

Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools

Brad A. Myers, Carnegie Mellon University

Amy J. Ko, University of Washington

Thomas D. LaToza, George Mason University

YoungSeok Yoon, Google

Human-centered methods can help researchers better understand and meet programmers' needs. Because programming is a human activity, many of these methods can be used without change. However, some programmer needs require new methods, which can also be applied to domains other than software engineering.

Two key (and surprisingly controversial) observations behind the research of Carnegie Mellon University's Natural Programming Group (www.natprog.org) are that developers are humans and that software development languages and

environments are the user interfaces through which developers interact with computers. This means that researchers should use conventional human-computer interaction (HCI) methods to investigate how developers use programming languages and tools. There are many perspectives on how to use HCI methods to improve software development and software products, including user-centered design, participatory design, agile methods, and the wide range of requirements-engineering methodologies.



See www.computer.org/computer-multimedia for multimedia content related to this article.

Here, we focus on the HCI methods (see Table 1) that our group has used over the last 30 years to improve the usefulness and, specifically, usability of tools for individual developers. Many others have performed significant and important research on other aspects, including how groups of developers can communicate and be effectively managed.¹⁻³

Table 1 lists the development activities in which we have used each HCI method, highlighting the breadth of activities that the methods can support. These methods can also be used in other ways to support additional activities. This article presents small case studies illustrating how we have used the methods and discusses each method's strengths and weaknesses. It contributes to understanding the broad range of human-centered approaches that can help improve programmers' tools.

APPLYING HCI METHODS

HCI is a well-established field that studies the interaction between people and technology. Many people might think HCI methods are primarily for usability testing (evaluating how well people can use a particular technology).⁴ However, HCI has developed or adapted a variety of human-centered methods for answering many kinds of questions about user interfaces. In fact, HCI masters' programs, such as those at Carnegie Mellon University and the University of Washington, teach over 30 methods for discovering information about people and their interactions with technology.

In this article, we use "HCI" and "human-centered" interchangeably to describe methods involving people, even methods that might be from other fields or might be widely used

elsewhere. We are not suggesting that these methods are exclusively under the purview of HCI or even that HCI has a greater claim on these methods than other fields, such as psychology or anthropology.

Another possible misconception is that HCI focuses on only the surface presentation (such as colors, fonts, icons, and screen layouts). HCI is concerned with everything the user encounters, including functionality, usefulness, information structure and meaning, content, presentation, layout, navigation, speed of response, emotional impact, context (the social environment in which a tool is used), and documentation and help. HCI also applies a variety of measures, including learnability (how well the tool or concept can be learned), productivity and effectiveness (how quickly tasks can be performed), and errors (how often people make mistakes during use). Many of the methods can provide data about these measures that can make decision-making more objective.⁴ Generally, for any question a tool maker has about a new tool or tool feature, an HCI method can probably help provide answers.

Our group has studied different kinds of developers—professional programmers (who generally have a computer science degree or an equivalent), novice programmers (who are learning how to be professional programmers), and end-user programmers (who program to automate a task rather than to ship code for others to use).⁵ In this article, we use "programmer" and "developer" interchangeably to apply to all people who work with code (among their other activities), including people who do the programming, software engineers, system architects, and testers. To avoid confusion, we use

"experimenter" to refer to people who use HCI methods to study programmers or to design or evaluate new tools or processes for programmers.

HCI methods can be applied to everything the developer encounters, including

- › tools such as editors and integrated development environments;
- › reusable components such as APIs, libraries, and software development kits;
- › documentation for all the tools, APIs, processes, and context used to organize development; and
- › the design of the programming languages themselves.

