://www.sciencedirect.com/science/article/abs/pii/S1071581900904075>.
2. Asay, M. 2016. Open source is not to blame for a lack of industry standards. TechRepublic; <https://www.techrepublic.com/article/open-source-is-not-to-blame-for-a-lack-of-industry-standards/>.
3. Automatic Proof Search (Auto). 2017. Agda; <https://agda.readthedocs.io/en/v2.5.3/tools/auto.html>.
4. Brandt, J., Guo, P. J., Lewenstein, J., Klemmer, S. R. 2008. Opportunistic programming: how rapid ideation and prototyping occur in practice. *Proceedings of the 4th International Workshop on End-user Software Engineering*, 1–5; <https://doi.org/10.1145/1370847.1370848>.
5. Claburn, T. 2021. Accused murderer wins right to check source code of DNA testing kit used by police. *The Register* (February 4); https://www.theregister.com/2021/02/04/dna_testing_software/.
6. Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13 (6), 377–387; <https://dl.acm.org/doi/10.1145/362384.362685>.
7. Czaplicki, E. 2015. Compiler errors for humans. Elm; <https://elm-lang.org/news/compiler-errors-for-humans>.

8. Gardner, M. 1970. The fantastic combinations of John Conway's new solitaire game "life." *Scientific American* 223, 120–123; <https://web.stanford.edu/class/sts145/Library/life.pdf>.
9. Grove, A. S. 1995. *High-Output Management*, 2nd edition. Vintage.
10. Guo, P. 1993. Philip Guo - UC San Diego. 2020; <https://pg.ucsd.edu/>.
11. Harel, D. 1987. *Algorithmics: The Spirit of Computing*. Addison-Wesley.
12. Henley, A. Z. 2021. Why is it so hard to see code from 5 minutes ago? *Hacker News*; <https://web.eecs.utk.edu/~azh/blog/yestercode.html>.
13. International Organization for Standardization. 2018. ISO 9241-11:2018: Ergonomics of human-system interaction—Part 11: Usability: definitions and concepts; <https://www.iso.org/standard/63500.html>.
14. Jacko, J. A., ed. 2012. *Human–Computer Interaction Handbook*, 3rd edition. CRC Press.
15. Karimi, H., Masoudi Alavi, N. 2015. Florence Nightingale: the mother of nursing. *Nursing and Midwifery Studies* 4(2), e29475; <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4557413/>.
16. Knuth, D. E. 1984. Literate programming. *The Computer Journal* 27(2), 97–111; <https://academic.oup.com/comjnl/article/27/2/97/343244>.
17. Lamb, G. 2001. Improve your UI design process with object-oriented techniques. Microsoft Developer Network, Visual Basic Developer; [https://web.archive.org/web/20130814153652/http://msdn.microsoft.com/en-us/library/aa227601\(v=vs.60\).aspx](https://web.archive.org/web/20130814153652/http://msdn.microsoft.com/en-us/library/aa227601(v=vs.60).aspx).

18. Lambdabot. 2021. Haskell; <https://wiki.haskell.org/Lambdabot>.
19. Mark, G., Gonzalez, V. M., Harris, J. 2005. No task left behind? Examining the nature of fragmented work. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 321–330; <https://dl.acm.org/doi/abs/10.1145/1054972.1055017>.
20. Mou, B. 1999. The structure of the Chinese language and ontological insights: a collective-noun hypothesis. *Philosophy East and West* 49(1), 45–62; <https://www.jstor.org/stable/1400116>.
21. Naveh-Benjamin, M., Guez, J. [2000]. Effects of divided attention on encoding and retrieval processes: assessment of attentional costs and a componential analysis. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 26(6), 1461–82; <https://psycnet.apa.org/record/2000-12254-009>.
22. Nielsen, J. 1994. 10 usability heuristics for user interface design. Nielsen Norman Group; <https://www.nngroup.com/articles/ten-usability-heuristics/>.
23. Norman, D. 2013. *The Design of Everyday Things*. New York, NY: Basic Books.
24. O’Conaill, B., Frohlich, D. 1995. Timespace in the workplace: dealing with Interruptions. *Conference Companion on Human Factors in Computing Systems*, 262–263; <https://doi.org/10.1145/223355.223665>.
25. Python Tutor. (n.d.). Visualize Code Execution; <http://pythontutor.com/>.
26. Rust-analyzer. (n.d.); <https://rust-analyzer.github.io/>.
27. Safety Research & Strategies Inc. 2013. Toyota unintended acceleration and the big bowl of “spaghetti”

code (November 7); <https://www.safetyresearch.net/toyota-unintended-acceleration-and-the-big-bowl-of-spaghetti-code/>.

28. Sierra, K. 2013. Your app makes me fat. Serious Pony (July 24); <http://seriouspony.com/blog/2013/7/24/your-app-makes-me-fat>.

29. Weinschenk, S. 2012. The true cost of multi-tasking. *Psychology Today* (September 18); <https://www.psychologytoday.com/us/blog/brain-wise/201209/the-true-cost-multi-tasking>.

Ayman Nadeem is a senior software engineer at GitHub, building security infrastructure for abuse mitigation at scale. She loves exploring how languages, both human and programming, are interpreted by minds and computers alike.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.