Research, Teaching, and Service Statement

Research

As a society, we are creating a software-based world that few of us can comprehend. Users wrestle with intricate and endlessly changing software applications, asking many questions but finding few answers. Aspiring programmers, faced with the staggering complexity of modern programming languages and tools, give up learning all too early. And professional software developers, despite their expertise, often struggle to understand the code that they and others have created, making software slow to test, debug, and evolve. As we further leverage software to solve grand challenges in health, energy, education, security, and science, I predict that program understanding challenges such as these will become an increasingly significant barrier to human progress.

I seek to lower these barriers through **interactive program understanding tools** that enhance people's ability to understand why programs behave as they do, so that they change its behavior by modifying its code or configuration. I view program understanding as a task in which people build mental representations of the relationships between program inputs, program outputs, and the source code that maps one to the other. What makes this task difficult is that input, code, and output are inherently disjoint in time and space: developers must search for causal relationships across disparate program elements executing at imperceptible speeds; end users must somehow infer these same causal relationships from input and output alone, with no access to or ability to understand a program's internal logic.

I invent interactive program understanding tools that help people search for and understand the causes of software behavior.

My interest in program understanding has emerged from a lifelong interest in lowering barriers to people using software to express themselves and solve problems. As a teenager, I created digital tools to help my friends create art, music, and games, gaining firsthand experience with the challenge of designing software that people understand. As an undergraduate, I investigated testing and debugging tools for spreadsheets users with no formal training in computer science. As a Ph.D. student, I invented the *Whyline* debugging tool [15,16,18] and published studies of the barriers that end users, novice programmers, and professional developers face when reasoning about software behavior.

In the past five years as an assistant professor, I have broadened my focus to four new aspects of program understanding: (1) helping developers find and fix bugs by automatically discovering relationships between code and output, (2) helping developers manage and interpret users' descriptions of software problems, (3) helping end users answer questions about software behavior through new forms of help retrieval, and (4) helping aspiring programmers acquire program understanding skills through a new genre of educational debugging games. Throughout this work, I have also contributed a more robust understanding of how program understanding affects society through descriptive social science on large data sets of software problems from open source [7,14,11], social media [10], and the news [8].

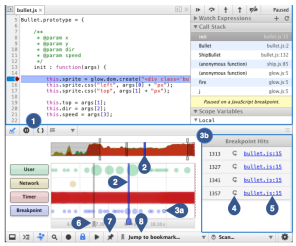
My approach to this research is both iterative and interdisciplinary. I conduct social science on how people describe and reason about software behavior, including how designers, developers, and users interact with each other to fix software problems and respond to its failures. I then use the discoveries from these studies to invent and evaluate new technologies that facilitate software understanding. My work is also highly collaborative: I conduct most of my research with Ph.D. students, helping them to perform research within the framing and direction that I provide. My publications from the past five years, which I discuss throughout the rest of this section, are therefore a mix of first-authored and student-authored work.

In the past five years I have worked closely with four fantastic Ph.D. students, co-authoring much of my research with them as first authors.

Finding and Fixing Bugs by Linking Code and Output

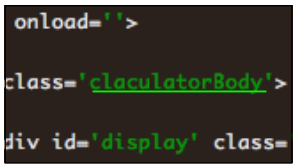
A fundamental challenge in evolving software is quickly and accurately understanding a program's behavior. Whereas most research in this space focuses on automating ths understanding through formal verification, my research has focused on interactive program analysis tools that help developers acquire knowledge about program behavior.

One example is *Timelapse* [1], designed by my co-advisee Brian Burg. The key insight behind *Timelapse* is that program state is ephemeral, making it difficult to observe, control, and therefore understand. *Timelapse* can record replayable input traces that allow developers to navigate to arbitrary points of an execution and inspect program behavior, with virtually zero recording or replay overhead. *Timelapse* is the first to do so in an interactive, on-demand manner that integrates seamlessly with existing tools, such as breakpoints and logging. This makes it possible to adapt powerful debugging tools such as my dissertation work on *Whyline* [18,16,15] into lightweight, on-demand tools that minimize a developers' need to plan their tool use in advance. In our evaluations, we showed that *Timelapse* changes how developers think about program execution, shifting it from something that moves uncontrollably forward to something that can be explicitly navigated, indexed, and searched to support program understanding.