The dangers of not using these human-centered practices have been well-documented: languages, documentation, and tools that are confusing, difficult to learn, or, worse yet, useless. We have identified two dimensions of usefulness: that an important problem is addressed and that the problem is actually solved. For a problem to be important, it must happen frequently or have a large impact and be difficult for the developer to solve (which might be measured by the effort or time that solving it takes). The frequency, impact, and difficulty can all be measured with the HCI methods we discuss in this article. Surveys have shown that developers complain that researchers sometimes address unimportant problems.⁶ Researchers can avoid this by using human-centered data to help decide which problems to research.

Furthermore, a research result might not actually solve the problem. Sometimes a new tool or process

TABLE 1. Human–computer interaction (HCI) methods our group has used.

Method	Tool development activities supported	Key benefits	Challenges and limitations
Contextual inquiry	Requirements and problem analysis	<ul style="list-style-type: none"> » Experimenters gain insight into day-to-day activities and challenges. » Experimenters gain high-quality data on the developer’s intent. 	<ul style="list-style-type: none"> » Contextual inquiry is time consuming. » Recruiting professionals might be a challenge.
Exploratory lab studies	Requirements and problem analysis	<ul style="list-style-type: none"> » Focusing on the activity of interest is easier. » Experimenters can compare participants doing the same tasks. » Experimenters gain data on the developer’s intent. 	The experimental setting might differ from the real-world context.
Surveys	<ul style="list-style-type: none"> » Requirements and problem analysis » Evaluation and testing 	<ul style="list-style-type: none"> » Surveys provide quantitative data. » There are many participants. » Surveys are (relatively) fast. 	The data is self-reported and is subject to bias and lack of participant awareness.
Data mining (including corpus studies and log analysis)	<ul style="list-style-type: none"> » Requirements and problem analysis » Evaluation and testing 	<ul style="list-style-type: none"> » Data mining provides large quantities of data. » Experimenters can see patterns that emerge only with large corporuses. 	<ul style="list-style-type: none"> » Inferring or reconstructing the developer’s intent is difficult. » Data mining requires careful filtering.
Natural-programming elicitation	<ul style="list-style-type: none"> » Requirements and problem analysis » Design 	Experimenters gain insight into developer expectations.	The experimental setting might differ from the real-world context.
Rapid prototyping	Design	Experimenters can gather feedback at low cost before committing to high-cost development.	Rapid prototyping has lower fidelity than the final tool, limiting what problems might be revealed.
Heuristic evaluations	<ul style="list-style-type: none"> » Requirements and problem analysis » Design » Evaluation and testing 	<ul style="list-style-type: none"> » Evaluations are fast. » They do not require participants. 	Evaluations reveal only some types of usability issues.
Cognitive walkthroughs	<ul style="list-style-type: none"> » Design » Evaluation and testing 	<ul style="list-style-type: none"> » Walkthroughs are fast. » They do not require participants. 	Walkthroughs reveal only some types of usability issues.
Think-aloud usability evaluations	<ul style="list-style-type: none"> » Requirements and problem analysis » Design » Evaluation and testing 	Evaluations reveal usability problems and the developer’s intent.	<ul style="list-style-type: none"> » The experimental setting might differ from the real-world context. » Evaluations require appropriate participants. » Task design is difficult.
A/B testing	Evaluation and testing	<ul style="list-style-type: none"> » Testing provides direct evidence that a new tool or technique benefits developers. » It provides objective numbers. 	<ul style="list-style-type: none"> » The experimental setting might differ from the real-world context. » Testing requires appropriate participants. » Task design is difficult.

might just change the problem. For example, visualizations often just change the developers’ search task from looking through the code to looking through a complicated graphical presentation, which might not be faster. Or, developers might find a new tool’s user interface too difficult to use, even when the tool’s functionality is inherently useful.⁷ Again, researchers can use HCI methods to

measure the results of using the tool or process, to see whether developers perform better.

REQUIREMENTS AND PROBLEM ANALYSIS

The first activity of a project likely involves getting a better understanding of the users’ problems and needs, as a way to solidify the requirements. This activity is obviously relevant for

commercial systems; we have found it useful and often necessary for research projects, too. Four methods that we have often used in this activity are *contextual inquiry* (CI), *exploratory lab studies*, *surveys*, and *data mining*.

