A Qualitative Analysis of Lecture Videos and Student Feedback on Static Code Examples and Live Coding: A Case Study

Derek Hwang
UC San Diego
dehwang@ucsd.edu

Vardhan Agarwal*
UC San Diego
v7agarwa@ucsd.edu

Yuzi LYu*
UC San Diego
yul134@ucsd.edu

Divyam Rana*
UC San Diego
dirana@ucsd.edu

Satya Susarla
UC San Diego
sbsusarl@ucsd.edu

Adalbert Gerald Soosai Raj
UC San Diego
gerald@eng.ucsd.edu

Abstract

One of the goals of computing education research is to understand and document the effectiveness of pedagogical strategies in computing. Among the many methods available to teach programming, two commonly used techniques to present code in Computer Science classes are static code examples (where pre-written code snippets are used during lectures) and live coding (where code is written before the students during the lecture). Even though prior research has tried comparing the effectiveness of these two teaching techniques on student learning and cognitive load, little is known about the structure of these code presentation techniques. In this study, we analyze the lecture recordings of a mid-level Computer Science course which uses both static code examples and live coding for teaching code snippets. We analyze these recordings with the intent to understand what these pedagogical techniques for teaching and learning programming consist of. We also analyze student feedback about both these pedagogical strategies to better understand these teaching methods from the students' perspective. We believe that our work will shed light on the usefulness of static code examples and live coding in Computer Science courses.

CCS Concepts

• **Social and professional topics** → **Computer science education**.

Keywords

Static Code Examples; Live Coding; Student Survey; Video Analysis

ACM Reference Format:
Derek Hwang, Vardhan Agarwal*, Yuzi LYu*, Divyam Rana*, Satya Susarla, and Adalbert Gerald Soosai Raj. 2021. A Qualitative Analysis of Lecture Videos and Student Feedback on Static Code Examples and Live Coding: A Case Study. In *Australasian Computing Education Conference (ACE '21), February 2–4, 2021, Virtual, SA, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3441636.3442317

*These authors contributed equally to this work.

ACE '21, February 2–4, 2021, Virtual, SA, Australia

ACM ISBN 978-1-4503-8976-1/21/02.
https://doi.org/10.1145/3441636.3442317

1 Introduction

Computer Science (CS) is often taught in lectures using a variety of methods including but not limited to traditional lecture using a blackboard or document camera, PowerPoint slides, Peer Instruction, Think-Pair-Share, and Process Oriented Guided Inquiry Learning (POGIL). Code snippets in CS classes are usually presented using two methods: 1) static code examples [37], i.e., code snippets that are prepared beforehand and are presented in lectures to explain how they work, and 2) live coding [33, 39], i.e., writing code snippets from scratch (or starting from some boilerplate code) in front of the students during lecture.

In static code examples, instructors often present pre-written code snippets to the classroom first, followed by explanations about the purpose of the code [37]. Although static code examples often lack exposure of the programming process, such examples are well-structured and thoroughly tested before class. Moreover, the code files can be shared with students before the class to better prepare them for the lectures.

In live coding, instructors will often incrementally develop their program in a step by step manner. By using iterative development, instructors often expose the various cognitive processes that they use when writing programs [9].

Although prior studies have reported the (in)effectiveness of live coding on student learning [33, 37] and student preferences toward this technique [30, 38, 39], to the best of our knowledge there has been limited research on static code examples even though it is one of the two popular techniques for presenting code in CS lectures. Also, prior studies have not documented the various processes involved in presenting code snippets using static code examples and live coding.

In our case study, we shed some light on the different pedagogical techniques involved in these two code presentation methods by analyzing video lectures in a mid-level CS classroom which consists of students who are also working professionals. Further, we also tie our observations from these video lectures to a student survey which gathered data on students' preferred method of teaching programming (static code examples, live coding, or a combination of both). By analyzing both the recorded lecture videos and a student feedback survey, we hope to share some insights on the structure and process involved in static code examples and live coding sessions and their benefits reported by students.

Our research questions are as follows:

(1) What are the pedagogical techniques used in CS lectures involving static code examples and live coding?

(2) Which aspects of static code examples and live coding do students appreciate?

Our work makes the following important contributions to computing education: 1) documentation of the processes involved in presenting code snippets using static code examples, 2) a deeper understanding of the pedagogical techniques involved in two commonly used code presentation methods, and 3) a better understanding of the reasons why students prefer one code presentation method over the other.

We believe that our case study is one of the initial steps in better understanding the code presentation techniques in CS. We hope that our work will help new (and experienced) CS teachers to evaluate and choose a code presentation technique according to their needs.

2 Related Work

Analysis of the lecture model of teaching reveals the structure of the lecture process as well as its advantages and disadvantages. By drawing on previously held notions about the lecture style, Kaur provides a broad view of present day lectures [17]. We consider our work to be tangentially related as we are also conducting a qualitative analysis of teaching strategies, namely static code examples and live coding. Similar to Kaur's work, we will be providing definitions of these two pedagogical approaches and break down the distinct processes of such practice. One difference between our work and Kaur's work is that our work seeks to observe these teaching phenomena (static code examples and live coding) as they happen in one particular case, as opposed to a broad review of static code examples and live coding.

