CAREER: ENABLING AND EXPLOITING EVIDENCE-BASED BUG TRIAGE

Evolving software is no simple task: somehow, amongst innumerable bug reports, feature requests, and project plans, software teams must decide *which* of these issues deserve the team's limited time and resources. To make these decisions, most teams engage in a process of **bug triage**, comparing estimates of the *frequency* and *severity* of each issue, among other factors.

While estimating frequency and severity is more disciplined than using sheer instinct, the lack of large-scale data about software issues means that **most of these estimates are based on intuition**. Worse yet, the data that teams *do* have is limited: technical support feedback is difficult to analyze because of its unstructured nature; automatic crash and hang reports are only a subset of the issues that users experience; and reports written by users directly usually come from *power users*, biasing reports to expert use. Because of these limitations, teams have no choice but to rely on their **subjective impressions** of software use and user needs.

The proposed work will replace these impressions with large-scale data about software issues. To do this, the PI will invent techniques that **detect software issues through peoples' use of automatic help tools**. These tools will allow users to get explanations about unexpected program behavior by choosing *how* and *why* questions about program output. The use of these tools will capture a wide range of software issues in a consistent, structured form. Unlike voluntary feedback, these tools will be part of users' normal work, increasing the representativeness of *frequency* and *severity* estimates, while also capturing new kinds of underreported issues such as non-fatal errors and a wide range of usability and understanding problems.

To explore this approach, the PI will extend his prior work on automatic help tools to a collection of widely-adopted *web-based courseware applications* developed at the University of Washington. The use of these help tools will be captured in the field to detect software issues. The PI and his team will then prototype several ways of exploiting this data, including 1) aggregation tools that **group** issues into generalized, executable test cases, 2) triage tools that **analyze** issues across time, version, and customer data, and 3) maintenance tools that **automate** fault localization, report assignment, and impact analysis. To assess these prototypes, the PI will measure the *representativeness* of reported issues, the *effectiveness* of the triage and maintenance tools, and the *objectivity* of triage decisions relative to current practices.

These research plans are closely aligned with the PI's educational goal, which is to **redefine software quality assurance education**. To achieve this, the PI plans 1) new projects for the PI's *User-Centered Design* course that directly involve students in the evaluation of the research, 2) a new course that teaches theories and skills relevant to software engineering *teamwork*, and 3) a peer and professional mentoring event that informs students about the day-to-day lives of software quality experts. The PI will evaluate these initiatives by comparing the enrollment and job placement outcomes of students who do and do not participate. All initiatives will involve participation by Seattle software companies, leading to technology transfer and a stronger bond between the University of Washington and the software industry.

The **intellectual merits** of this work include: 1) moving bug triage from an art to a science, 2) techniques for implementing automatic help tools in web applications, 3) new forms of automatic help, 4) knowledge about privacy issues in reporting user feedback, 5) statistical approaches for separating reported issues into meaningfully distinct groups, 6) new tools for supporting evidence-based bug triage and field data analysis, 7) software maintenance tools that exploit user feedback to streamline bug fixing and support decision-making, and 8) evidence of the feasibility, effectiveness, and utility of the above contributions.

The **broader impacts** of this work include: 1) software that better meets the needs of its users, increasing user productivity and reducing frustration, 2) help tools that reduce users' need for technical support to resolve software issues, 3) empowering users to contribute to software evolution through their normal use of software, 4) students who are more *informed* about careers in software quality, 5) students who are more *prepared* for teamwork aspects of software development work, 6) usability improvements to courseware, 7) undergraduate participation in research, and 8) broadened participation in computing.

1. Project Motivation and Approach

Despite significant advances in software technology, most people find software unreliable and difficult to use. For instance, a recent survey showed that **48% of adults needed to ask for help to use or configure their software**, and that of those, 38% had to contact technical support, 15% never fixed their problem, only 2% found help online, and half felt "discouraged and confused by their efforts" [42]. Most software companies are acutely aware of these issues, spending an average of 21% of corporate expenditures on technical support and software maintenance [81]. The result of these efforts is usually a vast collection of bug reports, feature requests, and other issues to address in the next software release [9, 48].

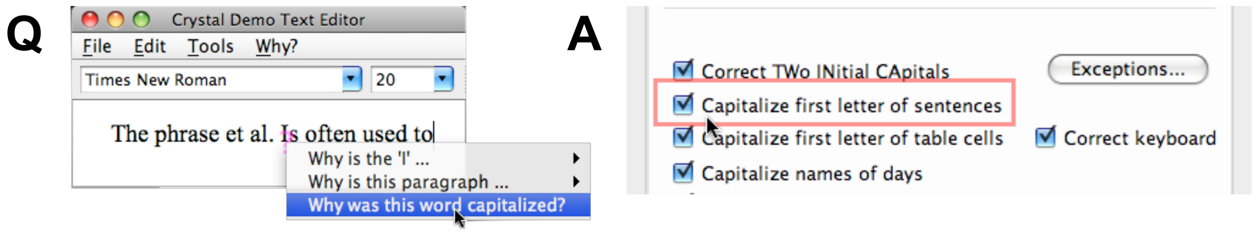
While software teams would ideally address *all* of these issues, this is rarely feasible: not only do teams work with limited time and resources, but in many cases, ways of resolving issues may be in direct conflict. Therefore, teams must decide *which* of these issues are most deserving of the team's attention. To make these decisions, most teams engage in frequent **bug triage** [48], prioritizing the team's efforts by comparing estimates of the *frequency*, *severity*, *cost*, and *risk* of each issue.

While this process is more disciplined than using sheer instinct, the lack of large-scale data about software use means that **most of these estimates are based on intuition**. Worse yet, the data that teams *do* have to support these estimates is limited:

- *Technical support feedback*, while capturing a wide range of issues, is difficult to analyze and aggregate because of the unstructured nature of text and speech. This feedback also lacks adequate context to reproduce or understand an issue, making it difficult for teams to act upon.
- *Automatically reported crashes and hangs*, while capturing stack traces and other data that enables teams to isolate their causes automatically [33], only capture a subset of the issues that users experience. Error reporting APIs, which enable custom error reports, require teams to *anticipate* the issues that users might experience.
- *Direct reports from users* often come from vocal minorities such as power users. This potentially biases frequency and severity estimates towards expert software use. Most users do not report issues because they blame themselves when software misbehaves [65].

Because of these limitations, teams have no choice but to form **subjective impressions** of which issues are the most *frequent* and *severe* to users, speculating about how *many* users are experiencing issues and whether users find these issues problematic [33, 71].

In my research, I plan to replace these *impressions* with evidence. To do this, I will invent new techniques that **detect and report software issues through peoples' use of automatic help tools**. These tools, which are based on my prior work [62], allow users to choose *questions* about unexpected program output and get explanations about its causes. For example, consider Figure 1, in which a user clicks on the word 'Is' and selects *"Why was this word capitalized?"* The tool responds by showing the checkbox that caused the auto-capitalizing to occur. This approach can be used to answer questions about unexpected errors, incorrect values, and a variety of other kinds of problematic program output.



The phrase et al. Is often used to

☑ Correct TWo INitial CApitals
☑ Capitalize first letter of sentences
☑ Capitalize first letter of table cells
☑ Correct keyboard
☑ Capitalize names of days

Figure 1. The *Crystal* automatic help tool, answering a question about an unexpected capitalization.

My research will **report the use of these help tools as indicators of software issues**. Compared to other forms of user feedback, these *help questions* have several distinct benefits:

- *They are more representative than voluntary reports* because users already seek help as part of their normal software use. The more users that provide feedback, the more faithfully a team's collection of reported issues will represent users' experience with software.
- *They indicate underreported but critical issues*, including feedback about both *functional failures* such as incorrect output, and *usability issues*, such as confusing error messages, workflow inefficiencies, and configuration problems (some of the most costly issues to support [81]).
- *They inherently indicate severity* by allowing users to make an explicit, machine-observable decision to troubleshoot an issue; simply counting the number of users who *choose* to troubleshoot issues can enable a team to compare issue severity.
- *They capture lightweight execution histories* as part of answering users' questions, facilitating the automatic aggregation of issues at a large scale. These histories can also be used to create a new class of evidence-based bug triage and software maintenance tools.

Figure 2 illustrates my plans for enabling these help questions and exploiting the user feedback that they capture. These plans include five major activities:

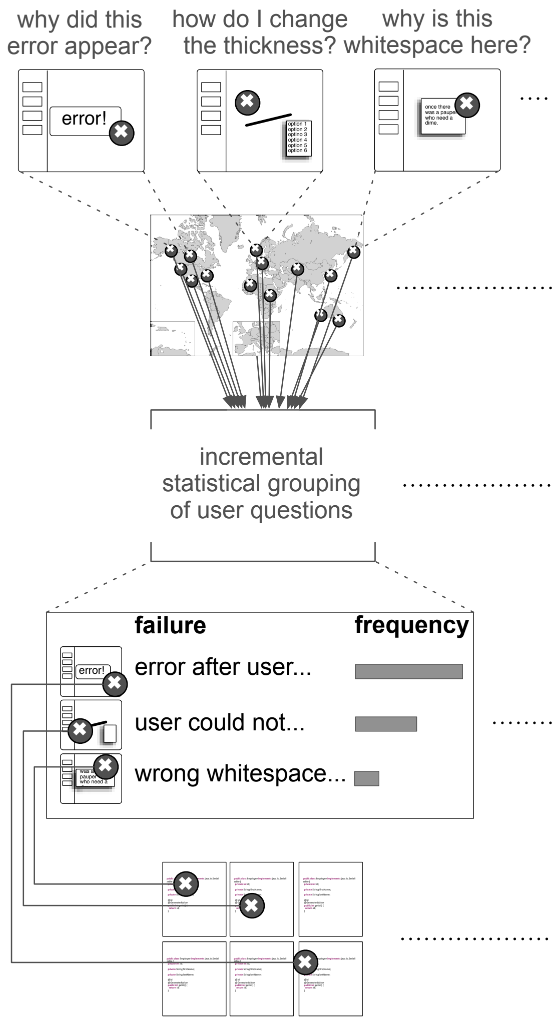

why did this error appear?
how do I change the thickness?
why is this whitespace here?
error!
incremental statistical grouping of user questions
failure
frequency
error after user...
user could not...
wrong whitespace...

