Improving Students' Program Comprehension in Large Code Bases

Adalbert Gerald Soosai Raj and William G. Griswold

Overview

A plethora of works in computing education have identified the academia-industry gap—the gap between expectations of new graduates' abilities in the workforce and their skills learned from undergraduate computer science programs. An eye-opening study in 2018 concluded that "in spite of nearly two decades since the gap in industry/academic coding experiences was identified and nearly ten years since researchers made recommendations for curricular changes to address the gap, it is still quite wide" [10]. There is a notable difference between industry work, which typically includes adding code to large, existing code bases, and university coursework, which typically includes writing small, stand-alone programs from scratch. As a result, prior works have consistently called for students to work on large, existing code bases in university courses [4, 10].

We will address the academia-industry gap by focusing on a specific process of working on a large code base: program comprehension. Program comprehension accounts for more than 50% of the time to modify or debug part of a large code base [29] and impacts productivity, communication, and quality of code changes. Despite the importance of program comprehension, limited work has focused on *teaching* program comprehension strategies in large code bases [31]. In fact, Tony Clear wrote in an ACM SIGCSE Bulletin in 2005: "Our state of the art seems akin to studying reading without the key notions of comprehension level or reading age" [8].

The PIs of this proposal have already made progress towards teaching students comprehension strategies in large code bases. Specifically, the PI designed and taught a course called "Working with Large Code Bases" in Spring 2023 in which students learned code navigation, code comprehension, and code management techniques for large, pre-existing code bases. Using this course as a starting point, we will conduct a three-phase research plan to 1) identify student struggles related to program comprehension in a large code base, 2) design a research-based, scalable curriculum to address those struggles, and 3) evaluate the curriculum on students' program comprehension strategies and abilities. We will ground our work in relevant theories—the Block Model (a theory of program comprehension) and Cognitive Apprenticeship (a teaching and learning theory)—to design a curriculum that imparts effective comprehending strategies in a large code base.

Intellectual Merit

The proposed work will advance our understanding of students' struggles while comprehending part of a large code base. These findings will lay the groundwork for future research to develop and evaluate pedagogical approaches to address these struggles related to program comprehension strategies in a large code bases—a notable gap in prior work. By grounding our empirical work to influential theories, we seek to advance our theoretical understanding of students' comprehension strategies and how we can use teaching approaches to impact effective strategies. This theoretical underpinning will help bridge empirical findings to a theoretical basis in computing education research.

Broader Impacts

Our work will help level the playing field between students who get internships (which may correlate to prior programming experience before college) and those who are learning computing for the first time in college by offering the experience of working in a large code base. Authentic, hands-on learning environments, like the one we aim to develop, are shown to increase student enthusiasm for computing and persistence in the field, especially for students from underrepresented groups. Further, since our goal is to significantly scale up the course size and make our course materials freely-accessible, we hope that anyone interested in learning strategies for working with large code bases can benefit from our work.

Improving Students' Program Comprehension in Large Code Bases

IUSE Engaged Student Learning—Exploration and Design

1 Introduction

Years of research into the academia-industry gap, which describes the gap between industry expectations and students' abilities, in the software engineering field has revealed deficiencies in new developers' ability to adjust to working on a large, pre-existing code base. Interviews from new developers and managers highlight the difference between the type of work done in industry—which typically involves developers contributing to a large code base—and academia—which typically involves students developing their own programs from scratch. The goal of this proposal is to develop and evaluate a curriculum to address part of the academia-industry gap. Specifically, this work will focus on program comprehension—the most time consuming process in a developer's workflow, yet an under-emphasized learning goal in software engineering courses.

The computing education research field includes a sizable amount of literature on novices' program comprehension skills [39, 47], including a Working Group at the 2019 ITiCSE conference [23]. However, the vast majority of these works study novice program comprehension strategies in *small* programs that are typically contained to a single file [23, 31, 32, 39, 47]. On the other hand, a significant body of work focuses on how expert and professional software developers comprehend their large code bases [37, 45, 54, 65]. Despite these two areas of work, there is little to no work within computing education research that covers how to *teach* code comprehension strategies [31], especially in the context of large code bases.

The impact of limited emphasis on teaching code comprehension in large code bases can lead to struggles among new developers who did not learn or experience how to comprehend a large code base. Begel and Simon conducted a months-long observational study of new software developers at Microsoft and identified several areas of deficiencies, including debugging, testing, and code comprehension in a large code base [5]. One of the key findings from their work was that students struggled to manage the complexity of a large system, especially with learning about all the interconnected parts of the code base [5]. More recently, Craig et al. conducted interviews with early-career software developers. A common theme that emerged from the interviews was that the majority of developers had not worked with legacy code bases in college [10]. One developer remarked: "The biggest difference between school and real life is that everything I did in college was greenfield and almost nothing since" [10]. A key difference in the process of working on a project from scratch and the process of working on an existing code base is the program comprehension step to understand the existing code. As a result, *both* studies mentioned above include the same recommendation to universities: explicitly teach students how to work on a large, pre-existing code base [4, 10].

We propose to fill the gap between the significant body of work to identify new developers' struggles when starting a software engineering job [4, 10, 42, 43] and the many explicit calls to action to *teach* students about working on a large code base [4, 6, 10]. Specifically, our work aims to 1) identify students' struggles to comprehend a large code base in terms of the Block Model, 2) design and scale a curriculum based on the Cognitive Apprenticeship learning theory to address those student struggles, and 3) evaluate the curriculum on students' ability to comprehend, explain, and modify code in a large code base. The first step will be to conduct observations and think-aloud interviews to identify student struggles as they try to comprehend a large code base. We will contextualize our findings using the Block Model—a theory of program comprehension that describes the separate aspects of a program that should be understood. Following this

step, we will leverage key methods of the Cognitive Apprenticeship learning theory (modeling, scaffolding, coaching, articulation, reflection, and exploration) to redesign our existing course to address the struggles related to comprehending large code bases. Finally, to evaluate our curriculum, we will compare students who have taken our course to those who have completed their required CS courses *without* taking our course. Specifically, we will analyze the impact of our course on students' comprehension, explanations, and productivity on the tasks within a "Skill Demonstration" that we design in which students must complete a small modification to a large code base. By the end of our work, we will produce a full curriculum design and a set of teaching approaches to improve students' ability to comprehend a large code base upon graduating from an undergraduate CS program. We aim for our work to contribute to a more diverse and productive set of CS graduates who will be immediate contributors to the country's workforce.

