CAREER: Inquisitive Programming Environments as Learning Environments for Novices and Experts

1 Overview, Significance, and Impact

Programming is a cognitively demanding task that requires **continuous learning**. But it is not just novices that are learning, since professional programmers must learn unfamiliar software projects, new libraries and frameworks, and even understand the design decisions and intent of code that other programmers wrote. This is particularly challenging since much of the information needed to complete programming tasks never gets written or shared. In fact, numerous studies have found that programmers spend a tremendous amount of time foraging for information and they often never find what they are looking for [69, 92, 93, 123]. Alleviating these barriers involved in learning during programming tasks is of paramount concern to support a diverse population of novices and experts to work in computing fields.

My overarching research plan is to **understand how inquisitive feedback loops can facilitate learning during software development tasks for novices and experts**. An inquisitive feedback loop is a system that elicits information from the programmer that is often locked away in their mind or someone else's mind, and then uses that information to provide more effective feedback. One motivating application of an inquisitive feedback loop is a system that detects complex code and entices the programmer to provide a structured explanation of the code snippet. Self-explanation has shown to be an an effective means of learning, especially in the context of learning to program, and the elicited explanation could be used to automate tedious programming tasks (e.g., writing documentation).

Toward this goal, I have three specific contexts I will design tools with inquisitive feedback loops for:

- (1) Supporting novice programmers in learning to effectively understand and explain how their programs work with CodeInquisitor.
- (2) Supporting expert programmers in learning, documenting, and sharing design decisions about code with CodeWitness.
- (3) Supporting programmers in the transition from novice to expert through onboarding with DevCourier.

By studying these three important contexts that pose considerable challenges to programmers, I strive to understand and validate the usefulness of inquisitive feedback loops through triangulation. Each context has unique characteristics that will come together to form a body of evidence on the effectiveness of inquisitive feedback loops.

I aim to advance knowledge in the areas of *human-computer interaction*, *software engineering*, and *computing education* to facilitate learning through self-explanation and revolutionize the ways in which programming tools can assist people during software development. In fact, I first validated an inquisitive feedback loop in my NSF CRII work for helping novices overcome misconceptions about their code. Our results were so promising that I received overwhelming support and feedback on our paper published [43] at the *ICSE'21 Education track* to continue this line of research. This proposal's research plan and education plan are tightly integrated through my industry collaborations, courses, and open source software projects. Therefore, I would like to take this promising approach and apply it to broader and more complex contexts during my research and education career.

2 PI Qualifications

My prior research can be broadly categorized as (1) studies on programmer's information seeking behavior [19, 92, 93, 94, 107] and (2) programming tools that provide information more effectively [1, 43, 44, 45, 47, 48, 49, 103]. In fact, I have conducted numerous qualitative studies to understand the barriers that programmers face, and then designed, implemented, and evaluated 8 programming tools in the last 7 years that are aimed at addressing the barriers we identified. Through my industry collaborations (e.g., with [redacted]) and academic collaborations (e.g., within and outside University of Tennessee), I am able to study programmers in both professional and educational settings. See the 8 letters of collaboration included with this proposal.

For example, we used a mixed-methods approach to study collaborative code reviews at Microsoft that involved programmers discussing and documenting design decisions about code changes. We designed CFar [48] to facilitate these design discussions by automatically displaying feedback in a way that includes the entire team. To evaluate our design, we conducted a small lab study, deployed the tool to 98 professional programmers for 15 weeks to collect log data, and surveyed 33 programmers to better understand their usage and opinion of the tool. Moving forward, I will use my experience, skills, and network to effectively conduct the research proposed to transform feedback loops for programmers.

3 Background & Related Work

There are many studies on the *information needs* of programmers and *tools* to support those information needs. However, much of the information may not exist outside of another programmer's mind (e.g., a design decision another programmer made [65, 77]) or the programmer may not yet realize they need information until going down the wrong path (e.g., a code change has cascading implications [65, 92]). Through an inquisitive feedback loop, programmers could benefit from *self-explanation* as a means of learning, documenting, and sharing knowledge.

3.1 Programmers' Information Needs

Developers need a variety of information to complete their development tasks. In fact, researchers have performed numerous empirical studies to better understand developers' dynamic information needs during development tasks (e.g., [65, 74, 75, 78, 95, 104, 108]). One notable field study found that programmers spent 58% of their time understanding code [123]. Another notable study identified 44 kinds of questions that developers ask during development tasks, such as *"Why isn't control reaching this point in the code?"* [104]. Similarly, researchers observed developers spending a substantial amount of their time trying to answer *reachability questions*, which are those concerned with about the paths in code or the relationships between code modules [74].

Despite programmers spending 35–50% of their time seeking information in code and documentation [69, 95], they are often unsuccessful in satisfying their information needs. In fact, developers often investigate irrelevant code in an attempt to find the code that will yield the information they need [69]. One cause for this may be that tools do not support developers' information needs, given that an analysis on questions that developers ask found that there is often a mismatch between the developer's question and the answers that tools provide [104]. Moreover, a survey of 179 professional developers revealed 94 distinct questions that they believe are hard to answer even with tools [75]. These difficulties to find information matches the results of

our recent study, which found that 50% of navigations yielded less information than developers expected, and 40% required more effort than the developer predicted [93]. Another key reason for the difficulty finding information is that it is often not written anywhere and requires asking another programmer [65, 77].

3.2 Tools for Satisfying Information Needs

To address the inefficiencies of development tools, researchers have designed tools for answering developers questions more efficiently. A popular approach is to integrate recommendation systems into the development environment (e.g., Mylar [64], Hipikat [116], Stacksplorer [62], the PFIS recommender [91], and Prodet [4]). In this context, a recommendation is any information that is "estimated to be valuable for a software engineering task" [99]. Such recommenders have utilized a variety of factors, often multiple, to produce recommendations of code. Notable factors include code structure [4, 50, 52, 62, 72, 76, 119], natural language [50, 124], navigation history [64, 91, 106], code edits [64], collaborative information [23, 106], and project documents [116].

A particularly relevant tool to our proposed research is the Whyline [66, 67, 68], which enables programmers to record program executions and then ask "why" and "why not" questions about the program's behavior. For example, a programmer can use the tool to click on a specific part of their program's interface and ask the question, "why is this button red?" The Whyline will then provide an answer (i.e., the relevant code) and allow the programmer to ask follow-up questions, such as "why did this event not occur?" To provide these answers, Whyline records an extensive amount of information from a program's execution and uses a combination of static and dynamic program slicing to derive the answers about the program behavior [67]. In an empirical study, participants using Whyline were able to fix bugs in half of the time than those using a traditional debugger [68].

Although Whyline is effective for debugging, it requires the programmer to have already identified a problem and to be able to select an appropriate question. In contrast, our proposed work on an inquisitive feedback loop is designed to support novices and experts *prior* to realizing there is a problem. For example, novices often have a different definition of program correctness [70] and they struggle with misconceptions even when their program does work as expected [63], so they would not be able to use the Whyline in such situations. Additionally, an inquisitive feedback loop can be used to assist expert programmers in tedious tasks that may be overlooked for extended periods of time (e.g., writing documentation or test code).

