An Analysis of Students' Program Comprehension Processes in a Large Code Base

1st Anshul Shah
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

2nd Thanh Tong
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

Elena Tomson
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

4th Steven Shi
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

5th Bill Griswold
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

6th Gerald Soosairaj
Computer Science and Engineering
University of California, San Diego
La Jolla, CA

***Abstract*—Program comprehension (PC) literature typically focuses on industry professionals comprehending large code bases or novice programmers comprehending short programs. As a result, limited work has aimed to understand how intermediate programmers comprehend large code bases, especially with the goal of supporting learner's incremental development of program comprehension expertise. Through the lens of the Block Model—a theory to support research on and teaching of PC—we aim to uncover 1) the comprehension process that intermediate programmers follow in terms of the Block Model (i.e., *top-down, bottom-up, etc.*), and 2) common mappings between comprehension techniques used by intermediate programmers and comprehension blocks in the Block Model. We present a qualitative analysis of students' "process journals" in which they described their PC process while modifying the open-source `idlelib` code base. Our results showed that students typically followed a *top-down* and *Text-first* approach to understand a feature in the `idlelib` code base. Our findings also reveal *how* students used various program comprehension techniques (such as code navigation, using the IDE-based debugger, making experimental code changes, etc.) in terms of the Block Model. These findings make progress toward bridging our theoretical understanding of novices' comprehension process in small programs and expert's code comprehension process in large code bases by presenting a high sample size investigation of *intermediate programmers'* PC processes in a large, existing code base. Finally, instructors can use our findings to understand which blocks in the Block Model are targeted by various PC techniques, which can enable intentional teaching activities to impart PC skills.**

***Index Terms*—Program Comprehension, Block Model**

I. Motivation

The *academia-industry gap* in software engineering refers to the difference between new graduates' abilities upon graduating and the industry expectations placed upon them [1]. Studies have uncovered new graduates' deficiencies in areas related to technical communication, testing, design, and much more [1], [2]. A potential cause for this gap is the stark difference between the type of programming tasks done in industry, which typically includes *brownfield* development (i.e., working with a legacy code base), and those done in university coursework, which typically includes *greenfield* development (i.e., creating a program from scratch) with smaller programs [3]. In fact, numerous studies that have investigated the academia-industry gap have explicitly called on universities to teach students the skills of working with large, existing code bases so that they are prepared for industry [3]–[5]. Unfortunately, a recent study by Craig et al. 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" [3].

While in industry, program comprehension represents a significant portion of a new graduate's time. Studies have placed the amount of time spent navigating a large system to be 35% of a developers' total time [6] and the time spent finding and understanding the relevant source code to be 70% percent of the workflow to modify existing code in a large code base [7]. Similarly, an emphasis on novices' program comprehension ability is crucial from a pedagogical perspective. In fact, in Xie et al.'s theory of instruction for introductory computer science, program comprehension is recommended as the first skill to teach novices [8]. However, CS Education researcher Colleen Lewis argued in 2023 that "while there is a long history of identifying expert code-comprehension strategies, there has been less work to understand and support the incremental development of code comprehension expertise" [9].

As Lewis points out, foundational research in program comprehension has sought to compare, from a theoretical perspective, the program comprehension approaches and abilities of novice and expert programmers [9]–[12]. For example, Lister et al. showed that experts tend to form abstract representations of a program based on the code's purpose whereas novices are primarily concerned with how the code functions [12]. Interestingly, Pennington showed that professional programmers start with an understanding of procedural elements, such as the program's control flow, before developing an understanding of the program's purpose [11]. These and other theories provide an understanding of how programmers comprehend code at various stages in their programming career. However, we argue

that the relative lack of theoretical understanding of *intermediate programmers'* program comprehension abilities contributes to the ineffective program comprehension pedagogy at the university level.

With the broader goal of developing a comprehensive program comprehension pedagogy for upper-division undergraduate students, this study aims to lay the groundwork for understanding this populations' program comprehension processes. We present a qualitative analysis on students' self-reported program comprehension process while they made a modification to a large code base—a task specifically chosen due to its similarity with industry tasks. Our work seeks to extend the empirical work related to a popular program comprehension theory—the Block Model—to the context of large code bases. The goals of this study are to 1) understand the techniques and processes that senior-level undergraduate students use when comprehending a large code base and 2) apply a program comprehension theory called the Block Model to synthesize our findings with the broader theoretical findings related to program comprehension processes by novices and experts. To accomplish these goals, we ask the following two research questions:

1) What patterns (i.e., top-down, bottom-up, text-to-purpose, etc.) in the program comprehension process do upper-division undergraduate students follow when making a feature modification in a large code base?
2) How do students use various program comprehension techniques for large code bases (i.e., code navigation, using a debugger, diagramming) to comprehend parts of a large code base in terms of the Block Model?

II. Theoretical Framework: The Block Model

Theories of program comprehension describe the process of comprehension as a "top-down" process guided by hypotheses about the code [13], a "bottom-up" process in which lines or blocks of code are grouped into abstractions [11], or an "as-needed" process where programmers use various strategies based on the task at hand [14]. Though competing program comprehension theories exist, we will focus on the Block Model presented by Schulte et al. [15]. We choose to use this framework because the Block Model synthesizes elements of well-established program comprehension theories and was specifically designed to support both research *and* teaching of program comprehension [10]. In fact, the Block Model serves as a useful framework to categorize the various approaches to program comprehension that may exist for a specific task.

The Block Model is presented as a table with three *dimensions* and four *hierarchies*. Table I depicts the 12 "blocks" in the Block Model as it was presented by Schulte et al. in his seminal work [15]. The dimensions—Text, Execution, and Purpose—represent the different axes of comprehension [15]. *Text* refers to understanding the syntax and meaning of the code itself, *Execution* refers to the program behavior at runtime, such as data flow and control flow, and *Purpose* refers to the goals and function of the code in some extrinsic context. The hierarchies—Atoms, Blocks, Relations, and Macro Structure—represent the levels of abstraction in the comprehension process [15]. Importantly, the literature around the Block Model has considered these hierarchies to be dependent on the specific code comprehension activity [16]. For example, the size of an *Atom* may be an individual token for smaller snippets of code or may be a line of code (or even several lines of code) for larger tasks [16]. Regardless, *Atoms* are the smallest units such as tokens or lines of code, *Blocks* are "regions of interest" in the code that are larger than Atoms, *Relations* are the relationships and references between Blocks, and *Macro Structure* is the high-level structure, execution, or purpose of the code.

