**Accepted for publication in: IEEE Computer,
Special issue on UI Design, vol. 49, issue 7, July, 2016.**

Programmers are Users Too: Human Centered Methods for Improving Tools for Programming

Brad A. Myers (IEEE Fellow)
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
(412) 268-5150
FAX: (412) 268-1266
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Andrew J. Ko
The Information School
University of Washington
Box 352840
Seattle, WA 98195
206-419-3990
ajko@uw.edu
http://faculty.uw.edu/ajko

Thomas D. LaToza
Department of Computer Science
Volgenau School of Engineering
George Mason University
4400 University Drive, MS 4A5
Fairfax, VA 22030
(703) 993-1677
tlatoza@gmu.edu
https://cs.gmu.edu/~tlatoza/

YoungSeok Yoon
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
(650) 253-8630
youngseokyoon@google.com
http://research.yyoon.net/

Abstract

Software development continues to be one of the most difficult human tasks. Over the past thirty years of research, we have found that the methods and approaches from the field of Human-Computer Interaction (HCI) are effective in better understanding and meeting the needs of programmers of all types: novice, professional, and end-user programmers (EUPs). We have applied these methods across all activities of software development: requirements and problem analysis, design, development, testing, and deployment. Since programming is a human process, we have found that many of these HCI methods can be used without change to answer many useful questions, but for other questions, we have needed to create new human-centered methods, which can also be usefully applied to systems in other domains in addition to software development. This article summarizes our use of new and existing HCI methods across nearly 30 years of research on improving software development.

Keywords

D.2 Software Engineering; D.2.0.b/H.1.2.d Software psychology; F.3.3 Studies of Program Constructs; O.2.8 Evaluation studies

Introduction

A key (and surprisingly controversial) observation behind the work of our "Natural Programming Group" at CMU (www.natprog.org) is that developers are humans, and software development languages and environments are the user interfaces through which developers interact with computers. This means that conventional Human-Computer Interaction (HCI) methods can and should be used to investigate developers using programming languages and tools. There are many perspectives on how HCI methods can be used to improve software development and software products, including user-centered design, participatory design, Agile methods, and the wide range of requirements engineering methodologies. In this paper we focus on methods that our group has used over the last 30 years to improve the usefulness and usability specifically of developer tools by individual developers. Many others have performed significant and important research on other aspects, including how groups of software developers can communicate and be effectively managed.[6-8]

HCI is a well-established field that studies the interaction between people and technology. While many may think HCI methods are primarily for *usability testing* (evaluating how well people can use a particular technology),[4] HCI has developed or adapted a wide variety of human centered methods for answering many kinds of questions about user interfaces. For example, the *contextual inquiry* method[1] is useful for understanding people's real needs and requirements, and *A/B testing* with formal statistical analyses are the "gold standard" for experimentally comparing whether people can do better with design *A* or design *B*. In fact, Masters of HCI programs, such as those at Carnegie Mellon University and the University of Washington, teach over 30 methods that can be used to discover a wide variety of different kinds of information about people and their interactions with technology. In this paper, we are using "HCI" and "human-centered" interchangeably as descriptors for methods that involve people, even for methods that may have come from other fields or be widely used elsewhere. In particular, we are *not* trying to suggest that these methods are exclusively the purview of HCI or even that HCI has more "claim" on these methods compared to other fields, such as psychology or anthropology.

Another possible misconception is that HCI focuses only on the surface presentation (such as the colors, fonts, icons and layout of screens). In fact, HCI is concerned with *everything* the user encounters, including such aspects as 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), documentation and help, etc. HCI also applies a wide variety of *measures*, including: learnability (how well the tool or concept can be learned), productivity and effectiveness (how quickly can tasks be performed), errors (how often do people make mistakes during use), etc. Many of the methods can provide *data* about these measures which can make decision-making more objective.[4] In general, for any question that a tool maker has about a new tool or tool feature, there is probably an HCI method that can help provide answers.