Figure 2. The proposed research.

I will **enable help questions** in a set of web-based courseware applications developed at the University of Washington (in addition to in-house test applications). To do this, I will generalize the prototype in Figure 1 to interactive web applications, adding support for new kinds of "why" and "how" questions to extend the range of detectable issues.

I will gather users' help questions at a large scale, exploring different ways of requesting users' permission to report information. I will also explore how **privacy**, **consent**, and **training** influence the representativeness of reported field data.

I will invent techniques for incrementally **grouping** questions over time, aggregating large-scale feedback into meaningfully distinct groups. These tools will analyze the input and output contexts of users' questions, producing generalized, executable test cases.

I will invent **analysis** tools for helping software teams explore field data over time, successive software versions, and customer data. These tools will integrate data from other forms of software testing, such as crash reports and user tests. I will assess these tools' ability to help teams make more evidence-based judgements of issue *frequency* and *severity*.

I will invent tools that **exploit** help questions to streamline other maintenance activities. For example, the execution contexts captured can be used to automatically locate faults and features that contribute to issues, to recommend which developers should work on issues, and to perform automatic impact analysis.

This research plan is closely tied to my educational goal, which is to **redefine software quality assurance education**. As part of achieving this goal, I propose three initiatives that integrate, enhance, and exploit, my proposed research:

- *Engaging User-Centered Design students in the proposed research*, by having students measure the *representativeness* of the issues reported by the proposed help tools. More than 70 students will gather data about the software issues that courseware users experience in practice, using the usability methods taught in class. They will then compare these to the issues reported with the automatic help tools, learning about the strengths and weaknesses of different evaluation methods.
- *Teaching software quality teamwork in a new studio-based project course*. Topics in this new course include small group communication, coordination in software teams, and aspects of version control, issue trackers, and other tools that support these teamwork challenges. Students will learn about software quality from industry experts and help evaluate the efficacy of research prototypes.
- *Organizing a peer and professional mentoring event on software quality*, in which student interns and industry professionals meet with pre-major college students to share experiences in software quality assurance. The event will give prospective *Computer Science* and *Informatics* students a more concrete understanding of careers in software quality, facilitate mentoring relationships between academic and industry, and build social networks of peers with similar career goals.

The **broader impacts of my plans will ultimately be software that better meets users' needs**. My research and education plans achieve this in three ways: 1) by providing tools that help users troubleshoot software issues, software will become easier to use and more helpful; 2) by enabling the capture of richer and more representative data about how software is used, software teams will be able to design more useful and helpful software; and 3) by improving education about software quality assurance, future generations of software teams will be better prepared to gather and utilize user feedback. Throughout this work, I will contribute scientific knowledge about software design and invent several new technologies.

This proposal is organized as follows. In the next section, I review prior work in detecting software issues. I then discuss my preliminary and proposed research in Section 3. I discuss my education plans and their ties to my research in Section 4. In the remaining sections, I detail my five-year plan, the intellectual merit and broader impacts of the proposed work, and my qualifications.

2. PRIOR WORK ON DETECTING AND TRIAGING SOFTWARE ISSUES

Researchers have explored several ways of detecting problems with software. Perhaps the most promising approach in the last decade has been **gathering large-scale empirical data about software failures.** For example, *automated crash and hang reports* are a widely-deployed example of this approach, particularly Microsoft's Windows Error Reporting (WER) system [33]. WER not only streamlines the reporting of crashes and hangs, but it allows application developers to generate custom reports for custom program events. Researchers have explored similar methods of crowdsourcing[1] the capture and repair of field failures. For example, Liblit et al. recorded snapshots of execution history from programs running in the field and combined these histories to support the diagnosis of crashes [55] and concurrency issues [83]. Others have focused on the *reproduction* of software issues, recording lightweight execution histories in the field [19]. These can then be used with other techniques to index and aggregate multiple histories to find incorrect outputs [57]. Tucek et al. describes a similar system that *diagnoses* failures in the field by performing checkpointing and root cause analysis while a program runs [84]. Because users need only to click a button to send feedback, these approaches give software teams large-scale, aggregated frequency data that is more representative than intuition.

[1] *Crowdsourcing* [23] is taking a task traditionally performed by a specific population and distributing it to a larger community (in this case, the task is detecting software issues).

The limitation of the above approaches is that they only work for crashes, hangs, and custom-defined, pre-anticipated events. They do not work for other critical issues such as incorrect (but non-fatal) computations, usability problems, configuration issues or other unanticipated errors. One approach to detecting these other issues is to gather **large-scale usage data**. For example, Microsoft Office users can send "Software Quality Metrics" (SQM) data to Microsoft. Such data usually involves *usage statistics*, which can inform Microsoft about which features are being used regularly, which features are not, and by whom. Researchers have also investigated the instrumentation of open source software [82], gathering more detailed information about users' documents and their relationship to other data in a users' work. More sophisticated techniques identify usability problems by statistically modeling *undo* and *erase* events in usage data, distinguishing between actual undos and help-seeking [1]; others have detected anomalies in console logs using statistical methods [88]. Google takes an experimental view of usage data, deploying different versions of web sites to perform experiments about low-level decisions, such as button placement and labeling [73]. *Google Analytics* supports similar analyses of web traffic.

While the above techniques can help software teams know *what* users are doing, interpreting *why* users are doing it can be highly subjective [41]. This interpretation requires one to distinguish between actions that move users towards a goal (such as undoing a mistake) and actions that help a user think (comparing two versions of sentence using undo) [45]. Moreover, teams have to *imagine* what patterns might exist in feature usage data before testing for them [43]. The primary way to avoid these interpretation challenges is to **perform *user tests* of critical use cases**. Usability testing [30, 35], which is a trade that has grown by 5000% in the past 15 years [86], involves devising representative tasks, recruiting representative users, having users work on these tasks, and identifying *breakdowns* that occur in users' work. These breakdowns are then documented as software issues, alongside other bugs and feature requests.

Unfortunately, user testing rarely scales because of the cost of paying users to participate [64]. Moreover, it is often difficult to find enough *representative* users [79] and to test a wide enough variety of tasks [56] to gain confidence in the quality of a design. Some researchers have tried to address these limitations by proposing ***remote* usability testing** tools that have users evaluate software *online* [22]. Remote users *can* successfully report critical incidents [14], but they only find half as many problems as trained usability testers in the lab [13], and they need to be *trained* and *incentivized* to acquire useful data [80]. Because of these limitations, usability testing often faces skepticism: managers often view user tests as unnecessary overhead [35, 76] and developers view the *results* of usability tests with skepticism because of their small samples [36].

There are other sources of large-scale user feedback, such as the questions and often detailed usability critiques that users post on **technical support web sites and user forums** [38]. Unfortunately, most of this data is unstructured, lacks execution context, and is of inconsistent quality, making it difficult and costly to analyze and aggregate systematically. Even when teams have the time to process this feedback, it is cumbersome to aggregate and does not result in reliable estimates of frequency and severity. Furthermore, this online feedback is *volunteered* by users, and thus often biased towards vocal minorities such as power users or other software developers. This potential bias makes it difficult to know how common the issues they report actually are [76].

Compared to the considerable research on *detecting* software issues, there is little technology to support the *triage* of software issues. Some researchers have explored ways of automatically assigning bug reports to developers [4], building machine learning classifiers on programmers' implementation expertise [5]. Others have explored the possibility of automatically detecting *duplicate* reports [87] and improving bug report *readability* [27]. Beyond these techniques, software teams have little support for making-evidence based estimates of the frequency and severity of software issues. Consequently, authority, stereotypes about user behavior, and technical issues tend to dominate design decisions [50], even when these decisions are inconsistent with teams' project goals.

3. ENABLING AND EXPLOITING EVIDENCE-BASED BUG TRIAGE

The proposed research will investigate a **new approach to detecting software issues by analyzing people's use of automatic help tools**. This approach will produce more reliable estimates of *frequency* and *severity*, enabling teams to perform more evidence-based bug triage. As illustrated earlier in Figure 2, this research will involve several activities, including 1) the creation of new help tools to capture issues, 2) research on how privacy aspects affect the representativeness of the software issues reported, 3) techniques for aggregating issues into generalized executable test cases, 4) several new bug triage analysis tools, and 5) an array of software maintenance tools that exploit help question data. In this section, I discuss the preliminary work that supports these plans, and then discuss these plans in detail.

3.1. PRELIMINARY WORK ON AUTOMATED HELP

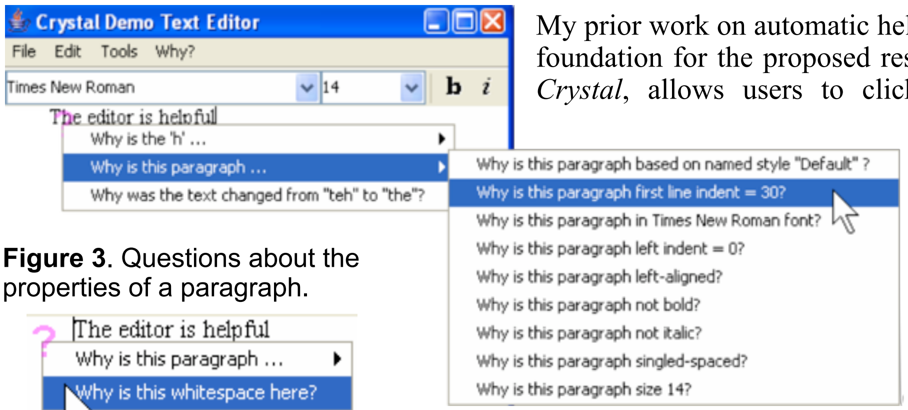

Crystal Demo Text Editor
File
Edit
Tools
Why?
Times New Roman
14
b
i
The editor is helpful
Why is the 'h' ...
Why is this paragraph ...
Why was the text changed from "teh" to "the"?
Why is this paragraph based on named style "Default" ?
Why is this paragraph first line indent = 30?
Why is this paragraph in Times New Roman font?
Why is this paragraph left indent = 0?
Why is this paragraph left-aligned?
Why is this paragraph not bold?
Why is this paragraph not italic?
Why is this paragraph singled-spaced?
Why is this paragraph size 14?