Macro Structure	Overall structure of the program text	Understanding the "algorithm" of the program	Purpose of the program (in its context)
Relations	References between blocks, such as 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, method, or ROI (as sequence of statements)	Function of a block (i.e., subgoal)
Atoms	Language elements	Operation of a statement	Function of a statement
	Text Surface	**Execution**	**Purpose**
Duality	**Structure**		**Function**

TABLE I: The Block Model [15]

Since the Block Model was presented in 2008, many works have used it as a theoretical framework for understanding students' program comprehension and algorithm comprehension processes [9], [16]–[22]. The value of the Block Model lies in its ability to represent different approaches to program comprehension, whether those approaches are top-down (where programmers start with the Macro Structure hierarchy and narrow down to Blocks or Atoms), bottom-up (starting with Atoms or Blocks and proceeding up the hierarchies), or procedural-before-functional understanding (which would start with the Text Surface or Execution dimensions and move towards Purpose). In fact, the early theoretical work on the Block Model aimed to map existing program comprehension theories, such as those by Letovsky, Pennington, von Mayrhauser and Vans, and more [10].

The most relevant empirical work to our present study comes from Jayathirtha et al., who conducted an analysis on high school students' comprehension processes for an Arduino program that impacted a physical artifact [18]. To do this, the authors conducted think-aloud interviews with the students as the students reasoned through the program. The transcripts were then analyzed for all "program-related" utterances and those utterances were then mapped to specific blocks from the Block Model [18]. The results showed that while students engaged with the *Purpose* dimension at various hierarchies, there were hardly any occurrences of students understanding the runtime behavior (*Execution*) or abstract understandings of

the *Text* dimension. Although our context of a large code base that outputs a software artifact is different from Jayathirtha et al.'s study, we were inspired by the analytic process of connecting students' processes to paths on the Block Model.

Jayathirtha et al.'s work and other empirical evaluations of the Block Model are part of the important avenue of work to understand how students approach program comprehension and to inform educators about potential interventions to target students' program comprehension skills. However, one notable gap in the literature surrounding the Block Model is its application to program comprehension in *large code bases*. Indeed, the majority of works that cite the Block Model are about students' program comprehension on smaller programs at the novice level [9], [16]–[21]. This lack of emphasis on students' program comprehension beyond simple programs only exacerbates the issue of teaching advanced program comprehension skills identified by Lewis [9].

III. Study Context

A. Course Design

In the Spring 2023 term at our public, R1 university, we taught a class in the Computer Science department that aimed to teach students techniques for program comprehension, code management, and project management. The course was offered as an upper-division elective course. The goal of the course was to introduce students to the workflow of comprehending and contributing to a large code base.

Out of the 20 total lectures, the first 10 covered program comprehension techniques. We have included a deeper description of the lecture topics included in the "Code Comprehension" unit in Table II. The order of the lecture topics in Table II is also the chronological order that we taught these topics to students. The remaining lectures covered unit testing (3 lectures), Git Workflow (4 lectures), and miscellaneous topics such as an overview on open-source software and two guest lectures from industry professionals. Lectures followed a consistent structure throughout the term to facilitate the transfer of techniques from instructor to students. In each lecture, the instructor first demonstrated the technique to students (i.e., using the debugger to step through a part of the code base), then motivated and assigned a short activity for students to use the technique on their own for a different part of the code base (i.e., using the debugger to understand a different feature than the one from the demonstration), and finally allowed students to discuss their approach and solution to the activity with peers (similar to Peer Instruction [23]).

TABLE II: Lecture topics taught in the Code Comprehension unit of the course.

Lecture Topic	Description
Code Navigation	Locate and navigate to relevant parts of a code base. Navigation shortcuts with VSCode were taught under this topic.
Experimental Code Changes	Making changes to the code such as print statements, logging, functional changes, etc. to see the relevant output or changes.
Diagramming	Using diagrams as a means to explain how various parts of the code base work at a high level.
The Debugger	Using the various features of the VSCode debugger to understand code.
Using Online Resources	Knowing when to use online resources, which online resources to use, and parsing through documentation.
Reading Test Cases	Reading test cases as a way to understand how developers intended certain parts of the code base to behave.

B. Participants

We administered a survey to all students in the first week of the course. Of the 51 students who responded, 86.3% were men, 9.8% women (the others preferred not to specify). Nearly 33% of the students were in their third-year of the undergraduate program while the remaining 67% were in their final (fourth) year. All students in our course had completed a required course called "Software Engineering" in which students learn about Git version control, AGILE workflow, software design patterns, continuous integration, and testing practices. In that course, students work in teams of 5 students to build a simple web or mobile application from scratch that makes API calls to some external service (examples include Spotify, Google Fit, etc.).

Most students were fairly confident in Python, with 84% having done Object Oriented Programming in Python and only 4% having not used Python before the course. All of our students had used Git before, 71% of students had been exposed to IDE-based debuggers, 77% of students had experience with unit-testing, and 67% of students had experience with Continuous Integration and Development.

Industry experience among students was more varied. Around 45% of students had no prior internship experience, 41% of students had completed one internship, and 14% of students had two or more internships. In terms of experience with large code bases, 35% of students had only worked on small programs (less than 1000 lines of code), 41% of students had worked with moderately-sized code bases (small open-source projects or projects created from scratch in a previous course), and 24% of students have worked on large open-source projects or enterprise systems.

C. The Programming Task

The course revolved around the `idlelib` code base. IDLE is a simple integrated development environment for Python, typically used by novice programmers learning to program in Python. It is an open-source project maintained by an active developer community. In our course, students downloaded the 3.12.0a6 Python version (the "a" represents an *alpha* version). A high-level description of the `idlelib` code base is below:

- The code base has roughly 23,358 lines of code (including 9,644 lines of test code).
- The code base contains 128 Python files (including 67 files of test code).
- The code base is written entirely in Python.
- The code base is part of a large open-source community.