Previous work done on analyzing video recordings provides a broad description of the gathering and interpreting of video recording based data. Situations that are described to be worthwhile to be analyzed through video recordings are plenty, but children interactions and classroom instruction are some of the more prevalent ones. Specific practices such as recording time stamps and categorization of observations using specific keywords are also included as a video analysis methodology [10]. As we consider our study to be a qualitative analysis of a classroom environment, we follow the following procedures as described in this paper in which we locally store all relevant video material, create timestamps for relevant portions and categorize them based on specific keywords (phrases). Furthermore, as we make observations on phenomenon occurring, important context such as the time stamp, topic introduced, and a relevant keyword (phrase) association is also made. The methodology for analyzing long videos as described in this paper guides our own methodology.

Prior research in Physics shows that while demonstrations during lecture are a positive addition, they were not as effective as instructors had hoped. It was found that when students simply observed the demonstration and did not have follow up comprehension activities, they did not show better understanding of the materials compared to the group that did not observe the demonstration at all [7]. In Milner-Bolotin's study, they observe an increase in students' correct responses when the students are asked to perform a prior discussion before seeing the demonstration [25]. The students that made the prediction prior to the demonstration had a 30% higher correct response rate. The students that not only made the prediction but also discussed with their peers have a 50% higher correct response rate [25]. We consider our work to be an extension of this prior work in computing education where we focus on two code presentation techniques, one of which is an in-class demonstration (live coding).

One popular way of presenting static code examples in CS courses is the use of slides. Since there is little research done on the presentation of static code examples through slides, we relate to other pedagogical studies about the effect of slides in the classrooms for other courses. While there are critiques about the foreshortening of thinking and the effectiveness of slides [40], there has been one study that reveals an 8.3% boost in students' performance with the addition of slides in lectures. This study conducted by Weinraub consists of two sections of the same corporate finance course [42]. The section in fall utilized overhead projection and printed handouts; while the winter section received the computer slides and printed copy of slides afterwards. Weinraub spotted an 8.3% higher performance from the section that adopted computer slides [42]. Although this late 1990s study suggests improvement from the use of slides, more recent studies have been reporting no significant differences.

Beets and Lobingier found no significant difference in students' performance between the use of chalkboard, overhead projector, and slides [1]. While arousal of attention is being spotted when lectures incorporate computer-generated slides, the effect or benefits of such arousals is inconclusive. However, studies do suggest that students experience enhanced learning when they have access to the copies of the slides[22]. Among all the studies one commonality surfaced which is the strong preference for computer slides coming from students' feedback, and such preference extends beyond the United States [4]. Clark concludes that students are able to recognize the benefit of slides through the application of color and movement; nevertheless, the success of the class is more closely related to the instructor's ability of incorporating such software [6]

Live coding as a tool for teaching programming has been previously studied in detail [30, 33, 37–39]. Although the effectiveness of live coding vs. static code examples is still unknown [33, 37], students' preferences toward live coding is overwhelmingly positive [33, 34, 38, 39]. Rubin's study analyzed the findings of two professors each teaching two sections, a control and an experiment section of an introductory programming course [33]. Through post class surveys, Rubin's study found that 90% of the experimental group preferred coding examples, whereas only 67% of the control group preferred these coding examples [33]. Soosai Raj et. al. conducted live coding sessions for two groups of students in a data structures course and analyzed student feedback on these live coding sessions[39]. Their analysis of the student feedback found that live coding helped students understand programming as a process and also helped students learn debugging. This notion of live coding being positively received by students is further affirmed through subsequent works that report on students' positive reactions when asked about what they thought about live coding [34, 39]. As previous papers have done so, we also intend to collect student feedback on live coding and link these notions to specific processes that a live coding session consists of. Additionally, we also intend to conduct the same analysis for static code examples, an area of research that is lacking compared to live coding.

Previous work, though limited, has studied live coding as a process. Previous work done by Bennedsen and Capersen note the following as processes of live coding: use of the IDE, incremental development, testing, and error handling [2]. Their work provides a basis for understanding how to teach programming as a process. Our study is marginally related to their work, as our study seeks to examine the various parts of the static code examples and live coding teaching techniques, and the various processes involved in these methods. Additionally, Gantenbein writes that the following processes can be used in beginning CS courses as a way of "incorporating the software life cycle": define the problem, select an approach, design a solution, implement the solution, and test the solution [9]. In our study, we follow in the spirit of these previous works as we look to outline the structure for not only live coding, but also another form of code presentation, static code examples.

Two prior works examined the effectiveness of live coding on student learning [33, 37]. Rubin found that students in the live coding group performed better on the final project when compared to students in the static group but there was no difference with respect to students' conceptual knowledge measured using exams [33]. Another study on the effectiveness of static code examples and live coding found no significant difference between the two methods [37]. We believe our qualitative work will help us better understand these two code presentation techniques so that we could evaluate their effectiveness on student learning using a new perspective.

3 Methodology

In this section, we describe the details of the course that we observed for our case study.

3.1 Participants

Our study was conducted at a research intensive university in the United States. Our study spanned over five weeks of an Intro to Computer Systems course. This course had 70 students who attended the weekly lectures. Each student was a full time employee working at a neighboring software company. One of the researchers of this study was an instructor for this course.

3.2 Course Details

Intro to Computer Systems is a mid-level computer science course. It is the third course in a sequence of courses required in order for the employees within the company to switch to the software development department. Prior knowledge on introductory Java programming and data structures was a pre-requisite for this course.

