Software engineering is a decidedly human process; I want to understand the human factors that affect this process and design technologies to ensure its success. To achieve this, I combine knowledge and techniques from both **Software Engineering** and **Human-Computer Interaction** (HCI), grounding my grounding my ideas, studies, and tool designs in empirical observations of actual people, whether software developers, end-user programmers, or just regular software users.

PROGRAM UNDERSTANDING

My dissertation work concerns software **debugging**: both what makes it difficult and how to make it more successful. To this end, I have conducted a series of studies to explore cognitive, social, and organizational aspects of software development. I began by investigating the cognitive causes behind the introduction of errors into code [14]. This work led to an exploration of the technical barriers that prevent people from identifying coding errors, revealing that most developers begin debugging activities by asking "why" questions, 60% of which are "why didn't" questions [10,12]. I then explored the use of Eclipse by Java programmers on a variety of debugging and enhancement tasks, finding that developers spend 30% of their time re-navigating to relevant code [4,10]. I also also assessed the impact of workplace interruptions, building statistical models of developers' interruptibility [9]. I then extended the scope of my studies to one of the largest software development companies in the world, using ethnographic techniques to explore diagram use in the software industry [3] and to identify crucial information needs in collocated software development teams [2].

While I learned a great number of things from these studies, the most important insight for debugging was this: the central limitation of today's tools is that they all require a developer to *guess* about the cause of a program's failure based on extremely limited data. For example, if a program was supposed to produce some output in response to a command, but does not, a developer's only recourse is to first guess about what caused the problem and only then can they acquire data to investigate the cause, using breakpoints, print statements, or other tools. Because these guesses are informed mainly by intuition, experience and the perception of a program's failure, and not by the program's actual execution, the guesses are often wrong and require a long period of refinement; in fact, anywhere from **50-90% of developers' initial guesses are completely incorrect** [4,10]. Guesses can also lead to *new* bugs [14], as well as inaccurate knowledge about the runtime behavior of the program [2,4].

DEBUGGING BY ASKING QUESTIONS

In response to my empirical findings, I invented the concept of a *Whyline*, which allows a developer to choose *why did* and *why didn't* questions directly about a program's output (or lack thereof). A Whyline uses the program as a specification of what output is possible. In response to a question, the tool provides an answer using static and dynamic program analyses, displaying only the code and events relevant to the question. Thus, instead of guessing, developers can immediately gather concrete data about the causes of the symptoms they see.

I have implemented Whyline prototypes for a variety of contexts. The first supported programs written in the Alice programming language, an educational environment for creating interactive 3D worlds. In this prototype, the system automatically identified relevant questions about the program's output, from which users could choose using a global "why" menu (at the top of Figure 1). Compared to the traditional debugging tools, novice and expert programmers who used the Whyline in a lab study **found bugs 8 times faster** and **completed 40% more tasks** [13].