Contextual inquiry

In a CI, the experimenter observes developers performing their usual work where it actually happens.⁸ For

example, in one of our projects, we wondered what key barriers developers face when fixing defects.⁹ So, we asked developers at Microsoft to work on their own tasks while we watched and took notes about the issues that arose. A key problem for 90 percent of the longest tasks was understanding the control flow through code in widely separated methods, which the existing tools did not adequately reveal.

CIs are a good way to gather qualitative data and insights into developers' real issues. However, they do not provide quantitative statistics, owing to the small sample size. Also, a CI can be time consuming, especially if it is difficult to recruit representative developers to observe.

Exploratory lab studies

In these studies, the experimenter assigns specific tasks to developers and observes what happens. Unlike usability analysis and A/B testing (which we discuss later), the participants normally use conventional tools (rather than a new design). The key difference from a CI is that here the participants perform tasks provided by the experimenter instead of their own tasks, so there is less realism. However, the experimenter can see whether the participants use different approaches to the same task.

For example, we collected a detailed dataset at the keystroke level of experienced developers performing maintenance tasks in Java.¹⁰ We discovered that the developers spent about one-third of their time navigating around the code base, often using manual scrolling. This highlights an important advantage of these observational techniques. For example, when we asked the participants about barriers when performing these tasks, no one

mentioned scrolling because it did not rise to the level of salience. However, it became obvious to us that this was a barrier when we analyzed the logs of what the developers actually did. Knowing about such problems is the first step to inventing solutions.

Surveys

In other cases, asking developers questions can be helpful. We often use surveys to collect numerical data (hopefully from a large number of people) about how pervasive our observations from CIs and exploratory lab studies are.

For example, after we observed in our CIs that interprocedural control flow was important, we wanted to find out how often developers have questions about control flow and how hard those questions are to answer. So, we performed a survey. The developers reported asking such questions on average about nine times a day, and most felt that at least one such question was hard to answer.⁹ By adding open-ended questions to this survey, we were able to collect many other hard-to-answer questions about code that capture real problems developers face in their everyday work.

Data mining

Data mining is probably more closely associated with fields other than HCI. It might include corpus studies or log analyses, depending on the kind of data being analyzed. Often, this can be a good way to investigate how pervasive a pattern is.

For example, we wondered how often developers backtrack (return code to a previous state) while editing code, possibly by using the editor's undo command. So, we analyzed 1,460 hours of fine-grained code-editing logs

from 21 developers. We detected 15,095 backtracking instances, for an average rate of 10.3 per hour. This motivated us to create Azurite, an Eclipse plug-in that provides more flexible selective undo, in which developers can undo past edits without necessarily undoing more recent ones.¹¹

In another study, we performed a linguistic analysis of the titles of nearly 200,000 bug reports from the repositories for five open source projects to see how developers describe software problems. Our analysis suggested designs for more structured bug report forms that better match people's phrasing of problems, while enabling tools to more easily reason about reports. However, in data mining, the large amount of data often makes manual inspection impossible, which can make it more difficult to validate results or understand the developer's intent.

DESIGN

Once experimenters have investigated a problem, they might want to design a tool or process to mitigate any discovered issues. HCI methods can help answer questions about what the design should be so that the result will be attractive to and effective for developers. The methods we have used for this include *natural-programming elicitation* and conventional *rapid prototyping*.