This course consisted of weekly 3 hour long lectures and was split into three sections: 1) C programming (first five weeks), 2) Assembly (next five weeks), and 3) Systems Programming (last six weeks). This paper will concentrate on the first five weeks of this course. We are interested in only the first five weeks of lecture content because it focuses primarily on C programming and the student feedback survey was given at the end of the first five weeks.

When teaching C programming, students were taught using a combination of static code examples and live coding by two instructors – the lead instructor and their teaching assistant (TA). The instructor taught during the first, third, and fourth week whereas the TA taught in the second and fifth week. Both instructors taught using a combination of static code examples and live coding. The lead instructor taught mostly with live coding sessions and the TA taught mostly with static code examples. The following topics were covered during the first five weeks of this course: variables, conditionals, loops, arrays, pointers, structures, stack vs heap, linked lists, file input/output, low-level C Programming (bit-wise operations), C preprocessor, makefiles, valgrind, and multi-file compilation

3.2.1 Lecture Format This course taught the C programming language in a traditional lecturing style using overhead projection and included code demonstrations in the form of static code examples and live coding. An approximately equal amount of time was allotted to both forms of code demonstration over the duration of 5 weeks, though an unequal amount of time was allotted for static code examples and live coding in each specific lecture. Lectures were recorded, uploaded to the course website, and are publicly available for viewing. These videos are available here: http://bit.do/static_live_lectures_full .

Lectures consisted of 3 parts: handwritten notes, static code examples and live coding. Hand written notes were used to teach concepts in C programming (e.g., how linked lists work). Static code examples and live coding were used to teach code snippets (e.g., a function to insert a node at the end of a linked list). All of the resources created in class were available for future use by students from the course website. The course website can be found here: http://pages.cs.wisc.edu/~gerald/cs354/Spring18/

3.2.2 Student Feedback Survey At the end of the fifth week, an anonymous student feedback survey was given. A total of 70 students attended this course, 55 responses were recorded from the student feedback survey. The student feedback survey asked the following:

(1) While teaching C code, which of the following teaching strategies helps you learn better?
    a) static code examples, b) live coding, c) Both, d) Other (Please specify)
(2) Why do you think static code examples or live coding helps you learn better?

Student responses to this survey had no impact on their grade in the course.

4 Data Analysis

In this section, we describe how we analyzed both the lecture videos and the student feedback survey. Additionally, we present how we assessed the quality of our analysis.

4.1 Video Analysis

We conducted video analysis to identify the structure of static code examples sessions and live coding sessions that were used in this course. We followed a well documented analysis method for analyzing long videos [10]. This method consists of locally storing video materials, creating timestamps, and categorizing phenomena using specific keywords. We downloaded all the lecture videos and kept track of specific phenomena with a corresponding time stamp. All observations were recorded on a spreadsheet with a short description of its significance. Once completed, all spreadsheets were shared among all researchers.

4.1.1 Initial Watch The first author watched the videos two times. The first watch was used to get a general sense of the content

presented. During the second watch through of the videos, detailed notes were made about the topic presented, what method was used (static code examples or live coding), and specific observations with a timestamp and keyword.

4.1.2 Initial Analysis The first author assigned approximately equal lengths of video spread across five different videos for three other researchers to observe phenomena and take notes on.

The three researchers were not given any information about specific phenomena to be observed or what presentation method(s) were used. This was done in order to minimize the risk of any one specific researcher's biases affecting the work of the other researchers.

4.1.3 One-on-one Discussions Discussions about observations occurred between researchers who had completed their section of video analysis. All researchers participated in one-on-one discussions at least once. Discussions were approximately one hour and thirty minutes in duration. The first author met with the second, third, and fourth author in order to have one-on-one discussions. A total of four meetings were held.

(1) **Discussion:** Researchers discussed their observations, making notes of the common phenomena observed. If researchers agreed on any of the phenomena, then a corresponding note would be made on a shared document. If researchers disagreed on any of the phenomena observed, then they would discuss and seek agreement.
(2) **Addressing Disagreement:** Both researchers viewed the video clip associated with the disagreed upon phenomena at the same time. They would discuss their observations orally. Any agreements here were noted. All disagreements were settled in this way.
(3) **Developing Processes:** The researchers would summarize all observed phenomena upon completing discussion on a single video. Here, any lingering disagreements were resolved.

4.2 Student Feedback Survey Analysis

Of the fifty-five collected student responses, thirty-seven responses (four static code examples preference, eighteen live coding preference, fifteen both preference) were analyzed. A total of eighteen responses were dropped since students did not mention why they preferred a certain teaching method and so researchers were unable to associate a code with these responses.

As the number of student feedback surveys were relatively smaller compared to the video data, all researchers read all of the survey responses.

4.2.1 Initial Surveys Reading All researchers independently read all responses to the student survey and separated responses based on a student's preference. For responses without a clear preference, a preference was inferred (see example 1). Several responses indicated a preference but did not provide an comprehensible reason and therefore were not considered during the analysis (see example 2).