Our group has studied different *kinds* of developers – professional programmers (who generally have a degree in computer science or equivalent), novice programmers (who are learning how to be professional programmers), and also end-user programmers (EUPs) who program to automate a task, rather than to ship code for others to use.[9] Note that in this paper, we are using the term "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, testers, etc. To avoid confusion, we will use the term "experimenter" to refer to people who are *using* the HCI methods to study the programmers or to design or evaluate new tools or processes for the programmers.

HCI methods can be applied to everything the developer encounters, including tools such as editors and integrated development environments (IDEs), reusable components such as application programming interfaces (APIs), libraries, and software development kits (SDKs), documentation for all the tools and APIs, the processes and context used to organize the development, and even the design of the programming languages themselves.

The dangers of *not* using these human-centered practices have been well-documented, resulting in 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*, and/or have a large impact and be *difficult* for the developer to solve (which might be measured by the amount of effort or time it takes to solve). The frequency, impact and difficulty are questions that can all be measured with the HCI methods discussed below. Surveys have shown that developers complain that researchers can address unimportant problems,[10] which can be avoided by using human-centered data to help decide which problems to research. Furthermore, a research result may not actually *solve* the problem. Sometimes a new tool or process may just *change* the problem, rather than solve it. For example, often visualizations just change the developers' search task from looking through the code to instead be looking through a complicated graphical presentation, which may not be faster. Another example is that a new tool may have such a poor user interface that developers find it too difficult to use, even when its functionality is inherently useful.[11] Again, HCI methods can be used to measure the results of using the new tool or process, to see whether developers perform better.

HCI Method	Tool development activities supported	Key benefits	Challenges and limitations
Contextual inquiry[1]	Requirements & problem analysis	Insight into day to day activities & challenges; high quality data on intent	Time consuming; may be challenging to recruit professionals
Exploratory lab studies	Requirements & problem analysis	Easy to focus on activity of interest; compare participants on same tasks; data on intent	Experimental setting may differ from real world context
Surveys	Requirements and problem analysis; evaluation and testing	Quantitative data; many participants; (relatively) fast	Self-reported data subject to bias and participant awareness
Data mining (including corpus studies and log analysis)	Requirements and problem analysis; evaluation and testing	Large quantities of data; ability to see patterns which emerge only through large corpuses	Difficult to infer or reconstruct developer's intent; requires careful filtering
"Natural programming" elicitation[3]	Requirements and problem analysis; design	Insight into developer expectations	Experimental setting may differ from real world context
Rapid prototyping[4]	Design	Gather feedback at low cost before committing to high cost development	Lower fidelity than final tool, limiting what problems may be revealed
Heuristic evaluation[4]	Requirements and problem analysis; design; evaluation and testing	Fast; does not require participants	Only reveals some types of usability issues
Cognitive walkthrough[5]	Design; evaluation and testing	Fast; does not require participants	Only reveals some types of usability issues
Think-aloud usability evaluations[4]	Requirements and problem analysis; design; evaluation and testing	Reveals usability problems & developer intent	Experimental setting may differ from real world context; requires appropriate participants; task design is difficult
A/B testing lab experiments	Evaluation and testing	Provides direct evidence that new tool or technique benefits developers	Experimental setting may differ from real world context; requires appropriate participants; task design is difficult

Figure 1. An overview of 10 HCI methods our group has used.

This article discusses how our research group has used 10 different HCI methods and principles to investigate problems and solutions across all activities involved in developing a new tool for developers (summarized in Figure 1), along with 5 usability recommendations. Although we list the tool development activities supported by each method, we are not suggesting that these are

the only ways to use the method – we are just emphasizing the breadth of activities that can be helped by HCI methods, and the particular ways we have used these methods. The next sections present small case studies of how we used the methods along with discussions about strengths and weaknesses of each. This paper contributes to understanding the broad range of human-centered approaches that can help to improve tools used by programmers.

Methods for requirements and problem analysis

The first activity of a project is likely to involve getting a better understanding of the users' real problems and needs, as a way of solidifying the requirements. Although obviously relevant for commercial systems, we have found this to be useful and often necessary for research projects as well. Four methods that we have often used in this activity are contextual inquiry, exploratory lab studies, surveys, and data mining.