Natural-programming elicitation

Because programming requires developers to map their intent into a form the computer can execute, researchers can make the process easier by bringing the form of expression closer to how the developer thinks. To better understand how developers think about their problems, we created

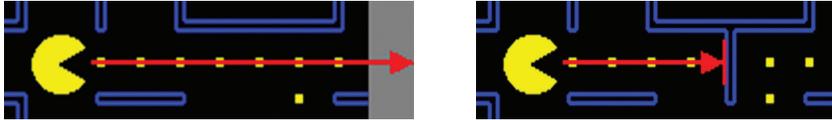


FIGURE 1. Using pictures to prompt novice programmers or nonprogrammers to express conditional situations. We asked participants how they would instruct the computer to have Pac-Man behave according to the pictures. Fifty-four percent of the participants used an if-then style (“If Pac-Man hits a wall, then he stops”), about 18 percent of the participants used a constraint-based style (“Pac-Man cannot go through a wall”), and the remaining participants used other styles.

natural-programming elicitation.¹² This involves getting developers to provide their own descriptions or designs, which then inform the design of tools and languages.

For example, our early research examined how novice developers expressed various programming concepts.¹² In one study, we showed them pictures like those in Figure 1 and asked how they would instruct the computer to achieve that behavior. We found they often used an event- or rule-based structure, in which actions occurred in response to events. For example, for Figure 1, 54 percent of the participants used an if-then style, such as “If Pac-Man hits a wall, then he stops.” In contrast, about 18 percent of the participants used a constraint-based style, such as “Pac-Man cannot go through a wall,” with the remaining participants using other styles.

We used the result of this and other studies to guide the design of a programming language for children.¹² For example, on the basis of the results for Figure 1, the system uses the if-then style for event handling.

We have also used this method to understand how developers want to access functionality through APIs.¹³ We gave developers a blank screen or piece of paper and asked them to design

the API for a particular functionality. This helped us understand the most natural vocabulary and organization of classes and methods, and provided recommendations for future API designs.

Rapid prototyping

A key HCI guideline is to rapidly iterate on the design, using prototypes.⁴ Typically, the first step employs paper prototypes, which are quickly created using drawing tools or even just pen and paper. For many of our tools, we use this rapid prototyping to test whether our ideas will likely work.

For example, when trying to help developers understand the interprocedural control flow of code, we used OmniGraffle to draw mockups of a possible new visualization and printed them on paper (see Figure 2a).⁹ We then asked the developers to pretend to perform tasks with them. We discovered that the initial visualization concepts were too complex to understand yet lacked information important to the developers. For example, a key requirement was to preserve the order in which methods are invoked, which was not shown (and is not shown by other static visualizations of call graphs, either). In the final visualization (see Figure 2b), the lines coming out of a method show the order of invocation.

EVALUATION AND TESTING

Developers are likely familiar with using HCI methods to evaluate the usability of the products they make. However, they might not have tried using those methods to measure their development tools’ usability. We routinely evaluate the usability and performance of our tools for developers, using *expert analyses*, *think-aloud usability evaluations*, *A/B testing*, and *data mining using log analyses*.

Expert analyses

In expert analyses, people who are experienced with usability methods perform the analysis by inspection. For example, *heuristic evaluation* employs 10 guidelines to evaluate an interface.⁴ We used this method in our collaboration with SAP. We found that the really long function names violated the principle of error prevention because the names could be easily confused with each other.¹³

Another expert-analysis method is a *cognitive walkthrough*.¹⁴ It involves carefully going through tasks using the interface and noting where users will need new knowledge to be able to take the next step.

Using both of these methods, we helped SAP iteratively improve a developer tool for Visual Studio. Because SAP used agile development,³ it could address our recommendations immediately.

Think-aloud usability evaluations

Another set of methods is empirical and involves testing the tools with the target users. For development tools, this requires having the developers perform realistic tasks. The first result of these evaluations is an understanding of what participants actually do, to see how they work with the tool.

In addition, we recommend using a think-aloud study, in which the participants continuously articulate their goals, confusion, and other thoughts. This provides the experimenter with rich data about why users perform the way they do, so problems can be found and fixed. As with other usability tests, the principle is that if one participant has a problem, others will likely have it too, so it should be fixed if possible.