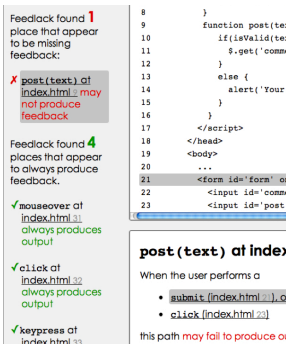
Timelapse supports interactive replay of web applications.

Program execution is not only ephemeral, it is also highly variable, making it difficult to comprehensively understand. I have explored interactive static analysis techniques that help developers acquire this comprehensive view. The first of these projects is *Cleanroom* [20], which I designed, prototyped, and then evaluated in collaboration with Jacob Wobbrock. *Cleanroom* addresses a limitation of dynamically typed programming languages, in which there are no guarantees that a name will refer to a value at runtime. I observed that programs use names both to define *and* refer to values, and so names that only appear once in a program are likely to be wrong (because they either define but not use a value or use but not define a value). *Cleanroom* [20] exploits this insight, providing keystroke-level, immediate feedback warnings about singleton variable names. In our evaluation, we found that *Cleanroom* detects many legitimate errors with few false positives and that it helps developers find these errors more quickly and reliably [20].



Cleanroom provides keystroke-level feedback about invalid names for dynamically typed languages. It won **best paper** at VL/HCC 2010.

Like *Cleanroom*, *FeedLack* [21] also helps developers acquire a comprehensive view of program behavior, but it searches for a more complex type of defect: scenarios in which a user interface provides no response to user input, such as error conditions in which buttons clicks are ignored and other edge cases that fail to provide feedback. The insight behind *FeedLack* was that this design principle—that user interfaces should always provide feedback in response to user input—is something that can be operationalized and analyzed automatically, by verifying that each code execution path that originates from user input should produce output. *FeedLack* finds execution paths that violate this property in web applications, highlighting paths that do not make a visible change to the web page content or appearance. Across both HCI and software engineering research, *FeedLack* is one of the first automated verifications of an interactive aspect of user interface usability.



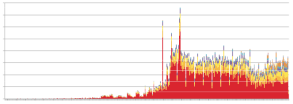
FeedLack finds scenarios in which a user interface fails to provide feedback in response to user input.

I have made many other contributions in this space through collaborations. My work with student Kayur Patel on *Gestalt* has explored how to help developers reason about the link between data and *machine learned* program behavior [29]. With Margaret Burnett and her students at Oregon State University, I have also worked on tools for troubleshooting machine learned programs such as spam filters [22,23]. I have also collaborated with Microsoft UX Designer Yann Riche on studies of the domain knowledge required to successfully use software APIs [19]. With over a dozen researchers, I have also synthesized a decade of research on end-user software engineering tools into a frequently cited survey paper [13] and contributed research methods for evaluating software engineering tools [9].

Interpreting Descriptions of Software Problems

Every day, millions of people report software problems in bug reports, support requests, and social media, writing vague, incomplete, and sometimes incorrect natural language descriptions of problematic program behaviors. Whereas prior work has investigated automated techniques for mining and processing this user feedback, my research has focused on understanding the content of this feedback, the contexts in which users contribute it, and the meaning that developers extract from it.

Most of my work on software problems has been descriptive, analyzing contexts in which people describe software problems to software teams, who must then interpret, triage, and manage these descriptions. With the help students Parmit Chilana and Mike Lee, I have studied the problems that users report to open source projects [7,14], the arguments between users and developers in bug reports about application behavior [11], the support requests received by software teams and how they process them [12,27], and the changes in reported software problems between beta and release versions of software [28]. I have also analyzed how teams leverage user feedback to improve software, studying the data-driven design processes at Facebook [2] and the design challenges in domains such as aviation, bioinformatics, and health care [5,6]. I have also examined popular press on critical software problems, analyzing over 500,000 news articles published since 1980 [8].

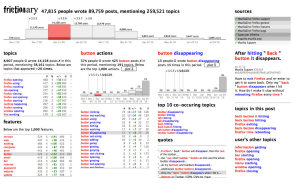


The frequency of critical software problems reported in global news since 1980 based on a corpus of 500,000 English news articles.