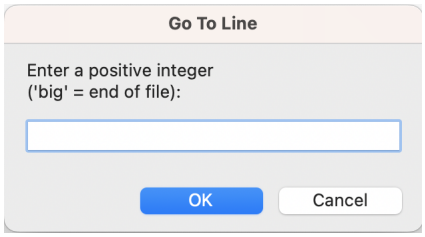


Fig. 1: Image of the IDLE "Go to line" feature shown to students in the task instructions.

In the third and fourth week of the course, students were tasked with updating the "Go to line" feature in the IDLE code base. The task was presented to students as if they were members of a software engineering team being assigned a task. The exact instructions are as follows:

> Our marketing team has just conducted extensive user testing of developers that use IDLE. One of the main issues identified by our team involved the 'Go to line' feature found under 'Edit' when the Python Editor Window is open.
>
> Users reported being confused about what 'big' meant (whether it meant they should type in the word big or type in a big number). We would like you to update the 'Go to line' popup to fix this issue. Our competitors, such as VS Code, have an effective implementation of such a feature. When the user sees the popup for 'Go to line', the maximum allowed input (i.e., the number of lines within a file) is shown to the users.
>
> **Required Features**
>
> You must implement the following features to receive full credits for this assignment. Specifically, edit the popup to fulfill these requirements:
>
> 1) Change the "Go to line" popup such that the popup should show the user the range of values allowed (i.e., show the total number of lines).
> 2) If the user enters a value that is too large (bigger than the total number of lines), a red error message should be shown to users that says 'Enter a number between 1 and N, inclusive' where 'N' is the number of lines in the file.
> 3) Implement the logic to handle a negative number using reverse indexing. For example, -1 should go to the last line, -2 should be the second to last line, etc. If a user enters a number that is too negative such that it extends past the first line, users should be shown the message 'Negative entries must be between -N and -1, inclusive' where 'N' is the number of lines in the file.

Students submitted the project on the course Github repository hosted on Github Classroom [1] via a pull request.

This task represents only one of the many types of programming tasks that could be completed in a large code base. This task required students to understand an existing feature and add functionality to it, whereas other tasks in a large code base could include adding a completely new feature to the code base, triaging and debugging an error, writing test cases, etc. However, we specifically chose this programming task to be representative of the nature of programming tasks in industry (although many industry code bases are significantly larger than `idlelib`). In fact, in their paper detailing aspects of the academia-industry gap, Craig et al. note that interview subjects reported spending much more adding features to the existing code base rather than writing stand-alone code [3]. The task described above aims to capture the process of understanding existing code and extending its functionality.

D. Solution to the Programming Task

To understand the skills and techniques students used to complete the programming task, we will describe the extent of the code navigation, code comprehension, and code writing that was needed to implement the requirements.

The "Go to line" feature is implemented in two files—`editor.py` and `query.py`. The `goto_line_event` method is defined on line 693 in `editor.py` [2], which is called when the user invokes the "Go to line" feature. This method initializes a `Goto` object, which is defined in line 229 in the `query.py` file [3]. Students should recognize that the `Goto` class is a subclass of the `Query` class and can look at the constructor for the `Query` class to understand the `Goto` class. Once students locate the relevant code in these two files, the main technical challenge was finding out how to calculate the total number of lines in the Python editor (i.e., the maximum number a user can enter in the "Go to line" popup). There are a number of ways students could find the solution, including using the debugger to step through code, using Google or ChatGPT, reading documentation about Tkinter text widgets, or looking at similar features in the code base to find the correct code. Regardless of how students' approached the challenge, they should recognize that the Python editor in IDLE is a Tkinter text widget, allowing them to find the relevant function and parameter combination to obtain the number of lines in the active IDLE editor. Ultimately, the solution required students to add code to the `goto_line_event` method to access the number of lines in the Python editor and to modify the "Go to line" popup prompt. Students also needed to create additional input validation checks in the `Goto` class in `query.py`. Overall, the requirements could be implemented by adding or modifying only six lines of code.

IV. Methods

A. Data Collection

During the project, students were required to maintain a *process journal* to document the steps they took to complete

[1] https://classroom.github.com/classrooms

[2] https://github.com/python/cpython/blob/v3.12.0a6/Lib/idlelib/editor.py\#L693

[3] https://github.com/python/cpython/blob/v3.12.0a6/Lib/idlelib/query.py\#L229

each project. The instructions for maintaining the process journals was given to students along with the task instructions (shown above). The instructions for the process journals are replicated below:

> Please also create a "Process Journal" called "journal.md" where you describe your code navigating and code understanding process as best as you can. Try to reflect on how you went about understanding the parts of the codebase that you need to change and how you went about adding the feature. Please describe major obstacles you faced or decisions you had to make as well as how you got unstuck from those obstacles (using Google, office hours, discussed with others, etc.)

The process journals were submitted along with the code changes via a pull request on Github for teaching assistants to evaluate, and assignments were not accepted without a process journal. During the lecture section, we also described example details to include in the process journal, such as the VSCode shortcuts students used, online resources students consulted and for what purpose, specific lines of code that were confusing, etc. A consequence of this self-reported data collection is that students' process journals varied in length and quality. This is an important limitation of our study which we will expand upon in Section VI-D. Overall, we collected 48 process journals from the task we assigned to students.

B. Data Processing

Because students submitted the process journals alongside their code changes in a file called "journal.md", we could easily access the text of the process journal. Our data analysis process is described in the block of pseudocode below:

```
procedure analyzeProcessJournals():
    //T is the set of PC techniques
    T = {Code Navigation, Debugger,
        Diagramming, Experimenting,
        Online Help-Seeking, Office Hours,
        Code Reading, Documentation,
        Prior Knowledge, Unknown}
    //D is the set of Dimensions
    D = {Text, Execution, Purpose}
    //H is the set of Hierarchies
    H = {Atoms, Blocks, Relations, Macro}
    for each PJ = Process Journal:
        read PJ
        divide PJ into Comprehension Steps
        S = set of Comprehension Steps
        for each Comprehension Step in S:
            Assign one Technique from T
            Assign one Dimension from D
            Assign one Hierachy from H
    END
```

We will elaborate upon the two main qualitative steps in our analysis. The first stage of the process was to divide the process journal into "comprehension steps" where each step represents one technique or action to understand the code base. This step was necessary because many process journals included details beyond just the strategies and techniques they used for understanding the code. Many students included details such as implementation steps, including the code that they added, the errors they faced, and general reflections about the project work. Therefore, we extracted only the snippets related to code comprehension. The main factor we considered for whether to include a snippet as a "comprehension step" was whether there was 1) an implied or explicit action that led to the understanding and 2) there was a fact or piece of understanding gained from the act. The research team conducted weekly meetings in which we discussed examples of agreed-upon comprehension steps and how to distinguish comprehension steps from implementation details in order to have a shared understanding of comprehension steps. Because of the potential subjectivity in deciding which snippets to mark as "comprehension steps," each process journal was analyzed by two members of the research team who were extremely familiar with the programming task, the program comprehension techniques taught in the course, and the Block Model. When the two researchers disagreed on whether something should be a step, there was a discussion conducted via comments on a collaborative Google Sheets spreadsheet.