2 Background

2.1 Theoretical Frameworks

A Program Comprehension Theory: The Block Model

Theories of program comprehension describe the process of comprehension as a "top-down" process guided by hypotheses about the code [7], a "bottom-up" process in which lines or blocks of code are grouped into abstractions [41], or an "as-needed" process where programmers use various strategies based on the task at hand [33]. Though many such program comprehension theories exist, we will focus on the Block Model presented by Schulte. The Block Model synthesizes elements of well-established program comprehension theories [47] and was specifically designed to support research *and teaching* of program comprehension. The Block Model, depicted in Table 1 consists of 12 "blocks" that each represent one "aspect of the understanding process"(i.e., one step of the program comprehension process and one learning goal in a course) [46]. The 12 blocks are arranged across three dimensions—Program Text, Program Execution, and Program Function. Each of these three dimensions are a different aspect of the program to understand (the Text pertains to the written programming language, the Execution refers to the order and flow of the program, and Function refers to the meaning of the code). Each dimension includes a hierarchy of elements—Atoms (the smallest level), Blocks, Relations, and Macro Structure (the largest level)—that covers a different comprehension level of the program.

Importantly, the Block Model can apply to either students comprehending a single program text (a micro-sequence) *or* an instructor designing a course that covers various blocks (a macro-sequence). Schulte notes that when planning programming courses, instructors may vary the sequence in which they introduce the blocks and may not need to cover all 12 blocks in a course [46]. For example, one sequencing option is to start teaching the Atoms of the Program Text dimension and build towards the Macro Structure of the Program Function in the top right. Alternatively, instructors could start at the Program Function dimension by introducing common code constructs (i.e., an accumulator pattern) and their purposes before explaining the language elements and execution of those constructs. Despite the flexibility afforded to instructors in sequencing the blocks, Schulte notes that educators must take care to teach students to comprehend both the Structure *and* Function of a program (rather than comprehending the Structure without understanding the purpose of the program) [46].

Many of the papers that cite Schulte's Block Model tend to focus on shorter, novice-level programs [3, 19, 22, 24, 26]. We will use similar approaches to these prior works that evaluated

Macro Structure	Understanding the overall structure of the program text	Understanding the "algorithm" of the program	Understanding the purpose of the program (in its context)
Relations	References between blocks, e.g.: method calls, object creation, accessing data	Sequence of method calls	Understanding how function is achieved by subfunctions/subgoals
Blocks	'Regions of Interests' (ROI) that syntactically or semantically build a unit	Operation of a block, a method, or a ROI (as sequence of statements)	Function of a block, maybe seen as subgoal
Atoms	Language elements	Operation of a statement	Function of a statement
	Program Text	**Program Execution**	**Program Function**
Duality	**Structure**		**Function**

Table 1. The Block Model [46]

the Block Model, but will focus our analysis to the context of large code bases. We aim to evaluate and potentially extend the model to explain the comprehension process for large code bases. Schulte calls for further research into 1) code reading and comprehension strategies, which he writes are "neglected topics" in our curricula, and 2) "code purpose" questions that ask students to explain and summarize the function of code. We will map specific lecture topics and program comprehension strategies to blocks from the Block Model. By the end of our work, we aim to contribute to a broader theoretical understanding of students' program comprehension strategies *in large code bases*.

Cognitive Apprenticeship

Program comprehension is a process [7, 20, 46]. In fact, many works regarding program comprehension compare the processes that experts use to the processes novices use [45, 64]. Fortunately, the central tenet of the Cognitive Apprenticeship (CA) learning theory is for an expert to "make their thinking visible" to learners to facilitate the transfer of *implicit processes and expertise* [9]. Specifically, the theory enumerates 6 teaching methods—modeling, scaffolding, coaching, reflection, articulation, and exploration—to help learners observe, practice, refine, and master the strategies needed to complete a task [9]. CA has been empirically evaluated across many reasoning-based fields, such as engineering, writing, and nursing [12]. The PI of this proposal has used CA methods extensively in the past while working to evaluate live coding—a modeling method—on students' code writing process. Based on this prior experience, the PI conducted a literature review that synthesizes the use of CA approaches within computing education research that appears in the SIGCSE 2024 Technical Symposium [48]. The literature review reveals a strong emphasis on modeling, scaffolding, and coaching methods but an under-emphasis on reflection, articulation, and exploration methods. However, the review uncovered several benefits of Cognitive Apprenticeship methods, such as improved student enthusiasm for computing [25, 30, 57], retention in computing courses [2, 11, 27, 28], and higher pass rates [14, 61]. Several works also found that Cognitive Apprenticeship methods helped instructors manage a larger course size by improving students' self reliance, leading to less help requests [15, 38, 58].

Our proposed work will draw upon the PI's experience and knowledge of CA teaching methods and will apply such methods for program comprehension in a large code base. For example, we will leverage the *modeling* method to demonstrate an instructor's approach to understanding part of the code base, the *articulation* method by asking students to use code explanations as a learning exercise, and the *exploration* method by assigning an open-ended task in which students must draw upon all the techniques they've learned to complete a task on their own. By the end of our work, we aim to associate teaching methods to learning outcomes and evaluate the effectiveness of those methods on students' program comprehension abilities in a large code base.

2.2 Ineffective Program Comprehension Strategies in Large Code Bases

This section discusses prior work to identify ineffective program comprehension strategies that developers use in large code bases. Our work will replicate some of these studies on a student population, draw connections between comprehension strategies and the Block Model, and explore reasons that students' rely on such techniques. Eventually, we will use these prior works, combined with our own findings, to develop teaching approaches to reduce students' reliance on such ineffective strategies.

Ineffective Behavior 1: Thrashing

Sharafi et al. conducted an eye-tracking study of 36 students completing realistic tasks related to maintenance and debugging in a large code base [53]. The authors discovered that a significant predictor of a student unsuccessfully completing the task is *thrashing*—the process of excessively switching between code elements [53]. Unsuccessful students displayed this thrashing behavior 35% more than successful students, leading to the authors recommending educational interventions to explicitly improve students' code navigation strategies [53]. This work largely confirmed the findings of Robillard et al., who examined the code investigation strategies of software developers and found that ineffective developers resorted to techniques such as "code-skimming" and scrolling in an effort to stumble upon the relevant piece of code [45].

Ineffective Behavior 2: Poor IDE Usage

Ineffective IDE (Integrated Development Environment) usage can cause developers to waste a significant amount of their time. Minelli et al. used an instrumented IDE to analyze professional developers' interactions with an IDE [37]. Though the study included several serious threats to validity, including half the data coming from a single developer, the authors found that nearly 17% of developers' time was spent fiddling with IDE features [37]. Similarly, an industry-focused study by Beller et al. revealed that software professionals demonstrated low usage of the IDE-based debugger—only one-third of developers used the debugger, amounting to just 13% of their actual development time (lower than previous estimates of nearly 50% use) [6]. The authors found that knowledge of how to use the debugger was surprisingly shallow, with developers noting that they had never received formal education on how to use the debugger or had learned from a senior developer on the job [6]. The authors include a specific recommendation for CS curricula in universities to teach students how to use the debugger.