3.3 Self-Explanation

Self-explanation is the process of explaining a concept to one's self [20]. Studies have found that students that perform self-explanation in programming courses make fewer coding mistakes, have a better understanding of their programming solutions, and perform better on tests [11, 96, 117]. *Rubber duck debugging* is a common strategy among programmers that involves explaining your code to an inanimate objects so that "you must explicitly state things that you may take for granted when going through the code yourself. By having to verbalize some of the assumptions, you may suddenly gain new insights into the problem" [113]. Moreover, there is evidence that explanations can also be useful in group settings [12].

However, studies in other domains have found that it is rare for people to perform self-explanation without being prompted [20, 41]. For example, a seminal study on problem-solving strategies found that only 10% of participants self-explained on their own [20]. When participants are prompted to self-explain, they show positive learning outcomes and achieve similar

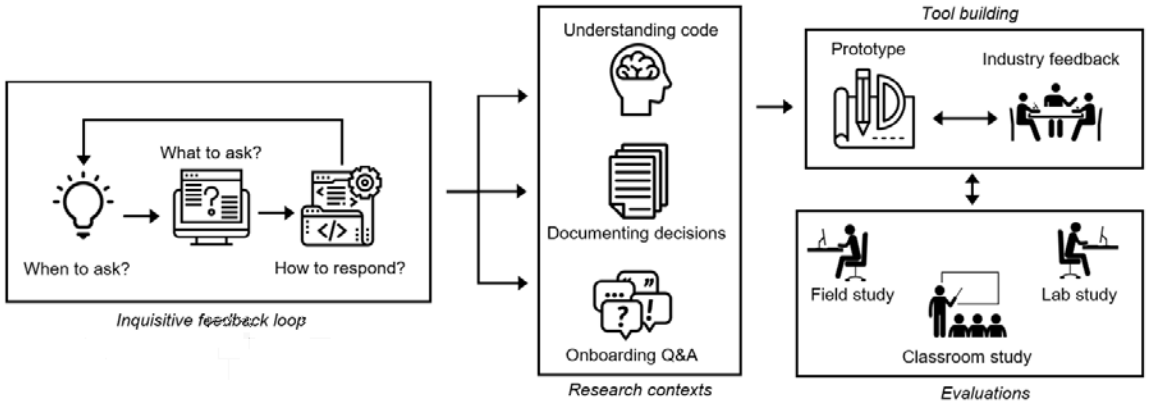

Figure 1: Overview of our proposed research to understand how inquisitive programming environments can be used as learning environments.

performance as those who self-explained without prompting [11, 21, 41]. These promising result motivates the need for further study on inquisitive feedback loops as a means of prompting novice and expert programmers to self-explain.

4 Proposed Research

Towards studying the effectiveness of inquisitive feedback loops in aiding programmers, I will design, implement, and evaluate three tools that support different information needs. In particular, the three contexts are: (1) novice programmers learning to understand and explain how their programs work, (2) expert programmers learning, documenting, and sharing design decisions about code, and (3) programmers transitioning from novice to expert during onboarding. By investigating these activities, we can study the effect on novices versus experts, individuals versus teams, and programming task (i.e., program understanding, documentation, and onboarding).

An overview of the proposed research is shown in Fig. 1. The inquisitive feedback loop is entirely dependent on the context, though the common elements include *when* to elicit information, *what* information to elicit, and *how* to use the information. We will then use a user-centered design approach to design and implement our tools, and then use mixed methods to evaluate them. The remainder of this section is organized into the three aims that strive to **understand how inquisitive feedback loops can promote learning during software development tasks for novices and experts**. Because of the overlapping nature of the three aims, including a shared codebase for implementing the tools, I believe the proposed research is feasible in a 5-year scope.

4.1 Aim #1: Support Novices in Program Understanding with CodeInquisitor

The number of people interested in learning computer programming has increased dramatically in recent years. However, novice programmers face numerous barriers while attempting to learn how to code that may deter them from pursuing a computer science degree or career in software development. To address the barriers that novices face while learning to code, I propose to build CodeInquisitor based on my NSF CRII research, a tool for eliciting explanations from novices about how they believe their program will behave while they are actively engaged in coding tasks.

4.1.1 Related work

Novice programmers make a lot of mistakes in their code [2, 9, 14, 28, 53, 57, 109]. Although the majority of the mistakes are syntax errors (e.g., unbalanced parentheses), these errors are often fixed quickly [2]. When the program behaves differently than expected, known as a semantic error, is also common [2, 14] but more challenging to fix since compilers often do not provide any warnings about them. A key reason that novice programmers have this issue is they often have a different definition of program correctness than professional programmers [70]. One study found that many students believe a program is correct when there are no compiler errors, but the students indicated little regard to the program's behavior [110]. Furthermore, students are often unable to accurately explain their own code [63, 79]. Even if the code produces the correct output, some students were still unable to explain the code or indicated a misconception about the behavior [63].

Researchers have proposed other techniques and tools that may alleviate various errors that novices make. One common approach to aiding students with syntax errors is for tools to provide enhanced compiler error messages that are easy to comprehend and may provide suggestions as to how to address the error. However, studies have had inconsistent or inconclusive results as to how effective they are in benefiting students [8, 24, 34, 86, 90, 98] Another technique is to provide visualizations of how the program will behave to aid programmers in understanding, such as Python Tutor, which displays visualizations of the program's data structures at each step of the code [36]. Other tools include Omnicode, which displays a scatterplot matrix of all run-time values for every variable in the program [61], Theseus, which annotates functions in the code editor with the number of times it was called during the current execution [81], and a tool extension that displays a small graph of how each variable changes over time during execution [51]. Researchers have also proposed tools that generate content to help programmers, such as hints [35, 42, 56, 89], examples [54, 55, 87], tutorials [39], and recommendations [32, 40, 56, 85, 112, 120]. Recently, researchers have began exploring how to enable users to efficiently tutor students over the web [37, 38, 118]. All of these approaches are synergistic with the proposed inquisitive feedback loop.

4.1.2 Preliminary work

My NSF CRII work led to an inquisitive code editor for addressing novice programmers' misconceptions of program behavior [43]. The inquisitive editor (see Fig. 2) follows a three step process: first, it identifies misconceptions by identifying potentially problematic code and prompting the programmer with specific multiple-choice or number-entry questions about the program's behavior. Second, it attempts to correct the misconceptions by providing an explanation based on the program's actual behavior. Third, it can prevent further misconceptions by inserting test code (e.g., assertions) and code comments.

Thus far we have prototyped the concept as a code editor plugin and gathered preliminary feedback from students and instructors. To identify potential misconceptions, we first investigated the viability of off-the-shelf code smell detectors (e.g., SonarSource). However, these tools are predominately designed for professional software developers. By examining the rules employed by such tools and showing them to undergraduate students, we found that the majority of them are of little to no value to novice programmers and could actually cause further confusion. In contrast, we trained machine learning models to identify code smells that are specific to mistakes that novices make. While it is trivial to detect individual mistakes reported by instructors and researchers (e.g., [2]), these may not generalize to other programming languages or programming

assignments. In an effort to overcome this limitation, we have combined several large data sets, including Blackbox [15], Stack Overflow, and GitHub.

From our initial results, we have found that the primary benefits come from the novice answering questions about their code and explaining the code to themselves. Originally, we were concerned with identifying and fixing specific code errors. However, by applying the inquisitive feedback loop more generally, we could support students in better understanding their code and also train them to do so more effectively.