Example 1: The student's preference was inferred to be live coding based on their response. The identifier represents the place where the student's response can be found in this spreadsheet: http://bit.do/static_live_survey_live. The letter represents the sheet in which the response is present and the number represents the row number.
Identifier: L23
Preference: Give an exercise then review the solution (like 3/14 lecture)
Reasoning: "Static is hard to follow; hard to understand what the coder was thinking when they wrote the code. Live lets you see why each line of code was written because the coder talks out loud."

Example 2: The student indicated a live coding preference, but inadequate reasoning was given.
Identifier: L4
Preference: Live-coding
Reasoning: "live"

4.2.2 Coding Several student responses included comments on the other teaching method (students who preferred static code examples mentioned live coding and vice versa). Such sentiments were also recorded by researchers by analyzing specific keywords or phrases of the student's response. For survey responses where a preference of both was indicated, these responses were analyzed as long as any reason was given for either static code examples or live coding. Researchers associated at least one code that best summarized the reason for their preference as described by the student response. When developing codes for these responses, researchers used "units of meaning" as described by the individual coder [11, 21]. This method was used since students may have imbued multiple meanings when they noted their reason for preference and we wanted to, as accurately as possible, code their reason. All responses where at least one code was unable to be assigned were not considered for further analysis (see example 3).

Example 3: This response was not considered for further analysis since no code could be assigned.
Identifier: L3
Preference: Live-coding
Reasoning: "live coding"

4.2.3 One-on-one Discussions for Developing a Coding Scheme Discussions on the coding scheme occurred when researchers finished coding all relevant responses. All researchers participated in one-on-one discussions at least once. Discussions were approximately two hours in duration. A single, shared coding scheme was constantly edited and revised during discussions.

(1) **Discussion:** Researchers would discuss their set of codes and reasoning for assignment. If both researchers agreed, the code would be adopted. If the code was worded differently, but the researchers agreed on its meaning, then it was rewritten and adopted. Any disagreements were settled the following way.
(2) **Addressing Disagreement:** Researchers disagreed in two ways: (1) conflicts in definitions and (2) conflicts in reasoning. Conflicts in definitions were resolved through discussions and mutual agreement on a clarified definition. Upon agreement, the code was adopted. Conflicts in reasoning were resolved through negotiation. Both researchers read the corresponding student response and explained their reasoning. Upon agreement, the code was adopted. In all cases, all disagreements were settled this way.

(3) **After Discussion:** All researchers who had participated in prior discussions would independently verify that the newly generated coding scheme still matched their own. In all cases, the most recent coding scheme was accepted and adopted by all the reviewing researchers.

4.2.4 Finalizing the Coding Scheme and Linking Processes The process of linking our developed coding schemes to specific processes of static code examples and live coding sessions happened as a two step process. A group of four researchers separated into two groups of two jointly made connections between codes and processes. The coding scheme and processes were shared to both groups. These links were made on a shared spreadsheet with rows that included the name of the code, which responses were coded using that code, and at least one associated process. During this time, some codes were found that could not be associated with processes that arose from an analysis of the lecture video recordings. For this, an external process called "outside of in-class processes" was created to cover these codes.

Upon completion, both groups of two researchers met together.

(1) **Discussion:** All researchers would discuss their set of codes and the associated process. If all researchers agreed, then the code and process were finalized. Any disagreements were addressed in the following way.
(2) **Addressing Disagreement:** Researchers disagreed in two ways: (1) conflicts in definitions and (2) conflicts in process association. Conflicts in definitions were resolved through discussions and mutual agreement on a clarified definition. Upon agreement, the code and process would be finalized. Conflicts in process association were addressed using negotiation. All researchers consulted the student response and a shared list of processes and explained their reasoning. If an agreement was reached, then the code and the process was finalized. If an agreement was not reached, then the code and process was dropped from the finalized list.

4.2.5 Assessing Analysis In order to assess the soundness of our analysis, we used inter-rater reliability and inter-rater agreement [16, 19, 24, 41]. We did four rounds of comparisons between the codes that four researchers independently came up with. It is important to note that all researchers were looking at the same set of responses so the processes such as unitization [18] were not used. Whenever disagreements about what code to be used occurred, researchers used a "negotiated agreement" approach in order to resolve these conflicts [11]. Interpersonal dynamics during negotiations was not an issue since all researchers that participated in the development of codes are all same-level peers studying at the same university. In order to calculate inter-rater reliability, we used the proportion-agreement method [26]. By using proportion-agreement, we are able to calculate a measure of agreement between different researchers. This metric is calculated by taking the total number of agreements and dividing it by the total number of agreements and disagreements. Although this method does not account for chance agreements between coders [3], we feel that since we had a numerous number of codes (31 unique codes), chance agreements were unlikely to happen. Furthermore, each researcher was required to have a corresponding phrase directly from the student response to support their code. As we consider our study to be exploratory in nature, previous work has argued that the proportion-agreement method is an acceptable approach for such a study [21].

Our set of codes undertook four stages of review. The first three stages involved the one-on-one discussions with the other three researchers. The fourth stage occurred during the group meeting when codes and processes were finalized. In both cases, the total number of agreements and disagreements were counted once the meeting was over. These numbers were used to calculate a proportion-agreement metric.

The proportion-agreement metric for the first three stages are noted in Table 1. The percentages below denote how many agreements occurred between the participating researchers.