The second stage of our analysis was to label each "comprehension step" with a Technique, Dimension, and Hierarchy in a similar process to Jayathirtha et al.'s work [18]. The technique labels were applied when students made a specific mention of the technique they used, such as a code navigation feature, making an experimental code change, using the debugger, etc or when it was clear which technique a student used. We applied a label called "Unknown" when students did not specify any technique they used and the technique could reasonably be multiple options. Tables III and IV present the descriptions of the Dimensions and Hierarchies that our group applied when labeling our data. For this phase, two reviewers independently labeled the comprehension steps for the same process journals and then deliberated to resolve any disagreements. We conducted this process of independent labeling and deliberating of sets of roughly 10 to 12 process journals at a time. By the final round of deliberation, the reviewers had a shared understanding of when to apply each Technique, Dimension, and Hierarchy. In the final round, the two reviewers aligned on 52/57 (91.2%) Technique labels, 45/57 (78.9%) Hierarchy labels, and 47/57 (82.4%) Dimension labels. Across both stages of our data processing, we consistently had two reviewers deliberate until an agreement was reached. As a result, we do not have any inter-rater reliability statistics to report since there was never an occurrence of a single reviewer deciding the steps or labels.

By the end of the data processing, we had 41 process journals. Although there were 48 students that submitted an assignment, we excluded 7 of the process journals because they lacked any details on the program comprehension process and described their implementation process instead (such as code they tried to write, errors they encountered, etc.).

TABLE III: Definitions of dimensions in the Block Model

Dimension	Description
Text	Understanding syntax, file structure, code organization, method calls, and object types (i.e., *What/where is the code?*)
Execution	Understanding the sequence of calls, tracking variable values, control flow, data manipulation (i.e., *How does the code run?*)
Purpose	Understanding how the code relates to subgoals and goals in the context of the IDLE application (i.e., *What does this code do?*)

TABLE IV: Definitions of hierarchies in the Block Model

Hierarchy	Description
Macro Structure	Understanding high-level patterns in the code base, such as file structure and big-picture purpose of classes
Relations	Understanding method calls, object creation, and data flow *between* classes, methods, and files.
Blocks	Understanding a specific code snippet of interest, typically entire methods or several lines of code
Atoms	Understanding a specific token, variable, or line of code

C. Data Analysis

We conducted analyses to determine 1) the typical order of blocks that students explored (RQ1), 2) the typical order of Dimensions that students explored (RQ1), 3) the typical order of Hierarchies that students explored (RQ1), and 4) the most common pairings of techniques to blocks in students' comprehension processes (RQ2).

1) RQ1 Analysis: To understand the patterns in students' program comprehension processes (i.e., top-down vs bottom-up, Text-to-Purpose vs Purpose-to-Text, etc.), we calculated the average position of each Dimension and Hierarchy within students' comprehension process. Since the process journals varied in the number of comprehension steps, we could not simply use the raw position values to understand patterns in students' processes. Instead, we used the *relative order* of each Dimension and Hierarchy. We accomplished this by first assigning a *progress percentile* to each comprehension step in a process journal based on its order within the process journal and the total number of steps in the process journal. We chose our calculation for the *progress percentile* such that the first step in each progress journal would have a percentile of 0% and the last step would have a percentile of 100%. For example, in a process journal with 9 comprehension steps, step 1 would have a progress percentile of 0%, step 2 would have a progress percentile of 12.5%, step 3 would have a progress percentile of 25%, and so on until step 9 (the last step), which would have a progress percentile of 100%.

We conducted separate analyses to identify the *average progress percentile* of each Dimension and Hierarchy. By knowing the relative progress percentiles of the Dimensions, we can identify the general order of Dimensions that students target in their comprehension process. Similarly, the general order of Hierarchies sheds light on whether students' typical comprehension process for our programming task was a top-down approach (as posited by Brooks [13]), a bottom-up process (as posited by Pennington [11]), or an ad hoc, opportunistic search (as posited by Littman [14]).

2) RQ2 Analysis: Our analysis for RQ2 is more straightforward. To understand which comprehension techniques map to specific Dimensions and Hierarchies, we conducted a frequency analysis for each of the comprehension techniques we observed. The frequency analysis consists of a simple count of pairings between comprehension techniques and blocks from the Block Model.

D. Example Process Journal Analysis

Below is a snippet from the beginning of a process journal:

> **Finding Relevant Code**
> The first step in implementing this feature was to figure out where exactly the current 'Go To Line' feature is implemented. In order to find it, I searched for the keyword 'big' (with the quotation marks) in VS Code's global search. This resulted in one location: goto_line_event() in `editor.py`. In order to make sure I found the correct code, I added another line of text in the prompt box and tested the change by using the feature in IDLE.
> **Finding The Endex**
> After finding where the 'Go To Line' feature is implemented in the codebase, the next step is to display the valid line numbers within the event prompt. In order to do so, I first looked into the self.text part of the code, which did not yield much on its own, but I did see that the text is represented by a MultiCallCreator object. After reading through the object definition in `multicall.py`, I concluded that there was no way I could get the line number range from the text object itself.