These studies have revealed much about how developers, organizations, and society manage information about software problems:

- The imprecise nature of natural language leads to many inefficiencies in the description, reproduction, diagnosis, and resolution of software problems [7,14], causing issue tracking systems, technical support forums, and support knowledge bases to grow unbounded with redundancy [12,27]. Developers are desperate for more precise ways of capturing and observing problematic program behavior (e.g., as made possible by *Timelapse* recordings [1]).
- Because end users struggle to observe, analyze, and comprehend the causes of problematic software behavior, most simply abandon their tasks, avoiding complex diagnostic work [27]. Those that do troubleshoot problematic behavior often rely on poorly curated and vetted web resources, causing additional problems that further confound problem diagnosis.
- To users, what constitutes "correct" software behavior depends less on the developers' intents and more on how they have appropriated software. This disparity between intent and appropriation, and the inherent power imbalance between designers' and users' ability to change software, leads to heated arguments about what constitutes the "right" design [17,14].

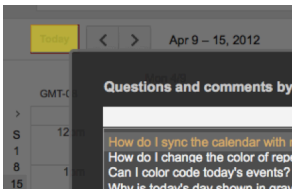
While these findings are important in their own right, they also suggest the need to help developers make sense of the endless stream of support forum posts, tweets, and e-mails on the web. To facilitate this, I invented *Frictionary* [10], which aggregates descriptions of software problems from the web over time. The key insight was that software problems could be discovered and organized by using the words that appear in a user interface. *Frictionary* indexes these user interface labels, using the index to mine the web for sentences that describe software problems. It then extracts, normalizes, and aggregates these sentences into human-readable problem descriptions, providing extensive time-based statistics and visualizations of software problem trends. I evaluated Frictionary on over 100,000 posts about Firefox, showing that the technique reliably aggregates descriptions of software problems and that Firefox designers and technical support staff found it useful for discovering new problems [10].



Frictionary mines the web for software problems described by end users in support forums, helping software teams discover unreported software problems.

Explaining Software Behavior to Users

One reason for the software problem reports discussed in the previous section is that everyday, millions of users struggle to understand inscrutable error messages, confusing features, complex workflows, and poor feedback pervasive in modern software. Only the most persistent of users try to overcome these problems, and even these users must rely on the web, writing and rewriting queries until they find a potential answer buried in a discussion forum. With my student Parmit Chilana and her co-advisor Jacob Wobbrock, we envisioned a simpler and more effective approach to finding software help called *LemonAid* [4]. The key insight behind *LemonAid* is that it is much easier for people to point to a problem than it is for them to describe it accurately; we found empirically that people are also be more likely to point to the same things for the same problem than they are to describe a problem in the same way. *LemonAid* exploits these insights, allowing users to select a label or image in a user interface as their query. *LemonAid* then uses the selection and its context to retrieve previously asked questions that were asked in a similar context. This approach effectively crowdsources a mapping between user interface elements and relevant Q&A, placing help content where users expect to find it, rather than in a decontextualized help resource that is difficult to search and browse.



LemonAid crowdsources a mapping from user interface elements to Q&A, enabling end users to find help by recognizing relevant elements rather than describing them.

Our evaluations of *LemonAid* have involved both a large scale simulated user community on Amazon's Mechanical Turk and a 6-month field deployment to four software teams at the University of Washington, including the UW Libraries [3]. We found that *LemonAid* is a highly effective search mechanism, retrieving relevant Q&A in the top 5 results for over 80% of help requests. We have also found that both users and site maintainers found *LemonAid* to be superior to web search, live chat, phone support, and knowledge base searches for finding and provisioning software help. Site owners also found the analytics that *LemonAid* provided a unique view into the problems that users are having, without having to explicitly investigate users' perspectives with traditional usability methods [4]. The *LemonAid* algorithms are patent pending and Parmit Chilana, the student leading the work, will be starting as an Assistant Professor at the University of Waterloo in Fall 2013.

Learning Program Understanding

The fourth research thread I discuss focuses on *teaching* program understanding skills rather than facilitating them. This focus emerged from an investigation I performed of 80 young adults' first encounters with computer programs, which I solicited in the form of "code autobiographies" [17]. I discovered that many young adults' first encounters with code were not in the classroom, but in informal settings, such as learning to write a script to customize a game or creating a web page to represent a social group online. Unfortunately, once these young adults began writing their first program, they were immediately faced with debugging their mistakes, which for many was an insurmountable challenge. Many expressed that they had similar experiences in their first computer science classes, where they were taught much about how to write programs, but little about how to read and understand them.