Figure 3. Questions about the properties of a paragraph.

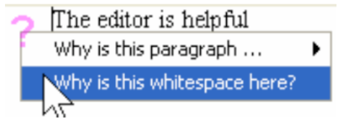


Figure 4. A question about a paragraph's whitespace.

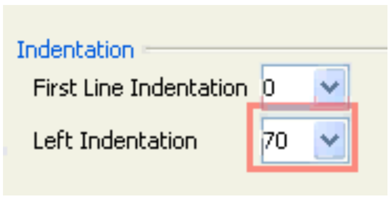


Figure 5. An explanation of a paragraph's whitespace, showing the field that caused it.

My prior work on automatic help tools [62] is the conceptual foundation for the proposed research. The prototype, called *Crystal*, allows users to click on elements of a user's document, user interface controls, and even whitespace, and select "why" questions about the the selected program output. For example, Figure 3 shows a user who has clicked on a paragraph to ask questions about its properties. Figure 4 shows a user who has clicked on the *whitespace* to the left of a paragraph. Figure 1, shown earlier, shows a user asking a question about text that was auto-corrected. In all of these cases, after a user has chosen a question, *Crystal* analyzes the history of user input, the program defaults, and the logic of the program's event handlers to determine the *causes* of output in question. These causes are then shown by highlighting the user interface controls that were used to *change* the behavior of the application, immediately showing the user how they can rectify the problem. For example, Figure 5 shows the answer to the question in Figure 4, highlighting the "Left Indentation" control that determined the paragraph's whitespace.

Conceptually, *Crystal* derives answers to users' help questions by **computing the user-modifiable subset of a precise dynamic slice** [8] on the selected output. For example, the whitespace in Figure 4 has a large number of *internal* dependencies that determined its size, but only a small number that the user has control over. These *user-modifiable dependencies* are the ones shown in the answer (and in the answer in Figure 5, there was only one such dependency).

To avoid having to record a complete execution history to perform dynamic slicing, *Crystal* exploits the editor's *undo* history. Most modern undo support store a list of changes in an application as some form of *command objects* [63]. For example, if a user deleted an event in a calendar application, the undo history stores an object that contains the deleted data and the context necessary to restore it (such as the calendar to which it was attached). To answer questions, *Crystal* adds two kinds of data to undo history. The first is the **history of values for all user-modifiable program state** and the input event that caused each. For example, when a user disables the "auto-correct" feature, the system stores the new value for the *auto-correction enabled* state and remembers that the user performed this change by clicking on a particular checkbox. This data allows *Crystal* to highlight the input or system event that caused the unexpected output in its answer. The second kind of data that *Crystal* adds to undo history is the **control and data**

dependencies used to decide whether to make a change. For example, before auto-capitalizing a word, the system needed to check whether the auto-capitalizing feature was *enabled*. *Crystal* records this data dependency and uses it to determine the causes of the output a user has questioned.

These additions to undo history required **little developer effort** beyond the effort necessary to support undo. For example, the paragraph questions in Figure 3 are automatically generated from the current state of the paragraph and user interface string constants; these answers are computed with no additional code. The questions and answers that *did* require custom code involved converting hard-coded data and behaviors into first class data and events. For example, Figure 4 shows a question about a paragraph's margin *whitespace*; this required the layout algorithm to remember the location of white space and the user interface a way to inquire about it. Similarly, answering questions about auto-corrected words (as in the question about 'teh' in Figure 3) required the developer to convert the hard-coded auto-correct behavior into a first-class undoable system action.

Of course, these help tools cannot answer *all* questions, nor can it answer all questions with complete precision. For example, users can ask about *visible* program output, such as document state, user interface controls, and values computed by the program, but users cannot ask about output that *does not* appear in a general way. Some answers may also involve some irreducible complexity that is difficult to explain. For example, while most questions produce a *single cause*, such as *"the word is capitalized because auto-capitalize is on."* or a single *chain* of causes, such as *"the paragraph is indented because it has the "Body" style, and the "Body" style inherits from the "Normal" style, which is indented."* some questions have *multiple* chains of causes. This might occur for output that results from complex computations, such as explanations for why an e-mail was marked as spam: such output often depends a large set of user tags, and a complex latent semantic analysis algorithm. (Explaining these is with precision is *not* the goal of this proposal, though as I discuss later, there may be benefit in providing simplified explanations).

Even with these limitations, Crystal proved quite successful in helping users with common problems. In one study of nine problematic use cases, **Crystal users resolved 30% more problems, 50% faster** than those using documentation and online help [62]. After using the automated help only once, users relied on it *exclusively* for help, preferring it over documentation and the Internet [62]. Users said this was because the help was particular to their document and use of the software.

3.2. Extending Automatic Help to Interactive Web Applications

In the proposed research, I will first generalize the *Crystal* concept to **interactive web applications**. This research will demonstrate the feasibility of the automatic help tools in client-server based applications, while also supporting a rapidly growing platform for software applications.

Much of the work necessary to adapt *Crystal* to web applications is straightforward. Instrumenting client-side applications to capture program events is feasible, well-supported by tools and web servers, and can be done without modifying the client [44]. Furthermore, many of the challenges with implementing *undo* in web applications has been resolved by industry (though these undo techniques are not widely adopted). I will adapt this work to support automatic help in web applications, building several test applications to demonstrate the feasibility of the tools.

In addition to these in-house applications, I will **collaborate with the *Catalyst Tools* team** at the University of Washington to help deploy the ideas to an existing population of application users (documented in an attached letter). *Catalyst* develops several web-applications, including a grade book, a course web site design tool, survey and portfolio authoring tools, web-based e-mail, and several other applications. These are widely adopted by the over 40,000 students, staff, and faculty at University of Washington campuses. All of these tools include undo support, which is the basic requirement for implementing the automatic help tools. Incorporating these help tools into a diverse set of deployed software will help assess how well these tools scale, what kinds of software issues can be feasibility reported, and what kinds of help the tools can feasibly provide. Furthermore, because the *Catalyst Tools* have an existing user base, my team can focus on *research*, rather than *deployment*.

Beyond these implementation issues, there are several research challenges in generalizing Crystal to web applications. In particular, there are challenges in tracking modifications to data that are triggered in the client, processed by the web server and then changed in a database. Tracking this data flow and then using it to answer users' questions will require new methods of capturing lightweight execution histories between multiple languages and machines. I will explore techniques that unify the capture of these execution history that are low-overhead and efficient to transmit between the client and server. This will result in **new frameworks for capturing lightweight execution histories in web applications**.

3.3. Extending the Range of User Feedback with New How and Why Questions

In addition to adapting Crystal's features to the web, I propose **three new kinds of help questions** to extend the range of user feedback that can be captured through help tools. These include new support for *how* questions, why questions about *performance*, and why questions about *complex behaviors.*

"Why" questions allow users to ask about *existing* features in a program; to compliment these, I will explore support for **"how" questions that capture large-scale, aggregated feature request data**. These questions will allow users to ask about features and options that they *expect* to find but cannot. For example, suppose a student is searching Amazon.com for the *shortest* book on a topic, but the search results do not list page length. Support for "how" questions would allow the user to click on the "Results" header at the top of the results and type "how do I search on page length?" The system's response would either find a feature that supports the desired behavior (building upon work on *keyword programming* [61]) or it would tell the user, "We don't yet support this feature, but thousands of others have asked for a similar feature. Stay tuned." As part of this work, I will explore ways of applying natural language processing techniques aggregate user feedback and to detect and filter spam.

The "why" questions in the *Crystal* prototype focused on functional requirements, but not other software qualities. Therefore, I will support new **why questions about performance issues**. For example, users will be able to ask "why is this progress bar slow?" and get answers such as "the system is waiting for data from the Internet" or "the system is halfway through a large data set." In the absence of a specific progress display, users will be able to click on the area that they *expect* to have a result and ask, "why isn't this updating?" The goal of these questions would not be to offer precise diagnostics to users, but to provide more detailed, context-specific feedback on demand. While these answers will not make the algorithms faster, they will help users decide if they want to cancel an operation or find some other way of completing their task. These questions will also enable teams to learn about user-critical performance issues at a large scale.

The third kind of question that I will explore are **"why" questions about complex behaviors**, to provide explanations of output computed by algorithms and processes that have some irreducible complexity. For example, a user may want to know why an e-mail message was marked as spam or why a link was recommended from a recommendation system. To answer these questions, I will explore ways of simplifying the complex dependencies behind these calculations and providing general descriptions of how these features work. For example, rather than trying to explain the precise reason to a user, these questions will provide simplified explanations, as is done in recommendation systems ("this movie was recommended because of your interest in..."). These answers will perform program analyses on these computations to generate **context-specific examples** from the user's own data to explain system behavior.