Table 1: Proportion Agreement

Discussion	Static Code examples	Live Coding	Both
1	83.33%	79.17%	82.76%
2	83.33%	81.81%	79.31%
3	83.33%	82.8%	80.5%

For the first three stages of review, there were no prevailing disagreements after engaging in "negotiated agreement" and as such, inter-rater agreement increased to 100% in all cases. Our reported inter-rater reliability scores reveal a satisfactory level of agreement [15, 21] (between 70% and 94%) between the four researchers that participated in the data analysis for this exploratory study.

For the fourth stage of review, which involved all four researchers, codes and associated processes were finalized. We report the inter-rater reliability percentages for this section as 70.97% before negotiation and 93.53% after negotiation. As our reported inter-rater reliability scores reveal that satisfactory level of agreement (nearly 70% before negotiation and 90% after negotiation), was satisfactory level of agreement between the four researchers.

All student responses can be found here: http://bit.do/static_live_survey_raw

5 Results

In this section, we report the results of our case study using lecture video analysis and mid-semester student survey analysis.

5.1 Video Analysis

The different pedagogical aspects of static code examples and live coding that arose from the researchers' video analysis are captured and presented below.

The lecture videos can be found here: http://bit.do/static_live_lectures_full

5.1.1 Structure of Static Code Examples

Lead Up: In this process, the instructor provides some theory using visuals and/or handwritten notes to introduce and elaborate the concepts depicted in upcoming code snippets by adjusting the level of explanation according to the students' familiarity of the concept.

Example: In a lecture, the instructor introduced the concept of C pointers by drawing memory diagrams and explaining using pseudo memory addresses. This was followed by a static code example in

which she showed the implementation while pointing out some good coding practices in reference to the topic and showed the output of the code snippet. The link to the video snippet can be found here : http://bit.do/vidsnippet_leadup_static

View Code / Multiple Files: Static code examples present code snippets in their entirety. Code snippets presented are prepared beforehand and adhere to the instructor's proper style guidelines. Multiple code snippets may be prepared by the instructor in order to explain the concept to the students.
Example: In a lecture, the instructor showed a C code snippet to the students to explain logical shifts by repeatedly explaining a couple of lines from the snippet followed by compiling and running the code to show the output. The students could view the whole code at once. The link to the video snippet can be found here: http://bit.do/vidsnippet_viewcode

Explain: Explanations of static code examples can occur as either a general overview of what the code is doing or a line-by-line explanation. The instructor may also choose to emphasize important concepts and skip simple parts of the code.
Example: In a lecture when the instructor explained the makefile, she went through pre-written code of the makefile while giving proper definition of the dependencies, rule, and target. The link to the video snippet can be found here: http://bit.do/vidsnippet_explain

Execute Code: After the explanation of concepts, the instructor compiles and runs the code, producing an output which may have been explained before compilation of code. Apart from that, the instructor may also choose to let students predict the output.
Example : The instructor explained the concept of macros in C. For doing so, the instructor first gave an explanation of the code and then asked the class to predict the output of the code snippet. The link to the video snippet can be found here: http://bit.do/vidsnippet_executecode

5.1.2 Structure of Live Coding

Lead Up: In this process, the instructors usually have some handwritten notes which include some theory and diagrams prior to the live coding sessions. They try to make sure that the students understand the concept before diving into the live coding example.
Example: Since the instructor wanted to show the output of numbers in decimal and hexadecimal representation in C using live coding, he gave a brief overview of 2's complement and unsigned binary numbers and made sure that the students grasped the concept before moving on to the live coding example. The link to the video snippet can be found here: http://bit.do/vidsnippet_leadup_live

Iterative Development Cycle: This development cycle consists of three distinct processes, namely, thought process, demonstrate writing process, and debugging process. The thought process and writing process happen together as the instructor explains their reasoning while writing code. The debugging process occurs once the instructor has entered a phase of testing and repetitive edits to increase functionality of the already written code.

The first two processes of this cycle happen at the same time. In this part, instructors can choose to implement the live coding sessions in two ways. They can either write the code themselves while explaining each line of code and its need, or they can start by asking guiding questions about what students think. Based on the students' response, the instructor may choose to narrate the reasons while writing a line of code or after they have finished writing. There may be some student initiatives in which student-led code is written and explained. The writing process is non-linear, meaning the instructor usually starts writing and explaining the basic functionality of code, which is followed by added edge cases and conditions. After these series of steps, the instructor might either face an unintentional bug or may intentionally add bugs. In this debugging section, the instructor may ask for student participation in order to identify a present bug and suggest a fix. At this point, the instructor may encounter two types of bugs: lapses in judgement for syntax leading to syntax errors or conceptual errors. The instructor fixes the syntax error and the students may or may not have noticed them. However, the conceptual errors are often led by student involvement.

The three processes that we identified under iterative development cycle are explained below.