2.3 The Value of Program Comprehension

We have chosen to focus on program comprehension in a large code base due to the impact of comprehension on various factors, which we will describe in this section.

Code Writing

Early work by Von Mayrhauser and Vans on the impact of program comprehension on software maintenance showed that a strong understanding of the source code is a prerequisite to make correct, high-quality code modifications [62]. Similarly, Hoadley et al. showed that developers with stronger comprehension of the code were more likely to reuse methods when applicable [21]. These works, along with others that study how novice programmers read and write small programs [32, 34], consistently show a link between comprehension and ability to correctly write code. Therefore, we posit that improving students' program comprehension skills should have a downstream effect on students' ability to correctly modify large, existing code bases.

Explanations

Explanations of code, whether written explanations in documentation or verbal explanations to a colleague or classmate, are a by-product of program comprehension [16, 55]. In fact, one avenue of program comprehension research seeks to evaluate developers' summarizations of code as a means to assess developer's program comprehension skill [16]. Given prior work on the academia-industry gap identifying communication as a consistent weakness of new developers [42], we argue that improved program comprehension skills may help new developers with technical communication and documentation generation.

Productivity

Studies have placed the amount of time spent *navigating* a large system to be 35% of a developers' time [29] and the time spent *understanding* the relevant source code to be between 58% and 70% [37, 65]. Xia et al. found that more-experienced developers spend *less* time on program comprehension activities than less-experienced developers, indicating the efficiency with which experienced developers go about understanding a code base [65]. Given the time-consuming nature of the program comprehension process, we contend that improving students' comprehension strategies and skills can enable more productive workflows and reduce developer frustration.

2.4 Our Preliminary Work to Impart Strategies for Working with Large Code Bases

The PI of this proposal has already made progress towards understanding and imparting students' program comprehension strategies in large code bases [52]. In the Spring 2023 quarter, the PI designed and taught an initial version of a new course called "Working with Large Code Bases" that aims to teach students program comprehension and maintenance strategies for large code bases. An experience report titled "Working with Large Code Bases: A Cognitive Apprenticeship Approach to Teaching Software Engineering" about this course will appear in the SIGCSE 2024 Technical Symposium [52]. The experience report describes the theoretical motivation, lecture activities, and programming projects in the course. The course covers code comprehension techniques for large code bases—code navigation, using the IDE-based debugger, diagramming, reading unit tests, and more—and project management techniques—Git workflow, documentation tips, and task management. Two of the main contributions of the experience report about the course include 1) introducing teaching approaches to explicitly teach comprehension strategies in a large code base and 2) a course design that could be managed by one instructor and two graduate teaching assistants (TAs) for a course of 50 students (previous work described much lower ratios of roughly 1 TA for every 6 students [56]). Importantly, student feedback during our

course shows a growing level of students' self-efficacy in working with large code bases, which may have downstream impacts on students' persistence and retention in computing [52].

The PI has also received a $50,000 grant from our university to support 1) the creation of an free, online, interactive textbook to accompany the course and 2) the redesign of the course material for the 2023-2024 academic year. We eventually aim for the textbook to serve as a stand-alone learning resource for anyone who wishes to improve their ability to comprehend a large code base. The course and textbook, which have been planned and developed over a full academic year, offers a strong starting point to study how students' approach the program comprehension process.

3 Our Proposal

The primary goal of our proposal is to develop and evaluate teaching methods to address students' struggles to comprehend large code bases. We aim to accomplish this goal in three phases

- **Phase 1:** Identifying the struggles students encounter when comprehending a large code base.
- **Phase 2:** Designing and scaling a theory-based and research-based curriculum to address the struggles we find.
- **Phase 3:** Assessing the impact of the designed curriculum on students' ability to accurately comprehend, explain, and modify a part of a large code base.

3.1 Phase 1: Identifying Student Struggles while Comprehending a Large Code Base

Motivation

Since the Block Model applies to students' comprehension strategies for a specific task *and* the sequencing of the topics in a course, a key motivation of Phase 1 is to 1) identify the struggles students encounter while comprehending a large code base and 2) connect those struggles to specific blocks in the Block Model. By doing so, we hope to motivate specific teaching methods to address the areas in which students struggle. As mentioned, prior work has revealed the struggles that new developers experience in their first software engineering job. These works, such as the studies by Craig et al., Radermacher et al., and Begel and Simon involve semi-structured interviews and anecdotes from professional developers [5, 10, 43]. However, our motivation for Phase 1 is to gain a deeper understanding of comprehension-related struggles that students face using *observations* of students' programming processes and *statements* from stimulated recall interviews. Using these in-depth, qualitative methods, we will identify the strategies that students use and the struggles students face during the program comprehension process.

Research Questions

- What struggles in the program comprehension process do students experience when understanding and modifying part of a large code base?
- In terms of the dimensions and hierarchy of the Block Model, how do students approach the task of understanding and modifying part of a large code base?

Methods

In Phase 1, we will use qualitative analyses to understand the variety of struggles students experience while comprehending large code bases. First, we will observe students completing

a "Skill Demonstration" in which they work to understand and modify a part of a large code base. A "Skill Demonstration" is a type of assessment used in our CS department in which students complete a task on their own and submit artifacts at various checkpoints to demonstrate completion of subtasks. For example, one Skill Demonstration may ask students to set up a development environment for a specific code base, which could include subtasks such as cloning a Github repository, installing dependencies, configuring the build settings, and building the code base. At each of these subtasks, students submit a required file, filepath, screenshot, etc. to show that they have correctly completed the subtask. We have already created a "pilot" Skill Demonstration in which students must find and modify a part of the IDLE code base based on our initial offering of the "Working with Large Code Bases" course. In this Skill Demonstration, students must locate and modify the `Go-to-Line` feature (an existing feature in IDLE that allows users to type in a line number to move the cursor to) in the IDLE code base according to requirements that we have specified. The subtasks in this Skill Demonstration include 1) locating the code for the feature within the code base, 2) understanding the relevant code in the feature to decide on the necessary changes, 3) modifying the code to satisfy requirements, 4) verifying the changes by running the program, and 5) writing unit tests for the changes. When assigning this task to students in the preliminary version of the course, the most time-consuming and difficult step was locating the code for the feature, which will allow us to observe the variety of comprehension strategies that students may use.

We aim to recruit 30 students to complete a two-hour Skill Demonstration. We will motivate students by informing them that they will earn money for each subtask they correctly complete, though we will give the complete amount to all students at the end of the Skill Demonstration. A similar approach to condition the monetary award based on correct completion, but to ultimately reward students with the full amount, was also used by Ko et al. in their study of code navigation strategies [29]. The activity will be framed as an opportunity for students to 1) gain experience working on a large code base that may help with their resume and project experience and 2) earn money for completing a task for research purposes. Students will be recruited via classroom announcements and emails sent to fourth-year undergraduates. We will complete these Skill Demonstrations over several months to allow for multiple rounds of recruitment and flexibility for scheduling with students. We have already conducted a pilot Skill Demonstration with 5 students without offering the monetary compensation, so we are optimistic that we can recruit enough undergraduates for our study.