To assess the effectiveness of these "why" and "how" questions in answering users' questions, I will perform both **lab and field studies that compare their utility to other forms of help** such as Q&A sites [38], technical support [42], and friends. I will perform these evaluations as part of a course assignment in my *User-Centered Design* course, having over 70 students apply a variety of usability evaluations on the research prototypes. For example, they will apply the *heuristic evaluation* technique [64] to identify potential design problems in advance and perform lab-based user studies of specific troubleshooting use-cases. Specifically, students will measure how quickly users are able to rectify problems compared to traditional forms of help and to what extent users rely on the help tools to troubleshoot their problems.

3.4. Reporting Representative, Anonymous Help-Seeking Feedback

With the help tools in the previous sections in place, a central research challenge will be **reporting *uses* of the help tools as indicators of software issues**. This problem has similarities to desktop crash and hang reporting services, such as *Windows Error Reporting* [33]; these systems generally pose two major challenges: 1) deciding what data needs to be sent and 2) getting users' permission to send it. Because the proposed research will capture new kinds of *user-defined* issues, I will also explore to what extent the issues captured are *representative* of the issues that users actually experience.

The data that needs to be sent will likely be the **same data necessary for *answering* users' questions**. This includes *user interface state*, the *data dependencies* and *control flow decisions* [8] involved in handling user input events, and the user's *undo history*. For example, when a user asks why a spreadsheet cell background is red, to explain that there was an error in the cell's formula, the system would need to report the source code that decided to highlight the cell red and the data that was used to make this decision, such as the formula itself and error reporting settings in the system preferences. I will explore to what extent this data, when aggregated at a large scale, is sufficient to reproduce an issue (compared to other forms of execution history, such as "whole" execution traces [89], and lightweight field failure traces [19]). This same data will be used to aggregate issues and to enable new kinds of bug triage and software maintenance tools (both discussed later).

Another challenge with reporting issues from the field is obtaining users' *permission* to send their data. In the case of crash reporting, users are asked to "opt-in" each time the system wants to send information. This consent process would be different for web-based applications, because much of the data is **already stored on the web server** as part these applications' normal data flow (this is because of the widespread use of AJAX, a collection of techniques for implementing interactive web applications that do not require a page reload). In this case, users have already consented to having some of their application use monitored by a web server. I will assess to what extent the additional data may require new forms of permission and informed consent.

I will use the above studies to design **anonymization and redaction techniques** that remove identifying information, while preserving the problematic aspects of reported issues. These techniques will particularly focus on redacting *textual* information, since it is the most likely to contain personally identifying information. This research will focus on program analyses that identify when textual data plays a direct role in a reported problem and finding automatic ways of preserving the characteristics of the data that caused the issue. For example, users of the *Catalyst Tools* grading application will have a wide variety of privacy-sensitive data including student names and grades; users may use the help tools to indicate problems with spreadsheet formula calculations. The proposed research will detect that such a dependency exists and report the *dependencies*, but remove the specific text from the data reported.

Even if the help questions are helpful, the data capture is low-overhead, and the privacy issues are overcome, it is highly unlikely that *all* users would report issues using the automatic help tools. Therefore, a critical part of the proposed research is **evaluating the representativeness of the data reported from help tools**, characterizing the biases imposed by the help technology. To measure representativeness, I will have the undergraduates in my annual course on *User-Centered Design* perform field observations of the target applications to evaluate 1) how many users are using the help tools and 2) what kinds of help users are able to get from the help tools, and 3) what kinds of issues users are resolving through other channels. Students will apply a variety of evaluation methodologies, including lab-based user studies, field observations, and *experience sampling* approaches [18], which ask users to document issues they experience periodically. These studies will establish a baseline for the issues that users experience in practice, allowing us to measure how many of these issues were reported by the automatic help tools. We will then use this feedback to extend and refine the kinds of help that the tools support. (I discuss the educational aspects of these plans in Section 4.4).

3.5. Grouping Help Questions into Generalized, Executable Test Cases

With these reporting mechanisms in place, a central part of the proposed research is to **automatically and incrementally producing meaningful groupings** of the data as it arrives. For example, imagine that a team receives 10,000 questions about an error dialog that says some data cannot be exported. Underlying these questions are one or more reasons why the dialog was shown, corresponding to different conditions that were checked before displaying the error. The challenge is to group these different cases by the different error conditions from which they emerged, while generalizing away details such as the particular data that users were trying to export. Prior work performed grouping on *whole* execution traces [69]; I will aggregate field data by exploiting properties of the *lightweight* traces, across the phases in Figure 6.

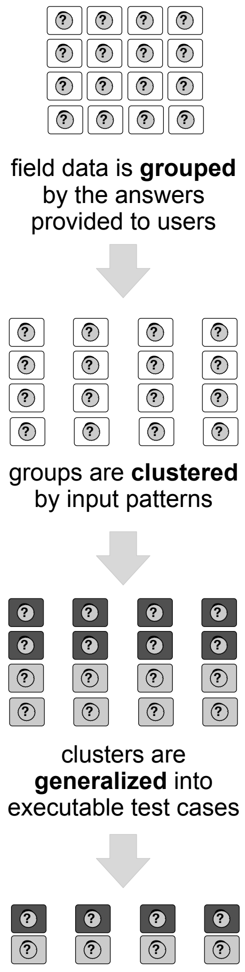

field data is grouped by the answers provided to users
groups are clustered by input patterns
clusters are generalized into executable test cases

Figure 6. Field data is aggregated across phases.

The first phase will **group issues by the *subject* of questions asked by users and the *dependencies* involved in the answer**. For example, imagine an application with 70 unique error dialogs: the system would split the issues by these error dialogs, and then further subdivide those groups by the particular conditions that produced the error. This grouping could be performed incrementally as data arrives from the field, since the subjects of questions and the answers depend largely only on the application source code and not on each particular reported case. As part of this grouping, issues would also be segregated by program versions.

Because the number of issues within each of the above groups may still be large, the second phase will **generalize the specific input values that caused the program to produce the unexpected output**. To detect which values contributed to the errors, I will apply unsupervised machine learning techniques to separate the input that contributed to failure into distinct groups. Because teams will likely depend on these groupings to be *stable* (for example, by writing bug reports against the groups), I will explore incremental adaptions of these learning techniques that handle the continual arrival of data from the field.

In the final phase, I will develop techniques that **generate a representative, executable test case** from the classes of the previous phase. Because the field data reported will not be complete execution histories, they cannot be used to directly reproduce issues. Instead, I will exploit the large scale of user data and extend the execution history captured on the web server to create a single complete execution trace for a representative example of an issue. These execution traces will enable the reproduction of failures, addressing a major challenge in software maintenance [9, 48].

In addition to this multi-level grouping, I will also explore techniques for **filtering reports that may not represent real issues**. For example, the reported data is likely to contain some *learning* activity, since the help tools can be used to understand how a program's features work (as opposed to overcome an issue). I will explore statistical methods of tracking individual users' reports to distinguish between novice and experienced use. Finally, some users may attempt to "game" the reporting mechanism, repeatedly sending feedback about a problem in order to try to influence its frequency. I will explore filtering approaches that detect statistical outliers or limit reports about specific issues to a certain number per day.

To evaluate the above work, I will measure the extent to which the generalized test cases reveal **meaningful and useful distinctions between reported issues**. For example, I will perform studies that assess whether the groupings of issues faithfully represent the concerns expressed by users, whether software teams perceive the issues in the same way that users do, and whether software developers find *technically* meaningful distinctions in the groupings. I will use the results of these studies to iterate and refine the techniques described above.

3.6. Analyzing Field Data with New Bug Triage Tools

While the techniques proposed in the previous section will help group and reproduce issues, a list of issues is not in itself useful. Therefore, a significant part of the proposed work will be to design **evidence-based bug triage tools** that help teams analyze and prioritize issues more objectively. These tools will:

- **Visualize field data as "hot spots" in user interfaces**. One way of supporting the exploration of the data is to let teams view issues *as users experienced them*. For example, if a field in a form was particularly error-prone based on the field failure data, the proposed system would apply highlighting to the field using estimates of the issue's frequency. This user interface would also allow teams to view and compare issues, issue frequency, and the kinds of input that caused them.

- **Track issues over time and software versions**. Because field data would arrive continuously, I will explore ways of applying unsupervised machine learning techniques to attributes of field data to highlight *new* issues or *new* situations in which issues are occurring. This could be particularly helpful after deploying software updates or revisions to a web site, providing automated feedback about whether software changes successfully resolved an issue.

- **Enable discussion about how to respond to user feedback**. Because bug triage is a *team* activity, and many teams are separated by distance, the proposed tools will provide online, asynchronous discussion tools to help teams evaluate and respond to user feedback.

- **Integrate other forms of user feedback.** Because the field data captured by the proposed work would be *complimentary* to both large scale crash report data and small-scale from software and usability testing, I will explore ways of integrating these various sources of user feedback. These features will link to data from other sources and provide a repository for storing it and will involve program analyses that automatically relate issues from different sources.

In addition to the above features, the triage tools will also support a scripting API that enables teams to analyze, filter and compare data in custom ways to better support triage. This API will allow teams to:

- **Compute custom frequency data**. While showing the *absolute* frequency of a particular type of issue could be helpful, there are more subtle kinds of frequency that may be more helpful in making business decisions about the issue's priority. For example, a team may want to know how many times an issue occurs in a day or how many times individual users experience the issue.