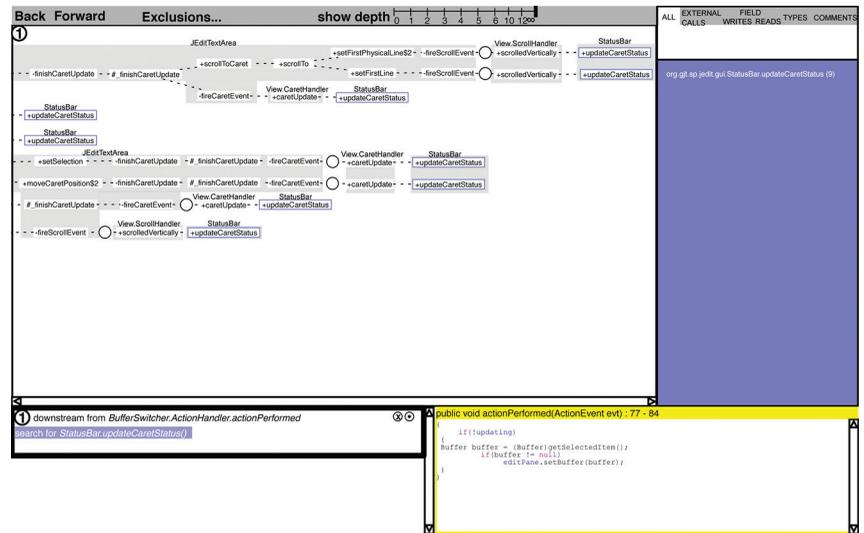
Research shows that a few representative users can find a great percentage of the problems.⁴ In our research, when we have evidence of usefulness from early needs analysis through CI and surveys, it is often sufficient to show usability of tools through think-alouds with five or six people. However, the evaluations should not involve participants who are associated with the tool, because they will know too much about how the tool should work.

A/B testing

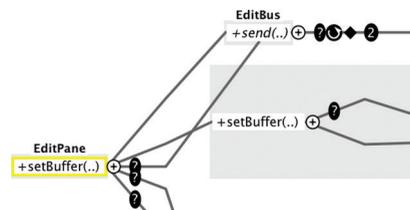
Unlike expert analyses and think-aloud usability evaluations, which are informal, A/B testing uses formal, statistically valid experiments. This is the key way to demonstrate that one tool is better than another regarding some measure.

For example, we showed that developers using our Whyline debugging tool were more than three times as successful, in one-half the time, as the control group.¹⁵ Similarly, we tested Eclipse with Azurite against regular Eclipse, and developers using Azurite were twice as fast.¹¹

Such formal measures are important for research papers. The resulting numbers might also help convince developers and managers to try new tools or change work habits, because they might find numbers more persuasive. However, these experiments



(a)



(b)

FIGURE 2. An example of rapid prototyping. (a) A paper prototype of a visualization drawn with OmniGraffle. (b) The final version of the tool, called Reacher. The prototype revealed that the order of method calls was crucial to visualize. The method `EditPane.setBuffer(..)` makes five method calls (the five lines exiting `setBuffer` shown in order from top to bottom, with the first and third being calls to `EditBus.send(..)`). Lines with “?” icons show calls that are conditional (and thus might or might not happen at runtime). The circular arrow indicates calls in loops, diamonds indicate overloaded methods, and numbers indicate that multiple calls have been collapsed.

can be difficult to design correctly and require careful attention to many possibly confounding factors.¹⁶ In particular, it is challenging to design tasks that are sufficiently realistic yet doable in an appropriate time frame for an experiment (an hour or two).

Data mining through log analyses

A tool’s evaluation does not have to stop when that tool leaves the

lab. Adding detailed logging to a tool can help tool designers better understand how developers use that tool.

For example, we used our Fluorite logger to investigate how developers used Azurite.¹¹ We found that developers often selectively undid a block of code, such as a whole method, restoring it to how it used to work and leaving the other code as is.

