

RESEARCH STATEMENT EMERSON MURPHY-HILL

Referenced papers (e.g., [1]) listed in vita.

Undeniably, software systems are becoming increasingly complex for users, both in terms of the sheer number of constituent features offered and the intricacies of each individual feature. My research focuses on one complex software system: integrated development environments for software developers. The Eclipse Java development environment, for instance, contains thousands of different tools, many of which present the results of sophisticated analysis to the software developer. When vendors choose to make their development environments extensible, the complexity worsens as third parties add even more tools. But people's ability to deal with complexity remains relatively fixed, so that a great chasm has opened between what environments *are capable of doing* and what developers *actually do* with them. My research suggests that this chasm exists in software development because it is too difficult for developers to gain, maintain, and exercise knowledge of the capabilities of integrated development environments.

I will simply call this the *capability chasm*, a problem that I seek to answer in my research by pursuing the question:

How can developers better harness the capabilities of their tools?

In the remainder of this statement, I chronologically discuss my research in four sections. First, research prior to my dissertation helped me form several research principles that I have used ever since. Second, in my dissertation, I addressed the capability chasm for refactoring tools. Third, my current research explores the breadth of the capability chasm. Finally, my planned research will delve further into the chasm, both in development environments and beyond.

Pre-Dissertation Research: the Formation of Research Principles

My early, formative research experiences helped me recognize three research principles that have since influenced my approach to research.

Principle: Collaboration Expands Breadth and Depth. In my first research experience as an undergraduate, I worked on two problems with my advisor, Dr. Judy Cushing. For the first problem, in collaboration with Dr. Douglas Toomey (University of Oregon, geophysics), I built a system to make a specialized marine geology application accessible to scientists through the internet. For the second problem, in collaboration with Dr. Nalini Nadkarni (Evergreen, forest canopy research) and several other ecologists, I helped build a family of tools to help scientists work more effectively with databases, including a web portal, a template-based database generator, and a forest canopy visualization tool [7,32]. Addressing both of these research problems illustrated that collaboration helps my research explore more possibilities, expand broadly across domains, and extend deeply into important problems.

Principle: Solving Practical Problems Increases Impact. As a graduate student, I investigated a programming language concept called infopipes, a pattern for data streaming, similar to the pipe-and-filter pattern. In collaboration with Dr. Robert Bertini (PSU) from the Intelligent Traffic Systems Laboratory, I worked with Dr. Andrew Black (PSU, distributed systems and programming languages), Dr. Jonathan Walpole (PSU, operating systems), and another graduate student to investigate how infopipes could be used to build software that analyzes automobile traffic congestion [27,28]. In focusing on traffic congestion, this experience illustrated that addressing practical problems can have a substantial impact on the world.

Principle: Empirical Methodology Validates Solutions. In the Multiview research group with Andrew Black, I investigated the difficulties that programmers have when implementing cross cutting features in programming languages. The research group introduced a new language feature, called traits, which purported to make it easier to implement some kinds of cross cutting features. As someone unfamiliar with traits, I was asked to implement a significant software project with and without traits, and write up my experiences as the first empirical study of the language feature [5,33]. This research provided my first, though somewhat limited, experience with evaluating software engineering innovations empirically. Continuing my investigation of traits in collaboration with another graduate student, I developed an implementation of traits for the Java language in Eclipse, helping to define how the language feature could be used in a mainstream programming language [4,12]. In building new tools for programmers, it remained unclear if the average programmer would find Java traits usable. Thus, my hunger grew for stronger empirical methodology.

Dissertation Research: from Poor Usability to the Capability Chasm

After infopipes and traits, I investigated a kind of programming tool that I suspected did not meet programmers' needs. These tools, called refactoring tools, semi-automatically transform code on behalf of the programmer by changing the structure of code without changing its behavior. The evidence I collected suggested that (1) people were not using refactoring tools as much as they could, and (2) poor usability was contributing to that underuse. As a consequence, despite refactoring tools' ability to improve code quality, programmers were not harnessing the power of these tools. Motivating and solving this problem in several dimensions became my dissertation:

- I conducted a survey of 112 people, largely from industry, to determine why programmers choose not to use refactoring tools [15]. One of the major findings was that the tools are not flexible enough, that programmers do not understand how refactoring tools work, and that programmers believe that they can refactor faster by hand than they can by using a tool. Another major finding was that when refactoring tools are available, programmers estimate that they do not use the tools 32% of the time, on average.
- I postulated a theory that people more often refactor in short, frequent bursts, rather than infrequently and protractedly, a theory that helps explain why refactoring tools, which largely support protracted refactoring, are underused [6,12,23]. In collaboration with Chris Parnin (Georgia Tech, software engineering) and Dr. Danny Dig (MIT, software engineering), I later validated this theory in several contexts by analyzing the program interaction histories of 41 developers, the source-code history of a multi-million line program, the tool usage history of about 13,000 Java developers, and the refactoring logs from four developers on the Eclipse project [2,10,18]. In this study, by replicating experiments performed by other researchers, I also confirmed (and disconfirmed) several previously held assumptions about refactoring. One of the most surprising findings was that the four programmers studied did not use refactoring tools for 89% of the refactorings that they performed, even when those refactoring tools were available. At least two other institutions are currently conducting follow-up research based on the my results.
- By combining other researchers' work and my own observations of how programmers refactor, I synthesized a detailed model of how programmers use conventional refactoring tools [8]. This model helped me structure my dissertation and create innovative refactoring tools.
- I investigated how programmers use tools to identify opportunities for refactoring by finding potential design problems in their code, consulting with Dr. Barry Anderson (PSU, psychology). I then created and evaluated an ambient visualization to allow developers to find such opportunities more effectively [9,16,19,20,21]. From the evaluation, which included 6 graduate students and 6 developers from industry, the results suggest that developers can effectively identify opportunities for refactoring and make more informed judgments about whether or not to refactor. I am currently in discussions with the consulting firm Industrial Logic (<http://industriallogic.com>) on how the static analysis tools that I wrote for my visualization can be used to provide code-quality feedback to their clients.
- In a formative study, I observed that programmers have difficulty selecting code as input to a refactoring tool, and have difficulty interpreting error messages produced by refactoring tools. I then created and evaluated three new tools designed to solve these problems [3,17,24,25,26]. The results of a comparative laboratory experiment suggest that programmers can correctly select program elements twice as fast and correctly identify refactoring errors more than three times faster using the new tools that I designed and implemented, when compared to traditional program development tools.
- I investigated programmers' difficulties when initiating and configuring refactoring tools, and designed and evaluated two tools for initiation and configuration [11,15,22]. The evaluations took the form of an interview with 15 developers from industry, an analytic evaluation, and a paper-and-pencil experiment. One of the main findings of the experiment, which included 6 developers from industry, is that by bypassing the traditional name-based initiation of refactoring tools, exemplified by hotkeys and menus, gestures can be used to initiate refactoring tools in a way that it is highly memorable.

For each of my proposed tools, I demonstrated that usability could be measurably improved. Under the assumption that improved usability leads to improved adoption, my goal of getting people to use refactoring tools would have been accomplished. However, this assumption ignores one important fact, that a programmer has to know of the existence of a tool in the first place. Because complex development environments contain thousands of tools, all competing for the programmer's limited attention, even highly usable tools might never get used. Thus, I expanded my focus from improving usability to bridging the capability chasm.

Current Research: the Capability Chasm in Development Tools and Code

I am currently investigating two main aspects of the capability chasm. First, I am investigating how programmers gain knowledge of the existence of tools. Second, I am investigating how complexity can be reduced in integrated development environments by leveraging programmers' knowledge of code.

In collaboration with Dr. Gail Murphy (UBC, software engineering), I have been researching the question, "how do programmers discover new tools?" To answer the question, I conducted interviews with 18 programmers from industry; the preliminary results suggest that learning from other programmers during normal working activities is highly effective [14]. However, I found that such learning is also relatively infrequent, especially when compared to encountering a tool when exploring a system's user interface.

In collaboration with a doctoral student, a masters student, and Dr. Murphy, I have been researching how programmers' knowledge of code, derived from both writing it and interacting with it, can be used to improve software development [1]. In a case study with a large international software company, we showed that our degree-of-knowledge model can recommend software development experts better than an existing approach. In another case study with a smaller local software company, we showed that our degree-of-knowledge model can help filter a developer's stream of bug-report change notifications, so that only relevant notifications are displayed.

Planned Research: Bridging the Capability Chasm in Software Development and Beyond

My current research suggests that software developers effectively discover tools from peers, but it has only begun to suggest *why* discovery via peers is effective. For a more definitive answer to why it is effective, I plan to conduct a longitudinal study where I directly observe the causes and effects of discovery via peers. The results will help suggest how discovery can be supported by development tools and practices, so that discovery via peers occurs more often.

In distributed teams where face-to-face discovery via peers is impossible, my research suggests that programmers sometimes discover new tools via online *screencasts*, short videos of other developers recorded during programming activities. However, in practice screencasts are painful to produce, because deliberate, prolonged, and duplicated effort is required to create, edit, target, and distribute screencasts. I plan to investigate how to make this process easier by building and evaluating a system that records a history of a developer's tool usage, allowing the system to:

- Recommend tools that a developer may find useful by using collaborative filtering and sequential pattern mining to compare the developer's tool usage history with the rest of the team;
- Give a developer the option of automatically creating a screencast of a tool by replaying a previous use of the tool from the developer's history; and
- Share a developer's tool knowledge with peers by leveraging the developer's existing social network by, for example, generating posts for blogs, Twitter, or Facebook.

Beyond Software Developers. While I have studied software developers—one kind of knowledge worker that use complex systems—I also plan in the future to study other types of complex system users. One avenue of research is to investigate whether tool discovery via peers is effective for other types of software systems. While discovery via peers may be effective in software development because pair programming and code reviews are common practice, what practices are used by people who use word processors, for example, to make discovery via peers effective or ineffective? The results of this investigation will help determine how discovery via peers can be supported in other domains.

People discovering tools from others is one instance of a broader phenomenon that I plan to investigate: how people share knowledge about software systems. For example, how can a development team's accumulated knowledge of a code base be efficiently shared with new teammates? How does a database analyst know which tables are relevant to a particular task and how can that knowledge be explicitly communicated? Through a combination of interviews, experiments, and practical tools, I hope to answer these questions.

Helping software users—including software developers—gain, maintain, and exercise knowledge of relevant tools is a critical challenge to achieving the capabilities of software systems, from scientific databases, to development environments, to social networking websites, and beyond. I plan to meet that challenge head-on.