The Role of Science in Supporting Software Development

Andrew J. Ko
5000 Forbes Avenue, Pittsburgh PA, 15213
Human-Computer Interaction Institute
Carnegie Mellon University
ajko@cs.cmu.edu

ABSTRACT

Discusses the importance of scientific explanations in tool design, and various ways of forming such explanations.

Categories and Subject Descriptors

H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms

Design, Human Factors, Experimentation.

Keywords

Empiricism, science, design, tools, evaluation, notation, theory, measurement, prototyping, experts, ethnography, collaboration.

1. INTRODUCTION

The primary focus of this workshop is to reflect on how tools can support the social side of software development. In service of this goal, rather than using this space to espouse my own ideas about how this might be done, I would instead like to reflect on the methods by which we invent such tools.

It is difficult to invent useful tools without some understanding of how people develop software. Even the most biased of tool designers have some model in their minds of what is important to software developers. Of course, these models are largely based on personal experience. While experience can be a valuable form of inspiration, what differentiates research from experience is science—and scientists seek to *explain*.

Therefore, while *descriptions* of the social side of software development have captured many of its modern practices, descriptions are insufficient for design. We need to know *why* software development is social. Is it because developers prefer to be social or because they need to be? What do developers gain by communicating with their peers? We know that some of this is to maintain awareness [2] and some is to learn from experts [3]—but awareness and knowledge of what? Are coworkers the only source for such information, or just the preferred source?

These questions are more than scholarly: the explanations we derive by investigating these questions are fodder for design. The more we understand *why* developers are social, the better that tools can match developers' needs. The better we can explain *why* developers seek awareness and expert knowledge, the better we can evaluate tools and articulate their tradeoffs.

But how can we explain these phenomena? Empiricism and observation are essential tools, but I would argue insufficient. One of their limitations is that the forms of explanations that they generate—models, theories, diagrams, etc.—rarely do justice to reality. We need to proceed one step further and "create" explanations by prototyping new tools and notations. Then, when we describe our explanations of why developers maintain awareness of each others' work, we need not refer to a paragraph or a picture; we instead point to an interactive tool or a new language that explicitly represents our theory of what is important to software development and what is not. Just as mathematics is the language for theories in basic sciences, tools and notations can embody our theoretical explanations of reality. Unlike other fields of science, however, tools have the unique ability to *change* reality—they are Turing's *mechanized thought* [8] realized.

2. EXPLAINING THROUGH EMPIRICISM

I practice these ideas to the extent that I can. I began my doctoral work by studying software development in a collaborative context, with four groups of students prototyping interactive 3D worlds in the Building Virtual Worlds course at CMU [4]. In this context, the reason for communication was clear: each contributor had a different skill. The programmer wrote code, the audio engineer create sounds, the writer scripted scenes, and the artists modeled characters. Communication in these groups occurred along technical dependencies: the programmers needed character models before they could write code to make characters behave; this meant that they needed to track the modeler's work.

When observing students trying to learn Visual Basic.NET to prototype user interfaces [5], communication was less about dependencies and more about expertise. When less experienced students reached an impasse, they would immediately seek out more experienced students for advice: where should I put my breakpoint? How do you use a timer? What can store a date?

Even in a lab study of lone developers' repairing bugs and adding features [5], I observed a great reliance on other people, through developers' use of documentation and example code. Moreover, the artificiality of the study emphasized the importance of collaboration: each time a developer sought some information about the code, rather than using information from other people, they were forced to resort to their own mind. Had I simply provided some documentation or some comments from the program's designer, their task would have been greatly simplified.

Most recently, I did a field study of 17 Microsoft product groups, documenting the information that developers sought, where they found it, and what prevented them from acquiring it. Coworkers were a central source of knowledge and bug reports were a hub for hints, discoveries, and decisions in the form of conversations. Of course, the surprising thing was not that developers relied on each other, but for *what* they relied on each other. One of the most important and difficult to find types of information was *design* knowledge. Why did you write this code this way? What is the program supposed to do in this scenario? For what purpose is this data structure intended? These questions refer not to technical aspects of code, but to the rationale and decisions of the code's authors. Therefore, code was a social and cognitive construct, only partially represented by the text in a source file.

3. EXPLAINING THROUGH DESIGN

Prototyping new technologies has played an equally important role in my studies. As with any design, my inventions did not follow directly from the understanding I have gained through observation. Rather, they are a culmination of the understanding I have gained about software development, both from my own investigation and from the decades of research that came before.