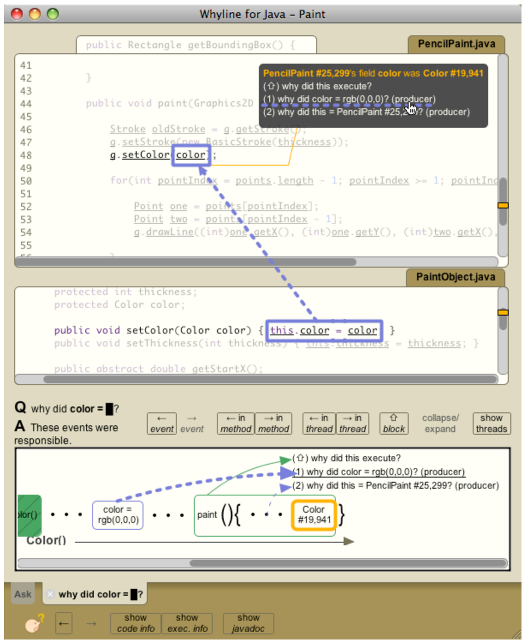
- **Estimate severity from context**. While the primary goal of the proposed work is to obtain more representative *frequency* data, the data captured in the execution histories from the field could also be used to estimate the *severity* of issues from a user perspective. For example, by providing teams with ways to compute *features* of the field data, these histories could be used to estimate how much data was lost as a result of non-fatal error messages or how often users abandoned the application after experiencing an issue. While these severity estimates would be crude, they would be an improvement over the speculation that developers currently rely upon [50].

- **Identify and compare user groups**. Even when a majority of users are experiencing an issue, they may not be the the most important user population to a team. For example, there may be a feature that a majority of users do not understand but that a small group of lucrative corporate users need to work in an important but obscure way. I will investigate ways of comparing field data across different user groups by providing teams with support for defining user groups based on customer data. I will also invent tools that allow teams to identify user groups with unsupervised machine learning on attributes of the field data.

To evaluate the above contributions, I will **assess the above tool features in a series of controlled experiments**, each comparing groups of users with the experimental feature to a control group without the tools. These assessments will provide evidence of the efficacy of the individual features.

3.7. Exploiting Field Data to Streamline Software Maintenance

In addition to using the field data to support more evidence-based bug triage, I also plan to exploit the field data to streamline software maintenance and debugging in other ways. These tools include:



```
public Rectangle getBoundingBox() {

    }

    public void paint(Graphics2D

        Stroke oldStroke = g.getStroke();
        g.setStroke(new BasicStroke(thickness));
        g.setColor(color);

        for(int pointIndex = points.length - 1; pointIndex >= 1; pointInd

            Point one = points[pointIndex];
            Point two = points[pointIndex - 1];
            g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(),
```

PencilPaint #25,299's field color was Color #19,941
(⇧) why did this execute?
(1) why did color = rgb(0,0,0)? (producer)
(2) why did this = PencilPaint #25,2[illegible]? (producer)

```
protected int thickness;
protected Color color;

public void setColor(Color color) { this.color = color; }
public void setThickness(int thickness) { this.thickness = thickness; }

public abstract double getStartX();
```

Q why did **color** = █?
A These events were responsible.

(⇧) why did this execute?
(1) why did color = rgb(0,0,0)? (producer)
(2) why did this = PencilPaint #25,299? (producer)

Figure 7. The Whyline, which allows developers to ask "why" questions about program output, will be extended to support more precise *feature* and *fault* localization.

Automatic fault and feature localization. One part of making bug triage decisions is estimating the amount of work necessary for implementing a change by identifying the code responsible for a particular program behavior [48]. I will exploit the large scale of the help question data to extend my prior work on the the *Whyline* [49] (shown in Figure 7) to support more precise and automatic *fault and feature* location. This will provide teams with a measure of how much of a program contributes to an issue across the range of reported cases, facilitating evidence-based estimates of the *cost* and *risk* of making the change.

Predicting the user consequences of design changes. Every time a code change is made, a critical challenge is predicting the *consequences* of a change to users' experience with software. I will explore ways of comparing the execution patterns that led to issues in the field data to the execution patterns of designs that teams have prototyped but not yet deployed. For example, imagine a developer has created a new web form with error validation that relies heavily on validation code that, in the past, led to a high predominance of questions. These new analyses could provide automatic warnings to developers, allowing them to learn from the feedback from prior deployments. Evaluations of this work will focus on assessing the rate of false positives reported by these techniques.

Automatically assigning issues to developers. A significant part of bug triage is deciding which developers to assign issues. Prior work has explored using text classification to automatically assign bug reports to developers [4], but this approach does not take advantage of information about the *components* involved in the failure. The execution histories captured by the proposed work could be used to make stronger associations between the components involved in a software issue in the field and the developers who have experience maintaining these components.

Detecting issues from feature usage data. Although the proposed research will attempt to increase the representativeness of field data by increasing the number of users who provide feedback, not all users will use these mechanisms, nor will users rely on them universally. However, there are opportunities to still learn from these users and situations by combining usage data with the questions that other users have asked. I will explore ways of statistically modeling the input contexts that lead to *questions* and then using these models to try to detect the prevalence of similar issues from *feature usage* data. These analyses increase the representativeness of frequency estimates.

Identifying usage patterns to support design decisions. Though most software teams already have some idea about who their users are, the field data captured by the proposed research can provide evidence about how different users understand and use software features. For example, the help question data could reveal that users who have difficulties authoring mail filtering rules also have difficulty configuring spam filters. I will explore machine learning techniques that help teams identify these associations between feature usage, identifying opportunities to improve how software features are partitioned and presented in user interfaces. It may also reveal opportunities to design new features that overcome the features that users struggle to use.

4. REDEFINING SOFTWARE QUALITY ASSURANCE EDUCATION

My research on evidence-based bug triage is tied closely to my educational goal, which is to **redefine software quality assurance education**. This vision is directly aligned with my teaching duties, which are primarily to teach undergraduate *Informatics* and *Computer Science* students who pursue jobs as software testers, software developers, usability experts, and managers of software teams. In this section, I describe my teaching approach and philosophy and then detail three educational initiatives that contribute to this educational vision, while enhancing, exploiting, and integrating my proposed research.

4.1. ACCOMPLISHMENTS IN EDUCATING WITH DIRECT EXPERIENCE

My educational approach is to **give students direct experience with team-based software design**. For example, my courses mix lectures with engaging classroom activities and lab sections that help students apply their knowledge to real design projects. All of my courses revolve around team projects that help students learn to work in interdisciplinary teams. For example, last fall I taught two courses in which students applied user-centered design methods to design and evaluate technologies to support the 2008 U.S. elections. Students observed campaign center visitors and interviewed blind and elderly users and prototyped and evaluated technologies to support these user groups as interdisciplinary teams.

As a new teacher, I have quickly demonstrated my commitment to teaching. For instance, I have a **4.85/5.0 average** student evaluation score across two courses required courses. Several students wrote unsolicited feedback to my dean, saying, for example, "*We were constantly shown real world examples and given real context to support the theories being presented. Andy didn't simply stand on a pedestal and preach philosophies that we must take on faith*." I hope to sustain this teaching success partly by learning from experienced teachers. For example, I have **weekly meetings with senior computer science lecturers** to discuss teaching approaches. I also gather data about the experiences of my *Computer Science* and *Informatics* undergraduates, publishing this work to educational communities [51].

4.2. ENGAGING USER-CENTERED DESIGN STUDENTS IN THE PROPOSED RESEARCH

One way that I will integrate research and education is by **involving undergraduates in evaluations** of the proposed research. For instance, one aspect of my proposed research will involve comparing the set of issues reported by *help-seeking tools* to the full set of issues that users actually experience. To perform this comparison, I will enlist the help of the 70 students I teach annually in my *User-Centered Design* course. During the course, students perform user studies, field observations, interviews, and surveys, all revealing a set of problems with a particular user interface. Once the automatic help tools are implemented, I will have students in this class explicitly compare the set of issues that users experience *in practice* to the subset of issues identified by the help tools. In the process, students will learn about the strengths and weaknesses of different usability testing methods, but they will also learn about the goals of the research project and be part of assessing the representativeness of the issues reported.

I expect this integration to lead to several outcomes. For example, students may more express interest in participating in research or pursuing graduate school. Students may also be more engaged in the course material because it involves a goal that extends beyond the classroom. I will assess these outcomes by **comparing students in different offerings of the course**, tracking whether they pursue graduate school, what kinds of jobs they pursue after graduating, and whether they find the course useful in their careers.

4.3. TEACHING TEAMWORK IN A STUDIO-BASED PROJECT COURSE

Another way that I will integrate research and education is by offering a new course in which to disseminate the results of my research. This course, to be offered in 2011, will **explicitly teach teamwork aspects of software engineering** to *Computer Science* and *Informatics* students. Knowing how to work in a team is particularly important for software testers and usability practitioners, since people in these roles collaborate closely with developers, managers, and customers to ensure software quality [50].

The course will have two main components: (1) instruction about the fundamentals of team coordination and small group communication, and (2) a team software development project, in which students will apply this instruction to specific roles in their software team. While part of the course will focus on issue tracking, revision control, and code review skills, I will explain the importance of these skills by describing more theoretical observations about human aspects of software teams. For example, I will demonstrate that modularity and object-orientation, two of software engineering's most successful ideas, are essentially ways to reduce *coordination requirements* in teams [12, 21, 15, 66]. I will also teach the consequences of coordination challenges, explaining that most knowledge about bugs and software architecture is *tacit,* in that it rests in the minds of the team, but is not documented [6, 54]. This instruction will involve a great deal of industry participation, particularly from software quality professionals at Microsoft (see attached letter).

To assess the course, I will compare **peer evaluations**, **grades**, and **job placement outcomes** of students who take the course and students who do not. I will also maintain contact with students after they take jobs, gathering feedback about the impact of the course on their professional experiences. The results of these assessments and the course materials, will be made publicly available for other instructors.

4.4. Organizing a Peer and Professional Mentoring Event on Software Quality

A third way that I will integrate research and education is to bring together students and professionals in an **annual peer and professional mentoring event on software quality**. This event will involve a day of small group discussions and presentations meant to inform students about the day-to-day life of software testers and usability experts. Participants will include pre-major college and high school students interested in careers in the software industry. Mentors will include other students who have completed internships and industry professionals from local companies, including Microsoft, Amazon, and Boeing.