Two members of the research team then deliberated and agreed upon the following list of comprehension steps from the snippet above. The comprehension steps we extracted were:

1) *I searched for the keyword 'big' (with the quotation marks) in VS Code's global search. This resulted in one location: goto_line_event() in* `editor.py`.
2) *In order to make sure I found the correct code, I added another line of text in the prompt box and tested the change by using the feature in IDLE.*
3) *I first looked into the self.text part of the code, which did not yield much on its own, but I did see that the text is represented by a MultiCallCreator object.*
4) *After reading through the object definition in* `multicall.py`*, I concluded that there was no way I could get the line number range from the text object itself.*

Once the comprehension steps were finalized, two members of the research team independently assigned a technique, dimension, and hierarchy to each step. After comparing labels and deliberating to resolve disagreements, we assigned the following labels for the comprehension steps above, which are listed in Table V.

TABLE V: Example analysis of process journal.

Step	Technique	Dimension + Hierarchy	Justification
1	Code Navigation	Text: Blocks	Using keyword to find region of interest (ROI)
2	Experimenting	Purpose: Blocks	Experimenting to validate understanding of code base and IDLE application
3	Code Reading	Text: Atoms	Found a useful variable but not sure how to use it
4	Code Reading	Execution: Blocks	Understanding the data in the object for a specific block (object definition)

Progress Percentiles per Block

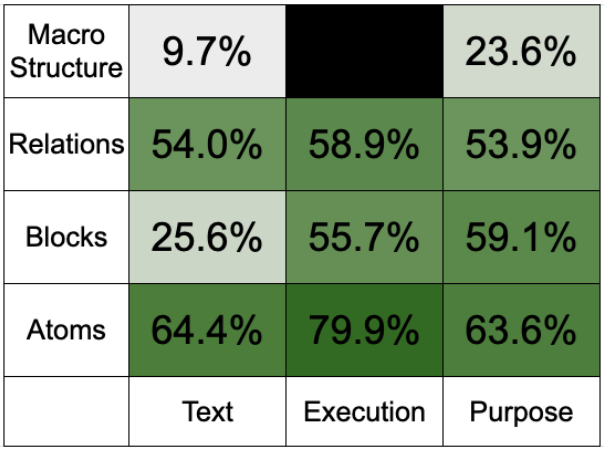


Fig. 2: A heatmap of the progress percentiles of the blocks in the Block Model. A darker hue indicates a higher progress percentile (i.e., a *later* average position of that block in the PC process). A black tile indicates a sample size of 0.

V. Results

A. RQ1 Results

Our first research question seeks to understand students' program comprehension process through lens of the Block Model. Figure 2 is a heatmap of the progress percentiles of the blocks in the Block Model. Note that the larger percentiles (which correspond to a darker hue in the figure) indicate that this block was visited later in the comprehension process, on average. For example, the *Text: Macro Structure* and *Purpose: Macro Structure* blocks have low progress percentiles, indicating that these blocks were visited early in the comprehension process, on average. Conversely the highest progress percentiles (and thus, the darkest tiles) are present in the *Atoms* row, indicating that these tiles were visited later in the comprehension process, on average. Interestingly, the tiles in the *Relations* and *Blocks* row have very similar values ranging from 53.9% to 59.1%, with the exception of the *Text: Blocks* tile which has a progress percentile of 25.6%. The black tile on the *Execution: Macro Structure* block indicates a sample size of 0 in our data.

Figure 3 shows the count of comprehension steps for each

Count of Comprehension Steps per Block

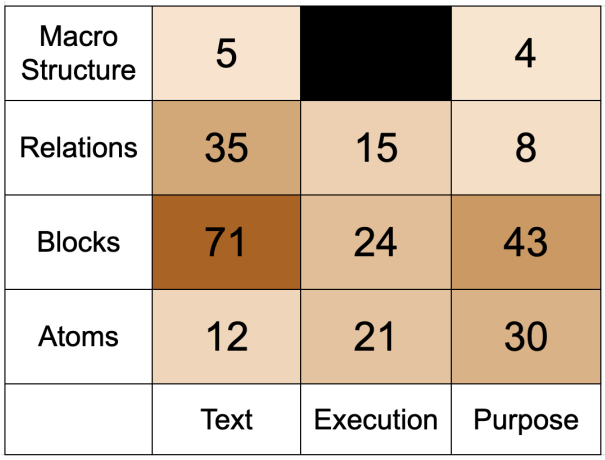


Fig. 3: A heatmap of the count of total comprehension steps made by students for each block. A darker hue indicates more comprehension steps targeting that block. A black tile indicates a sample size of 0.

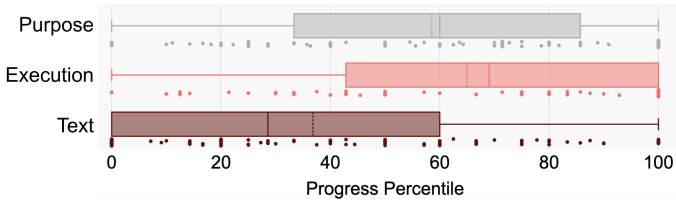
block in the Block Model. Overall, only 9 total comprehension steps about *Macro Structure* were mentioned by all students.

We also present two boxplots (Figures 4(a) and 4(b)) to compare the average progress percentiles across Dimensions and Hierarchies, respectively. In Figure 4(a), *Text* has the lowest median progress percentile at 28.6% ($n = 128$), then *Purpose* at 60.0% ($n = 85$), and *Execution* at 69.0% ($n = 60$). Figure 4(b) shows the average progress percentiles of the Hierarchies. *Macro Structure* has the lowest median progress percentile at 11.1%, although the sample size is small ($n = 9$). After *Macro Structure*, *Blocks* has the next lowest median progress percentile at 40.0% ($n = 138$), then *Relations* at 53.6% ($n = 58$), and finally *Atoms* at 80.0% ($n = 63$).

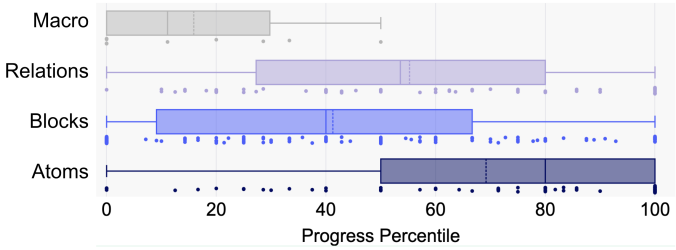
B. RQ2 Results

Our second research question aims to understand which program comprehension techniques students tended to use for certain blocks of the Block Model. We present four heatmaps (Figure 5) to visualize how specific program comprehension techniques were used (Code Navigation, Experimental Code Changes, Code Reading, and the Debugger, respectively). We chose to include these figures in the main text of the paper because of their relevance to our interpretation and key takeaways of our findings; however, the heatmaps of the remaining techniques are included in the Appendix. Note that a larger number of steps using that program comprehension technique for a particular block in the Block Model correspond to a darker hue in the figures.