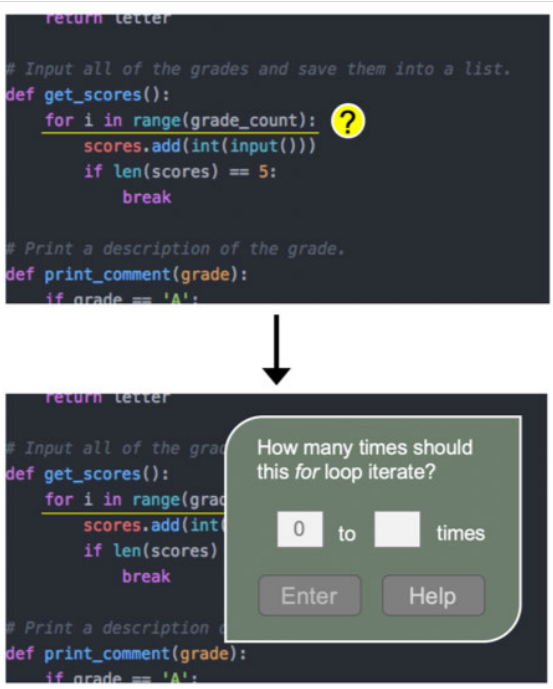

Figure 2: The Atom code editor with a preliminary prototype of our tool extension for identifying misconceptions. Code patterns that correlate with misconceptions are automatically annotated with a question mark (top). Clicking the annotation displays a question for the programmer to answer about how they think the program will behave (bottom).

4.1.3 Research plan

CodeInquisitor will extend my preliminary work by generalizing the inquisitive feedback loop in the context of novice programmers attempting to understand their code. In particular, we will expand the work beyond identifying and fixing specific misconceptions from a limited set of known problems that novices have. Instead, CodeInquisitor will elicit general explanations about code as a means of correcting high-level issues in their understanding. Interestingly, Lehtinen et al. recently discussed the potential benefits of automatically asking students about their code [80], which encourages the need of such work.

When to elicit explanations? A fundamental aspect of this work is knowing *when* to elicit an explanation that will be most beneficial to the novice while minimizing disruption. For example, asking for an explanation early in their coding session will yield unhelpful explanations (i.e., little to explain or not enough understanding yet to articulate an explanation) and asking too late means the learning opportunity has passed (i.e., they have already figured it out or given up). Additionally, there is the dimension of frequency of prompting. In our prior work on overcoming misconceptions, we prompted the novice with a specific code question whenever they wrote a line of code that was flagged by our code smell detector with a maximum of one prompt per two minutes. In that context, our configuration was adequate, likely because the novices believed it was the compiler helping them solve bugs and the compiler already gives frequent feedback in the form of errors.

We have identified a number of indicators that can be calibrated to trigger the elicitation event, including running the program, adding a breakpoint, or completing a block of code with sufficient complexity. In fact, our preliminary work found that repeatedly running the program or adding a breakpoint is indicative of a misconception. In contrast, passing a unit test (often provided by instructors) that was previously failing indicates a breakthrough. Both of these events may be useful times to elicit an explanation since students struggle with explaining their code, even when it appears to run correctly. Eliciting explanations that are not specific to a bug To entice

the novices to provide the explanations using CodeInquisitor and not ignore them, we will use *surprise-explain-reward* [101]. This is a technique that has been used to entice end-users to test software in a variety of contexts, such as spreadsheets, by provoking their curiosity and then rewarding them [17, 58, 122]. Additionally, we will use our classroom study to compare the different indicators and parameters (e.g., how frequently to elicit explanations), explained further in the Evaluation Plan.

What information makes for a good explanation? It is an open question *what* makes for a good explanation to help novices understand their code. In our prior work, we asked specific questions about code (i.e., "how many times will this loop iterate?"), which were able to aid in overcoming misconceptions, but only for a set of predefined issues. With CodeInquisitor we aim to assist them in understanding their code from a higher-level. However, it will not be adequate to simply give the novice a textbox and ask them to explain their code in natural language. We will investigate the novel idea of *structured self-explanations* that provide scaffolding to assist the novice in providing useful explanations with necessary details for CodeInquisitor to generate feedback. Our method to operationalize these structured self-explanations is by (1) displaying a set of automatically generated short explanations that (2) must be drag-and-dropped into the correct order by the novice and (3) allow some minor edits about key program behaviors. This idea is based heavily on Parsons problems for programming tests [88, 27], which involve selecting and rearranging a set of predefined code snippets to form a solution and have been shown to have a number of benefits over asking students to write code from scratch [25, 31, 30, 29, 84]. In fact, we can use similar techniques (i.e., program analysis and a database of pre-identified issues based on student data) to generate these structured prompts as we did for generating the code questions [43].

How can explanations be used to generate effective feedback? The elicited explanation can be used by CodeInquisitor to provide feedback to the novice that otherwise would not be possible. In our prior work, we used the answers to the specific code questions to generate short explanations as why they were correct or incorrect as well as inserting code documentation and assertions. Although the short explanation was beneficial to the students, the generated code documentation and assertions were not. They were either ignored, removed, or in one case, caused the student considerable confusion as to why the generated assertion was failing later. To overcome this short coming, CodeInquisitor will not generate any code or documentation for the novices, but instead act as an interactive debugging assistant. When a student provides an explanation that CodeInquisitor can detect is incorrect (using program analysis similar to our prior tool [43]), it will instruct the novice how to test if their explanation is correct. Afterwards, CodeInquisitor will display the novice's provided explanation and ask them to modify it until it is correct, allowing for multiple rounds of feedback.

4.1.4 Potential risks

- Enticing programmers to use our tools, rather than to ignore them, is a primary concern. To overcome this, we will incorporate techniques such as surprise-explain-reward [101].
- Identifying indicators of when to elicit explanations is essential for our tools to be beneficial. We will analyze several public sources of bugs and errors that novices introduce (e.g., Blackbox [15]) to identify correlations between problematic outcomes and debugging behaviors. Additionally, we will deploy the tool in classroom situations to calibrate the parameters.
- Overwhelming the programmer with information could greatly decrease the usefulness of our tools, so we will employ a user-centered design approach to frequently get feedback on our tool designs from programmers.

- Providing the right scaffolding for structured explanations is an open-ended problem. We will first investigate structured explanations in a laboratory setting to isolate the necessary information before integrating it into CodeInquisitor.
- Measuring the effect of such tools is complex and is difficult to discern. By using mixed methods we will triangulate our findings with quantitative and qualitative data from laboratory studies and classroom studies.

4.2 Aim #2: Support Experts in Documenting Decisions with CodeWitness

To "transcend the individual human mind" [3] remains a challenging and relevant problem, particularly in collaborative software engineering. A considerable amount of information about software design is never documented, likely due to the significant amount of effort required to document, organize, and maintain the information. Toward addressing the tedious and time consuming task of documenting design decisions about code, I propose to build CodeWitness, a tool for eliciting design decision rationale from experts while they are actively engaged in coding tasks.

4.2.1 Related work

Programmers often need information about code that is not written and thus requires seeking out another person who may have the answer [65, 77]. In one study, 66% of the participants agreed that understanding the rationale behind a piece of code is a serious problem and 39% said that finding the right person to talk to about a piece of code is a challenge they face [77]. The same study found that even when design documents do exist, programmers found them too difficult to use. Similarly, another study found that programmers often ask "why was this code implemented this way", but are often unable to find an answer [65].