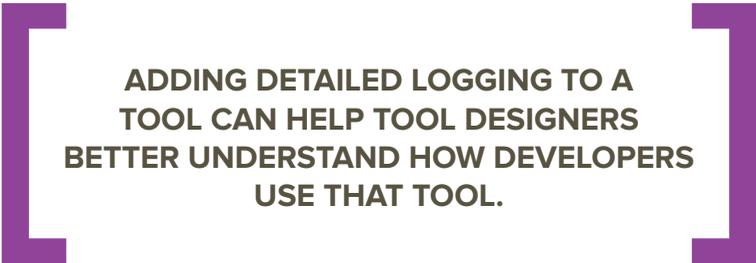
Similarly, we created Apatite, a Web-based documentation tool that presents Java methods by association. When the user selects a method or class, Apatite shows all the other methods and classes that are often used with it. In our field study, because Apatite is a Web tool, we could easily log every user action by using conventional Web analytics tools. The logs

showed that few users found this association feature useful. Instead, they used Apatite mainly to quickly find a method or class by name because it autocompletes any name entered. Although we were disappointed that Apatite did not prove more useful, the study revealed valuable insights into what developers really want.

designers. In addition, usability issues can be a barrier to uptake and use, even when the functionality is useful, as we mentioned before. To help resolve these issues, experimenters can engage people skilled in graphic and interaction design. For example, graphic designers can help with colors, icons, selection of controls, and layout. This might be

especially important for visualization tools. We have consistently found that improving our tools' presentation, interaction flow, and layout has improved their usability, popularity, and evaluations.

tion on providing easy navigation to the relevant code for each step. Similarly, the Reacher tool (see Figure 2b) shows only a summary of method invocations. Users click on the lines and icons to see the corresponding code snippets to get an understanding of the associated code.



ADDING DETAILED LOGGING TO A TOOL CAN HELP TOOL DESIGNERS BETTER UNDERSTAND HOW DEVELOPERS USE THAT TOOL.

showed that few users found this association feature useful. Instead, they used Apatite mainly to quickly find a method or class by name because it autocompletes any name entered. Although we were disappointed that Apatite did not prove more useful, the study revealed valuable insights into what developers really want.

DESIGN AND DEVELOPMENT RECOMMENDATIONS

Besides the HCI methods we just discussed, human-centered techniques can improve tools for developers in other ways. We have found the following observations useful.

Good aesthetic and interaction design

During design, we have found it useful to apply good aesthetic and interaction design principles. Even when building tools for people like themselves, software engineers (and researchers) are not necessarily the best interaction

designers. In addition, usability issues can be a barrier to uptake and use, even when the functionality is useful, as we mentioned before. To help resolve these issues, experimenters can engage people skilled in graphic and interaction design. For example, graphic designers can help with colors, icons, selection of controls, and layout. This might be

The primacy of viewing code

We have found that a key use for visualizations is to guide developers to the right code to look at, rather than the visualizations being an aid to understanding on their own. For example, Whyline aids debugging by visualizing why a particular output did or did not happen, through dynamic and static analysis of full execution traces.¹⁵ Whyline's first version targeted the Alice educational programming language and provided an elaborate visualization of the control flow and dataflow. However, when we targeted the tool at Java, the visualization became unwieldy and not understandable. Therefore, we focused the visualiza-

The importance of search

It is no surprise to any developer how useful it is to search the Web for answers to a variety of questions, from how to use APIs to what error messages mean. Our Mica project augmented Google search to make the returned results even more useful by highlighting which API methods and code examples the search returned.

We have found search useful in many other ways. For example, Reacher lets users search forward and backward along the feasible control flow (rather than searching through all the code), to specifically answer questions such as, "Are there any paths by which this method can be reached without first calling the initialization method?"