We will analyze students' performance on the Skill Demonstration using the Block Model to guide our analysis. Prior to the Skill Demonstration, we will ask students to complete a survey about internship experience, past experiences with large code bases, and comfort with the Python language to understand their initial *knowledge base* (which Schulte notes is under-examined in program comprehension research [46]). During the Skill Demonstration, we will record students' screens as they complete the activity to manually detect their code navigation and comprehension strategies, similar to the methods of prior works [29]. The Skill Demonstration will include comprehension questions to assess their understanding of the program text, structure, and function per the Block Model [46]. For example, one question may ask students to identify the regions of interest (Program Text: Blocks), one may ask students to list out the sequence of method calls (Program Execution: Relations), and one may ask students to explain the function of a specific statement (Program Function: Atoms). Together, our observations and students' responses to the Skill Demonstration questions will help us understand students' code comprehension strategies and, importantly, the struggles students face in terms of the Block Model.

Following the Skill Demonstrations, we would like to gather more in-depth data on students' comprehension strategies in large code bases to make stronger connections to the Block Model.

Therefore, we will conduct video-stimulated recall interviews [40] in which we show students recordings of their work and ask students to explain their thought process and reasoning during that part of the Skill Demonstration. Similar to past theory-driven works related to program comprehension in *small* programs [7, 33, 41], we aim to use our observations of student processes and quotes from students to improve our understanding of *what strategies students use* and *why they attempt those strategies* when comprehending *large* code bases.

3.2 Phase 2: Designing and Scaling a Theory-Based Curriculum to Address Student Struggles

Motivation

Much of the prior work we have introduced in this proposal has only *identified* struggles of new developers. However, given the PI's background and prior work in evaluating teaching approaches for computing education, we aim to design a course to *address* the struggles we identify in Phase 1. We will leverage Cognitive Apprenticeship theory, which offers an instructional model for facilitating the transfer of expert strategies from instructor to learner. Our literature review about Cognitive Apprenticeship teaching methods in computing education revealed key benefits of such methods, such as improved student persistence in computing, improved student ability to complete programming tasks, and improved ability for instructors to manage a large class size [48]. We aim to design a course that uses Cognitive Apprenticeship teaching methods, such as modeling, scaffolding, articulation, and exploration, to realize these benefits.

Another key motivation is to accommodate at least 300 students in the course. Previous works that describe project-based, software engineering courses rarely report on the ability to scale up the course. The few works that do discuss logistics related to accommodating more students either have a very low student-TA ratio (Tafliovich et al.: 5-6 students per TA [56]) or leverage industry connections to mentor students ([1, 18]). These options are not necessarily available to an instructor when budget constraints limit the number of TAs per course or when industry connections are not present. Therefore, the goal of this phase is not only to design a theory-driven and evidence-based course curriculum but also to accommodate more students in the course.

Research Questions

- Based on students' perspectives, which mappings exist between various code comprehension techniques (i.e., the IDE-based debugger, code navigation, etc) and "blocks" in the Block Model (i.e., Relations in Program Execution, Atoms of Program Text, etc)?

Methods

The main considerations for Phase 2 are the content of course topics to teach (i.e., what will be taught) and the techniques to manage a large class size (i.e., how to accommodate more students).

To decide the content of course topics, we will identify mappings between blocks in the Block Model and comprehension techniques that we teach. We will use Version 1 of our course (to be taught in Spring 2025) to ask students to reflect on the usefulness of code comprehension techniques they have used, which will help us create these mappings. Our preliminary version of the course already elucidated several such mappings. For example, students found the IDE-based debugger helpful to follow the order of method calls, which maps to the Relations and Blocks aspects of the Program Execution dimension in the Block Model [52]. Similarly, code navigation shortcuts (such as using `Find All References` or `Go to Definition`) map to the Relations aspect

of Program Text since these commands can help understand references between blocks. Based on these mappings, we will design teaching approaches to impart these code comprehension techniques. Notably, we may identify struggles that are not directly mapped to the Block Model due to our context being large code bases (rather than introductory-level programs). In these cases, we will aim to extend the Block Model to explain the comprehension process in a large code base and will still design the course to capture these struggles.

Using these mappings, we will design Cognitive Apprenticeship-based lecture activities to impart code comprehension strategies to students. For example, we will likely begin with a **modeling** step in which an instructor demonstrates the strategy to students, a **scaffolding and coaching** step in which students use the strategy under instructor supervision during an in-lecture activity, and an **articulation** step in which students describe how they used the strategy and its usefulness for a task. Finally, we aim to include an **exploration** phase for students to use the strategy in an open-ended task after the lecture. We aim for these activities to promote active learning and to incorporate authentic tasks for students to complete based on our findings in our Cognitive Apprenticeship literature review [48].

In this phase, we will also explore and evaluate several techniques that aim to accommodate a larger class size each iteration of the course. One avenue of exploration will be to focus on students' independent help-seeking techniques to limit students' help requests to instructors. Our first iteration of the course allowed students to use LLM tools such as ChatGPT and Github Copilot. Student feedback showed that students' felt these tools were extremely useful for overcoming obstacles during projects. We aim to explore student perspectives and the pedagogical impacts, in terms of student learning and managing enrollment growth, of teaching students effective techniques for interacting with LLMs for working with large code bases. Another potential technique we may explore are peer-code reviews in which students provide an initial code review to each other to reduce the time course staff need to spend on code reviews. The goal of exploring these techniques is to help instructors of project-based, software engineering courses implement such techniques to accommodate more students in their courses.

3.3 Phase 3: Assessing the Impact of our Curriculum on Student Skills

Motivation

While we will develop and administer a course to impart effective comprehension strategies, the course will revolve around a single code base. Therefore, our intervention begs the questions of whether students who took our course can transfer the skills we taught to a new code base. As a result, we need to compare students that have completed our course to students who will graduate without completing our course to understand the effectiveness of our course in preparing students for working on a large, existing code base. This phase is crucial for the generalizability of our results. Since we aim to disseminate our course materials, teaching methods, and course policies to the broader computing education community, we should be able to identify learning outcomes of the course related to students' ability to comprehend, modify, and explain part of a large code base.

Research Questions

- How does usage of program comprehension strategies differ between students who took our course but did not complete an internship, students who did not take our course but completed an internship, and those who neither took our course nor completed an

internship?
- How does performance on code modification tasks, Explain-in-Plain-English (EiPE) questions, and comprehension questions related to a large code base differ between those groups?