This event will serve many purposes. It will address a common student misperception that software development is anti-social and isolating; students will learn that, to the contrary, software and usability testing jobs involve a great deal of communication, persuasion, and discussion. As this may **increase enrollment in Computer Science and Informatics**, I will compare enrollment outcomes of students who do and do not attend the event. I also will use the event as an opportunity to systematically interview industry participants, learning about their team's approach to bug triage and how my proposed research might integrate into their processes. This will further ground my research in practice and foster new research ideas and opportunities for technology transfer.

5. Five-Year Project Timeline

Figure 8 shows the timeline for my research and education plans. The activities are grouped into three parallel activities: *prototypes* (in which my team implements systems that test the feasibility of tools), *deployments* (in which we collaborate with *Catalyst* teams to deploy and gather user feedback), and *education* (in which I plan and execute my teaching and outreach efforts).

The education aspects of the work will be ongoing efforts across all five years. For example, the peer and professional mentoring event and the new course on teamwork will begin in 2010 and can be planned independently of the other proposed work. I teach *User-Centered Design* annually. The collaborations with the *Catalyst Tools* team will also occur in parallel, but offset from the main research activities by about one year. This will allow my research team to prototype help into web applications to help *Catalyst* assess the feasibility of making the changes in their applications.

The main research efforts will exhibit more of a sequence. In **Year 1**, we will design automatic help-seeking tools for web applications. This work will focus largely on architectural and assessment aspects enabling and answering "why" questions. In **Year 2**, we will focus on reporting, privacy and consent issues, testing approaches with small-scale deployments of simple test applications; collaborations with the *Catalyst Tools* will begin. In **Year 3**, we will work on aggregation and grouping techniques. Students

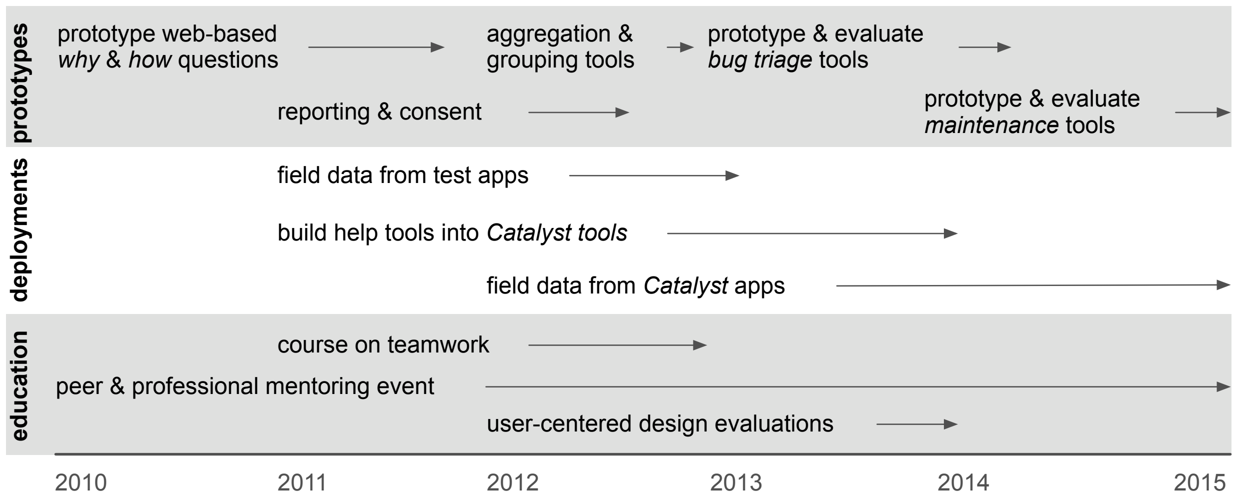


Figure 8. The timeline for the proposed research and educational plans.

in *User-Centered Design* will compare field data to data from traditional usability methods. In **Year 4**, we will prototype new bug triage tools and evaluate their effectiveness. In **Year 5**, we will prototype new software maintenance tools and evaluate their effectiveness. The predominance of field data in these later years and the increased size of my research team will enable a variety of projects to occur in parallel.

The budget for this proposal includes support for one graduate research assistant, but the project team will also include Ph.D. students with other forms of support, undergraduates and masters students working on senior capstone projects, and students in my classes. All students will be from both the *Information School* and *Computer Science*, but all will have software development backgrounds. In general, the *Information School* recruits several students with computer science and software development backgrounds who want to apply their knowledge in a human-centered way.

6. INTELLECTUAL MERIT AND BROADER IMPACTS

The **intellectual merits** of the proposed work include contributions to both scientific understanding of software development teamwork and technical designs of software technologies. Contribution to **knowledge** include new findings about current bug triage practices and their limitations, discoveries about the strengths and weaknesses of different software quality assessment methodologies, and assessments of the efficacy of a wide range of new technologies. Contributions to **design and technology** include a new class of automatic help tools that enable software to explain its behavior to users, methods of reporting and aggregating uses of these help tools as indicators of software issues, ways of filtering, processing, and analyzing help question data, and new forms of software maintenance tools that exploit the help question data. This research will also lead to several new research opportunities in more advanced help technologies and other kinds of feedback mechanisms to support software evolution.

The **broader impacts** of the proposed work will affect several populations. For **software users**, these impacts include enabling users to troubleshoot software, empowering users to influence software design through direct feedback, and increasing overall software quality and usability. These impacts will ultimately increase users' productivity and reduce frustration with technology. For **software companies**, impacts include several free and open source software maintenance tools, an ability to work objectively with data from the field, and thus improved success for their organizations. For **students**, impacts include a new course on teamwork in software design, increased relevance of education to students entering the software industry, increased student participation in research, usability improvements to courseware developed at the University of Washington, and opportunities to directly participate in research. Graduate students will be essential members of the project team, learning skills in field research, user interface design, usability evaluation, empirical studies, software engineering, and research prototyping.

7. CAREER OBJECTIVES AND ACCOMPLISHMENTS

My overall career goal is to **support human aspects of software engineering**; enabling and exploiting evidence-based bug triage is just one part of this larger goal. For example, my dissertation work sought to understand and support the "why" questions that software developers ask when they are debugging. Over the course of six years, I performed several studies about program understanding and debugging, spanning observational studies in educational settings [46], lab studies with expert developers [31, 47], and field studies of nearly twenty software development teams at Microsoft [16, 50]. The major finding of these studies was that debugging is difficult because of the *guesswork* that it involves*:* because developers cannot trace back directly from the visible symptoms of failures, they have to guess what is causing the failure. Their initial guesses are usually incorrect, costing considerable time.

To avoid this guesswork, I invented a tool called the *Whyline* [46, 49], which allows developers to click directly on the visible symptoms of failures and choose "why" questions about the properties of these symptoms. The tool then uses a both static and dynamic program analyses to isolate the causes of the failure, helping developers quickly navigate control and data dependencies. My prototypes dramatically reduced debugging time compared to regular breakpoint debugging tools [49, 52]. This work has received **international press**, **four best paper awards at top conferences**, and resulted in over 30 peer-reviewed articles in 7 venues across *Human-Computer Interaction* and *Software Engineering* venues.

In addition to this work, I have extensive experience in developing other kinds of software development tools. For example, Figure 9 contains snippets of my work on other debugging tools, novel code editors, new programming languages. I also have considerable background in designing performing empirical studies in the classroom, in the lab, and at software development companies. I also have over ten years of experience in *Computer Science*, *Psychology*, and *Design* disciplines, giving me a unique interdisciplinary perspective on *Software Engineering* research. With my adjunct status in Computer Science & Engineering, **I also plan to advise several Computer Science Ph.D. students**, in addition to more technically skilled Information School Ph.D. students.

The diversity of my expertise and my deep understanding of the nature of software development practice has also led to several strong ties to industry. In just one year in Seattle, I have established connections with over 30 local companies; I have worked closely with Visual Studio teams to support future versions of debugging and code navigation tools; I also work with the Firefox and Bugzilla open source community to understand their bug triage and global communication challenges. I intend to act as a key mediator between academia and the software industry throughout my career.

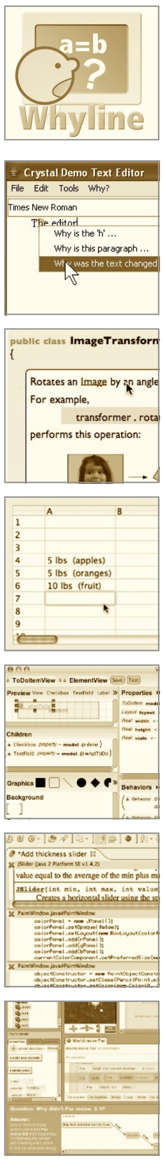


Figure 9. Software development tools I have invented.

8. RESULTS FROM PRIOR NSF SUPPORT

I have no prior research funding from NSF. I have received one service grant to support student travel for a 2009 graduate student consortium, bringing together 12 Ph.D. students from around the world to discuss their research on broadening participation in computing.

My Ph.D. work was supported by an *NSF Graduate Research Fellowship* and a *National Defense Science and Engineering Graduate Fellowship*. I was also supported by NSF grant IIS-0329090 entitled *Lowering the Barriers to Successful Programming*, awarded to my Ph.D. advisor Dr. Brad A. Myers of Carnegie Mellon University; I wrote significant portions of this grant and was the main executor of the work itself.

REFERENCES

1. Akers, D., Simpson, M., Jeffries, R., and Winograd, T. 2009. Undo and erase events as indicators of usability problems. *ACM Conference on Human Factors in Computing Systems*, Boston, MA, April, 659-668.

2. Ambriola, V. Bendix, L. and Ciancarini, P. (1990). The evolution of configuration management and version control. *Software Engineering Journal* , 5(6), November, 303-310.