Contextual inquiry[1] (CI) is a method where the experimenter observes developers performing their usual, real work where it actually happens. For example, for one of our projects, we were wondering what the key barriers are when developers fix defects, so we asked developers at Microsoft to work on their own tasks, and we watched and took notes about the issues that arose.[2] One result was that a key problem for 90% of the longest tasks was understanding the control flow through code in widely separated methods, since this is not adequately revealed by existing tools.[2] CIs are good ways to gather qualitative data and insights into developers' real issues, but they do not provide quantitative statistics due to the small sample size, and can be time consuming to perform, especially if it is difficult to recruit representative developers to observe.

Another useful way to gain insights is with *exploratory lab studies*, where the experimenter gives developers specific tasks to do, and then observes what happens. This differs from usability analysis and A/B tests (discussed below) in that the participants are usually using conventional tools (rather than a new design). The key difference from CIs is that here the participants perform tasks provided by the experimenter, instead of their own tasks as in CIs, so the experimenter can see if the participants use different approaches to the same task, but at the cost of realism. For example, we collected a detailed dataset at the keystroke level of experienced developers performing maintenance tasks in Java.[12] This dataset was so unique and valuable that it resulted in multiple award-winning papers about developers' behaviors. For instance, we discovered from this data that developers were spending about one-third of their time navigating around the code base, often using manual scrolling.[12] This highlights an important advantage of these observational techniques – when *asked* about barriers when performing these tasks, no-one mentioned scrolling, because it did not rise to the level of salience, but 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.

In other cases, asking developers questions can be very 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 interprocedural control flow as being important in our CIs, we wanted to find out how often developers have questions about control flow and how hard those questions are to answer. Our survey confirmed that developers report asking such questions on average about 9 times a day and most felt at least one was hard to answer.[2] By adding open-ended questions to this survey, we were also able to collect

a large number of other hard-to-answer questions about code that capture real problems faced by developers in their everyday work.

Another method we have used, which is probably more closely associated with other fields rather than HCI, is *data mining*, which may include corpus studies or log analyses, depending on what kind of data is being analyzed. Often, this can be a good way to investigate how pervasive some pattern is. For example, we were wondering how often developers *backtrack* while code editing ("backtracking" here means returning the code to a previous state), possibly by using the editor's *undo* command, so we analyzed 1,460 hours of fine-grained code editing logs collected from 21 developers. We detected a total of 15,095 backtracking instances, which gives an average backtracking rate of 10.3/hour. This motivated us to create a new tool, Azurite, which provides more flexible *selective undo*, where developers can undo code edits in the past without necessarily undoing more recent edits.[13] 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, which suggest 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, the large amount of data often makes manual inspection impossible, which can make it more difficult to validate results or understand developer's intent.

Methods for design

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

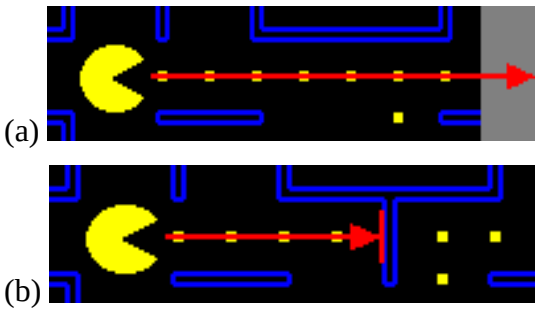


Figure 2. Pictures to prompt novice or non-programmers to express conditional situations. Participants were asked how they would instruct the computer to have Pacman behave according to the pictures. The difference is that in (b) PacMan hits a wall and stops, which can be described either using an "if-then" style ("if PacMan hits a wall, he stops") or a "constraint" style ("PacMan cannot go through a wall").