Methods

Phase 3 will evaluate the impact of our curriculum on students' ability to comprehend, explain, and modify parts of a large code base. Similar to Phase 1, we will conduct another screen-recorded Skill Demonstration. In this phase, however, the Skill Demonstration will occur with all students in our course (as a final assessment) and with fourth-year undergraduates who did not complete our course. We will likely have at least 100 students who complete the course. As a result, we hope to recruit at least 50 participants that did not complete our course. If our sample size is large enough, we hope to have one group of students who did not take our course nor complete an internship and another group who also did not take our course but has internship experience. The Skill Demonstration will be conducted on a different code base than the IDLE code base so that students from our course are not at an advantage over other students and students will be instructed to find and modify an existing part of the code base. The code base should be written in Java to further evaluate the generalizability of our course on students' program comprehension strategies. During the Skill Demonstration, we will include comprehension and EiPE questions that cover various blocks of the Block Model. Just like the Phase 1 Skill Demonstration, we will ask students a series of questions to understand each students' knowledge base (such as students' experience with large code bases, with Java, and in internships).

The analysis of the Phase 3 Skill Demonstration will be very similar to the analysis of the Phase 1 Skill Demonstration. For this phase, however, we will have one group of students that includes students who completed our course and one group of students who did not. We will use two-sample t-tests (or ANOVA tests if we have more than 2 groups) to compare the performance of the groups. We will manually analyze screen recordings from the Skill Demo to report on the variety and frequency of code comprehension strategies used by both groups. Further, we will evaluate the strength of code explanations using a rubric that we will develop for the EiPE questions. Finally, we will count the number of subtasks completed and the rate at which those were completed. The results from this phase will help us determine the impact of our Cognitive Apprenticeship approach on students' program comprehension strategies and comprehension-adjacent skills such as code explanations and correctness of code modifications.

4 Evaluation

We will update our external evaluator (EE)—Chris Hundhausen—after each phase. Dr. Hundhausen has extensive experience in software engineering education and has recently received an NSF IUSE grant (#1915198) related to teaching skills such as teamwork and reflection while working on "brownfield assignments," which are large, existing code bases. We believe that our proposed work to improve the technical skill of code comprehension in large code bases complements Dr. Hundhausen's existing proposal to promote the soft skills of working in a large code base. We will meet with Dr. Hundhausen to discuss ideas for conducting our studies and for understanding the results we obtain.

Evaluating the Skill Demonstration

We will take care to evaluate the two Skill Demonstrations (one in Phase 1 and one in Phase 3) we create. For each of these Skill Demonstrations, we will conduct an initial "pilot" test to determine if the instructions are clear and that the questions measure the skills that we intend to assess. We will consult with our EE on the design of the Skill Demonstration and may reach out to industry contacts (who have given guest lectures for our course) for feedback on the subtasks to include in the Skill Demonstrations. These industry contacts have years of experience as software developers and already understand the goals of our course given our prior collaboration.

Evaluating our Comparative Analyses

Both PI Soosai Raj and co-PI Griswold have experience conducting controlled experiments, including multiple course-long experiments to evaluate the impact of live coding on students' comprehension skills, programming processes, and lecture engagement [44, 49, 51]. Co-PI Griswold also has conducted qualitative studies on instructor perspectives of the academia-industry gap [59, 60]. For our current work, we will rely on similar methods that PI Soosai Raj and co-PI Griswold have used previously, such as controlling for various factors that impact students' ability to comprehend a large code base. For example, we will ensure that we collect data about students' internship experience, prior programming experience before college, experience with certain programming languages, GPA, and other characteristics we deem relevant. We will control for these factors the statistical tests we apply.

Theoretical Evaluation

Because the Block Model and Cognitive Apprenticeship learning theory play a major role in motivating each phase and the methods we employ, we will explain our results in terms of these theories. In this sense, the learning and teaching theories will help validate our findings. For example, in our prior work to evaluate live—a modeling method of Cognitive Apprenticeship—we did not observe an improvement to students' programming processes (measure in terms of adherence to incremental development and debugging techniques). However, we were able to explain our results using Cognitive Apprenticeship itself, since our intervention involved *only* a modeling method (live coding) without appropriate scaffolding, coaching, or reflection methods for students to practice these programming processes. We will make similar connections to theory in our proposed work, which can provide a theoretical basis for our findings.

5 Disseminating Results

Published Work

We will disseminate our findings via research papers and experience reports at premier computing education research venues such as the SIGCSE TS, ITiCSE, and ICER. The list of proposed papers below represents the "core" studies and reports we aim to create:

- Student Struggles in Program Comprehension on Large Code Bases (Research Paper)
- A Skill Demonstration to Assess Students' Ability to Understand and Modify a Large Code Base in Python (Experience Report)
- Towards Mapping Code Comprehension Techniques to the Block Model (Research Paper)
- A Revised Curriculum to Teach Working with Large Code Bases at Scale (Experience Report)

- An Empirical Evaluation of a Cognitive Apprenticeship-Based Course for Comprehending Large Code Bases (Research Paper)

However, we note the potential for many other findings given the large amount of data we plan to collect. For example, since we will be introducing students to LLM tools such as Cursor [13] (an AI-first IDE that integrates features such as Github Copilot and ChatGPT), we plan to collect data related to students' experiences and perceptions of using these tools for a large code base. Our Skill Demonstrations might shed light on the value and frequency of internships, which could also be a useful finding for our research community.

Pedagogical Artifacts

We will also produce several pedagogical artifacts to assist instructors who wish to teach a similar course at their own institution:

- A course website designed with recordings of lectures, assignment descriptions, a recommended syllabus, and other relevant course material.
- A free, online, interactive textbook on Stepik so that anyone can access the course material.
- A Skill Demonstration with instructions for implementation for instructors to assess their students' ability to comprehend a large code base.

6 Sustainability

This project will be sustained by graduate and undergraduate students at UC San Diego. A significant portion of the work will be completed by Anshul Shah, a third-year PhD student focusing on using Cognitive Apprenticeship techniques to improve students' programming processes (including code writing and code comprehension). Anshul was the lead student on the work to evaluate live coding. We also have a sizable number of undergraduate and master's students that are interested in this work. For example, two master's students (Jerry You and Thanh Tong) were interested in our first offering of the course and played a large role in its implementation. Jerry and Thanh are co-authors on the experience report of our initial course offering. In fact, several former students of our course have already expressed interest in being future teaching assistants for the course due to their positive experiences in the class.