With my Ph.D. student Michael Lee, we therefore began to explore the benefits of explicit instruction on program understanding. Our approach was to create an educational game that distilled the task of debugging programs into a simple game mechanics. Our game, called *Gidget* [26], gives players a series of defective programs written by Gidget the robot. Gidget explains that it does not know what is wrong with the programs and asks the player to help fix them. By completing a series of progressively more complex debugging puzzles, players learn a series of program understanding skills, learning the semantics of Gidget's programming language in tandem. The key principle behind this design was to convert programming from what is a traditionally *creative* task to a *comprehension* task, asking learners to understand program behavior rather than both define *and* understand it.

Gidget challenges players to find and fix defects in programs. Our studies examine the effect of the game design on learning and engagement.

We have already learned much about the benefits of the game. Over 2,000 have played it through a series of online lab studies, with many expressing that they did not know programming could be so much fun [26]. In observing people play the game in the lab, we observed that unlike creative tasks where each barrier encountered in expressing their creative vision was personal failure, in *Gidget*, each barrier is simply a puzzle to solve that is known to have some solution.

We have also discovered several important aspects of the game's design on learning and engagement. Across three online experiments, we have shown that giving the robot a face and having it use personal pronouns such as "I" and "we" led players to voluntarily complete more than twice as many levels as players who interacted with a faceless robot [26]. We have found that the more purposeful a goal, even if that purpose is meaningful only inside the context of the game, the more learners attend to the game's instruction and the longer they play [25]. Our most recent study has shown that explicit in-game assessments of learning not only increase how long learners play the game, but also the speed with which players learn [24].

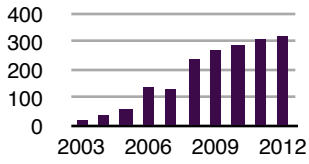
These results are building an evidence-base for the design of other informal computing education learning technologies, as well as a debugging-first approach to teaching programming, complementing existing research which has focused primarily on teaching people how to write programs, rather than read them. In the coming year we will evaluate the game in a series of summer camps for high school girls, begin a 3-month longitudinal field study of learning outcomes, and publicly release the game.

Contributions, Awards, Impact, and Funding

Taken together, my decade of research contributes an examination of program understanding and the interactive tools that can facilitate it. Some of these technologies help developers find and fix defects (*FeedLack*, *Cleanroom*, *Frictionary*), some help developers better control and inspect the execution of programs so that they may better understand them (*Timelapse*, *Whyline*), and some help non-developers better understand how software works (*LemonAid*, *Gidget, Crystal*). These tools, and more broadly, the genres of tools that they exemplify, help envision a world in which program understanding is a more accessible skill, streamlining people's ability to successfully create and use software.

In examining program understanding, I have also contributed several insights about the nature program understanding, its role in software development work, and more broadly, the importance of studying the human and social aspects of software engineering. Many of my most frequently cited papers are cited for their careful attention to the interaction between people and their tools, and the subtle ways in which software development is a social activity as much as a technical one.

My contributions have been recognized as rigorous, high quality, and significant. In the past 11 years, I have received 3 best paper awards at CHI and ICSE (two top conferences in HCI and software engineering), 2 best paper awards at VL/HCC (the main conference in my research area of interactive developer tools), and 1 best paper honorable mention at CHI. The first paper I wrote as a Ph.D. student in 2003, which proposed a framework for reasoning about software errors, was also recently nominated for a Most Influential Paper award at VL/HCC. My work has been cited over 2,100 times in the past decade by hundreds of researchers across HCI and Software Engineering. My *h*-index (the largest number *h* such that *h* publications have at least *h* citations) is **21** as measured by Google Scholar, which is similar to that of many of my recently tenured peers, including Scott Klemmer (24, Stanford), Jeffrey Heer (27, Stanford), Gillian R. Hayes (21, Irvine), and Martin Robillard (25, McGill). Nine of my articles have been cited more than one hundred times.



Citations to my publications per year, according to Google Scholar.