1) Thought Process: In this process, the instructor explains their reasoning for each line of code written. During this process, the instructor may also choose to involve students by asking for students to develop some part of the code. Additionally, the instructor may also choose to incorporate extra resources such as documentation or hand-drawn visual diagrams.
Example: The instructor opened a new file to write an implementation of linked lists. He walked through his ideas to demonstrate the need for every step he was doing. For instance, while writing the function prototype (see below) for appending a new node to the linked list, the instructor demonstrated the need for all the parameters he was writing ("Since we are going to make some modifications to the list, this function should know which list should be modified or to which list should I add some data at the end… ").
Function Prototype used in the above example:

```
void insert_at_end(struct node *head, int data);
```

The link to the video snippet can be found here: http://bit.do/vidsnippet_thought

2) Writing Process: In this process, the instructor writes code in front of a student audience. The instructor may choose to write code in a non-linear fashion, constantly going back to previously written code and re-modifying it. Furthermore, the instructor may choose to use placeholders (void return type) and return to modifying code that used placeholders. The notion of development being non-linear is shown in this step.
Example: In a lecture which involved writing functions while implementing a linked list, when the instructor started working on the null checks, he wrote a partially wrong code intentionally and asked the students if the code will work. This led to students providing their possible fixes and this is when the instructor also demonstrated the need to change the return type from void to struct node pointer, hence adding the return statements to the code and thus modifying the code to remove bugs. The link to the video snippet can be found here: http://bit.do/vidsnippet_writing

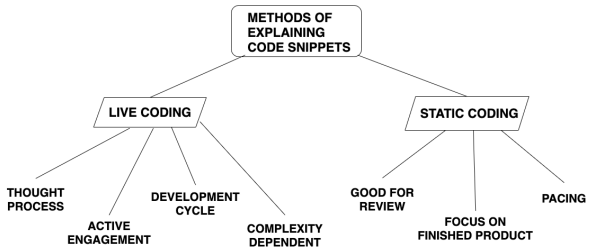
3) Debugging Process: This process involves the instructor making some intentional and unintentional bugs while live coding. During this process, they made some edits to the code to demonstrate the edge cases and to arrive at a final, correct solution. This step also involves the instructor asking questions to the students to identify the bug and suggest a fix, if any.

Example: In a lecture, while the instructor taught the class how to append a node in a Linked List, he wrote code for the same, and pointed out that he was missing a null check for the head of the linked list (intentional bug). Now, instead of fixing it directly, he goes to the terminal to show the segmentation fault to the students. However, when he runs it, he sees a totally different bug which was raised because NULL and malloc were not identified by the GCC compiler (unintentional bug) which led to the instructor including the stdlib.h library. Then he went on to show the segmentation fault and fixed it. The link to the video snippet can be found here: http://bit.do/vidsnippet_debugging

5.2 Survey Analysis

We present the key results of our student feedback analysis of the two teaching techniques, static code examples and live coding based on the survey responses. Using the methodology we mentioned in the previous section, we were able to notice some common features of static code examples and live coding addressed by students' survey responses. These codes (key features) were presented for both static code examples and live coding along with some students' survey responses from which the code was inferred. Figure 1 represents two methods of explaining code snippets that are discussed in this paper

Figure 1: Methods of Explaining Code Snippets



All responses were labeled using S, L, or B to denote the students preference as static code examples, live coding, or both respectively.

A list of all codes generated from the student feedback responses can be found here: http://bit‘.do/static_live_survey_codes

5.2.1 *Static Code Examples*

Student responses as referenced below can be found here: http://bit.do/static_live_survey_static

Good for reviewing: With the nature of being pre-written and easily distributed among students, we noticed that static code examples are often appreciated for their use as a review resource. Student responses show that static code examples are good for reviewing because students already know the topic and therefore understand the solution code. Students are also able to interpret the code functionality on their own due to its readability. Some of the corresponding responses depicting this idea are:
"*Static code is good for review because we should know the topic and understand how we got to the solution*" (L10). Here in this response, this student appreciates that they can refer back to static code examples when studying and rely on the fact that the instructor went over the specific examples in class so they understood the reasoning behind the static code examples.
Respondents also stress the fact that it is important for them to use static code examples while studying on their own. "*Static code is more useful for self-study where you can read it and figure out its function on your own*" (L12). The appreciation for the option to independently contemplate on the static code examples ties well to our next key feature: pacing.

Pacing: Static code examples allow the students to read through all of the code at once. Some students prefer to look at and follow through the code at their own pace, which static code examples enable them to do. While they prefer to look at the code at once on their own pace, some students find this effective only for easier concepts. They find the static code examples faster and thus time-saving. Also, students find static code examples not helpful for more complex problems. They find it difficult to understand the complex concepts due to the absence of line by line discussion strategy of the code snippet.
The following response emphasizes the importance of being able to read the code and form an initial understanding independently. "*For simple concepts, I prefer static code examples. There's no need to waste time typing out the code, plus it's fairly straightforward to understand all of the code at a glance. For more complicated concepts, live coding is useful, since you can explain the rationale at each step*" (B28) which indicates the students' preference towards static code examples in less complex concepts and preference of live coding for more complex concepts.
"*I like to be able to look at the code in at my own pace while it's on screen, which static code allows me to do*" (S5) indicate a pattern: the preference for static code examples surface when students need their space or time to read the code, which live coding does not offer since code will be written live in class.

Focus on finished product: Static code examples allow students to view a completed product with proper styling and inline comments. A few students feel that they can concentrate well on the code explanation when static code examples are prone to less errors because they have been developed by the instructor in their own time. Some of the corresponding responses are:
"*Static is more efficient and less prone to errors...*" (B26) This response appreciates an error-free version of code where as with live coding the instructor could make mistakes while typing in real-time.
"*Static coding lets you see more code all at once, and see the overall structure of a finished product.*" (B4) This response points out that static code examples are able to give a holistic view of the snippet that helps students to better understand the overall structure, which live coding lacks.