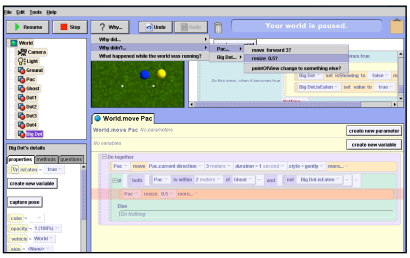

***Figure 1.* The Whyline for *Alice*.**

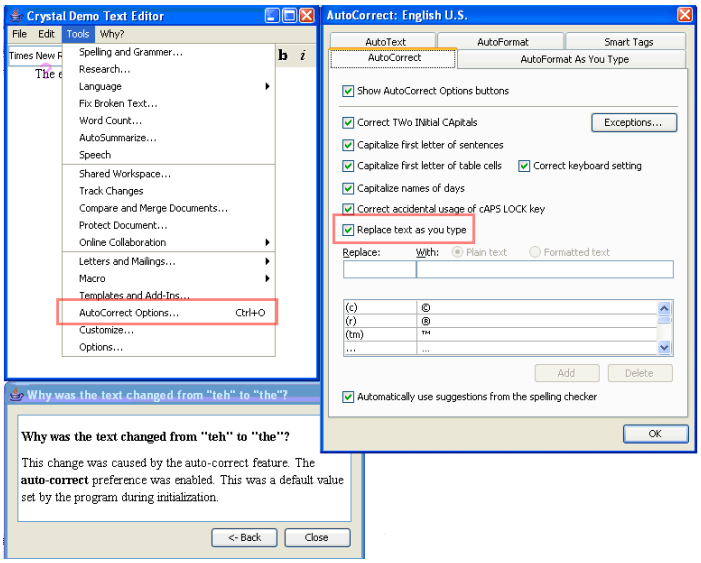

***Figure 2*. The Crystal text editor, supporting questions.**

Motivated by the success of the Alice prototype, we then explored the concept in the domain of end-user software, in particular word processors [8]. In this version, called Crystal, when users of a text editor are confused about the state of their document or the application (for example, wondering why the editor automatically corrected a misspelled word), they simply click on the word and choose the question, "why was this text changed?" The system knows to include this question in the "why" menu because of an augmented command and undo history, which allows a developer to support questions about each command. In response to the question, the application analyzes the data dependencies within the application's code and runtime state to discover that the correction occurred because the editor's "replace text as you type" checkbox was checked. It then shows this answer by opening the preferences dialog that contains the checkbox, and highlighting it (Figure 2). In a comparison to normal word processors, users with this tool were able to complete **30% more tasks, 21% faster**, than those without [6].

My latest exploration of the *Whyline* concept explores issues of scale by supporting questions about *Java* programs with textual and graphical output [1]. Not only do Java programs pose the technical challenge of being more complex than those used in the other prototypes, but they also raise questions about the nature of output, for *anything* a program does can be considered output from some perspective. Therefore, I use a notion of *code familiarity* to derive and filter questions. For example, it would be unreasonable to support questions about the internals of an API, which a user of an API would know nothing about, unless the developer is debugging the API itself. The Whyline also chooses questions specific to the program being debugged by identifying layers of *primitive output* in a program's code (pixels, rectangles, console output, etc.), and then propagates this knowledge along the program's call graph to identify program-specific types of output (such as *button*, in the case of GUIs, or the paint stroke shown in Figure 3). This allows the Whyline to support both *why did* and *why didn't* questions about program-specific output (as seen in Figure 3).

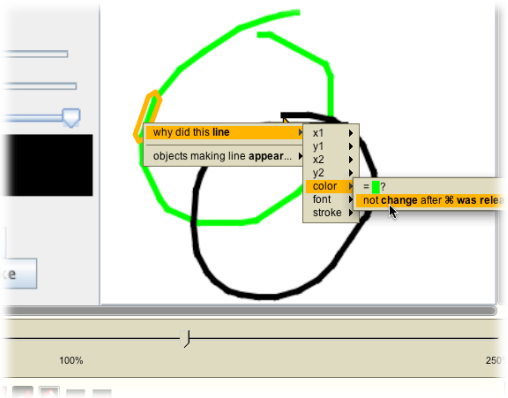

***Figure 3*. Question asking in the Whyline for *Java*.**

The Whyline for Java can support *why didn't* questions because it knows what output the program could potentially generate. However, in addition to identifying kinds of output, it also identifies concrete values that might be assigned to output-affecting state. For example, a button's *enabled* state typically affects its appearance on-screen. The Whyline discovers this by propagating its knowledge about primitive output along data flow paths through the program, identifying higher-level output-affecting program state. Therefore, the Whyline can support questions such as "Why was this button not enabled?" or and by finding uses of constants in assignments to output-affecting state, even questions such as "Why was this

panel not grey?" or "Why was this table's position not 0?"

Once the user asks a question, the Whyline answers it using a combination of call graph analyses, static and dynamic program slicing, and new algorithms that identify the space of possible explanations for why a particular line of code was not executed. This analyses are *time* and *identity* precise: they are specific to a particular point and a particular object in the program's execution. If a user asks about some aspect of a particular object on-screen at a particular time, the analyses identify why that particular aspect was or was not affected after that time. After computing the answer, the Whyline shows a visualization that combines data and control flow events, directly tied to their corresponding source code (see Figure 4).

Evaluations of the Whyline for Java have shown that people with no programming experience are **three times faster** at finding the cause of a bug **than experts using Eclipse**. In an ongoing study, I am evaluating the tool on a quarter-million line open source application, and expect to find similar results.