3. Andreasen, M. S., Nielsen, H. V., Schrøder, S. O., and Stage, J. (2007). What happened to remote usability testing? An empirical study of three methods. *ACM Conference on Human Factors in Computing Systems*, San Jose, CA, April, 1405-1414.

4. Anvik, J., Hiew, L., and Murphy, G.C. (2006). Who should fix this bug? *International Conference on Software Engineering*, Shanghai, China, May, 361-370.

5. Anvik, J. and Murphy, G.C. (2007). Determining implementation expertise from bug reports. *International Workshop on Mining Software Repositories*, May, 2.

6. Aranda, J. and Venolia, G. (2009). The secret life of bugs: Going past the errors and omissions in software repositories. *International Conference on Software Engineering*, May, Vancouver, B.C., Canada, 298-308.

7. Baker, R.S., Corbett, A.T., and Koedinger, K.R. (2004). Detecting student misuse of intelligent tutoring systems. *Lecture Notes in Computer Science*, 3220, 531-540.

8. Baowen, X., Ju, Q., Xiaofang, Z., Zhongqiang, W., and Lin, C. (2005). A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2) 1-36.

9. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What makes a good bug report? *ACM SIGSOFT International Symposium on Foundations of Software Engineering* Atlanta, GA, November, 308-318.

10. Beyer, H. and Holtzblatt, K. (1998). *Contextual design: defining customer-centered systems*. San Francisco, Morgan Kaufmann.

11. Brabham, D.C. (2008). Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence: International Journal of Research into New Media Technologies* 14(1), 75–90.

12. Brooks, F.P. Jr. (1975). *The mythical man-month: Essays on software engineering*. Addison Wesley, Reading, MA.

13. Bruun, A., Gull, P., Hofmeister, L., and Stage, J. (2009). Let your users do the testing: a comparison of three remote asynchronous usability testing methods. *ACM Conference on Human Factors in Computing Systems,* Boston, MA, April, 1619-1628.

14. Castillo, J.C. (1997). The user-reported critical incident method for remote usability evaluation. Master thesis, Virginia Polytechnic Institute and State University.

15. Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. 2006. Identification of coordination requirements: implications for the design of collaboration and awareness tools. *ACM Conference on Computer Supported Cooperative Work*, Banff, Alberta, Canada, November, 353-362.

16. 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*, San Jose, CA, USA, April, 557-566.

17. Christensen, C.M. (2003). The innovator's dilemma. Harper Collins.

18. Christensen C.T., Feldman Barrett, L., Bliss-Moreau, E., Lebo, K. and Kaschub, C. (2003). A practical guide to experience-sampling procedures, *Journal of Happiness Studies*, 4, 53-78.

19. Clause, J. and Orso, A. (2007). A technique for enabling and supporting debugging of field failures. *International Conference on Software Engineering*, May, 261-270.

20. Cleve, H. and Zeller, A. (2005). Locating causes of program failures. *International Conference on Software Engineering*, St. Louis, MO, USA, May, 342-351.

21. Conway, M.E. (1968). How do committees invent? *Datamation* 14(5), 28-31.

22. Cuddihy, E., Wei, C., Bartell, A.L., Barrick, J., Maust, B., Leopold, S.S., and Spyridakis J.H. (2007). Conducting an automated experiment over the Internet to assess navigation design for a medical web site containing multipage articles. In G. Hayhoe and H. Grady (Eds.), *Connecting people with technology: issues in professional communication*. Farmingdale, NY: Baywood Publishers, 31-41.

23. Brabham D.C. (2008). Crowdsourcing as a model for problem solving: an introduction and cases. *Convergence: The International Journal of Research into New Media Technologies*, 14(1), 75-90.

24. de Souza, C. R., Redmiles, D., and Dourish, P. (2003). "Breaking the code": moving between private and public work in collaborative software development. *ACM SIGGROUP Conference on Supporting Group Work*, Sanibel Island, FL, USA, November, 105-114.

25. de Souza, C. R., Redmiles, D., Mark, G., Penix, J., and Sierhuis, M. (2003). Management of interdependencies in collaborative software development. *IEEE International Symposium on Empirical Software Engineering,* September, 294.

26. de Souza, C. R., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. (2004). Sometimes you need to see through walls: a field study of application programming interfaces. *ACM Conference on Computer Supported Cooperative Work*, Chicago, IL, USA, November, 63-71.

27. Dit, B. and Marcus, A. (2008). Improving the readability of defect reports. *International Workshop on Recommendation Systems for Software Engineering*, November, Atlanta, Georgia.

28. Eisenstadt, M. (1997). “My hairiest bug” war stories. *Communications of the ACM*, 40(4), 30–-37.

29. Farnham, S., Chesley, H. R., McGhee, D. E., Kawal, R., and Landau, J. (2000). Structured online interactions: improving the decision-making of small discussion groups. *ACM Conference on Computer Supported Cooperative Work*, Philadelphia, PA, USA, 299-308.

30. Flanagan, J. (1954) The critical incident technique. *Psychological bulletin*, 51(4), 327-358.

31. 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*, Portland OR, April, 331-340.

32. Frøkjær, E. and Hornbæk, K. 2005. Cooperative usability testing: complementing usability tests with user-supported interpretation sessions. *Extended Abstracts on the ACM Conference Human Factors in Computing Systems*, Portland, OR, USA, April, 1383-1386.

33. Glerum K., Kinshumann K. Greenberg S., Aul G., Orgovan V., Nichols G., Grant D., Loihle G., and Hunt G. (2009). Debugging in the (very) large: ten years of implementation and experience. *ACM Symposium on Operating Systems Principles*, Big Sky, MT, to appear.

34. Mark, G., Gonzalez, V. M., and Harris, J. (2005). No task left behind? Examining the nature of fragmented work. *ACM SIGCHI Conference on Human Factors in Computing Systems*, Portland, OR, USA, April, 321-330.

35. Gould, J. and Lewis, C. (1985) Designing for usability: key principles and what designers think. *Communications of the ACM,* 28 (3), 300-311.

36. Gulliksen, J., Boivie, I. and Göransson, B. (2006) Usability professionals—current practices and future development. *Interacting with Computers,* 18(4), 568-600.

37. Gutwin, C., Penner, R., and Schneider, K. (2004). Group awareness in distributed software development. *ACM Conference on Computer Supported Cooperative Work*, Chicago, IL, USA, November, 72-81.

38. Harper, F.M., Moy, D., and Konstan, J.A. (2009). Facts or friends? Distinguishing informational and conversational questions in social Q&A sites. *ACM Conference on Human Factors in Computing Systems*, Boston, MA, USA, April, 759-768.

39. Harrold, M.J. and Soffa, M.L. (1988). An incremental approach to unit testing during maintenance. *International Conference on Software Maintenance*, October, 362-367.

40. Hertzum, M. (2002). The importance of trust in software engineers' assessment of choice of information sources. *Information and Organization*, 12(1), 1-18.

41. Hilbert, D. M. and Redmiles, D. F. (2000). Extracting usability information from user interface events. *ACM Computer Surveys*, 32(4), December, 384-421.

42. Horrigan, J. (2008). When technology fails. Pew Internet and American Life Project. *http://www.pewinternet.org/pdfs/PIP_Tech_Failure.pdf*, retrieved June 1st, 2009.

43. Howarth, J., Andre, T. S., and Hartson, R. (2007). A structured process for transforming usability data into usability Information. *Journal of Usability Studies*, 3(1), 7-23.

44. Kiciman, E. and Livshits, B. (2007). AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. *ACM Symposium on Operating Systems Principles*, October.

45. Kirsch, D. and Maglio, P. (1994). On distinguishing epistemic from pragmatic action. *Cognitive Science* 18, 513-549.

46. 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*, Rome, Italy, September, 199-206.

47. 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.

48. Ko, A. J. DeLine, R., and Venolia, G. (2007). Information needs in collocated software development teams. *International Conference on Software Engineering*, St. Louis, MO, USA, May, 344-353.

49. Ko, A.J. and Myers, B.A. (2008) Debugging reinvented: asking and answering why and why not questions about program behavior. *International Conference on Software Engineering*, Leipzig, Germany, May, 301-310.

50. Ko A.J. and Chilana P. (2009). The anatomy of design discussions in contentious open source bug reports. In review.

51. Ko A.J. and Myers B.A. (2009). Attitudes and self-efficacy in young adults' computing autobiographies. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Corvallis, OR, to appear.

52. Ko A.J. and Myers B.A. (2009). Finding causes of program output with the Java Whyline. *ACM Conference on Human Factors in Computing Systems*, Boston, MA, USA, April, 1569-1578.

53. Koschke, R. and Quante, J. (2005). On dynamic feature location. *IEEE/ACM international Conference on Automated Software Engineering*, Long Beach, CA, USA, November, 86-95.

54. LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. *International Conference on Software Engineering*, Shanghai, China, May, 492-501.

55. Liblit, B. (2007) Cooperative bug isolation. *Lecture Notes in Computer Science*, Springer-Verlag: NY.

56. Lindgaard, G. and Chattratichart, J. (2007). Usability testing: what have we overlooked? *ACM Conference on Human Factors in Computing Systems*, San Jose, CA, USA, April, 1415-1424.

57. Liu C., Zhang X., Han, J., Zhang, Y., and Bhargava, B.K. (2007). Indexing noncrashing failures: a dynamic program slicing-based approach. *International Conference on Software Maintenance*, October, 455-464.

58. McConnell, S. (1994). *Code Complete*. 2nd Edition. Redmond, Wa.: Microsoft Press.

59. McDonald, D. W. and Ackerman, M. S. (1998). Just talk to me: a field study of expertise location. *ACM Conference on Computer Supported Cooperative Work*, Seattle, WA, USA, November, 315-324.