My research has also been impactful in industry. I frequently consult with Microsoft's Developer Division, synthesizing insights from my research and others' for Microsoft product developers and designers. Some of my early ideas on the design of integrated development environments from 2005 are now part of Microsoft's *Debugger Canvas* plugin for Visual Studio. With Ph.D. student Parmit Chilana and colleague Jacob Wobbrock, I have also co-founded a company based on *LemonAid*. Over a dozen companies are using our technology to offer software help to hundreds of thousands of web site visitors. We have already raised $500,000 in venture capital investment, which will allow us to grow the company and further disseminate our research to the world. I view technology transfer such as this an essential part of my responsibilities as a public intellectual, providing a much needed and often missing bridge between research and practice.

In my five years as an Assistant Professor I have raised **$4,417,047** in grants and gifts to support my research. Most of this amount ($4,192,446) has been through two competitive National Science Foundation Grants as lead PI (a CAREER grant, which supports work on *LemonAid*, and a CE21 grant, which supports work on *Gidget*) and one as a Co-PI (an HCC Large grant, which supports new work on programming with variations). The remainder of my funding has come from gifts from Microsoft Research and Google Research, a competitive UW Royalty Research Fund grant, and several smaller grants to fund doctoral consortiums, workshops, and commercialization efforts.

Future Research

As the world becomes more complex, so will software, further compounding challenges in software understanding. Machine learning, big data, cloud services, and pervasive sensors are only part of this increased complexity. The growing weight of legacy systems, the rapid change in consumer applications, and the proliferation of smartphones with access to anything, anywhere will also increase these complexities, making the task of understanding how software works even more formidable and necessary than ever.

In my future work, I want to invent interactive program understanding tools that help humanity exploit these advancements. I want to invent software that can explain itself to people rather than requiring people actively seek explanations. I want to create an ecosystem of accessible educational games that not only teach program understanding effectively but also *efficiently*, empowering everyone who wants to understand computing to do with minimal barriers. Through these inventions, I want to begin to create a science of human-computer interaction that bridges our deep understanding of human cognition with our burgeoning knowledge of software architecture, creating a deeper understanding of how low-level software implementation choices in information technology affect high level interactive experiences. I believe these efforts are essential not only for enabling increases in software complexity, but also for humanizing what are often incomprehensible software systems.

Looking beyond tenure, I also plan to investigate research questions that require more time, resources, and planning, shifting from a research program that has been defined by individual papers, to a research program defined by a broader, more lasting vision. I am particularly excited about the vast array of open questions in computing education learning technologies, as they address a problem both significant to society and combine my lifelong interests in helping people better comprehend software. In the next 10 years, I anticipate creating a stronger presence in the computing education research community, conducting basic science on pedagogy and broadly impacting computing literacy through my inventions. I also hope to play a significant role in persuading the federal funding agencies in the U.S. to further invest in computing education research, which is still chronically underfunded apart from service efforts to broaden participation.

Teaching

My favorite teachers in life have all had one thing in common: every day, in small ways, they showed me that they personally cared about my learning. I have always aspired to achieve the same level of personal connection with my students and have worked hard to design my courses in a way to facilitate this. For example, in spearheading the design of our undergraduate Design Methods (INFO 360) and Collaborative Software Design (INFO 461) courses, I've planned classroom time to be primarily interactive and interpersonal, focusing on skill-development, team projects, individual feedback, and small group discussions about each class's big ideas. In all of these, I interact extensively with individual students and groups of students, posing and answering questions that are specific to each students' level of understanding, and developing relationships with each student that last beyond their time as students. In my Ph.D. courses, I treat students as advisees, working closely to tie course materials with their research. As an advisor to Ph.D. students, I collaborate closely with Ph.D. students as if they were my colleagues, helping them to develop into world-class independent researchers and supporting their own visions by seeking funding to support their research.

As a member of our teaching community, I frequently discuss teaching methods with colleagues in the iSchool and around UW, highlighting the pros and cons of flipping the classroom, discussing the merits and manifestations of peer instruction, and refining methods for teaching through team work. I read literature in learning sciences and education to improve my teaching and inform my research. I relish the challenge of designing courses that maximize learning and minimize burden and engage even the most disengaged of students. Since I began dreaming of being a teacher as a middle school tutor, my teaching responsibilities are not a distraction from research, but something that drives and informs the research I conduct.