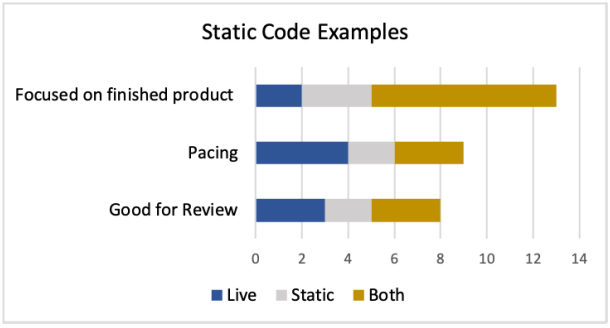
Figure 2 shows the distribution of student feedback responses across the different codes for static code example.

5.2.2 *Live Coding*

Student responses as referenced below can be found here: http://bit.do/static_live_survey_live

Thought Process: Live coding exposes the instructor's thought process during development and the reasoning behind their code. We have seen high frequency of respondents reporting that live

Figure 2: Distribution of Responses across Different Codes for Static Code Examples



coding helped with developing their thought-processing skills because they can see the code evolve from scratch and understand the need for adding a new line in the code which makes it easy to recall later. Some of the responses depicting this idea are:
"*Live coding is more useful when learning because it introduces a thought process.*"(L12)
"*...I'm able to see how the code evolves from function declarations to a working prototype. It also helps to explain why assumptions are made and certain things are written certain ways.*" (L2) As these responses suggest, live coding entitles students to the complete reasoning behind each line of code which is an integral part to help students to write their own program.

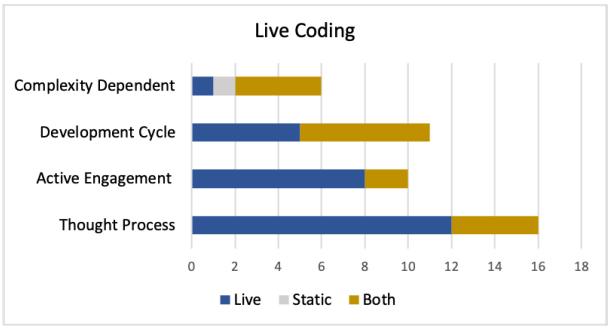
Active Engagement: Live coding allows students to be actively engaged during the presentation. Activities such as predictions, answering questions, and suggesting code were commonly mentioned. Many survey responses show that students appreciate they can be more involved writing code along with the instructor in live coding sessions. Some students mention that as the instructor writes a line of code, they like to speculate the next line of code or observe subtleties in the code. Some of the sample responses are:
"*I like anticipating the next line or answer without it being there. It helps me determine if I'm on the right track.*" (L6) This quote mentions an important distinction between static code examples and live coding. According to this response, the student is able to catch their error or misunderstanding early as they predict the next line.

Development Process: Live coding exposes the instructor's development process as a whole. Students are able to see all the steps behind the progression from a blank file to a fully working program. Some students indicate a better understanding of the overall development process (i.e., thought process, code writing skills, and debugging skills) and the specific logic behind the code. This is because they can visualize the code progression as the instructor builds up the concepts with explanation about the code as the instruction continues. One of the student responses that captures this idea:
"*I understand much better by doing, so being able to code along while the instructor does, and then actively compile and run the code I've written is very helpful. It's also more engaging to see something built up gradually.*" (L17) This response appreciates that live coding enables them to complete the development cycle from the beginning to the end. It also ties back to the previous point that the students are more engaged when they get to see actual code being developed before them.

Complexity Dependent: Live coding was preferred by students for discussion of more complex concepts. "*Live lets you see why each line of code was written because the coder talks out loud*"(L23). This response points out that thinking out loud and the more in-depth explanation by the instructor are very helpful. However, students also address the shortcoming of live coding that for simpler concepts, live coding slowed the class down and thus was not time efficient compared to static code examples. One of the students mentioned, "the only time I feel static code is better is when the example code is really simple since then live coding can slow the class down too much" (L19). We can see that although this student prefers live coding, they believe there are scenarios static code examples are more efficient than live coding.

Figure 3 shows the distribution of student feedback responses across the different codes for live coding.

Figure 3: Distribution of Responses across Different Codes for Live Coding



6 Discussion

In this section, we provide an interpretation of our results, state the limitations of our study, and suggest directions for future work.

6.1 Interpretation of Results

Our study directly compares the static code examples and live coding sessions and breaks them down into specific processes as outlined in Section 5.1.1 and Section 5.1.2. Although, our findings matched with previous papers [2, 9], it is important to note that the specific terms used differed. For example, terms such as incremental development, testing, and error handling [2] describe the same processes as the terms thought process, writing process, and debugging process. The specific terms to define the problem, adopt an approach, design a solution, implement the solution, and test the solution [9] describe what we believe to be the same processes. Our study affirms the findings from these previous studies on live coding and contributes more details about static code examples (which has not been documented as well as live coding). Even though our results may seem like common knowledge to instructors, to the best of our knowledge our study is one of the first that specifically states the anatomy and/or features of a static code example and a live coding session.