Figure 5(a) shows a heatmap of steps that used Code Navigation. Code Navigation was primarily used for blocks in the *Text* dimension. Within *Text*, the majority of steps cited code navigation for *Text: Blocks* ($n = 58$), with the second-highest block being *Text: Relations* ($n = 19$).



(a) Average Position of Dimensions



(b) Average Position of Hierarchies

Fig. 4: Boxplots of the progress percentiles of the Dimensions in the Block Model. Higher values indicate a *later* average position of that Dimension in students' comprehension process). The minimum, first quartile, median (the solid line in the boxes), mean (the dotted line in the boxes), third quartile, and maximum values of the progress percentiles are shown.

Figure 5(b) displays a heatmap of steps that used Experimental Code Changes. Experimenting was used primarily for *Execution* and *Purpose*, more specifically in the *Atoms* and *Blocks* Hierarchies. *Purpose: Blocks* had the highest number of steps associated with using experimenting as the program comprehension technique ($n = 19$).

Figure 5(c) shows a heatmap of how the Debugger was used for blocks in the Block Model. The Debugger was solely used for blocks in the *Execution* Dimension and was the least-used technique ($n = 9$) compared to the other techniques in the figure. The majority of steps used the debugger for *Execution: Blocks* ($n = 5$), followed by *Execution: Atoms* ($n = 3$), and finally *Execution: Relations* ($n = 1$).

Figure 5(d) is a heatmap of steps that used Code Reading as a code comprehension technique. This technique was distributed most evenly across *Text*, *Execution*, and *Purpose* compared to the other comprehension techniques listed (Code Navigation, Experimental Code Changes, and the Debugger). Code Reading was used most in the *Blocks* Hierarchy ($n = 32$), with *Purpose: Blocks* ($n = 13$) and *Execution: Blocks* ($n = 11$) being reported the most.

VI. Discussion

A. Interpretation of Results

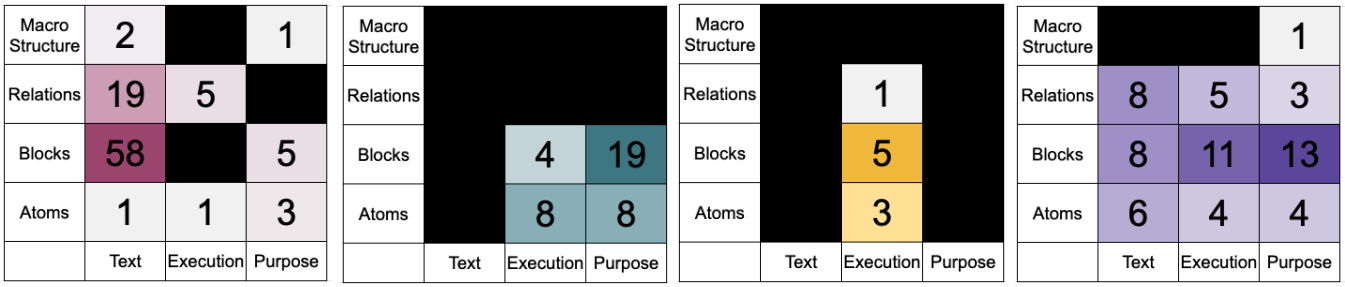
1) Research Question 1: Our results reveal several high-level approaches that our students followed. In terms of the hierarchies of the Block Model (Figure 4(b)), we observed a roughly *top-down* comprehension process–on average, Macro Structure understanding occurred, albeit rarely, before the lower hierarchies. Most students actually started with a step in the Relations or Blocks hierarchies, and proceeded to explore various code chunks and the relationships between those blocks, as seen by the frequency of progress percentiles at roughly 50% for the Relations and Blocks dimensions (Figure 2). Subsequently, students narrowed their search and comprehension to specific *Atoms*, which were typically the final steps of the comprehension process.

In terms of the dimensions of the Block Model (Figure 4(a)), we observed a *Text-first* comprehension process. In total, 36 of the 41 students started with a comprehension step in the *Text* dimensions, typically by locating a specific code snippet that implemented the feature they needed to modify. Following this initial understanding of the text surface, students typically proceeded to creating understanding about *Purpose* and *Execution* dimensions, with many students ending with understanding the specific data held in a specific variable, which would count as the *Execution* dimension.

As seen in Figure 3, students made more inferences about the *Text* dimension (123 comprehension steps) than the *Purpose* (85 steps) and *Execution* (60 steps) dimensions. Similarly, students made more mentions about the *Blocks* hierarchy (138 comprehension steps), than *Atoms* (63 steps), *Relations* (58 steps), and *Macro Structure* (9 steps).

We hypothesize that we observed this comprehension process because our task asked students to implement a modification of an *existing feature* without pointing students to specific pieces of code as a starting point. As a result, students needed to first *locate* the relevant code snippets, understand the relationships between those locations, and then finally need to understand the data contained in a specific variable (see Section III-D for a description of a solution to the task). However, other tasks performed in a large code base might involve a different set of instructions such that programmers would start at a specific line of code and work their way "up" the Block Model, resulting in a more *bottom-up* process. Therefore, we note that our observed results are certainly limited to specific programming tasks and programmers. Our discussion is not meant to apply to students' program comprehension processes across all large code bases, but rather is limited to a feature modification task in a large code base in which the output is a user-facing, observable application. We recognize that not all large code bases necessarily allow programmers to immediately run the code to detect the impact of code changes.