Across the 15 courses I have taught in the past 5 years, my course evaluations are consistently above **4.5** on a **5** point scale. I was nominated for my school's PROF award, with my dean writing, "*We are very fortunate to have such an enthusiastic, knowledgable, open and innovative professor on our faculty*." In a recent survey of our Informatics undergraduates, I was described by several students as "*One of the best things about the Informatics degree*." Students also send me unsolicited praise; for example, I received this LinkedIn recommendation from a former undergraduate two years after his graduation:

> *Andy is a truly brilliant, innovative, and inspiring teacher. I took several classes of his, and each of them were instrumental in my college education and career today. My favorite was "Collaborative software design," a class that he pioneered within the Information School. He had the keen insight to develop the class after identifying the need for real-world job skills and experience within the Informatics curriculum. He taught us how software teams design and build products, and gave many of us our first taste of iterative design, software specifications, testing, design thinking, version control, and more. The lessons I learned from his class are still relevant today, and I'm immensely proud of the web-application I developed in the class. His enthusiasm and encouragement were inspiring, and helped push me to do some of the best work of my college career… Andy Ko is an incredible asset to the University of Washington and I'm immensely grateful that I got to learn from him!*

Others have had similar experiences in my design courses:

> *I had lunch yesterday with [the Principle UX Manager of Microsoft]. We talked about a number of things, but naturally the conversation became focused on the topic of design, particularly the design process. Everything, and I mean literally everything, were things that I had learned in INFO 360. Obviously I was very excited to be able to be conversant! Design is a powerful skill because of its scalability: you can design a single icon or an entire system*

using the same process. This isn't something easily taught but I think you did an excellent job, considering those ideas are still with me 2 years after INFO 360.

Comments like these remind me that teaching is not just about transferring knowledge and building skills, but also developing wisdom, confidence, and judgment to youth that are at the beginning of a lifelong career in learning.

As a teacher and a mentor, I still have much to learn myself. In the short term, I will continue to lead efforts to define undergraduate and masters level education in HCI and Design at the University of Washington. In the long term, I will continue to innovate in my teaching, especially as the Internet continues to transform where, when, and how people learn. With my research in computing education, I am especially excited to begin integrating my research and teaching efforts, bringing the learning technologies I create in the lab into my classroom.

Service

As a boundary spanning researcher, I view service as an opportunity to bridge the HCI, Software Engineering, and Computing Education research communities, providing a conduit for exchange between three communities with significant conceptual overlap but little interaction. To achieve this, I have joined program committees at both CHI and UIST (two of the top HCI venues) and ICSE and FSE (two of the top software engineering venues). I have also served on several other program committees that bridge HCI and Software Engineering, such as VL/HCC and the CHASE workshop. I have also been selected for several leadership roles, acting as a Sub-Committee Chair at CHI in 2012 and 2013 and Technical Program Co-Chair of VL/HCC in 2011.

I play a similar interdisciplinary role at UW. As a core member of our DUB cross-campus HCI consortium. I participate in faculty and Ph.D. student recruiting and inviting internationally recognized HCI researchers to speak at our weekly seminar. I have been key in helping to define, launch, and staff our new Masters in HCI and Design. Within the Information School, I have served as a central member of our Informatics committee, refining the degree's curriculum and defining it for our faculty and students. I have also championed design in our curriculum at all levels, helping our faculty to recognize design as a critical perspective in information science. This has led to curriculum changes in all of our academics programs and a better understanding of the relationship between HCI, design, and the other disciplines represented in our school. I have also voluntarily joined iSchool hiring committees on top of my normal service duties, helping with the iSchool's recent and extensive faculty hiring efforts. In 2012, I was elected to our Elected Faculty Counsel, working with senior faculty on decisions about our school's future directions.

Selected Works Cited All papers below were published since starting at UW in 2008

1. Burg, B., Bailey, R., Ko, A.J. & Ernst, M.D. (2013). Interactive record/replay for web application debugging, *ACM Symp. on User Interface Software and Technology.*
2. Chilana, P. K., Holsberry, C., Oliveira, F., & Ko, A.J. (2012). Designing for a billion users: A case study of Facebook. *ACM Conf. on Human Factors in Computing Systems, Case Studies.*
3. Chilana, P.K., Ko, A.J., Wobbrock, J. (2013). A multi-site field study of crowdsourced contextual help: Usage and perspectives of end-users and software teams. *ACM Conf. on Human Factors in Computing Systems.*
4. Chilana, P.K., Ko, A.J., Wobbrock, J. (2012). LemonAid: Selection-based crowdsourced contextual help for web applications. *ACM Conf. on Human Factors in Computing Systems*.
5. Chilana P., Wobbrock, J.O., & Ko, A.J. (2010). Understanding usability practices in complex domains: Implications for training the next generation of usability professionals. *ACM Conf. on Human Factors in Computing Systems*.