Since programming is the process of mapping the developer's intent into a form that the computer can execute, the process can be made easier by bringing the form of expression closer to the way the developer is thinking. Our group invented a new method to better understand the ways that developers naturally think about their problems, which we call the *natural programming elicitation method.*[3] This involves getting developers to provide their own descriptions or designs, which then informs the design of tools and languages to match. For

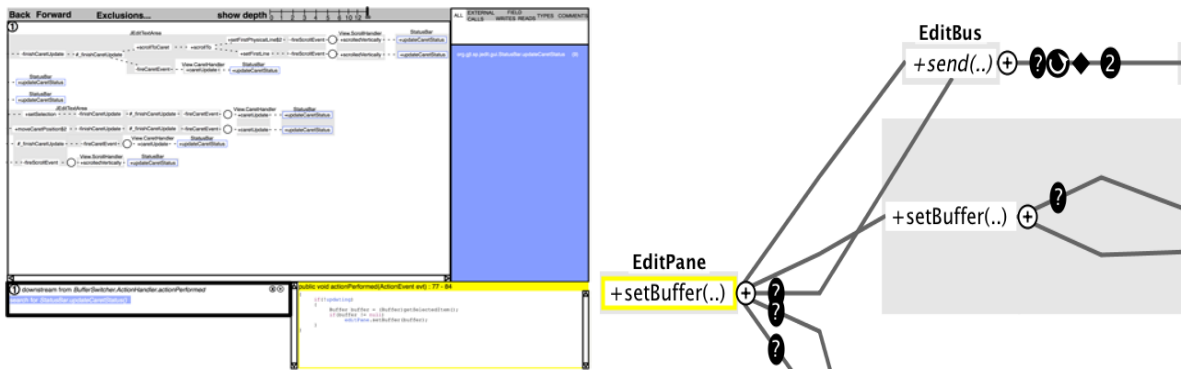
example, our early work examined how novice developers would express various programming concepts.[3] In one study, we showed them pictures like those in Figure 2 and asked them how they would want to instruct the computer to achieve that behavior. One result was that an event-based or rule-based structure was often used, where actions were taken in response to events. For example, for Figure 2, 54% of the participants said something like "If PacMan hits a wall, he stops." In contrast, about 18% of the participants used a constraint-based style, such as "PacMan cannot go through a wall", with the rest of the participants using other styles. We used the result of this and other studies to guide the design of a new programming language for children who were learning to program.[3] For instance, based on the results of Figure 2, the system uses the if-then style for event handling. We have also used this natural programming elicitation style to understand how developers want to access functionality through APIs.[14] The strategy here is to give developers a blank screen or piece of paper, and ask them to design the API for a particular functionality. This helps understand the most natural *vocabulary* and *organization* of classes and methods, and provides recommendations for future API designs.[14]

Additionally, a key HCI guideline is to rapidly iterate on the design, using *prototypes*.[4] Typically, the first step would be to use "paper prototypes" which are quickly created using drawing tools, or even just pen and paper. In many of our tools, we use this *rapid prototyping* technique to test whether our ideas are likely to work. For example, in trying to help developers understand the interprocedural control flow of code, we drew mockups of a possible new visualization using the OmniGraffle drawing tool and printed them out on paper (Figure 3a).[2] By putting these in front of developers, and asking them to pretend to perform tasks with them, we discovered that the initial concepts for the visualizations were too complex to understand, yet were missing information that was 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 (Figure 3b), the lines coming out of a method show the order in which methods are invoked.[2]

Methods for evaluation and testing

Developers are likely familiar with using HCI methods to evaluate the usability of the products they make, but may not have tried using those same methods to measure the usability of their own development tools. We routinely evaluate the usability and performance of our tools for developers, using four kinds of methods: *expert analyses*, *think-aloud usability evaluations*, formal *A-vs-B experiments*, and log analyses.

First, we have used *expert analyses*, where people who are experienced with usability methods perform the analysis by inspection. For example, *heuristic evaluation* is a method where 10 guidelines are used to evaluate an interface.[4] We used this method in our collaboration with SAP, where we found that the really long names for functions violated the principle of error-prevention, because the names were easily confused with each other.[14] Another expert analysis method is called *cognitive walkthrough*[5], and involves carefully going through tasks using the interface, and noting where users will need new knowledge to be able to take the next step. We helped SAP iteratively improve a developer tool for Visual Studio through using both of these expert analyses methods. Since SAP was using an agile development process,[8] they were able to address our recommendations immediately.