2) Research Question 2: Students used a variety of program comprehension techniques, such as code navigation features in VSCode, making experimental code changes, the IDE-based debugger, etc. Our analysis sheds light on *how* students used these various techniques. Notably, Code Navigation was primarily used for the *Text: Blocks* and *Text: Relations* tiles, likely because Code Navigation helps with locating regions of interest via global keyword searches and jumping between them via shortcuts such as `Go to Definition` and `Show All References`. Students typically used Experimental Code Changes for the *Execution* and *Purpose* dimensions. We believe this result represents the two typical reasons for making experimental code changes: first, students used a print



(a) Code Navigation (b) Experimental Code Changes (c) Debugger (d) Code Reading

Fig. 5: Comprehension Blocks targeted by the various program comprehension techniques exhibited by students.

statement to display variable values (data flow) or to see if a snippet of code was run (control flow), which is considered as *Execution* in the Block Model; second, students made changes to code such as changing the 'Go to Line' prompt and then ran the program to see the impact of the changes on the IDLE application, which falls under the *Purpose* dimension.

B. Related Work

Our results show notable similarities to prior empirical works on developers' comprehension processes. Sharafi et al. conducted an eye-tracking study of 36 student developers completing realistic tasks related to maintenance and debugging in a large code base [24]. The authors discovered that a significant predictor of a developer unsuccessfully completing the task is *thrashing*—the process of excessively switching between code elements [24]. Unsuccessful developers displayed this thrashing behavior 35% more than successful developers, leading to the authors recommending educational interventions to explicitly improve students' code navigation strategies [24]. This work largely confirmed the findings of Robillard et al., who examined the code investigation strategies of software developers and found that while effective developers followed a structured, methodical approach to understanding the source code, ineffective developers resorted to techniques such as "code-skimming" and scrolling in an effort to stumble upon the relevant piece of code [25]. Similarly, an industry-focused study by [26] 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) [26]. An important finding of this work was 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 [26]. The authors include a specific recommendation for CS curricula in universities to teach students how to use the debugger.

Our results largely align with the studies described above. Though our data collection does not enable us to know whether students were thrashing, our data shows a lack of varied comprehension techniques being used beyond code navigation and code reading (Figure 5). In general, students used features such as the keyword-search and shortcuts such as `Go to Definition` and `Find All References`, but we observed little use of making experimental code changes, the IDE-based debugger, reading test cases, and other strategies. These results indicate that the recommendation set forth by Beller et al. to explicitly instruct students on the IDE-based debugger may not necessarily translate to student use of the debugger [26], since the students in our course were aware of the debugger but chose not to use it (despite its clear value in the task). In contrast to these prior studies that used think-aloud protocol or recordings of participants' screens, our use of process journals provides a less-granular picture of students' program comprehension process. Instead, our results shed more light on the *inferences* made by students during the program comprehension process, allowing us to compare the processes used by our students to the processes described in established program comprehension theories.

C. Theoretical Implications

Perhaps the most relevant theory of program comprehension for our work comes from Von Mayrhauser and Vans, who noted the distinct lack of work related to program comprehension for large-scale code bases [27], [28]. Von Mayrhauser and Vans used a think-aloud protocol with 11 professional developers tasked with understanding part of a large code base (80,000 lines of code) [28]. The authors concluded that experts working to comprehend part of a large code base 1) rely heavily on domain knowledge of the code base and 2) use a multi-level approach that visits various hierarchies of abstraction [28]. Further, in Schulte's work to connect the Block Model to previous program comprehension theories, the authors describe how Von Mayrhauser and Vans' theory implies a bottom-up process of understanding the *Text* dimension and an abstracted, top-down process of understanding *Execution* and *Purpose* [10]. Our findings confirm part of this theory since students certainly used a multi-level approach to understand the "Go to Line" feature, as indicated by the concentration of values around 50-60% for *Blocks* and *Relations* (Figure 2). Students seemed to jump between locating blocks of code, identifying relevant atoms, and understanding the data flow of those atoms between blocks. However, our results do not show that students used the bottom-up approach to understand the

Text dimension, as their domain knowledge of the code base may have been insufficient to have a clear *Atom* or *Block* to start a bottom-up comprehension process.

Our findings also highlight the hierarchy of abstractions that these upper-division students make throughout the program comprehension process. A foundational study from Lister et al. in 2006 showed a clear difference between novices and experts in their ability to make *multi-structural* inferences beyond the atomic and syntactic elements of a program [12]. On one hand, students certainly progressed beyond grasping onto syntactic elements, as shown by the presence of many comprehension steps in the *Blocks* hierarchy. However, the relative lack of comprehension steps in the *Macro Structure* and *Relations* hierarchies demonstrate a limit to students' ability to make abstract representations of the code.

Our findings also imply that students demonstrated somewhat similar processes to those found by Pennington in her study of 80 professional developers [11]. Pennington showed that programmers started with a *procedural* (control flow) representation of the program before moving onto a *functional* (goal hierarchy) representation [11]. Although Pennington compares *text structure knowledge* to *plan knowledge*, whereas the Block Model further divides the *text structure knowledge* into the *Text* and *Execution*, both our work and Pennington's work found that *plan knowledge* (or *Purpose*) came later in the comprehension process than the *text surface knowledge*.

Given the time-consuming nature of studying programmers' *process* of comprehending code, our sample size of 41 students in their third- or fourth-year of an undergraduate CS program offers a valuable contribution to our understanding of students' "incremental development of code comprehension expertise" that Lewis motivated in her prior work [9]. With this newfound understanding of students' program comprehension process in a large code base, future studies can compare how developers with more domain knowledge approach the same or similar task. Such comparisons are valuable data points for educators aiming to develop students' program comprehension skills to be similar to experts' skills.

D. Limitations

Perhaps the biggest limitation to our study is in the study design and analysis. Unlike prior studies, we did not use think-aloud, stimulated-recall interviews. A trade-off between our approach and these longer-form interviews include sample size (due to the time-consuming nature of the interviews and analysis) and depth of observations. Although we certainly missed out on some key comprehension steps by using our approach instead of interviews, there were some advantages of our design including a higher sample size than prior works while capturing, in our view, the most salient comprehension steps students took.

However, due to the informal and ambiguous instructions for maintaining a process journal for the project (described in Section IV-A), these process journals varied in length and quality. Some students provided great detail about what actions they took to understand the large code base while other students wrote a short summary of what they accomplished in their project, which was not as comprehensive. Additionally, a subset of students focused heavily on how they *implemented* the project, rather than how they were able to *comprehend* parts of the code base. For students that focused on their project implementation, we were unable to analyze a majority of their process journal because they did not list their comprehension strategies to understand the code base.