We also plan to sustain this project by sharing our findings at top CS Education conference venues through research papers, experience reports, and workshops (as mentioned in Section 5). One explicit goal from our dissemination is to help instructors at other institutions offer this course using the materials we have produced. Because of co-PI Griswold's prior work in which CS faculty around the world were surveyed about their perspectives on the academia-industry gap, we are aware of the obstacles these faculty have reported that make industry preparation difficult [60]. These obstacles include lack of time to create materials and large class sizes, among others. Fortunately, our goals are to 1) create ready-to-use course materials for instructors and 2) accommodate hundreds of students in our course. Therefore, we are optimistic that our work can improve the sustainability of this course *at other institutions*.

7 Intellectual Merit

Our research community has repeatedly called for interventions to impart strategies of working with large code bases, from Tony Clear's opinion piece in 2005 [8] to recent studies that investigated

the academia-industry gap [10, 43]. Given the lack of work to *address* the academia-industry gap, as opposed to understanding perspectives and experiences related to the gap, our work will make important progress to narrow the gap pedagogically. Program comprehension is a significant part of a developer's workflow that impacts productivity, ability to debug, and communication. These far-ranging impacts warrant the significant research undertaking that we have proposed in order to prepare our students for industry work. Further, by relating our results in each phase to the Block Model, we aim to create a theoretical basis for our recommendations for teaching code comprehension in large code bases. The Block Model has received much attention since its introduction, though the work to evaluate the model has focused on small programs. Our work will build upon this existing body of work by extending these prior analyses to the context of large code bases. Through this line of work, we may encourage future researchers and educators to emphasize the important skill of program comprehension in large code bases.

8 Broader Impacts

The broader impacts of our work are two-fold: first, there are direct benefits to individuals from currently underrepresented groups and second, there are benefits to our future workforce. First, computing suffers from low retention rates among underrepresented groups, including women and BLNPI+ students. Fortunately, prior work, including a literature review on Cognitive Apprenticeship methods from the PI [48], has shown the positive impact of authentic and active learning environments on students' persistence in computing [27, 30]. Our preliminary experience teaching the course already showed students reporting an improved confidence throughout the term. Further, differences among incoming university students in terms of prior programming experience, which is linked to gender [35, 63] and socioeconomic status [36], likely impacts students' ability to obtain summer internships and thus, employment. Our research contributions *and* pedagogical artifacts will help level this playing field by providing an authentic software engineering experience for students who haven't obtained internship experience. Second, there is a clear impact on workforce readiness and industry productivity from our work. A large motivation for this proposal comes from prior works that describe the frustrations and confusions of new graduates in software engineering [5, 10, 43]. Since our work will aim to impart *relevant* and *high-impact* skills for developers, such as code comprehension and code management techniques, we expect our graduates to perform more productively in the workforce. Ultimately, we aim for our work to contribute to a more diverse and effective software engineering workforce.

9 Key Personnel

Our project team includes PI Adalbert Gerald Soosai Raj (UC San Diego), co-PI Bill Griswold (UC San Diego), and EE Chris Hundhausen (Oregon State University).

PI Soosai Raj has experience conducting controlled experiments to evaluate the impact of live coding (see Section 10 for more details), has 3 years of relevant software engineering experience in industry, and has designed and administered the initial offering of the Working with Large Code Bases course. His work with live coding included three quarter-long experiments to compare live-coding examples to alternative pedagogical techniques to understand the impact of modeling on students' programming processes. Part of the live coding work included developing a metric for incremental development—an effective software development technique to improve developer's productivity and mitigate errors. PI Soosai Raj's industry experience has helped him identify the

shortcomings in our current ability to teach students how to work with large code bases—a key motivation for him to conduct this research.

Co-PI Bill Griswold has extensive experience in software engineering research *and* computing education research—a combination of experiences that is crucial for our proposed work. Co-PI Griswold was a pioneer in code refactoring for software maintenance [17]. His experience in software engineering research—specifically in software maintenance and evolution—will be vital for the experimental design and analysis of our results related to program comprehension. Co-PI Griswold has also been involved in software engineering education research, including work to evaluate live coding, to identify at-risk students, and to understand perspectives of the academia-industry gap [60].

EE Chris Hundhausen also conducts research into software engineering education, including NSF-funded work to explore brownfield programming assignments (NSF DUE Award #1915196). He has authored the influential "IDE-based Learning Analytics for Computing Education," which even mentions using Cognitive Apprenticeship methods such as scaffolding and articulation within IDEs to support student learning.

10 Results of Prior NSF Support

PI Soosai Raj and co-PI Griswold worked on the project "Determining the Effectiveness of Live Coding on Student Learning in Introductory Programming" (IUSE #2044473). The goal of this project was to determine the impact of live coding (a modeling method of Cognitive Apprenticeship) on students' programming processes (such as incremental development, debugging, and testing), code comprehension skills, and lecture engagement in introductory programming courses. The work succeeded in 1) developing a metric called the Measure of Incremental Development to quantify a students' adherence to incremental development [50], 2) comparing a remote, live-coding pedagogy to a remote, static-code pedagogy [49], 3) comparing an in-person, live-coding pedagogy to an in-person, static-code pedagogy [51], and 4) comparing a live-coding pedagogy to an active live-coding pedagogy that engaged the scaffolding and articulation methods of Cognitive Apprenticeship (submission forthcoming). This prior work has been vital to motivating our current proposal. Not only have we gained a deeper understanding of applying Cognitive Apprenticeship methods in courses, but we have also realized the difficulty in expecting introductory programming students to adhere to effective incremental development and debugging practices. In a larger code base, with more time-consuming tasks, these programming processes become more important to reduce developer frustration and improve productivity. This realization was a motivating factor in the inception of our current proposed work.

Another key project that co-PI Griswold worked on was "Identifying and Aiding At-Risk Students in Computing" (DUE #1712508). Co-PI Griswold conducted work to understand faculty views on the academia-industry gap [59, 60]. Importantly, these works showed a dominant perspective from 57% of computing faculty around the world that a principal goal of university CS education is to prepare students for industry [60], which further motivated our current proposed work. We aim to develop a curriculum to prepare students for industry while also addressing the barriers that instructors have identified, such as large class sizes and the time required to develop course materials [60].

11 Timeline of Work, Outcomes, and Dissemination

Fall 2024 - Winter 2025	• Conduct and analyze Skill Demos and interviews to identify strategies and struggles while understanding large code bases.
Spring 2025	• Teach Version 1 of the course to 100 students. • Continue analyzing data from Skill Demos and interviews. • **Experience Report about *A Skill Demonstration to Assess Students' Ability to Comprehend a Large Code Base***
Summer 2025 - Winter 2026	• Analyze student feedback and artifacts from course • Update course content, textbook, and policies • **Research Paper about *Student Struggles while Modifying a Large Code Base***
Spring 2026	• Teach Version 2 of the course to 150-200 students • Conduct final exam/Skill Demonstration for Version 2 of our course to assess ability to comprehend a large code base • Begin conducting the same Skill Demo for students who did not take Version 1 or Version 2 of our course. • **Research Paper about *Mapping Code Comprehension Techniques to the Block Model***
Summer 2026 - Winter 2027	• Continue Skill Demos for students who did not take V1 or V2 of our course if needed • Analyze data from the Skill Demos • **Experience Report about *Scaling a Software Engineering Course about Working with Large Code Bases***
Spring 2027	• Teach Version 3 (the final version) of our course to 300 students • **Research Paper about *Evaluating a Cognitive Apprenticeship-Based course to Impart Strategies for Comprehending Large Code Bases***

References

[1] Zakarya Alzamil. Towards an effective software engineering course project. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 631–632, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139632. doi: 10.1145/1062455.1062575. URL https://doi.org/10.1145/1062455.1062575.