(a) (b)

Figure 3. (a) A paper prototype of the visualization drawn with the Omnigraffle drawing tool revealed that the order of method calls was crucial to visualize, as is shown in the final version of the tool (b), which is called "Reacher".[2] The method EditPane.setBuffer(..) makes 5 method calls (the 5 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 may or may not happen at runtime). Other icons on lines include a circular arrow to show calls inside of loops, diamonds to show overloaded methods, and numbers to show that multiple calls have been collapsed.

A second set of methods are *empirical* and involve testing the tools with the target users. For development tools, this requires testing the tools with developers using realistic tasks. The first result of these evaluations will be an understanding of what participants actually do, to see how participants work with the tool. In addition, we recommend using a *think-aloud study*, where the participants are asked to continuously articulate their goals, confusions and other thoughts. This provides the experimenter with rich data about *why* users are performing in the way that they are, so problems can be found and fixed. As with other usability tests, the principle is that if one participant has a problem, it is highly likely that others will have the same problem, so it should be fixed, if possible. In fact, research shows that a few representative users can find a great percent of the problems.[4] In our research, if we have tools where the evidence of *usefulness* comes from early needs analysis from CIs and surveys, it is often sufficient to show *usability* through think-alouds with five or six people. Note that it does not work to try to do a usability evaluation with any participants who are associated with the tool, since they will know too much about how the tool is supposed to work.

In contrast to the first two methods, which are informal, the third method uses formal, statistically valid *experiments.* This is the key way to demonstrate that one tool is *better* than another along some measure. For example, we tested the Whyline debugging tool and showed that developers using it were over 3 times as successful, in one-half of the time, compared to the control group.[15] Similarly, the Azurite backtracking plugin, which supports selective undo of code edits, was tested against regular Eclipse without the plugin. The test showed that developers were twice as fast when using Azurite.[13] Such formal measures are important for research papers, and they may help convince developers and managers to try new tools or change working habits, since engineers tend to find numbers persuasive. However, these experiments can be difficult to design correctly, and require careful attention to many possibly confounding factors.[16] In

particular, it is challenging to design the tasks for participants that have the right balance of realism yet are doable in an appropriate time frame of an experiment (taking an hour or two).

The evaluation of a tool does not have to stop when it leaves the lab. We have found that adding detailed logging to a tool can help the tool designer better understand how tools are used in the field. For example, our Fluorite logger was used to investigate how the Azurite backtracking tool was used in the field, in actual use, revealing that developers often selectively undo a block of code, such as a whole method, restoring it to how it used to work, and leaving the other code as is.[13] Similarly, we created a web-based documentation tool called Apatite, that presents Java methods *by association* – when the user selects a method or class, then Apatite shows all the other methods and classes that are often used with it. Since this was a web tool, it was easy to log every user action using conventional web analytics tools, which showed that few users found this association feature useful, and instead mainly used Apatite to quickly find a method or class by name since it provides autocomplete on any name entered. While it was disappointing that our research tool did not prove more useful, the field study revealed valuable insights into how developers wished to use tools within their work.

Design and development recommendations

In addition to the HCI *methods* discussed above, human-centered results can improve tools for developers in other ways. Here are five observations we have found useful.

First, an HCI practice that we have found useful during design is 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, and usability issues have been shown to be a barrier to uptake and use, even when the functionality is useful.[11] One can engage people skilled in graphic and interaction design to help resolve these issues. For example, graphic designers can help with colors, icons, selection of controls, layout, etc. This may be especially important for visualization tools. We have consistently found that improving the presentation, interaction flow, and layout of our tools has improved their usability, popularity, and evaluations.