4.2.2 Preliminary work

To facilitate discussion and documentation about design decisions in code, we designed CFar [48], a tool that automatically inserts feedback into code reviews and tracks them over time. Shown in Fig. 3, CFar's goal was to give programmers a starting point for discussions about code, since reviewers often fixate on low-level issues while ignoring high-level design problems and bugs [6, 102]. This is likely due to substantial time that code reviews take [5, 71, 77], despite less time spent on code reviews has been shown to correlate with more bugs [33, 83, 114]. Our studies on the effectiveness of CFar found that it increased communication, productivity, and code quality. Towards studying this problem from another perspective, we designed CodeDeviant [46], a tool that automatically records ad hoc tests (i.e., running the problem and manually testing inputs) that can be shared and rerun with teammates. Our initial evaluations of CodeDeviant found that programmers discovered significantly more bugs and fixed them in less time.

4.2.3 Research plan

The research on designing CodeWitness is divided into three steps: (1) identify events when design decisions are being made, (2) design a technique to elicit design decision rationale while minimizing disruptions, and (3) design a technique to document and share the rationale.

Identifying when design decisions are made in code. The first step in this project is to empirically study *when* design decisions are being made in the context of code. Since it is not possible to be a fly on the wall in the developers' offices, we will instead use their code changes as indicators

of design decisions. For example, a major refactoring may indicate that new design decisions have been made. As such, we will utilize existing techniques for large code-change detection (e.g., for refactoring code [115]) and code idiom detection (e.g., [60]). Once we have obtained a dataset of real-world instances of potential indicators, we can survey the developers to understand if and what decisions were actually made when they made the code changes.

Eliciting design decision rationale. Once we have validated the indicators that a design decision has been made, we can design CodeWitness to automatically detect the indicators and elicit more information about the design decision from the developer. This would entail annotating the relevant code in the code editor, and then when a developer clicks the annotation, prompting them with a question that asks for more information about the design decision. The set of possible questions would be generated using program analyzes so that it is specific to the context and reduces the amount of effort for the developer to answer. These annotations are based on *negotiable interruptions*, notifications that allow the user to attend to when they choose, and studies have found them to be preferred in long, continuous tasks [82].

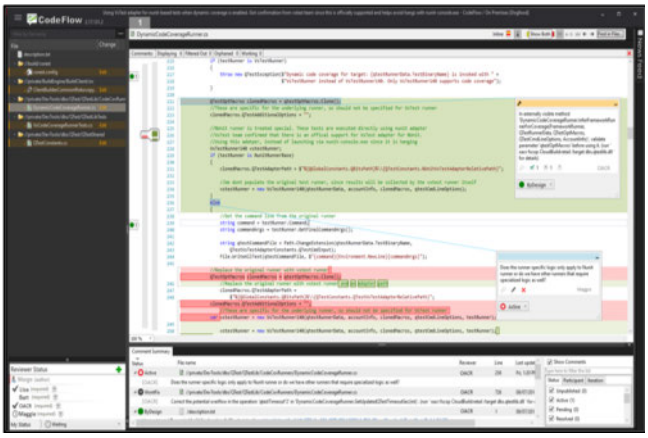

Figure 3: Our CFar tool integrated into Microsoft's CodeFlow code reviewing tool. CFar automatically inserts feedback into code reviews to initiate design discussions within teams.

Automatically generate and share documentation. Using the answer from the developer, the tool will automatically generate and share documentation given the specific code context. We will investigate how to effectively transform a developer's answer to the generated questions into documentation. The next feature of the tool will share the documentation in the relevant forms (i.e., comments in code, discussions in code reviews, and standalone design documents). It will do so by storing the comments in a standalone database and use a web service that integrates with other services, such as the code reviewing tool and code repository (as we did with CFar [48]). This integration is essential since programmers have already been shown to have difficulty in searching their documentation and instead will go ask someone [77]. To achieve a successful integration in our prior work, we deployed the tool to a small set of teams at Microsoft and used mixed-methods to rapidly iterate on the tool design before deploying to a larger population [48].

4.2.4 Potential risks

- The phrasing of questions is of paramount concern in order to elicit explanations that will serve as useful documentation to others. This open-ended research question will likely yield new knowledge to the research community that we will study by asking programmers from both perspectives (i.e., writing and reading explanations).
- Our prior work found that integrating into programmers' workflow is vital for adoption. We will seek feedback from our industry collaborators (see letters of collaboration) to iteratively solve these integration issues.
- To minimize disruptions to these programmers, we will use negotiable interruptions which have been found to be suitable in such contexts.

4.3 Aim #3: Support Q&A during Onboarding with DevCourier

Onboarding is the process that a programmer goes through to join a new software team or project. It is an important yet challenging time for organizations, since it is costly and time consuming for everyone involved [22, 73, 111]. In fact, one study found that programmers are not productive for the first several months of joining [105], and another found that programmers need several years to be fluent in a software project [125]. Toward addressing the barriers faced by new hires and teams during hiring, I propose to build DevCourier, a tool that semi-automatically records questions and elicits answers to technical questions.

4.3.1 Related work

The challenges of onboarding have been addressed with mentorship programs, pair programming, organized processes, and software solutions. Mentorship programs [7, 59]. An organized onboarding process [13, 73, 100]. Pair programming [10, 121]. Systems such as bots and collaborative agents have recently been applied to software development teams [26] as well as other onboarding contexts [16, 18]. Although each of these solutions are beneficial, the problem remains of getting information from an experienced team member to the new member efficiently.

4.3.2 Preliminary work

We studied the effect of annotations on code written by an expert for the use of onboarding new programmers to a project through Synectic [1], shown in Fig. 4. To validate annotations in Synectic, we conducted a user study comparing newcomer task support for information foraging and comprehension within a traditional programming environment (Eclipse) and our canvas-based tool. The results showed that providing the right information at the right place and time helped newcomers answer comprehension questions with significantly more accuracy and in less time.

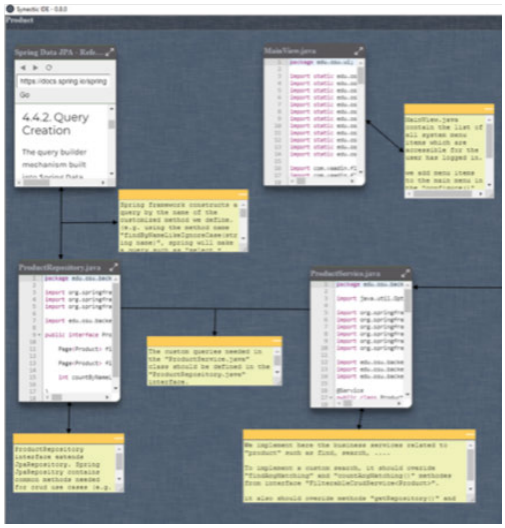

Figure 4: Synectic provides a canvas-based environment for arranging cards of information that can be linked to relevant code. The annotations are aimed to assist in onboarding programmers to a project.

4.3.3 Research plan

The research on designing DevCourier is divided into three steps: (1) investigate techniques for recording questions that new hires have, (2) efficiently elicit answers from experts automatically, and (3) archive the resulting Q&A where it can be used as a team resource in the future.