[2] Ray Bareiss and Martin Radley. Coaching via cognitive apprenticeship. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 162–166, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300063. doi: 10.1145/1734263.1734319. URL https://doi.org/10.1145/1734263.1734319.

[3] Roman Bednarik, Carsten Schulte, Lea Budde, Birte Heinemann, and Hana Vrzakova. Eye-movement modeling examples in source code comprehension: A classroom study. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365352. doi: 10.1145/3279720.3279722. URL https://doi.org/10.1145/3279720.3279722.

[4] Andrew Begel and Beth Simon. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, page 226–230, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937995. doi: 10.1145/1352135.1352218. URL https://doi.org/10.1145/1352135.1352218.

[5] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 3–14, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404522. URL https://doi.org/10.1145/1404520.1404522.

[6] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 572–583, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180175. URL https://doi.org/10.1145/3180155.3180175.

[7] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. ISSN 0020-7373. doi: https://doi.org/10.1016/S0020-7373(83)80031-5. URL https://www.sciencedirect.com/science/article/pii/S0020737383800315.

[8] Tony Clear. Comprehending large code bases - the skills required for working in a "brown fields" environment. *SIGCSE Bulletin*, 37:12–14, 06 2005. doi: 10.1145/1083431.1083439.

[9] Allan Collins, John Seely Brown, Ann Holum, et al. Cognitive apprenticeship: Making thinking visible. *American educator*, 15(3):6–11, 1991.

[10] Michelle Craig, Phill Conrad, Dylan Lynch, Natasha Lee, and Laura Anthony. Listening to early career software developers. *J. Comput. Sci. Coll.*, 33(4):138–149, apr 2018. ISSN 1937-4771.

[11] Timothy Davis, Robert Geist, Sarah Matzko, and James Westall. Texvn: Trial phase for the new curriculum. *SIGCSE Bull.*, 39(1):415–419, mar 2007. ISSN 0097-8418. doi: 10.1145/1227504.1227455. URL https://doi.org/10.1145/1227504.1227455.

[12] Vanessa Dennen and Kerry Burner. The cognitive apprenticeship model in educational practice. *Handbook of Research on Educational Communications and Technology*, 01 2008.

[13] Cursor Developers. Cursor, 2024. URL https://cursor.sh/.

[14] Matthias Ehlenz, Thiemo Leonhardt, and et al. The lone wolf dies, the pack survives? analyzing a computer science learning application on a multitouch-tabletop. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365352. doi: 10.1145/3279720.3279724. URL https://doi.org/10.1145/3279720.3279724.

[15] Maria Feldgen and Osvaldo Clua. Promoting design skills in distributed systems. In *2012 Frontiers in Education Conference Proceedings*, pages 1–6, 2012. doi: 10.1109/FIE.2012.6462229.

[16] Judith Good and Paul Brna. Program comprehension and authentic measurement:: a scheme for analysing descriptions of programs. *International Journal of Human-Computer Studies*, 61(2): 169–185, 2004. ISSN 1071-5819. doi: https://doi.org/10.1016/j.ijhcs.2003.12.010. URL https://www.sciencedirect.com/science/article/pii/S1071581904000023. Empirical Studies of Software Engineering.

[17] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, USA, 1992. UMI Order No. GAX92-03258.

[18] Steven M. Hadfield and Nathan A. Jensen. Crafting a software engineering capston project course. *J. Comput. Sci. Coll.*, 23(1):190–197, oct 2007. ISSN 1937-4771.

[19] Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. Evaluating beacons, the role of variables, tracing, and abstract tracing for teaching novices to understand program intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 329–343, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600140. URL https://doi.org/10.1145/3568813.3600140.

[20] Regina Hebig, Truong Ho-Quang, Rodi Jolak, Jan Schröder, Humberto Linero, Magnus Ågren, and Salome Honest Maro. How do students experience and judge software comprehension techniques? In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 425–435, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379588. doi: 10.1145/3387904.3389283. URL https://doi.org/10.1145/3387904.3389283.

[21] Christopher Hoadley, Marcia Linn, Lydia Mann, and Mike Clancy. When and why do novice programmers reuse code? In *Empirical Studies of Programmers - Sixth Workshop*, pages 109–130, 01 1996.

[22] Cruz Izu and Claudio Mirolo. Comparing small programs for equivalence: A code comprehension task for novice programmers. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 466–472, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368742. doi: 10.1145/3341525.3387425. URL https://doi.org/10.1145/3341525.3387425.

[23] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450375672. doi: 10.1145/3344429.3372501. URL https://doi.org/10.1145/3344429.3372501.

[24] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450375672. doi: 10.1145/3344429.3372501. URL https://doi.org/10.1145/3344429.3372501.

[25] Wei Jin and Albert Corbett. Effectiveness of cognitive apprenticeship learning (cal) and cognitive tutors (ct) for problem solving using fundamental programming concepts. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, page 305–310, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305006. doi: 10.1145/1953163.1953254. URL https://doi.org/10.1145/1953163.1953254.

[26] Maria Kallia. The search for meaning: Inferential strategic reading comprehension in programming. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 1–14, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600135. URL https://doi.org/10.1145/3568813.3600135.

[27] Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen. Carry-on effect in extreme apprenticeship. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 1150–1155, 2013. doi: 10.1109/FIE.2013.6685011.

[28] Maria Knobelsdorf, Christoph Kreitz, and Sebastian Böhne. Teaching theoretical computer science using a cognitive apprenticeship approach. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, page 67–72, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326056. doi: 10.1145/2538862.2538944. URL https://doi.org/10.1145/2538862.2538944.

[29] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 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, 2006. doi: 10.1109/TSE.2006.116.

[30] D. Brian Larkins, J. Christopher Moore, et al. Application of the cognitive apprenticeship framework to a middle school robotics camp. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 89–94, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445226. URL https://doi.org/10.1145/2445196.2445226.

[31] Colleen M. Lewis. Examples of unsuccessful use of code comprehension strategies: A resource for developing code comprehension pedagogy. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 15–28, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600116. URL https://doi.org/10.1145/3568813.3600116.