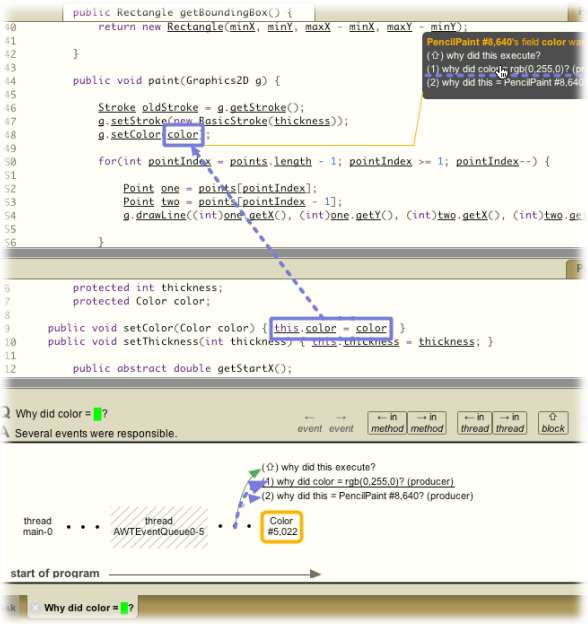

***Figure 4*. The source code and visualization shown for an answer in the**

OTHER PROJECTS

In addition to focusing on debugging, I have pursued several other projects, many in collaboration with undergraduate and masters students. Some of these have been technological advances. For example, I have invented a new kind of code editing environment called Barista that not only enables new ways to edit and modify code, but also new was to view code and context-relevant metadata [7]. Barista is implemented in a new programming language called Citrus, which supports novel language constructs designed to simplify event-based application development [8]. I have also explored ways of helping users identify errors in spreadsheet environments by propagating labels through spreadsheet formulas [11].

FUTURE RESEARCH

The common thread in all of my work is a deep fascination with software and the people who make it. In future research, I will strive both to improve the means by which people create software and also attempt to understand the role of software in society, influencing the process of software engineering accordingly.

First, there is a great deal of knowledge and practice in HCI research that has little presence in the Software Engineering research community. For example, in HCI, there are a number of sketching technologies for exploring software design ideas. My studies suggest that similar tools may be helpful in supporting software architecture and design conversations between software developers [4]. My studies have also identified the crucial nature of describing and documenting design rationale in industrial software engineering [3, 4]. HCI research has identified several design strategies for persuading users to provide data of future value; I hope to apply these strategies to the problem of capturing software design rationale. There are also a number of fundamental questions in software engineering about notions of requirements, testing, correctness, and design rationale, whose answers are limited by traditions of formalism. I am interested in augmenting these formalisms with the vast array of empirical and observational methodologies in HCI.

Second, the discipline of HCI itself could benefit from the techniques and technologies employed in Software Engineering and Computer Science. For example, there are a number of well-understand program analyses that could be adapted to support usability and interaction concerns, such as the importance of feedback and the challenge of identifying rare, but catastrophic interactive situations in safety-critical applications. Software in general suffers a notorious lack of memory: a program that crashes once will crash again, with no apparent knowledge of its previous failure. I am interested in designing languages and runtimes that allow programs to learn from their mistakes and adapt to their failures, much like just-in-time compilers adapt to new information about program usage to improve performance.

IMPACT

Throughout all of this work, I expect to maintain a strong presence in both the Software Engineering and HCI communities, making significant contributions to each field and to practice. I have already had considerable success in bridging these academic fields, assisting in writing interdisciplinary grants to Adobe, Microsoft, SAP, and NSF, all successfully funded. Together, my papers have been cited by others over two hundred times by more than fifty authors spanning HCI and software engineering.

My influence also extends into practice. My work on the Whyline has received international press both in print and online. I have demonstrated my technologies to Visual Studio and Office teams at Microsoft, to several teams at Adobe, Inc., and to the virtual machine team at Sun Microsystems, all of whom are interested in integrating my ideas into their future products. I am also working with several open source developers to integrate features of my Citrus programming language into their environments and possibly future versions of Python.

REFERENCES (CHRONOLOGICAL)

