CHAPTER 13

Human-Centered Methods to Boost Productivity

Brad A. Myers, Carnegie Mellon University, USA
Andrew J. Ko, University of Washington, USA
Thomas D. LaToza, George Mason University, USA
YoungSeok Yoon, Google, Korea

Since programming is a human activity, we can look to fields that have already developed methods to better understand the details of human interactions with technologies. In particular, the field of human-computer interaction (HCI) has dozens, if not hundreds, of methods that have been validated for answering a wide range of questions about human behaviors [4]. (And many of these methods, in turn, have been adapted from methods used in psychology, ethnography, sociology, etc.) For example, in our research, we have documented our use of at least ten different human-centered methods across all the phases of software development [11], almost all of which have impacts on programmer productivity.

Why would one want to use these methods? Even though productivity may be hard to quantify, as discussed in many previous chapters of this book, it is indisputable that problems exist with the languages, APIs, and tools that programmers use, and we should strive to fix these problems. Further, there are more ways to understand productivity than just metrics. HCI methods can help better understand programmers' real *requirements and problems*, help *design* better ways to address those challenges, and then help *evaluate* whether the design actually works for programmers. Involving real programmers in these investigations reveals real data that makes it possible to identify and fix productivity bottlenecks.

C. Sadowski and T. Zimmermann (eds.), *Rethinking Productivity in Software Engineering*,
https://doi.org/10.1007/978-1-4842-4221-6_13

For example, a method called *contextual inquiry* (CI) [1] is commonly used to understand barriers *in context*. In a CI, the experimenter observes developers performing their real work where it actually happens and makes special note of *breakdowns* that occur. For example, in one of our projects, we wondered what key barriers developers face when fixing defects, so we asked developers at Microsoft to work on their own tasks while we watched and took notes about the issues that arose [7]. 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. However, it is one of the best ways to identify what is *really* happening in the field that affects the programmers' productivity.

Another useful method to understand productivity barriers is doing *exploratory lab user studies* [14]. Here, the experimenter assigns specific tasks to developers and observes what happens. 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 data set at the keystroke level of multiple experienced developers performing the same maintenance tasks in Java [5]. 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—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 to the programmers' productivity when we analyzed the logs of what the developers actually did. Knowing about such problems is the first step to inventing solutions. And these kinds of studies can also provide numeric data, which can later be used to measure the difference that a new tool or other intervention makes.

Neither of these methods can be used to evaluate *how often* an observed barrier occurs, which might be important for calculating the overall impact on productivity. For this, we have used *surveys* [16] and *corpus data mining* [9]. For example, after we observed in our CIs that understanding control flow was important, we performed a survey to count how often developers have questions about control flow and how hard those questions are to answer [7]. 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. In a different study, we felt that programmers were wasting significant time trying to *backtrack* (return code to a previous state) while editing code. We had observed that this seemed to be error-prone as changes often had to be undone in multiple places. Therefore, we analyzed 1,460 hours of fine-grained code-editing logs from 21 developers, collected during their regular work [18]. We detected 15,095 backtracking instances, for an average rate of 10.3 per hour.

Once such productivity barriers have been identified, an intervention might be designed, such as a new programming process, language, API, or tool. We have used a variety of methods during the design process to help ensure that the intervention will actually help. *Natural-programming elicitation* is a way to understand how programmers think about a task and what vocabulary and concepts they use so the intervention can be closer to the users' thoughts [10]. One method for doing natural-programming elicitation is to give target programmers a "blank paper" participatory design task, where we describe the desired functionality and have the programmers design how that functionality should be provided. The trick is to ask the question in a way that does not bias the answers, so we often use pictures or samples of the *results*, without providing any vocabulary, architecture, or concepts.

Rapid prototyping [15] allows quick and simple prototypes of the intervention to be tried, often just drawn on paper, which helps to refine good ideas and eliminate bad ones. Sometimes it might be too expensive to create the real intervention before being able to test it. In these cases, we have used another recommended human-centered method called *iterative design* using *prototypes* [14]. Typically, the first step employs *low-fidelity prototypes*, which means that the actual interventions are simulated. For many of our tools, we have used *paper prototypes*, which are quickly created using drawing tools or even just pen and paper. For example, when trying to help developers understand the interprocedural control flow of code, we used a Macintosh drawing program called OmniGraffle to draw mock-ups of a possible new visualization and printed them on paper. We then asked 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 [7]. 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, the lines coming out of a method show the order of invocation, as shown in Figure 13-1.