Consider the Whyline [7], the first tool that I worked on in my doctoral work. The idea behind this debugging tool was to help developers ask questions about their program's output and reveal their implicit assumptions about what had occurred at runtime. While I used my observation of the Building Virtual Worlds class discussed earlier for inspiration, the idea ultimately originated from several months of reflection and reasoning about the work that I observed. and a careful study of other debugging tools described in the literature The understanding and theories I had gained from observations helped me to *evaluate* and *test* the merits of my ideas, but not to *form* them.

Furthermore, because the theories behind the tool's design were incomplete, people used the Whyline in surprising ways. For example, one of the participants in my evaluation study had used the Whyline a few times and it had pointed out some of the assumptions she had made about what happened while her program was executing. The next time she began to ask the tool a question, she hovered over the "Why" button, but said, "I don't even need to ask. I think I made the same assumption that I did last time." The tool was introducing participants to the very same notion of assumptions that had inspired the Whyline's design—in this sense it *embodied*, *validated* and even *elaborated* the theories that motivated it.

Another tool I was involved in designing, Jasper [1], followed a similar trajectory. The original idea was inspired by a finding that developers gathered many little pieces of a program for a particular task, but had no way to gather them together in a single place [5]. This led to navigational overhead, as they navigated back and forth between code snippets that were distributed amongst several files. While my colleague designed and implemented the tool, I was busy at Microsoft, watching developers do work. As I watched them consult each other for knowledge about what code was relevant to a bug report or feature, I realized that being able to gather together snippets was not only helpful in reducing navigational overhead, but a fundamentally important way to share the *context* of one's task with coworkers. This new understanding changed the purpose of the tool in my mind: rather than just a navigational aid, it was a medium for externalizing and sharing task context. Had I noted invented the idea, this realization would not have been possible.

4. EXPLAINING THROUGH EVALUATION

Understanding and invention are vital ingredients in improving software engineering, but they are little without a notion of success to guide our research efforts. Is my tool helpful? Is it effective? Does it improve productivity? Will people adopt it? These are the criteria by which we separate successful and unsuccessful design. Unfortunately, unlike success measures in other engineering disciplines, these are difficult to measure and not necessarily the same as those which users of our tools employ to evaluate tools.

One view on this issue is that "good" and "productive" should be defined by what a *developer* thinks is good and productive. Who better to evaluate the utility and fit of a tool than the people most familiar with a job's complexities? The challenge of this approach is that as researchers, we must often settle for creating prototypes rather than fully functional and usable products. This makes it difficult to know whether problems observed in evaluations are due to the tool's incompleteness or some underlying inadequacy.

Of course, a measure based on developers' reactions also suffers from bias, subjectivity, and considerable variation. There may be absolute measures of success that avoid these problems. For example, to what degree did a team create what it *intended* to create? Did the rates of information acquisition and decision making increase? Did the right quality attributes improve with the intervention? Although such measures are extremely difficult to compute, they may be necessary to pursue if we wish to clearly articulate the merits of our ideas to ourselves and to the world.

Whatever the merits of our measurements or the results of our evaluations, the key result of these studies is the elaboration of our explanations. By completing this loop between design and understanding, we inevitably improve the designs in our minds.

5. CONCLUSIONS

To support the social side of software development—or more appropriately, to decide whether to do so and why—researchers must explain why developers rely on each other in the ways that they do. As we rise to this challenge, let us remember that the *diversity* of our ideas, methods, skills and experiences are our greatest strength.

6. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under NSF grant IIS-0329090 and as part of the EUSES consortium under NSF grant ITR CCR-0324770. The first author was supported by an NDSEG fellowship.

7. REFERENCES

[1] Coblenz, M., Ko, A. J., Myers, B. A. (2006). Carnegie Mellon University, CMU-HCII-06-107.

[2] Gutwin, C., Penner, R. and Schneider, K. (2004). Group Awareness in Distributed Software Development. CSCW, Chicago, IL, 72-81.

[3] Hertzum, M. (2002). The Importance of Trust in Software Engineers' Assessment of Choice of Information Sources. Information and Organization, 12(1), 1-18.

[4] Ko, A. J. (2003). A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment. CHI, Fort Lauderdale, FL, 1036-1037.

[5] Ko. A. J., Myers, B.A., Coblenz, M. and Aung, H. H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. Transactions on Software Engineering, to appear.

[6] Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. VL/HCC, Rome, Italy, 199-206.

[7] Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. CHI, Vienna, Austria, 151-158.

[8] Sevenster, A. (1992). Collected Works of A.M. Turing: Mechanical Intelligence, Volume 1. Elsevier, New York:NY