6. Chilana, P.K., Palmer, C., & Ko, A.J. (2009). Comparing bioinformatics software development by computer scientists and biologists: An exploratory study. *Workshop on Software Engr. for Computational Science & Engineering*.
7. Chilana, P.K., Ko, A.J., Wobbrock, J.O. (2010). Understanding expressions of unwanted behaviors in open bug reporting. *IEEE Visual Languages and Human-Centric Computing.*
8. Ko, A.J., Dosono, B., Duriseti, N. (*in review*). Thirty years of software problems in the news.
9. Ko, A.J., LaToza, T.D., & Burnett, M.M. (in review). A practical guide to controlled experiments of software engineering tools with human participants.
10. Ko, A.J. (2012). Mining whining in support forums with Frictionary. *ACM Conf. on Human Factors in Computing Systems, alt.chi.*
11. Ko, A.J. & Chilana, P.K. (2011). Design, discussion, and dissent in Open Bug Reports. *iConf..*
12. Ko, A.J., Lee, M.J., Ferrari, V., Ip, S., & Tran, C. (2011). A case study of post-deployment user feedback triage. *Int'l Workshop on Cooperative and Human Aspects of Software Engineering.*
13. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M.M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B.A., Rosson, M.B., Rothermel, G., Shaw, M., & Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys.*
14. Ko, A. J. & Chilana, P.K. (2010). How power users help and hinder open bug reporting. *ACM Conf. on Human Factors in Computing Systems.*
15. Ko, A.J. & Myers, B.A. (2010). Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology.*
16. Ko, A.J. & Myers, B.A. (2009). Finding causes of program output with the Java Whyline. *ACM Conf. on Human Factors in Computing Systems*.
17. Ko, A.J. (2009). Attitudes and self-efficacy in young adults' computing autobiographies. *IEEE Visual Languages and Human-Centric Computing*.
18. Ko, A.J. & Myers, B.A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. *Int'l Conf. on Software Engineering.*
19. Ko, A.J. & Riche, Y. (2011). The Role of conceptual knowledge in API usability. *IEEE Visual Languages and Human-Centric Computing.*
20. Ko, A. J. & Wobbrock, J.O. (2010). Cleanroom: Edit-time error detection with the uniqueness heuristic. *IEEE Visual Languages and Human-Centric Computing*.
21. Ko, A.J. & Zhang, X. (2011). FeedLack detects missing feedback in web applications. *ACM Conf. on Human Factors in Computing Systems*.
22. Kulesza, T., Stumpf, S., Wong, W., Burnett, M.M., Perona, S., & Ko, A.J. (2011). Why-oriented end-user debugging of naive Bayes text classification. *ACM Interactive Intelligent Systems.*
23. Kulesza T., Wong W.K., Stumpf S., Perona S., White R., Burnett M.M., Oberst I., & Ko, A.J. (2009). Fixing the program my computer learned: Barriers for end users, challenges for the machine. *Journal of Intelligent User Interfaces*.
24. Lee, M.J., Ko, A.J., & Kwan, I. (2013). The effect of assessments in on engagement and learning efficiency in discretionary computing education. *Int'l Computing Education Research Conference.*
25. Lee, M.J. & Ko, A.J. (2012). Investigating the role of purposeful goals on novices' engagement in a programming game. *IEEE Visual Languages and Human-Centric Computing.*
26. Lee, M.J. & Ko, A.J. (2011). Personifying programming tool feedback improves novice programmers' learning. *Int'l Computing Education Research Workshop.*
27. Lee, M.J. & Ko, A.J. (2012). Representations of user Feedback in an agile, collocated software team. *Int'l Workshop on Cooperative and Human Aspects of Software Engineering.*
28. Li, P., Kivett, R. Zhan T., Jeon S., Nagappan, N., Murphy, B., & Ko, A.J. (2011). Characterizing the differences between pre- and post-release versions of software. *Int'l Conf. on Software Engineering, SEIP.*
29. Patel, K., Bancroft, N., Drucker, S., Fogarty, J., Ko, A., Landay, J.A. (201). Gestalt: Integrated support for implementation and analysis in machine learning processes. *ACM User Interface Software and Technology*.