Another key limitation is that we could not properly detect the impact of students' prior knowledge—an important part of the comprehension process as mentioned by both Schulte [15] and Von Mayrhauser and Vans [27]. Although we hypothesize that students' prior knowledge of the code base was rather limited due to the programming task occurring in the first four weeks of the course, we were unable to empirically assess students' reliance on prior knowledge.

Finally, the programming task described in this study was selected because it was similar in nature to programming tasks in industry, helping us understand students' program comprehension processes for large code bases. However, there are some inherent differences between this task and similar tasks in industry. For example, the open-source `idlelib` code base was written in Python, includes significant documentation on Tkinter (a library for creating text widgets), and is certainly smaller than enterprise code bases. As has been pointed out in previous works related to the theory of program comprehension, various tasks will necessitate different program comprehension approaches [11], [15], [28]. Therefore, our findings are limited to the type of programming task we provided to students.

VII. Conclusion

Our study makes two key contributions to the program comprehension literature:

1) A snapshot of intermediate programmers' approach to comprehending a large code base.
2) A methodology of mapping program comprehension processes to the Block Model, which can enable comparison with other studies that use a similar methodology.

Our students demonstrated some traits of novice program comprehension strategies, such as limited use of program comprehension strategies beyond code navigation and code reading and limited abstractions at the *Relations* and *Macro Structure* hierarchies. However, we observed some traits of professional developers identified by previous works, such as starting with a procedural (*Text* and *Execution*) understanding of the code and then making the connections to functional (*Purpose*) aspects of the code. Though the ultimate goal of this line of work is to contribute to program comprehension pedagogy, this study presents the initial steps towards understanding the strategies students use and the struggles they face while comprehending a large code base.

References

[1] A. Radermacher and G. Walia, "Gaps between industry expectations and the abilities of graduates," in *Proceeding of the 44th ACM Technical*

Symposium on Computer Science Education, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 525–530. [Online]. Available: https://doi.org/10.1145/2445196.2445351

[2] A. Radermacher, G. Walia, and D. Knudson, "Investigating the skill gap between graduating students and industry expectations," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 291–300. [Online]. Available: https://doi.org/10.1145/2591062.2591159

[3] M. Craig, P. Conrad, D. Lynch, N. Lee, and L. Anthony, "Listening to early career software developers," *J. Comput. Sci. Coll.*, vol. 33, no. 4, p. 138–149, apr 2018.

[4] A. Begel and B. Simon, "Novice software developers, all over again," in *Proceedings of the Fourth International Workshop on Computing Education Research*, ser. ICER '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 3–14. [Online]. Available: https://doi.org/10.1145/1404520.1404522

[5] ——, "Struggles of new college graduates in their first software development job," *SIGCSE Bull.*, vol. 40, no. 1, p. 226–230, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1352322.1352218

[6] 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 Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[7] R. Minelli, A. Mocci, and M. 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*, 2015, pp. 25–35.

[8] B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson, D. Dong, H. Kwik, A. H. Tan, L. Hwa, M. Li, and A. J. Ko, "A theory of instruction for introductory programming skills," *Computer Science Education*, vol. 29, pp. 205 – 253, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:86551630

[9] C. 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*, ser. ICER '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 15–28. [Online]. Available: https://doi.org/10.1145/3568813.3600116

[10] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson, "An introduction to program comprehension for computer science educators," in *Proceedings of the 2010 ITiCSE Working Group Reports*, ser. ITiCSE-WGR '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 65–86. [Online]. Available: https://doi.org/10.1145/1971681.1971687

[11] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, pp. 295–341, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:41137387

[12] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: novice programmers and the solo taxonomy," *SIGCSE Bull.*, vol. 38, no. 3, p. 118–122, jun 2006. [Online]. Available: https://doi.org/10.1145/1140123.1140157

[13] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737383800315

[14] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. USA: Ablex Publishing Corp., 1986, p. 80–98.

[15] C. 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*, ser. ICER '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 149–160. [Online]. Available: https://doi.org/10.1145/1404520.1404535

[16] C. Izu, C. Schulte, A. Aggarwal, Q. Cutts, R. Duran, M. Gutica, B. Heinemann, E. Kraemer, V. Lonati, C. Mirolo, and R. 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*, ser. ITiCSE-WGR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 27–52. [Online]. Available: https://doi.org/10.1145/3344429.3372501

[17] J. Salac, C. Thomas, B. Twarek, W. Marsland, and D. Franklin, "Comprehending code: Understanding the relationship between reading and math proficiency, and 4th-grade cs learning outcomes," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 268–274. [Online]. Available: https://doi.org/10.1145/3328778.3366822

[18] G. Jayathirtha and Y. B. Kafai, "Program comprehension with physical computing: A structure, function, and behavior analysis of think-alouds with high school students," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 143–149. [Online]. Available: https://doi.org/10.1145/3430665.3456371

[19] Q. Cutts and M. Kallia, "Introducing modelling and code comprehension from the first days of an introductory programming class," in *Proceedings of the 7th Conference on Computing Education Practice*, ser. CEP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 21–24. [Online]. Available: https://doi.org/10.1145/3573260.3573266

[20] C. Izu and C. 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*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 466–472. [Online]. Available: https://doi.org/10.1145/3341525.3387425

[21] M. Hassan, K. Cunningham, and C. 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*, ser. ICER '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 329–343. [Online]. Available: https://doi.org/10.1145/3568813.3600140

[22] P. Kather and J. Vahrenhold, "Exploring algorithm comprehension: Linking proof and program code," in *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3488042.3488061

[23] L. Porter, C. Bailey Lee, and B. Simon, "Halving fail rates using peer instruction: A study of four computer science courses," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 177–182. [Online]. Available: https://doi.org/10.1145/2445196.2445250

[24] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, "Eyes on code: A study on developers' code navigation strategies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1692–1704, 2022.

[25] M. Robillard, W. Coelho, and G. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[26] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 572–583. [Online]. Available: https://doi.org/10.1145/3180155.3180175

[27] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[28] ——, "Identification of dynamic comprehension processes during large scale maintenance," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 424–437, 1996.