Record questions from new hires. DevCourier will attempt to capture questions that new hires have, even if they normally would not ask the question to another personThis is of interest since it is indicative of a team or project's lack of documentation. DevCourier will be made up of three components: a Google Chrome plugin, a Slack plugin, and a VS Code plugin that will use the

new hire's behavior to know when they are seeking information. Based on our information foraging research, we were successful in predicting when programmers need additional information [92, 94, 107]. We will use similar techniques to know when a new hire needs information, but is not successful in finding it. Using their search behavior, we will generate a customized question that a Slack bot poses to the new hire to elicit what information they are looking for.

Elicit answers from experts. Once a question has been posed to the CodeWitness Slack bot, it can try to solicit an answer from a relevant team member in a non-disruptive fashion. For example, CodeWitness can forward the question and contextual information, such as where the new hire looked for the answer, to the first set of team members that appear to be active in Slack (i.e., they will not be interrupted while outside of Slack). Their response and any artifacts (e.g., links) will automatically be logged by CodeWitness and returned to the new hire. CodeWitness will then monitor the interactions between these two users for a period of time to log any follow-up or clarification questions on the same topic.

Archive and share the Q&A. Once the question has been answered, CodeWitness will automatically archive the question and all relevant details in a shareable form that integrates into the team's existing tooling. With this documentation, future team members that have similar questions can find the answer in the documentation. Furthermore, CodeWitness will remember every question and answer it has facilitated and be able to reference them automatically when it detects similar questions. Bots are used extensively throughout the software development process, and we had success in using a bot to facilitate code reviewing and design discussions at Microsoft [48].

4.3.4 Potential risks

- Capturing questions that programmers have is a complex problem-space, but we have been successful in modeling programmers' information needs and we will mitigate this risk by developing CodeWitness as Chrome, VS Code, and Slack plugins.
- It is possible that programmers might ignore our bot. However, bots are used extensively throughout the software development process and we were successful in building a bot at Microsoft that saw promising adoption by teams [48].

5 Evaluation Plan

We will be applying an iterative user-centered design approach to develop our tools, and we will seek feedback throughout the process (see Fig. 5 for a timeline). Because of the overlap in studying each of the three aims, including a shared codebase for the tools and similar study designs for portions of the aims, I believe the proposed research is feasible in a 5-year scope. Our methods will include informal user studies, gathering feedback from industry researchers, laboratory studies, classroom studies, field deployments, and interviews. Throughout the project, we will conduct frequent *informal user studies* with small samples of programmers and instructors to gather feedback on our designs. In our prior work, we found that these informal opportunities to obtain feedback and identify usability problems gave us crucial ideas for our designs while taking little time to collect and analyze (e.g., [45, 48]). Moreover, we will conduct multiple *laboratory user studies* to compare our designs to that of other tools in a controlled environment. To assist in data analysis and statistics, we are in collaboration with [redacted] see letter of collaboration).

Since it is important for our work to be applied to realistic settings, we will collaborate extensively with industry researchers. We will periodically seek their feedback on our tool designs,

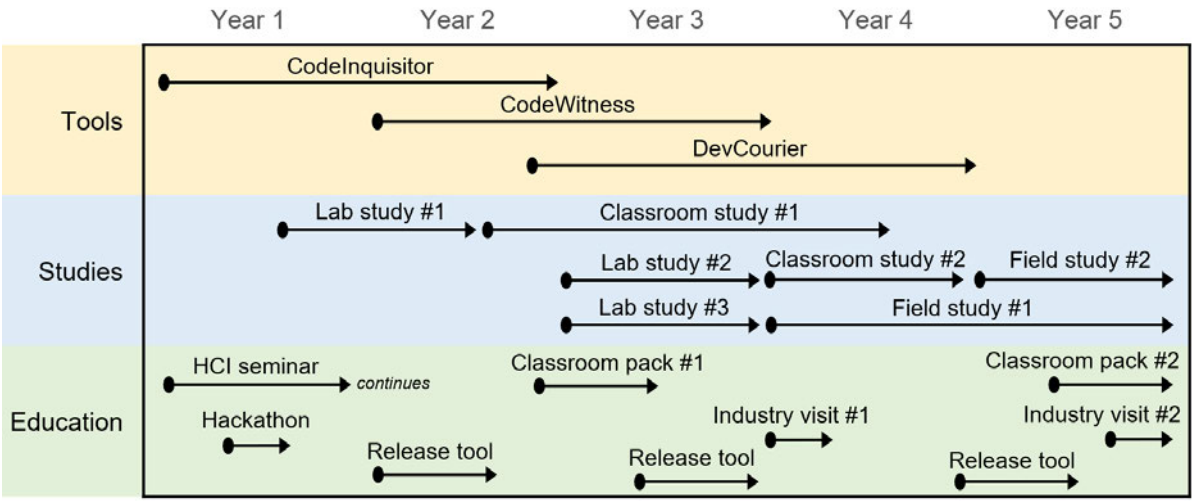

Figure 5: Timeline of the proposed work.

and also collaborate with them to conduct *field deployments* to gather usage data in professional settings. Prior to the field deployments, we will pilot the tools in an educational setting with *classroom studies*. Across all the tools and studies we will measure usage data to understand when the elicitations are ignored and how the generated feedback can be improved. When appropriate we will also gather subjective feedback and opinions on the tools and workflows through *semi-structured interviews*.

For CodeInquisitor, we are focused on measuring the quality and quantity of the elicited explanations. We will pilot the tool in lower-level computer science courses at University of Tennessee (see [redacted] letter of collaboration), which have 100+ students each per semester. The large amount of data will enable us to calibrate the tool to elicit an optimal number of explanations at opportune times. Then we will expand to training other instructors to adopt the tool for their courses (see [redacted] letter of collaboration from [redacted]), and eventually release a classroom package for any instructor to voluntarily use.

For CodeWitness, we are focused on identifying indicators of when design decisions are made, how to minimize disruptions, and the usefulness of the elicited explanations as documentation. We will first pilot the tool in upper-level undergraduate computer science courses at University of Tennessee that involve a significant group project (see [redacted] letter of collaboration), including my own COSC 340: Software Engineering course and my newly designed COSC 540: Advanced Software Engineering course. Then we will conduct field studies of the tool being used by professional programmers in ecologically valid settings (see the letter of collaboration from [redacted], [redacted], and [redacted]).

For DevCourier, we are focused on capturing the situations in which new hires have questions and effectively obtaining an answer from relevant experts. We will conduct a series of studies with industry partners to understand the complexities that arise during onboarding (see the letter of collaboration from [redacted]). Due to the two-sided nature of onboarding (new hire and mentor), it may be appropriate to use *wizard of oz* methodology to simulate one side in a controlled manner while studying the other (e.g., we will simulate the mentor's answers given a participant's questions).

For each our tools, we will release and open source as much of the data and tooling as is reasonable. Most interestingly, it could be beneficial to the software engineering community for us to release a well-polished website for each tool that includes instructions, demonstration videos,

source code, and binaries to reduce any barriers to adopting our tools. Additionally, I will travel to software companies to demonstrate our tooling, share our findings, and promote collaboration on our studies.