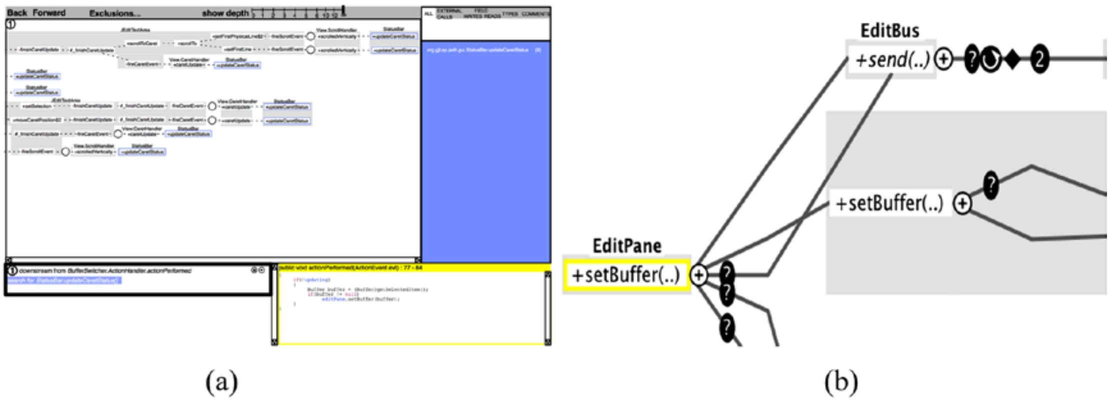

Figure 13-1. *(a) A paper prototype of the visualization drawn with the Omnigraffle drawing tool revealed that the order of method calls was crucial to visualize, as is shown in the final version of the tool (b), which is called Reacher [7]. The method `EditPane.setBuffer(..)` makes five method calls (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 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.*

No matter what kind of intervention it is, the creator might want to evaluate how well programmers can use it and whether it actually improves productivity in practice. For example, our observations about backtracking difficulties motivated us to create Azurite, a plug-in for the Eclipse code editor that provides more flexible selective undo, in which developers can undo past edits without necessarily undoing more recent ones [19]. But how can we know if the new intervention can actually be used? There are three main methods we have used to *evaluate* interventions: expert analyses, think-aloud usability evaluations, and formal A/B testing.

In expert analyses, people who are experienced with usability methods perform the analysis by inspection. For example, *heuristic evaluation* [13] employs ten guidelines to evaluate an interface. We used this method to evaluate some APIs and found that the really long function names violated the guideline of error prevention because the names could be easily confused with each other, wasting the programmer's time [12]. Another expert-analysis method is called *cognitive walkthrough* [8]. 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 a company iteratively improve a developer tool [3].

Another set of methods is empirical and involves testing the interventions with the target users. The first result of these evaluations is an understanding of what participants actually do, to see how the intervention works. In addition, we recommend using a *think-aloud study* [2], 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 evaluations, 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 [14]. 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.

Unlike expert analyses and think-aloud usability evaluations, which are informal, *A/B testing* uses formal, statistically valid experiments [6]. This is the key way to demonstrate that one intervention is *better* than another, or better than the status quo, with respect to some measure. For example, we tested our Azurite plugin for selective undo in Eclipse against using regular Eclipse, and developers using Azurite were twice as fast [19]. Such formal measures can be useful proxies for the productivity gains that an intervention might bring. The resulting numbers might also help convince developers and managers to try new interventions and change developers' behaviors because they might find having numbers more persuasive than just the creator's claims about the intervention. However, these experiments can be difficult to design correctly and require careful attention to many possibly confounding factors [6]. 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).

To get a more realistic evaluation of an intervention, it may need to be measured in actual practice. We have found this to be easiest to do by instrumenting the tools to gather the desired metrics during real use, and then we can use *data mining and log analysis*. For example, we used our Fluorite logger, which is another plugin for Eclipse, to investigate how developers used the Azurite tool [17]. We found that developers often selectively undid a selected block of code, such as a whole method, restoring it to how it used to work and leaving the other code as is, which we call *regional undo*, confirming our hypothesis that this would be the most useful kind of selective undo [19].

Many other HCI methods are available that can answer additional questions that creators of interventions might have (see Table 13-1 for a summary). 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. Based on our extensive use of these methods over many years, we argue that they will be useful for better understanding the many different kinds of barriers that programmers face, for creating useful and usable interventions to address those barriers, and for better evaluating the impact of the interventions. In this way, these methods will help increase the positive impact of future interventions on developers' productivity.

Table 13-1. Methods We Have Used *(Adapted from [11])*

Method	Cite	Software Development Activities Supported	Key Benefits	Challenges and Limitations
Contextual inquiry	[1]	Requirements and problem analysis.	Experimenters gain insight into day-to-day activities and challenges. Experimenters gain high-quality data on the developer's intent.	Contextual inquiry is time-consuming.
Exploratory lab user studies	[14]	Requirements and problem analysis.	Focusing on the activity of interest is easier. Experimenters can compare participants doing the same tasks. Numerical data can be collected.	The experimental setting might differ from the real-world context.
Surveys	[16]	Requirements and problem analysis. Evaluation and testing.	Surveys provide quantitative data. There are many participants. Surveys are (relatively) fast.	The data is self- reported and is subject to bias and participant awareness.
Data mining (including corpus studies and log analysis)	[9]	Requirements and problem analysis. Evaluation and testing.	Data mining provides large quantities of data. Experimenters can see patterns that emerge only with large corpuses.	Inferring or reconstructing the developer's intent is difficult. Data mining requires careful filtering.
Natural- programming elicitation	[10]	Requirements and problem analysis. Design.	Experimenters gain insight into developer expectations.	The experimental setting might differ from the real-world context.