Second, we have found that a key use for visualizations is to guide developers to the right code to look at, rather than being an understanding tool on their own. For example, we built a tool called the "Whyline" to help with debugging by visualizing *why* a particular output did or did not happen, through dynamic and static analysis of full execution traces. The first version of the Whyline targeted an educational programming language called "Alice" and provided an elaborate visualization of the control and data flow. However, when the tool was targeted at Java, the visualization became unwieldy and not understandable, so instead, we focused the visualization on providing an easy way to *navigate* to the relevant code for each step.[15] Similarly, the Reacher tool shown in Figure 3 shows only a summary of method invocations, and users click on the lines and icons to see the corresponding code snippets to get an understanding of the associated code.[2]

A third observation is the primacy of *search*. It is no surprise to any developer how useful it is to search the web for answers to a wide variety of questions, from how to use APIs to what error messages mean. Indeed, in one of our projects, called Mica, we augmented Google search to try to make the returned results even more useful, by highlighting which API methods and code examples were returned by the search. In addition, we have found search to be useful in many

other ways. For example, the Reacher tool in Figure 3 allows users to search forwards and backwards *along the feasible control flow* (rather than searching through all of the code), to specifically answer such questions as "Are there any paths by which this method can be reached without first calling the initialization method?". The Azurite tool[13] mentioned above is a plugin for a code editor that allows developers to search code backwards in *time*, to answer such questions as "when was this variable renamed"? In general, we found that developers often have very specific questions, so providing a means to answer these questions directly through tools can substantially improve the developers' productivity.

Our fourth observation is to augment what developers are already doing, so the new features are where they are looking anyway. This significantly increases familiarity and therefore usability and reduces the overhead of the new tools. For example, many developers' favorite way to explore an API is to use the autocomplete popup menus in the code editor. Although designed to reduce typing, this feature is commonly used by developers to see the list of available methods, and then the developer will try to guess which one might be useful. However, we found that the Eclipse autocomplete is not always useful when trying to *create* a new instance of a class (for example, invoking the autocomplete menu after typing an "=" does not provide any useful completions in Eclipse[14]), especially when the instance should be created using a "factory" or other indirect means. Therefore, we built new plugins for Eclipse that incorporate such entries directly into the autocomplete menus. The first plugin, called Calcite, makes the autocomplete menu that appears after an "=" is typed more useful by adding to the menu the most common ways to create instances based on an analysis of code found through web crawling.[14] Another plugin, called Dacite, adds autocomplete entries based on API annotations for various patterns such as "factories" and "helper methods." Our Graphite plugin provides mini-editors, which can be discovered through autocomplete, for defining colors and interactively authoring regular expressions, automatically generating the corresponding code to create Java objects. Finally, our Jadeite tool augments the popular JavaDoc documentation for Java with entries for methods that developers expect to exist in a specific class but are actually defined somewhere else. For example, we found that developers expect the email "send" method to be on the `Message` class, whereas it is actually on the `Transport` class. Therefore, Jadeite adds a "placeholder" entry in the JavaDoc for the `Message` class under "send" telling developers where to look.[14]

With respect to the process for developing programming tools, we recommend the same process that has always been recommended for creating more usable applications for consumers: [4,17] iterative design throughout all phases using human-centered methods as described above. Large companies creating developer tools, such as Microsoft and Google, already embed user interface specialists into their teams that are creating developer tools (such as in Microsoft's Visual Studio group), but even small teams can learn to use at least some of these methods themselves.[4]

Conclusions

This paper has illustrated 10 human-centered methods and principles that we have successfully used in our group to better understand developers and the tools that might help them. We also provide 5 other human-centered observations that help guide tool-building. Many other HCI methods and observations are also available which can answer other questions that tool developers might have. Hopefully, these can be used to increase the likelihood that future tools will help developers be more successful, effective, and efficient.

Acknowledgements

This article grows out of nearly 30 years of work by the Natural Programming group by more than 50 students, staff and postdocs in addition to the authors, and we thank them all for their contributions. The work summarized here has been funded by SAP, Adobe, IBM, Microsoft and multiple NSF 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 material are those of the authors and do not necessarily reflect those of any of the sponsors.

Biographical sketches

Brad A. Myers is a Professor in the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University. His research interests include programming environments, programming language design, and user interfaces. He received a PhD in in computer science at the University of Toronto. He is a Fellow of the IEEE and the ACM, and also belongs to the IEEE Computer Society and the ACM Special Interest Group on Computer-Human Interaction (SIGCHI). He can be reached at: bam@cs.cmu.edu.