6 Education Plan

Integrating CodeInquisitor into introductory programming courses. We will integrate CodeInquisitor into the first and second year programming courses at University of Tennessee (see [redacted] letter of collaboration; [redacted]). From our preliminary findings, this tool should dramatically reduce the tedium and frustrations involved in learning to program help our department retain students that may otherwise leave computer science.

Organizing a hackathon for educational programming tools. In an effort to get more students involved in research, I plan to organize and host a hackathon where teams of students will design and build educational programming tools, similar to that of CodeInquisitor and CodeWitness. I will also work with the teams after the hackathon event to assist them in releasing and evaluating their tools, source code, and demonstration videos. Ideally, this will get more undergraduates involved in my department's research and graduate programs.

Pedagogy development based on an inquisitive feedback loop. We are developing new curriculum and pedagogy that emphasize self-explanation through an inquisitive feedback loop. We will release these materials for instructors to integrate CodeInquisitor into courses. Additionally, I have been part of my department's CS101 committee with the goal of reforming the course (and the early CS curriculum).

Applying CodeWitness to group project courses. We will teach students to use CodeWitness at University of Tennessee in courses that involve a group programming project. In collaboration with [redacted] (see [redacted] letter of collaboration), we will initially use CodeWitness in COSC 340: Software Engineering and COSC 423/523: Artificial Intelligence. After piloting the tool for a year, we will expand it to COSC 540: Advanced Software Engineering and COSC: 425 Machine Learning. Furthermore, we will work with faculty at other universities to use the tool in their courses (see the letter of collaboration from [redacted] at the [redacted]).

Educating professional programmers about CodeWitness and DevCourier. We will prioritize gaining mainstream adoption of our tools by maintaining documentation and tutorials for using CodeWitness and DevCourier. In fact, I will visit companies to demonstrate and share the tools as well as assist them in incorporating them into their software development teams.

Developing a HCI presence at UTK. I have been co-leading an effort to build a human-computer interaction presence at the University of Tennessee. This includes starting a regularly held HCI seminar across departments, integrating HCI into fundamental computer science courses (e.g., software engineering, cybersecurity, and machine learning), and designing new courses focused on HCI. Additionally, we created a shared laboratory that can support three user studies being held simultaneously, and it houses large computer displays, virtual reality headsets, and eye trackers.

Student Research Mentorship. My hands-on and applied teaching style has proven effective in getting undergraduate students involved in research. In fact, I have mentored 11 undergraduate research assistants in the last 3 years, of which 7 have continued on to a graduate program (although most of them originally had no intention to do so). I currently advise 6 graduate students, several of whom have won prestigious awards, such as the Tennessee Fellowship for Graduate Excellence and the departmental Outstanding Graduate Teaching Assistant Award. Additionally, I co-mentor a postdoctoral researcher that in his first year has led a successful multi-disciplinary

collaboration at the University of Tennessee. To grow my initiative of involving undergraduate students in research, I became the faculty advisor for our ACM student chapter and have guest lectured for numerous engineering courses about user-centered design.

.

7 Expected Outcomes

- Techniques for eliciting structured explanations about program behavior and using them to address flawed understandings.
- Techniques for eliciting, archiving, and sharing design decision rationale.
- Techniques for capturing questions from new hires and eliciting answers from relevant experts.
- Novel tool designs for inquisitive feedback loops in various software development contexts.
- Open source implementations of our tool designs (e.g., VS Code plugins).
- Empirical evidence on the benefits of our techniques and tools designs, as well as design guidelines for other researchers and tool builders.
- Adoption packages for industry teams to integrate our tooling into their software development workflows.
- Teaching materials for instructors to incorporate our inquisitive tools into programming courses.

8 Broader Impacts

Development of a diverse, globally competitive STEM workforce. The proposed tools aim to lower the barriers programmers face during programming, improve accessibility for people to learn how to program, and streamline the new hire experience. Since learning to program has been considered particularly tedious to learn initially, reducing the tedium could make the skill available to far more people. Overall, this project strives to make considerable impact by helping a diverse population of novice and expert programmers learn throughout programming tasks.

Improved STEM education and educator development at any level. Through our pilot studies on CodeInquisitor and CodeWitness in the classes at University of Tennessee, I will develop classroom packages and pedagogy that I will promote to other educators. I am also determined to amplify the works of other educators by collaborating with them to disseminate their own educational materials. In fact, I assisted a colleague in writing and promoting a free undergraduate operating systems textbook and blog aimed to reduce the barriers in learning about systems programming. Additionally, I am working with our lecturers (non-tenure track instructors) to be engaged in education research and to attend conferences such as SIGCSE.

Full participation of women, persons with disabilities, and underrepresented minorities in science, technology, engineering, and mathematics (STEM). To promote diversity in my university, I volunteered to be on my department's CS101 ad hoc committee with a goal of restructuring our lower-level undergraduate curriculum to attract and retain underrepresented groups by making the material more interactive, collaborative, and project-driven. Furthermore, I have applied findings from educational psychology to my undergraduate software engineering course, such as the contributing student pedagogy, flipped classrooms, and active learning. I am also encouraging students to create a more inclusive culture through our local ACM student chapter, which I am

the faculty advisor for. Through that, I will organize and host a hackathon to get undergraduates involved in building and evaluating programming tools aims to overcome issues based on their education experiences, and ideally, this will get more students involved in our research programs.

Enhanced infrastructure for research and education. My research and education plan are tightly coupled to integrating my software tools and learning materials in classrooms and beyond. Towards this goal, I will release classroom kits for educators and tool postmortems for tool designers including a comprehensive website with design guidelines, lessons learned, and source, such that they can integrate inquisitive feedback loops into their software, teams, and teaching.

Increased partnerships between academia, industry, and others. I have collaborated extensively with industry (e.g., [redacted]) and will continue to do so (see letters of collaboration). Previously, I was a Visiting Researcher at Microsoft in 2019 and did 5 internships during graduate school between 2013–2017 at IBM Research, Microsoft Research, National Instruments twice, and First Horizon. These relationships have been a paramount factor in my education and research career, and I intend to facilitate those same companies (and others) in recruiting students at University of Tennessee. I have also budgeted for trips to present our research to companies as way of promoting our tools and establishing new collaborations.

Increased public scientific literacy and public engagement with science and technology. To share my research projects to a broader audience, I started a technical blog that has been viewed over 600,000 times, received over 1000 discussion comments on Reddit and Hacker News, and I've received over 100 messages and emails from educators and students about their experiences related to my writings. One notable outcome of the blog's popularity includes myself being featured on Stack Overflow's podcast discussing our information foraging research. In the coming year, I plan to put more effort into these writings and specifically aim to release a series on user-centered design for non-technical readers as well as a series on creativity and exploratory, bottom-up design in engineering.

9 Prior Support

Sole PI on *CRII: CHS: Overcoming Novice Programmers' Misconceptions of Program Behavior* (#1850027; $174,956; 09/01/2019–08/31/2022) which produced the preliminary idea and work for an inquisitive feedback loop. *Intellectual Merits:* Published the inquisitive tool concept [43]. *Broader Impacts:* Released CodeRibbon as an open source tool, discussed our research on the Stack Overflow podcast [97], and started a research blog that has reached 600,000 views.