As we mentioned before, Azurite lets developers search code backward in time, to answer questions such as, "When was this variable renamed?" Generally, we have found that developers have very specific questions, so providing a means to answer their questions directly through tools can substantially improve their productivity.

Augmenting what developers are already doing

Experimenters should augment what developers are already doing, so that the new features are where they are looking anyway. This significantly increases familiarity and therefore usability and reduces the new tools' overhead.

For example, many developers explore an API by using the auto-complete pop-up menus in the code editor. Although this feature aims to reduce typing, developers commonly use it to see the list of available methods and then guess which one might be useful. However, we found that the Eclipse autocomplete is not always useful when developers are trying to create a new instance of a class.¹³ (For example, invoking the autocomplete menu after typing “=” does not provide any useful completions.) This is especially true when the instance should be created using a factory method or other indirect means.

Therefore, we built Eclipse plug-ins that incorporate such entries directly into the autocomplete menus. The Calcite plug-in makes the autocomplete menu that appears after the developer types “=” more useful by adding to it the most common ways to create instances.¹³ It does this on the basis of analysis of code found through Web crawling. The Dacite plug-in adds autocomplete entries based on API annotations for various patterns such as factories and helper methods. The Graphite plug-in provides mini-editors, which can be discovered through autocomplete, for defining colors and interactively authoring regular expressions. It automatically generates the corresponding code to create Java objects.

Our Jadeite tool augments the popular JavaDoc documentation with entries for methods that developers expect to be in a certain class but that are actually defined elsewhere. For example, we found that developers expect the email send method to be in the Message class, whereas it is actually in the Transport class. So, Jadeite adds a placeholder entry in the JavaDoc for

the Message class under send, telling developers where to look.¹³

Iterative design

To develop programming tools, we recommend the same process that has always been recommended for creating more usable applications for consumers: iterative design throughout all phases, using human-centered methods.^{4,17} Large companies such as Microsoft and Google already embed user interface specialists into their teams that create developer tools (such as in Microsoft’s Visual Studio group). However, even small teams can learn to use at least some of these methods.⁴

Many other HCI methods and observations are available that can answer additional questions tool developers might have. Hopefully, tool creators can use these methods to help increase the likelihood that future tools will help developers be more successful, effective, and efficient. 

ACKNOWLEDGMENTS

This article grew out of nearly 30 years’ work by the Natural Programming Group involving more than 50 students, staff, and postdocs besides the authors. We thank them all for their contributions. The research summarized here has been funded by SAP, Adobe, IBM, Microsoft, and multiple US National Science Foundation grants including CNS-1423054, IIS-1314356, IIS-1116724, IIS-0329090, CCF-0811610, IIS-0757511, and CCR-0324770. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect those of any of the sponsors.

REFERENCES

1. J. Tomayko and O. Hazzan, *Human Aspects of Software Engineering*, Charles River Media, 2004.
2. A. Seffah, J. Gulliksen, and M.C. Desmarais, *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*, Springer, 2005.
3. D. Salah, R.F. Paige, and P. Cairns, “A Systematic Literature Review for Agile Development Processes and User Centred Design Integration,” *Proc. 18th Int’l Conf. Evaluation and Assessment in Software Eng. (EASE 14)*, 2014, article 5.
4. J. Nielsen, *Usability Engineering*, Academic Press, 1993.
5. A.J. Ko et al., “The State of the Art in End-User Software Engineering,” *ACM Computing Surveys*, vol. 43, no. 3, 2011, article 21.
6. D. Lo, N. Nagappan, and T. Zimmermann, “How Practitioners Perceive the Relevance of Software Engineering Research,” *Proc. 10th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE 15)*, 2015, pp. 415–425.
7. E. Murphy-Hill and A.P. Black, “Refactoring Tools: Fitness for Purpose,” *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.
8. H. Beyer and K. Holtzblatt, *Contextual Design: Defining Custom-Centered Systems*, Morgan Kaufmann, 1998.
9. T.D. LaToza and B. Myers, “Developers Ask Reachability Questions,” *Proc. 32nd Int’l Conf. Software Eng. (ICSE 10)*, 2010, pp. 185–194.
10. A.J. Ko et al., “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks,” *IEEE Trans. Software Eng.*, vol. 33, no. 12, 2006, pp. 971–987.