Thomas D. LaToza is an Assistant Professor of Computer Science in the Volgenau School of Engineering at George Mason University. His research focuses on software engineering environments, spanning empirical and design work in the areas of debugging, software design, collaboration, and crowdsourcing software engineering. He received his Ph.D. at the Institute for Software Research at Carnegie Mellon University in 2012. He can be reached at tlatoza@gmu.edu.

Andrew J. Ko is an Associate Professor at the University of Washington Information School and an Adjunct Associate Professor in Computer Science and Engineering. His research focuses on interactions between people and code, spanning the areas of human-computer interaction, computing education, and software engineering. He received his Ph.D. at the Human-Computer Interaction Institute at Carnegie Mellon University in 2008. He can be reached at ajko@uw.edu.

YoungSeok Yoon is a software engineer at Google Inc., USA. His research interests lie in making software developers more productive by providing better development tools and environments for code editing, debugging, testing, and build automation. He received his Software Engineering Ph.D. in the Institute for Software Research at Carnegie Mellon University in 2015. He can be reached at: youngseokyoon@google.com.

References

[1] H. Beyer, and K. Holtzblatt, *Contextual Design: Defining Custom-Centered Systems*, San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1998.

[2] T. D. LaToza, and B. Myers, “Developers Ask Reachability Questions,” in *ICSE'2010: Proceedings of the International Conference on Software Engineering*, Capetown, South Africa, 2010, pp. 185-194.

[3] B. A. Myers, J. F. Pane, and A. Ko, “Natural Programming Languages and Environments,” *Communications of the ACM*, vol. 47, no. 9, pp. 47-52, Sept, 2004.

[4] J. Nielsen, *Usability Engineering*, Boston: Academic Press, 1993.

[5] C. Lewis, P. G. Polson, C. Wharton *et al.*, "Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces," in *CHI'90: Proceedings of the SIGCHI conference on Human factors in computing systems*, Seattle, WA, 1990, pp. 235-242.

[6] J. Tomayko, and O. Hazzan, *Human Aspects of Software Engineering*, p.^pp. 338: Charles River Media, 2004.

[7] A. Seffah, J. Gulliksen, and M. C. Desmarais, *Human-Centered Software Engineering - Integrating Usability in the Software Development Lifecycle*: Springer Netherlands, 2005.

[8] D. Salah, R. F. Paige, and P. Cairns, "A systematic literature review for agile development processes and user centred design integration," in *(EASE'14) Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, London, UK, 2014, pp. Article 5, 10 pages.

[9] A. J. Ko, R. Abraham, L. Beckwith *et al.*, "The State of the Art in End-User Software Engineering," *ACM Computing Surveys*, vol. 43, no. 3, pp. Article 21, 44 pages, April, 2011.

[10] D. Lo, N. Nagappan, and T. Zimmermann, "How Practitioners Perceive the Relevance of Software Engineering Research," in *10th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, Bergmano, Italy, 2015.

[11] E. Murphy-Hill, and A. P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, pp. 38-44, 2008.

[12] A. J. Ko, B. A. Myers, M. Coblenz *et al.*, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 971-987, Dec, 2006.

[13] Y. Yoon, and B. A. Myers, "Supporting Selective Undo in a Code Editor," in *37th International Conference on Software Engineering, ICSE 2015*, Florence, Italy, 2015, pp. 223-233 (volume 1).

[14] B. A. Myers, and J. Stylos, "Improving API Usability," *Communications of the ACM*, vol. 59, no. 6, pp. to appear, July, 2016.

[15] A. J. Ko, and B. A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," in *ICSE'2008: 30th International Conference on Software Engineering*, Leipzig, Germany, 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 Softw. Engg.*, vol. 20, no. 1, pp. 110-141, 2015.

[17] J. Goodman-Deane, P. M. Langdon, S. Clarke *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., pp. 23-34, London: Springer London, 2008.