Co-PI on *CHS: SMALL: Collaborative Research: Adaptive Development Environments: Modeling and Supporting Cognitive Styles of Software Developers* (#2008408; $249,928 my portion out of $499,928 total; 10/01/2020–09/30/2022) that aims to model various program solving styles of programmers, and then integrate predictive models into developer tools to better support those needs. *Intellectual Merits:* Published a tool and study [1] on how providing the right information at the right time and place can better support new programmers. *Broader Impacts:* This work aims to make developer tools more accessible to programmers with varying problem solving styles.

References

[1] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma. Supporting code comprehension via annotations: Right information at the right time and place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, 2020.

[2] A. Altadmri and N. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM.

[3] E. Arias, H. Eden, G. Fischer, A. Gorman, and E. Scharff. Transcending the individual human mind—creating shared understanding through collaborative design. *ACM Trans. Comput.-Hum. Interact.*, 7(1):84–113, Mar. 2000.

[4] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Braunlich, and T. Fritz. A field study on fostering structural navigation with prodet. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 229–238, May 2015.

[5] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *ICSE*, pages 712–721. IEEE Computer Society, 2013.

[6] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *ICSE*, pages 931–940. IEEE Computer Society, 2013.

[7] S. Balali, I. Steinmacher, U. Annamalai, A. Sarma, and M. A. Gerosa. Newcomers'barriers. . . is that all? an analysis of mentors'and newcomers'barriers in oss projects. *Computer Supported Cooperative Work (CSCW)*, 27(3):679–714, 2018.

[8] B. A. Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, New York, NY, USA, 2016. ACM.

[9] B. A. Becker. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 296–301, New York, NY, USA, 2016. ACM.

[10] A. Begel and N. Nagappan. Pair programming: What's in it for me? In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, page 120–128, New York, NY, USA, 2008. Association for Computing Machinery.

[11] K. Bielaczyc, P. L. Pirolli, and A. L. Brown. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and instruction*, 13(2):221–252, 1995.

[12] K. Bielaczyc, P. L. Pirolli, and A. L. Brown. Collaborative explanations and metacognition: Identifying successful learning activities in the acquisition of cognitive skills. In *Proceedings of the sixteenth annual Conference of the Cognitive Science Society*, pages 39–44. Routledge, 2019.

[13] R. Britto, D. S. Cruzes, D. Smite, and A. Sablis. Onboarding software developers and teams in three globally distributed legacy projects: A multi-case study. *Journal of Software: Evolution and Process*, 30(4):e1921, 2018. e1921 JSME-17-0088.R2.

[14] N. C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[15] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 223–228, New York, NY, USA, 2014. ACM.

[16] C. J. Cai, S. Winter, D. Steiner, L. Wilcox, and M. Terry. "hello ai": Uncovering the onboarding needs of medical practitioners for human-ai collaborative decision-making. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), Nov. 2019.

[17] J. Cao, S. D. Fleming, M. Burnett, and C. Scaffidi. Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 27(6):640–660, Nov. 2015.

[18] P. Chandar, Y. Khazaeni, M. Davis, M. Muller, M. Crasso, Q. V. Liao, N. S. Shami, and W. Geyer. Leveraging conversational systems to assists new hires during onboarding. In R. Bernhaupt, G. Dalvi, A. Joshi, D. K. Balkrishan, J. O'Neill, and M. Winckler, editors, *Human-Computer Interaction - INTERACT 2017*, pages 381–391, Cham, 2017. Springer International Publishing.

[19] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik. *What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities*, page 1–12. Association for Computing Machinery, New York, NY, USA, 2020.

[20] M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, 13(2):145–182, 1989.

[21] M. T. Chi, N. De Leeuw, M.-H. Chiu, and C. Lavancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):439–477, 1994.

[22] J. Coelho and M. T. Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 186–196, New York, NY, USA, 2017. Association for Computing Machinery.

[23] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.

[24] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 273–278, New York, NY, USA, 2014. ACM.

[25] P. Denny, A. Luxton-Reilly, and B. Simon. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 113–124, New York, NY, USA, 2008. Association for Computing Machinery.

[26] J. Dominic, J. Houser, I. Steinmacher, C. Ritter, and P. Rodeghero. Conversational bot for newcomers onboarding to open source projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 46–50, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Y. Du, A. Luxton-Reilly, and P. Denny. *A Review of Research on Parsons Problems*, page 195–202. Association for Computing Machinery, New York, NY, USA, 2020.

[28] A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human Computer Studies*, 41(4):457–480, 1994.

[29] B. Ericson, A. McCall, and K. Cunningham. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, Koli Calling '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] B. J. Ericson, J. D. Foley, and J. Rick. Evaluating the efficiency and effectiveness of adaptive parsons problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 60–68, New York, NY, USA, 2018. Association for Computing Machinery.

[31] B. J. Ericson, L. E. Margulieux, and J. Rick. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling '17, page 20–29, New York, NY, USA, 2017. Association for Computing Machinery.

[32] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein. Emergent, crowd-scale programming practice in the ide. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2491–2500, New York, NY, USA, 2014. ACM.

[33] A. L. Ferreira, R. J. Machado, J. G. Silva, R. F. Batista, L. Costa, and M. C. Paulk. An approach to improving software inspections performance. In *2010 IEEE International Conference on Software Maintenance*, pages 1–8, 2010.

[34] T. Flowers, J. Jackson, and C. Carver. Empowering students and building confidence in novice programmers through gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.(FIE)*, volume 00, pages T3H/10–T3H/13 Vol. 1, 10 2004.

[35] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663, New York, NY, USA, 2014. ACM.

[36] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[37] P. J. Guo. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 599–608, New York, NY, USA, 2015. ACM.

[38] P. J. Guo, J. White, and R. Zanelatto. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 79–87, Oct 2015.

[39] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher. Automatically generating tutorials to enable middle school children to learn programming independently. In *Proceedings of the 12th International Conference on Interaction Design and Children*, IDC '13, pages 11–19, New York, NY, USA, 2013. ACM.

[40] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[41] R. G. Hausmann and M. H. Chi. Can a computer interface support self-explaining. *Cognitive Technology*, 7(1):4–14, 2002.

[42] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 3–12, Oct 2015.

[43] A. Henley, J. Ball, B. Klein, A. Rutter, and D. Lee. An inquisitive code editor for addressing novice programmers' misconceptions of program behavior. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 165–170, 2021.

[44] A. Z. Henley and S. D. Fleming. The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2511–2520, New York, NY, USA, 2014. ACM.

[45] A. Z. Henley and S. D. Fleming. Yestercode: Improving code-change support in visual dataflow programming environments. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 106–114, Sept 2016.

[46] A. Z. Henley and S. D. Fleming. Codedeviant: Helping programmers detect edits that accidentally alter program behavior. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 65–73, Oct 2018.

[47] A. Z. Henley, S. D. Fleming, and M. V. Luong. Toward principles for the design of navigation affordances in code editors: An empirical investigation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5690–5702, New York, NY, USA, 2017. ACM.

[48] A. Z. Henley, K. Muçlu, M. Christakis, S. D. Fleming, and C. Bird. Cfar: A tool to increase communication, productivity, and review quality in collaborative code reviews. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 157:1–157:13, New York, NY, USA, 2018. ACM.

[49] A. Z. Henley, A. Singh, S. D. Fleming, and M. V. Luong. Helping programmers navigate code faster with Patchworks: A simulation study. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '14, pages 77–80, July 2014.