1. Ko, A.J. and Myers, B.A. (2008) Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *To appear at the International Conference on Software Engineering (ICSE), Leipzip, Germany.*
2. Ko, A. J. DeLine, R., Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *International Conference on Software Engineering (ICSE)*, May 20-26, 344-353.
3. Cherubini, M., Venolia, G., DeLine, R. and Ko. A. J. (2007). Let's Go to the Whiteboard: How and Why Software Developers Draw Code. *ACM Conference on Human Factors in Computing Systems (CHI)*, April 28-May 3, 557-566.
4. Ko. A. J., Myers, B.A., Coblenz, M. and Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12), 971-987.
5. Ko. A. J., Myers, B.A., Chau, D. H. (2006) A Linguistic Analysis of How People Describe Software Problems. *Visual Languages and Human-Centric Computing (VL/HCC)*, Brighton, United Kingdom, September 4-8, 127-134.
6. Myers, B. A., Weitzman, D., Ko, A. J., Chau, D. H. (2006) Answering Why and Why Not Questions in User Interfaces. *ACM Conference on Human Factors in Computing Systems (CHI)*, Montreal, Canada, April 24-27, 397-406.
7. Ko, A. J., Myers, B. A. (2006) Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views for Code Editors. *ACM Conference on Human Factors in Computing Systems (CHI)*, Montreal, Canada, April 24-27, 387-396.
8. Ko, A. J. and Myers, B. A. (2005). Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data. *ACM Symposium on User Interface Software and Technology (UIST)*, Seattle WA, October 23-26, 2005, 3-12.
9. Fogarty, J., Ko, A.J., Aung, H.H., Golden, E., Tang, K.P. and Hudson, S.E. (2005). Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility. *ACM Conference on Human Factors in Computing Systems (CHI)*, Portland OR, April 2-7, 331-340.
10. Ko, A. J., Aung, H., and Myers, B. A. (2005). Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. *International Conference on Software Engineering (ICSE)*, St. Louis, MI, May 15-21, 126-135.
11. Coblenz, M. J., Ko, A. J., and Myers. B. A. (2005). Using Objects of Measurement to Detect Spreadsheet Errors. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September 23-26, 314-316.
12. Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Rome, Italy, September 26-29, 199-206.
13. Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. *ACM Conference on Human Factors in Computing Systems (CHI)*, Vienna, Austria, April 24-29, 151-158.
14. Ko, A. J. and Myers, B. A. (2003). Development and Evaluation of a Model of Programming Errors. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, October 28th-31st, 7-14.

Although I love research, teaching is the reason I pursued a Ph.D. My first experience was in fourth grade, when my teacher, prompted by my math skills, asked me to write an extra credit assignment on long division for my classmates. I asked what the students were struggling with and crafted a number of questions to target these difficulties. I convinced my teacher to set some time aside for the class to work on it and let me help those that needed help. For the first time in my young life, I learned that I could not only share my expertise with my peers, but that I had the patience and acumen to share it well.

Since then, I have pursued every opportunity I could find. I have tutored dozens of students, helping them prepare for the SATs, understand algebra and calculus, carefully craft written arguments for essays, among other things. As president of the Oregon State University chapter of the Association for Computing Machinery, I created a computer science tutoring program, recruiting nearly thirty computer science students to volunteer their time to help students struggling in introductory computer science courses. I coached teams of students competing in the ACM Programming Contest. I created a computing cluster of abandoned computers to help my peers learn more about parallel computing. I even created a campus-wide, and eventually statewide software engineering competition, recruiting local companies like Intel, Microsoft, and HP to not only sponsor the competition, but provide employees to mentor the teams who competed and judge their submissions. The contest continued for several years.

As a Ph.D. student, my affinity for tutoring has matured. With the aid of my advisor, I have advised six students on research projects and one through his undergraduate and masters theses. I have become known among the younger Ph.D. students in my department as something of a mentor: dozens of my peers come to me for advice about their research directions and struggles with their advisors. In the past year, I played this role more formally as the official ombudsperson for our department, both advising students through conflicts and crises and relaying concerns to our faculty. I also began a tradition of having a Friday lunch, where students meet and dine out, discussing the perks and perils of being a Ph.D. student. Throughout all of these experiences, I have found my individual relationships with students to be some of the most rewarding professional relationships in my career.

In addition to helping students individually, I have also pursued several classroom opportunities. When I was a sophomore at Oregon State University, many of my peers wished the department would offer small special topics courses on contemporary computing topics, such as 3D graphics, gaming, and the internet. I wanted to find a way to allow students to teach these courses for credit, as well as have students take them for credit, as part of our computer science curriculum. To support my case, I deployed a formal survey of my peers' interests, and reported the data to my department chair and college dean. They agreed that it was a fantastic idea and I volunteered to teach the first course. Thus, in my second year as an undergraduate, I designed and taught a class of my peers on the topic of 3D rendering algorithms, lecturing weekly, writing tests, and even holding office hours. Although the university eventually decided that the program would dilute certain university statistics and therefore stopped it, for a whole year, computer science undergraduates at Oregon State thrived on these student-taught courses, fostering an active and enthusiastic community of computing culture.

As a Ph.D. student, I have continued to teach, assisting in teaching three courses, more than required by my program. I have found ways to combine my enthusiasm for teaching with my research program, spending a semester assessing learning barriers in students' efforts to learn to program. This work led not only to publications, but insights into the rising attrition in undergraduate computer science departments and the lack of interest in computing careers in society. In the future, I would like to continue to combine my research interests in software engineering and computer science education with my time in the classroom, developing new kinds of educational technologies and applying my research findings and inventions to improve students' understanding of their discipline. To this end, I would be excited to teach not only courses on software engineering, human-computer interaction, and programming languages, but also introductory courses in computer science.