60. Millen, D. R. (1999). Remote usability evaluation: user participation in the design of a Web-based email service. *SIGGROUP Bulletin*, 20(1), April, 40-45.

61. Miller, R. C., Chou, V. H., Bernstein, M., Little, G., Van Kleek, M., Karger, D., and Schraefel, M. (2008). Inky: a sloppy command line for the web with rich visual feedback. *ACM Symposium on User interface Software and Technology*, Monterey, CA, USA, October, 131-140.

62. 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*, Montreal, Canada, April, 397-406.

63. Myers, B.A. and Kosbie, D.S. (1996). Reusable hierarchical command objects. *ACM Conference on Human Factors in Computing Systems*, Vancouver, BC, Canada, April 13-18, 260-267.

64. Nielsen, J. (1994) Using discount usability engineering to penetrate the intimidation barrier. In Bias, R. and Mayhew, D. (eds.) *Cost-justifying Usability*. Boston: Academic Press.

65. Norman, D. (2002). *The design of everyday things*. Basic Books.

66. Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.

67. Perlow, L. A. (1999). The time famine: toward a sociology of work time. *Administrative Science Quarterly*, 44(1), 57-81.

68. Perry, D. E., Staudenmayer, N. A., and Votta, L. G. (1994). People, organizations and process improvement. *IEEE Software*, July, 36–45.

69. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. *International Conference on Software Engineering*, Portland, OR, USA, May, 465-475.

70. Reeves, B. and Shipman, F. (1992). Supporting communication between designers with artifact-centered evolving information spaces. *ACM Conference on Computer-Supported Cooperative Work*, 394-401.

71. Reis, C.R. and de Mattos Fortes, R.P. (2002). An overview of the software engineering process and tools in the Mozilla project. *Open Source Software Development Workshop*, 155–175.

72. Rodham, K., Olsen, D.R. (1997) Nanites: An approach to structure-based monitoring. *ACM Transactions on Computer-Human Interaction*, 4(2).

73. Russel D., Kamvar S., Millen D., Butler K. (2009). Large data changes the way we think about HCI. *Human-Computer Interaction Consortium*, Winter Park, CO, USA.

74. Sandusky, R. J. and Gasser, L. (2005). Negotiation and the coordination of information and activity in distributed software problem management. *ACM SIGGROUP Conference on Supporting Group Work,* Sanibel Island, FL, USA, November, 187-196.

75. Sarma, A., Noroozi, Z., and van der Hoek, A. (2003). Palantír: raising awareness among configuration management workspaces. *Int'l Conference on Software Engineering*, Portland, OR, May, 444-454.

76. Scott, K.M. (2009). Is usability obsolete? *Interactions*, 16 (3), 6-11.

77. Seaman, C. B., V.R. and Basili, V. R. (1998). Communication and organization: an empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering*. 24(7), 559–572.

78. Sillito, J., Murphy, G. C., and De Volder, K. (2006). Questions programmers ask during software evolution tasks. *ACM SIGSOFT International Symposium on Foundations of Software Engineering* Portland, OR, USA, November, 23-34.

79. Spool, J. and Schroeder, W. (2001). Testing web sites: five users is nowhere near enough. *Extended Abstracts of the ACM Conference on Human-Factors in Computer Systems*, 285-286.

80. Steves, M. P., Morse, E., Gutwin, C., and Greenberg, S. (2001). A comparison of usage evaluation and inspection methods for assessing groupware usability. *ACM SIGGROUP Conference on Supporting Group Work*, Boulder, CO, USA, September, 125-134.

81. Tarter, J. (2008). Maintenance and services ratios. *Association of Support Professionals*. http://www.asponline.com.

82. Terry, M., Kay, M., Van Vugt, B., Slack, B., and Park, T. (2008). Ingimp: introducing instrumentation to an end-user open source application. *ACM Conference on Human Factors in Computing Systems* Florence, Italy, April, 607-616.

83. Thakur, A., Sen, R. Lu, S. and Liblit B. (2009). Cooperative crug isolation. *International Workshop on Dynamic Analysis*, Chicago, IL, USA, July, 35-41.

84. Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. (2007). Triage: diagnosing production run failures at the user's site. *ACM Symposium on Operating Systems Principles*, Stevenson, WA, Oct., 131-144.

85. Virvou, M. and Kabassi, K. (2000). An intelligent learning environment for novice users of a GUI. *International Conference on intelligent Tutoring Systems*, June, 484-493.

86. Vredenburg, K., Mao, J., Smith, P. W., and Carey, T. (2002). A survey of user-centered design practice. *ACM Conference on Human Factors in Computing Systems*, Minneapolis, MN, USA, April, 471-478.

87. Wang X., Zhang, L. , Xie, T. , Anvik, J., and Sun, J. (2008). An approach to detecting duplicate bug reports using natural language and execution information, *International Conference on Software engineering*, May, Leipzig, Germany.

88. Xu, W., Huang, L, Fox, A., Patternson, D., and Jordan, M. (2009). Mining console logs for large-scale system problem detection. *ACM Symposium on Operating Systems Principles*, Big Sky, MT, to appear.

89. Zhang, X. and Gupta, R. (2005). Whole execution traces and their applications. ACM Transactions on Architecture and Code Optimization. 2(3), September, 301-334.

90. Zeller A. and R. Hildebrandt (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), February, 183–200.

91. Zeller, A. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, SC, USA, November, 1-10.

BIOGRAPHICAL SKETCH

AMY J. KO, PH.D.

A. PROFESSIONAL PREPARATION

Oregon State University	Computer Science	B.S. 2002
Oregon State University	Psychology	B.S. 2002
Carnegie Mellon University	Human-Computer Interaction	Ph.D. 2008

B. APPOINTMENTS

Assistant Professor	The Information School, *University of Washington*, 2008 – present
Adjunct Assistant Professor	Computer Science & Eng, *University of Washington*, 2008 – present
Research Intern	*Microsoft Research*, Redmond, Summer 2006
Research Assistant	HCI Institute, *Carnegie Mellon University*, 2002 – 2008
Research Assistant	Computer Science Department, *Oregon State University*, 2000 – 2002

C. PUBLICATIONS RELATED TO THIS PROPOSAL

FIVE PUBLICATIONS RELATED TO THIS PROPOSAL

Ko A.J. and Myers B.A. (2009). Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology*, to appear.

Ko A.J. and Myers B.A. (2009). Finding causes of program output with the Java Whyline. *ACM Conference on Human Factors in Computing Systems* (CHI '09), Boston, MA, 1569-1578.

Ko, A. J., DeLine, R., and Venolia, G. (2007). Information needs in collocated software development teams. *International Conference on Software Engineering*, Minneapolis, MN, May 20–26, 344-353.

Myers, B. A., Weitzman, D., Ko, A. J., and Chau, D. H. (2006) Answering why and why not questions in user interfaces. *ACM Conference on Human Factors in Computing Systems*, Montreal, April, 397-406.

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*, Vienna, Austria, April, 151-158.

FIVE OTHER PUBLICATIONS

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*, Rome, Italy, Sept., 199-206.

Ko, A. J., and 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*, Montreal, Canada, April, 387-396.

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*, Portland, OR, 331-340.

Ko, A. J., Myers B. A., Coblenz, M. J., 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*, 33(12), 971-987.

Ko, A. J. and Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1-2), 41-84.

D. SYNERGISTIC ACTIVITIES

Development of course on teamwork on software engineering with software industry
Guest editor of IEEE Software special issue on End-User Software Engineering
Chair of doctoral consortium, IEEE VL/HCC 2009
Frequent reviewer for ACM CHI, ACM UIST, IEEE VL/HCC
Program committee member for ACM IUI, IEEE ICPC, IEEE VL/HCC

E. COLLABORATORS AND OTHER AFFILIATIONS

COLLABORATORS AND CO-EDITORS

Htet Htet Aung, *unaffiliated*, Alan F Blackwell, *Cambridge University*, Margaret M Burnett, *Oregon State University*, Duen Horng Chau, *Carnegie Mellon University*, Mauro Cherubini, *Telefonica I+D*, Michael J Coblenz, *Apple*, Robert DeLine, *Microsoft Research*, James Fogarty, *University of Washington*, Scott Hudson, *Carnegie Mellon University*, Thomas LaToza, *Carnegie Mellon University*, Brad Myers, *Carnegie Mellon University*, Mary Beth Rosson, *Penn State University*, Gregg Rothermel, *University of Nebraska Lincoln*, Christopher Scaffidi, *Carnegie Mellon University*, Gina Venolia, *Microsoft Research*, Susan Wiedenbeck, *Drexel University*, Jacob Wobbrock, *University of Washington*.

GRADUATE AND POSTDOCTORAL ADVISORS

Ph.D. advisor at Carnegie Mellon University: Dr. Brad A Myers
B.S. advisors at Oregon State University: Dr. Margaret Burnett (CS) and Dr. Bob Uttl (Psych)

PH.D. STUDENTS

Parmit Chilana, Information School, University of Washington
Michael Lee, Information School, University of Washington (starting Fall 2009)
Lydia Chilton, Computer Science & Engineering, University of Washington (starting Fall 2009)

B.S. THESIS SUPERVISION

Michael Coblenz, Carnegie Mellon University (now at Apple)
Ray Barnhart, University of Washington
Benjamin Berlin, University of Washington
Philip Phung, University of Washington
Don Bushell, University of Washington
Charles Ko, University of Washington

M.S. THESIS SUPERVISION

Michael Coblenz, Carnegie Mellon University (now at Apple)
Yoko Nakano, Carnegie Mellon University
Gregory Mueller, Carnegie Mellon University

STAFF SUPERVISION

David Weitzman, Carnegie Mellon University
Duen Horn Chau, Carnegie Mellon University
Sun Young Park, Carnegie Mellon University