[32] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *SIGCSE Bull.*, 41(3): 161–165, jul 2009. ISSN 0097-8418. doi: 10.1145/1595496.1562930. URL https://doi.org/10.1145/1595496.1562930.

[33] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, page 80–98, USA, 1986. Ablex Publishing Corp. ISBN 089391388X.

[34] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 101–112, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404531. URL https://doi.org/10.1145/1404520.1404531.

[35] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: Women in computing*. MIT press, 2002.

[36] Jane Margolis, Rachel Estrella, Joanna Goode, Jennifer Jellison Holme, and Kim Nao. *Stuck in the Shallow End: Education, Race, and Computing*. The MIT Press, 2017. ISBN 0262533464.

[37] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015. doi: 10.1109/ICPC.2015.12.

[38] S. Murray and et al. A tool-mediated cognitive apprenticeship approach for a computer engineering course. In *Proceedings 3rd IEEE International Conference on Advanced Technologies*, pages 2–6, 2003. doi: 10.1109/ICALT.2003.1215014.

[39] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 2–11, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349680. doi: 10.1145/3105726.3106178. URL https://doi.org/10.1145/3105726.3106178.

[40] Nga Thanh Nguyen, Amanda McFadden, Donna Tangen, and Denise Beutel. Video-stimulated recall interviews in qualitative research. 2013. URL https://api.semanticscholar.org/CorpusID:141084582.

[41] Nancy Pennington. *Comprehension Strategies in Programming*, page 100–113. Ablex Publishing Corp., USA, 1987. ISBN 0893914614.

[42] Alex Radermacher and Gursimran Walia. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 525–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445351. URL https://doi.org/10.1145/2445196.2445351.

[43] Alex Radermacher, Gursimran Walia, and Dean Knudson. Investigating the skill gap between graduating students and industry expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 291–300, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327688. doi: 10.1145/2591062.2591159. URL https://doi.org/10.1145/2591062.2591159.

[44] Adalbert Gerald Soosai Raj, Pan Gu, Eda Zhang, Arokia Xavier Annie R, Jim Williams, Richard Halverson, and Jignesh M. Patel. Live-coding vs static code examples: Which is better with respect to student learning and cognitive load? In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, ACE'20, page 152–159, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376860. doi: 10.1145/3373165.3373182. URL https://doi.org/10.1145/3373165.3373182.

[45] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004. doi: 10.1109/TSE.2004.101.

[46] Carsten Schulte. Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 149–160, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404535. URL https://doi.org/10.1145/1404520.1404535.

[47] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE Working Group Reports*, ITiCSE-WGR '10, page 65–86, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450306775. doi: 10.1145/1971681.1971687. URL https://doi.org/10.1145/1971681.1971687.

[48] Anshul Shah and Adalbert Gerald Soosai Raj. A review of cognitive apprenticeship methods in computing education research. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education*, SIGCSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704239. doi: 10.1145/3626252.3630769. URL https://doi.org/10.1145/3626252.3630769.

[49] Anshul Shah, Vardhan Agarwal, Michael Granado, John Driscoll, Emma Hogan, Leo Porter, William Griswold, and Adalbert Gerald Soosai Raj. The impact of a remote live-coding pedagogy on student programming processes, grades, and lecture questions asked. In *Proceedings of the 28th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '23, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3587102.3588846. URL https://doi.org/10.1145/3587102.3588846.

[50] Anshul Shah, Michael Granado, Mrinal Sharma, John Driscoll, Leo Porter, William Griswold, and Adalbert Gerald Soosai Raj. Understanding and measuring incremental development in cs1. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3545945.3569880. URL https://doi.org/10.1145/3545945.3569880.

[51] Anshul Shah, Emma Hogan, Vardhan Agarwal, John Driscoll, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. An empirical evaluation of live coding in cs1. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER

'23, page 476–494, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399760. doi: 10.1145/3568813.3600122. URL https://doi.org/10.1145/3568813.3600122.

[52] Anshul Shah, Jerry Yu, Thanh Tong, and Adalbert Gerald Soosai Raj. Working with large code bases: A cognitive apprenticeship approach to teaching software engineering. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education*, SIGCSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704239. doi: 10.1145/3626252.3630755. URL https://doi.org/10.1145/3626252.3630755.

[53] Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer. Eyes on code: A study on developers' code navigation strategies. *IEEE Transactions on Software Engineering*, 48(5): 1692–1704, 2022. doi: 10.1109/TSE.2020.3032064.

[54] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Towards understanding how developers spend their effort during maintenance activities. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 152–161, 2013. doi: 10.1109/WCRE.2013.6671290.

[55] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, sep 1986. ISSN 0001-0782. doi: 10.1145/6592.6594. URL https://doi.org/10.1145/6592.6594.

[56] Anya Tafliovich, Francisco Estrada, and Thomas Caswell. Teaching software engineering with free open source software development: An experience report. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 01 2019. doi: 10.24251/HICSS.2019.931.

[57] Neena Thota, Gerald Estadieu, Antonio Ferrao, and Wong Kai Meng. Engaging school students with tangible devices: Pilot project with .net gadgeteer. In *2015 International Conference on Learning and Teaching in Computing and Engineering*, pages 112–119, 2015. doi: 10.1109/LaTiCE.2015.26.

[58] R.L. Upchurch and J.E. Sims-Knight. Integrating software process in computer science curriculum. In *Proceedings Frontiers in Education 1997 27th Annual Conference. Teaching and Learning in an Era of Change*, volume 2, pages 867–871 vol.2, 1997. doi: 10.1109/FIE.1997.635990.

[59] Sander Valstar, Sophia Krause-Levy, Alexandra Macedo, William G. Griswold, and Leo Porter. Faculty views on the goals of an undergraduate cs education and the academia-industry gap. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 577–583, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366834. URL https://doi.org/10.1145/3328778.3366834.

[60] Sander Valstar, Caroline Sih, Sophia Krause-Levy, Leo Porter, and William G. Griswold. A quantitative study of faculty views on the goals of an undergraduate cs program and preparing students for industry. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 113–123, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370929. doi: 10.1145/3372782.3406277. URL https://doi.org/10.1145/3372782.3406277.

[61] Arto Vihavainen and Matti Luukkainen. Results from a three-year transition to the extreme apprenticeship method. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 336–340, 2013. doi: 10.1109/ICALT.2013.104.

[62] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995. doi: 10.1109/2.402076.

[63] Chris Wilcox and Albert Lionelle. Quantifying the benefits of prior programming experience in an introductory computer science course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 80–85, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159480. URL https://doi.org/10.1145/3159450.3159480.

[64] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, 56(4), nov 2023. ISSN 0360-0300. doi: 10.1145/3626522. URL https://doi.org/10.1145/3626522.

[65] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. doi: 10.1109/TSE.2017.2734091.