(continued)

Table 13-1. (*continued*)

Method	Cite	Software Development Activities Supported	Key Benefits	Challenges and Limitations
Rapid prototyping	[15]	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	[13]	Requirements and problem analysis. Design. Evaluation and testing.	Evaluations are fast. They do not require participants.	Evaluations reveal only some types of usability issues.
Cognitive walk-throughs	[8]	Design. Evaluation and testing.	Walk-throughs are fast. They do not require participants.	Walk-throughs reveal only some types of usability issues.
Think-aloud usability evaluations	[2]	Requirements and problem analysis. Design. Evaluation and testing.	Evaluations reveal usability problems and the developer's intent.	The experimental setting might differ from the real-world context. Evaluations require appropriate participants. Task design is difficult.
A/B testing	[6]	Evaluation and testing	Testing provides direct evidence that a new tool or technique benefits developers.	The experimental setting might differ from the real-world context. Testing requires appropriate participants. Task design is difficult.

Key Ideas

The following are the key ideas from the chapter:

- There are many methods used in human-computer interaction research that can also be used to study what hinders and improves software developer productivity, to help design interventions that increase productivity, and to then evaluate and improve their impact.
- The ten methods listed in this chapter have proven useful at various phases of the process.

References

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

[2] Chi, M. T. (1997). Quantifying qualitative analyses of verbal data: A practical guide. The journal of the learning sciences, 6(3), 271–315.

[3] Andrew Faulring, Brad A. Myers, Yaad Oren and Keren Rotenberg. "A Case Study of Using HCI Methods to Improve Tools for Programmers," *Cooperative and Human Aspects of Software Engineering (CHASE'2012)*, An ICSE 2012 Workshop, Zurich, Switzerland, June 2, 2012. 37–39.

[4] Julie A. Jacko. (Ed.). (2012). Human computer interaction handbook: Fundamentals, evolving technologies, and emerging applications. CRC press.

[5] Andrew J. Ko, Brad A. Myers, Michael Coblenz and Htet Htet Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*. Dec, 2006. 33(12). pp. 971–987.

[6] Ko, A. J., Latoza, T. D., & Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. Empirical Software Engineering, 20(1), 110–141.

[7] Thomas D. LaToza and Brad Myers. "Developers Ask Reachability Questions," *ICSE'2010: Proceedings of the International Conference on Software Engineering*, Capetown, South Africa, May 2-8, 2010. 185–194.

[8] C. Lewis et al., "Testing a Walkthrough Methodology for TheoryBased Design of Walk-Up-and-Use Interfaces," *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI 90)*, 1990, pp. 235–242.

[9] Menzies, T., Williams, L., & Zimmermann, T. (2016). Perspectives on Data Science for Software Engineering. Morgan Kaufmann.

[10] Brad A. Myers, John F. Pane and Andy Ko. "Natural Programming Languages and Environments," *Communications of the ACM*. Sept, 2004. 47(9). pp. 47–52.

[11] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," *IEEE Computer*, vol. 49, issue 7, July, 2016, pp. 44–52.

[12] Brad A. Myers and Jeffrey Stylos. "Improving API Usability," *Communications of the ACM*. July, 2016. 59(6). pp. 62–69.

[13] J. Nielsen and R. Molich. "Heuristic evaluation of user interfaces," Proc. ACM CHI'90 Conf, see also: `http://www.useit.com/papers/heuristic/heuristic_list.html`. Seattle, WA, 1–5 April, 1990. pp. 249–256.

[14] Jakob Nielsen. *Usability Engineering*. Boston, Academic Press. 1993.

[15] Marc Rettig. "Prototyping for Tiny Fingers," Comm. ACM. 1994. vol. 37, no. 4. pp. 21–27.

[16] Rossi, P. H., Wright, J. D., & Anderson, A. B. (Eds.). (2013). Handbook of survey research. Academic Press.

[17] YoungSeok Yoon and Brad A. Myers. "An Exploratory Study of Backtracking Strategies Used by Developers," *Cooperative and Human Aspects of Software Engineering (CHASE'2012)*, An ICSE 2012 Workshop, Zurich, Switzerland, June 2, 2012. 138–144.

[18] YoungSeok Yoon and Brad A. Myers. "A Longitudinal Study of Programmers' Backtracking," *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*, Melbourne, Australia, 28 July–1 August, 2014. 101–108.

[19] YoungSeok Yoon and Brad A. Myers. "Supporting Selective Undo in a Code Editor," *37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, May 16–24, 2015. 223–233 (volume 1).