IEEE TRANSACTIONS ON
BIG DATA

▶ **SUBSCRIBE
AND SUBMIT**

For more information on paper submission, featured articles, call-for-papers, and subscription links visit:

www.computer.org/tbd



TBD is financially cosponsored by IEEE Computer Society, IEEE Communications Society, IEEE Computational Intelligence Society, IEEE Sensors Council, IEEE Consumer Electronics Society, IEEE Signal Processing Society, IEEE Systems, Man & Cybernetics Society, IEEE Systems Council, IEEE Vehicular Technology Society

TBD is technically cosponsored by IEEE Control Systems Society, IEEE Photonics Society, IEEE Engineering in Medicine & Biology Society, IEEE Power & Energy Society, and IEEE Biometrics Council



ABOUT THE AUTHORS

BRAD A. MYERS is a professor in the Human–Computer Interaction Institute in Carnegie Mellon University’s School of Computer Science. His research interests include programming environments, programming-language design, and user interfaces. Myers received a PhD in in computer science from the University of Toronto. He is a Fellow of IEEE and ACM and belongs to the IEEE Computer Society and the ACM Special Interest Group on Computer–Human Interaction. Contact him at bam@cs.cmu.edu.

AMY J. KO is an associate professor in the University of Washington’s Information School and an adjunct associate professor in the university’s Computer Science & Engineering department. Her research focuses on interactions between people and code, spanning the areas of human–computer interaction, computing education, and software engineering. Ko received a PhD from Carnegie Mellon University’s Human–Computer Interaction Institute. Contact her at ajko@uw.edu.

THOMAS D. LATOZA is an assistant professor of computer science in George Mason University’s Volgenau School of Engineering. His research focuses on software engineering environments, spanning empirical and design work related to debugging, software design, collaboration, and crowdsourcing software engineering. LaToza received a PhD from Carnegie Mellon University’s Institute for Software Research. Contact him at tlatoz@gmu.edu.

YOUNGSEOK YOON is a software engineer at Google. His research aims to improve developer productivity by providing better development tools and environments for code editing, debugging, testing, and build automation. Yoon received a PhD in software engineering from Carnegie Mellon University’s Institute for Software Research. Contact him at youngseokyoona@google.com.

11. Y. Yoon and B.A. Myers, “Supporting Selective Undo in a Code Editor,” *Proc. 37th Int’l Conf. Software Eng. (ICSE 15)*, vol. 1, 2015, pp. 223–233.
12. B.A. Myers, J.F. Pane, and A. Ko, “Natural Programming Languages and Environments,” *Comm. ACM*, vol. 47, no. 9, 2004, pp. 47–52.
13. B.A. Myers and J. Stylos, “Improving API Usability,” *Comm. ACM*, vol. 59, no. 6, 2016, pp. 62–69.
14. C. Lewis et al., “Testing a Walk-through Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces,” *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI 90)*, 1990, pp. 235–242.
15. A.J. Ko and B.A. Myers, “Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior,” *Proc. 30th Int’l Conf. Software Eng. (ICSE 08)*, 2008, pp. 301–310.
16. A.J. Ko, T.D. LaToza, and M.M. Burnett, “A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants,” *Empirical Software Eng.*, vol. 20, no. 1, 2015, pp. 110–141.
17. J. Goodman-Deane et al., “User Involvement and User Data: A Framework to Help Designers to Select Appropriate Methods,” *Designing Inclusive Futures*, P. Langdon, J. Clarkson, and P. Robinson, eds., Springer, 2008, pp. 23–34.

Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>