[50] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng. (ASE*, pages 14–23, 2007.

[51] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 532:1–532:12, New York, NY, USA, 2018. ACM.

[52] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, May 2005.

[53] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 153–156, New York, NY, USA, 2003. ACM.

[54] M. Ichinco, W. Hnin, and C. Kelleher. Suggesting examples to novice programmers in an open-ended context with the example guru. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 230–231, Sept 2016.

[55] M. Ichinco, W. Y. Hnin, and C. L. Kelleher. Suggesting api usage to novice programmers with the example guru. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 1105–1117, New York, NY, USA, 2017. ACM.

[56] M. Ichinco and C. Kelleher. Semi-automatic suggestion generation for young novice programmers in an open-ended context. In *Proceedings of the 17th ACM Conference on Interaction Design and Children*, IDC '18, pages 405–412, New York, NY, USA, 2018. ACM.

[57] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Proceedings Frontiers in Education 35th Annual Conference*, pages T4C–T4C, Oct 2005.

[58] W. Jernigan, A. Horvath, M. Lee, M. Burnett, T. Cuilty, S. Kuttal, A. Peters, I. Kwan, F. Bahmani, and A. Ko. A principled evaluation for a principled Idea Garden. In *Proc. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*, pages 235–243, Oct. 2015.

[59] M. Johnson and M. Senges. Learning to be a programmer in a complex organization. 22(3):180–194, 2021/07/19 2010.

[60] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[61] H. Kang and P. J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 737–745, New York, NY, USA, 2017. ACM.

[62] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.

[63] C. Kennedy and E. T. Kraemer. Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 224–230, New York, NY, USA, 2019. Association for Computing Machinery.

[64] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 1–11, New York, NY, USA, 2006. ACM.

[65] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[66] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[67] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[68] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[69] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, Dec. 2006.

[70] Y. B.-D. Kolikant. Students' alternative standards for correctness. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 37–43, New York, NY, USA, 2005. ACM.

[71] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1028–1038, New York, NY, USA, 2016. Association for Computing Machinery.

[72] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers. Blaze: Supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2195–2200, New York, NY, USA, 2012. ACM.

[73] A. Labuschagne and R. Holmes. Do onboarding programs work? In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 381–385, 2015.

[74] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194. ACM, 2010.

[75] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[76] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124. IEEE, 2011.

[77] T. D. LaToza, G. Venolia, and R. DeLine. *Maintaining Mental Models: A Study of Developer Work Habits*, page 492–501. Association for Computing Machinery, New York, NY, USA, 2006.

[78] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Softw. Eng.*, 39(2):197–215, Feb. 2013.

[79] T. Lehtinen, A. Lukkarinen, and L. Haaranen. Students struggle to explain their own program code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[80] T. Lehtinen, A. L. Santos, and J. Sorva. Letâ€™s ask students about their programs, automatically. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 467–475, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[81] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM.

[82] D. C. McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction*, 17(1):63–139, 2002.

[83] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 192–201, New York, NY, USA, 2014. Association for Computing Machinery.

[84] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial. Subgoals help students solve parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 42–47, New York, NY, USA, 2016. Association for Computing Machinery.

[85] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann. Crowdsourcing suggestions to programming problems for dynamic web development languages. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1525–1530, New York, NY, USA, 2011. ACM.

[86] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 168–172, New York, NY, USA, 2008. ACM.

[87] S. Oney and J. Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2697–2706, New York, NY, USA, 2012. ACM.

[88] D. Parsons and P. Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163, 2006.

[89] B. Peddycord III, A. Hicks, and T. Barnes. Generating hints for programming problems using intermediate output. In *Educational Data Mining 2014*. Citeseer, 2014.

[90] R. S. Pettit, J. Homer, and R. Gee. Do enhanced compiler error messages help students?: Results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 465–470, New York, NY, USA, 2017. ACM.

[91] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, CHI '12, pages 1471–1480, New York, NY, USA, 2012. ACM.

[92] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? How production bias affects developers' information foraging during debugging. In *31st IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 11–20, 2015.

[93] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett. Foraging and navigations, fundamentally: Developers' predictions of value and cost. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 97–108, New York, NY, USA, 2016. ACM.

[94] D. Piorkowski, S. Penney, A. Z. Henley, M. Pistoia, M. Burnett, O. Tripp, and P. Ferrara. Foraging goes mobile: Foraging while debugging on mobile devices. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–17, Oct 2017.

[95] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3063–3072. ACM, 2013.

[96] P. Pirolli and M. Recker. Learning strategies and transfer in the domain of programming. *Cognition and Instruction*, 12(3):235–275, 1994.

[97] B. Popper, R. Donovan, and A. Henley. Podcast 347: Information foraging - the tactics great developers use to find solutions, Jun 2021.

[98] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 74–82, New York, NY, USA, 2017. ACM.

[99] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, July 2010.

[100] P. Rodeghero, T. Zimmermann, B. Houck, and D. Ford. Please turn your cameras on: Remote onboarding of software developers during a pandemic. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 41–50, 2021.

[101] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol.*, 10(1):110–147, Jan. 2001.

[102] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, pages 598–608. IEEE Computer Society, 2015.

[103] A. C. Short and A. Z. Henley. Towards an empirically-based ide: An analysis of code size and screen space. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 199–203, 2019.

[104] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July 2008.

[105] S. Sim and R. Holt. The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering*, pages 361–370, 1998.

[106] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[107] A. Singh, A. Z. Henley, S. D. Fleming, and M. V. Luong. An empirical evaluation of models of programmer navigation. In *IEEE Int'l Conference on Software Maintenance and Evolution*, ICSME '16, pages 9–19, 2016.

[108] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[109] J. C. Spohrer and E. Soloway. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, July 1986.

[110] I. Stamouli and M. Huggard. Object oriented programming and program correctness: The students' perspective. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 109–118, New York, NY, USA, 2006. ACM.

[111] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '15, page 1379–1392, New York, NY, USA, 2015. Association for Computing Machinery.

[112] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 107–115, Oct 2017.

[113] D. Thomas and A. Hunt. *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional, 2019.

[114] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 168–179, 2015.

[115] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494, 2018.

[116] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th Int'l Conf. on Software Engineering (ICSE '03)*, pages 408–418, 2003.

[117] A. Vihavainen, C. S. Miller, and A. Settle. Benefits of self-explanation in introductory programming. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 284–289, 2015.

[118] J. Warner and P. J. Guo. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 1136–1141, New York, NY, USA, 2017. ACM.

[119] F. W. Warr and M. P. Robillard. Suade: Topology-based searches for software investigation. In *Proc. ICSE*, pages 780–783, 2007.

[120] C. Watson, F. W. Li, and J. L. Godwin. Bluefix: using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*, pages 228–239. Springer, 2012.

[121] L. Williams, A. Shukla, and A. Anton. An initial exploration of the relationship between pair programming and brooks' law. In *Agile Development Conference*, pages 11–20, 2004.

[122] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 305–312, New York, NY, USA, 2003. ACM.

[123] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.

[124] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, New York, NY, USA, 2002. ACM.

[125] M. Zhou and A. Mockus. Growth of newcomer competence: Challenges of globalization. FoSER '10, page 443–448, New York, NY, USA, 2010. Association for Computing Machinery.