Additionally, it should be understood that teaching programming using static code examples and/or live coding are not mutually exclusive. It is possible to teach with static code examples and engage students with questions where they may predict outputs of code code snippets or guess the next line of code to be written. One of the anonymous reviewers of a previous version of this paper stated, "there are many ways to integrate live coding aspects in a power point presentation by using "animations" in the way of revealing step-by-step the next parts of the program". Likewise, a live coding session may have little to no interaction with the students and consist of only the instructor writing code to an audience. Although our findings did not result in such observations of static code examples and live coding, we acknowledge that they exist.

In a programmer's education, their code reading and tracing skills are often tested. Interviews with instructors of programming courses revealed a consensus that reading code was important, and yet, was often not taught directly [5]. Additionally, code reading is a part of the development cycle as programmers must read their code to refine their pre-written code [5]. The process of writing code is likely associated with code reading and tracing ability [23]. Code tracing plays a crucial role in the building of a solid foundation for a programmer's code writing skills [14, 20]. The use of static code examples provides a unique opportunity for instructors to explicitly incorporate these elements of code reading and tracing directly into the *View Code* (see Section 5.1.1) process since all of the code is visible to students. There were responses from students that indicated that they appreciated the opportunity to read through the code at their own pace and even review it later (see Section 5.2.1). We urge any new (and experienced) CS teachers teaching programming to consider explicitly adding elements of code reading and tracing when they present static code examples.

One of the most distinct features in the static code examples is their proper styling. The code was properly structured and the in-line comments thoroughly explained the code. According to previous research, well written code with comments make the code more comprehensible [29, 44], hence reducing the cognitive load on the students [8]. This was corroborated by our survey analysis where the respondents found the static code examples to be better for review than the code snippets used in live coding sessions.We suggest any new (and experienced) CS teachers to consider presenting properly styled code as our results suggest that students greatly appreciate code that they can refer back to.

Previous papers have reported and agreed to the notion that live coding is a process of designing and implementing the code "live" in front of a group of students. Previous papers have suggested that live coding helps students understand the thought process behind development [12, 13, 33, 39] and that debugging skills can be developed as a result of the instructor (un)intentionally writing erroneous code and debugging it [33, 39]. Our findings agree with these findings. We hypothesize that the inherent quality of live coding being able to show each line being written along with an explanation in real time from the instructor (Thought Process and Writing Process, see Section 5.1.2) directly contributes to the development of a student's programming skills. Further, gaining insights into the thought process of an instructor and building one's own debugging skills were among the most popular sentiments that were collected from the student feedback survey (See Section 5.2.2). We hope that our findings add validity to the notion of student perceived benefits of live coding.

Among the computing education research community, there are dissenting opinions on whether or not live coding can be considered an active learning technique. While there are some that suggest live coding involving student activity can be qualified as an active teaching method [38, 43], there are critics who state that "students can just passively copy notes or listen without any active thought" (from an anonymous reviewer for a previous version of this research paper). We also acknowledge that some of the burden to make live coding more active falls upon the instructor. Instructors have to make the decision to also be engaged in the process of live coding for the whole process to be more active [27, 28].

So, the question of how can CS instructors make live coding more active naturally arises. We noticed that there were many instances where the instructor asked for student suggestions. Asking students to predict the output of code snippets during live coding sessions using techniques like Peer Instruction [31, 32, 35, 36] is a way to make instructor-led live coding more active. Another way to make live coding more active is with the inclusion of student-led live coding [12, 13]. By directly involving students in the coding process, students will be more actively engaged with live coding. Interestingly, our survey revealed that there were several students who were "anticipating the next line" (L6) (See Section 5.2.2) during live coding sessions. We believe that further research on determining how to integrate effective active learning techniques with live coding would be beneficial. Additionally, it may be beneficial to extend this research into developing active learning strategies for use during code explanations using static code examples as well.

6.2 Limitations and Future Work

One of the major limitations of our study is that the static coding examples and live coding were not presented by the same instructor. The static code examples were presented by a teaching assistant (TA) while the live coding was conducted by the lead instructor. This may have affected the survey responses because their teaching styles and experience levels differed.

Another limitation is that our video analysis was finished before starting our survey analysis. This may have caused a subconscious bias when we were generating codes based on the student responses. Furthermore, our data set was quite small. Of the 55 collected responses, only 37 were used. Since we had to drop several responses, our analysis may have missed some edge cases that were discussed by students. Apart from that, this is a special case study about students who were working professionals. These students might have some bias towards live coding in the survey due to the adoption of the live coding strategies by their organization. Therefore, the results from the student feedback survey may not be able to be generalized to first year undergraduate students who are full-time students. Finally, students had to identify a preference at the end of five weeks. We acknowledge that five weeks is a very short time and that students may not have been able to form a concrete opinion on static code examples and live coding.

We would like to encourage more research on the effectiveness of static code examples and how students learn when taught with static code examples. To the best of our knowledge, there is a multitude of documentation on live coding and how students learn

when taught using live coding [30, 33, 37, 39]. We believe that, much like live coding, static code examples should also be studied as intensively so that we can compare and contrast these two code presentation techniques in a fair and unbiased manner. We consider our work to be one of the initial steps in this direction.

7 Conclusion

In our case study, we analyzed lecture videos and student feedback to better understand the processes involved in static code examples and live coding. We found that students are able to recognize the benefits and shortcomings from both these techniques. For static code examples, students appreciate the opportunity to strengthen their code reading and tracing abilities. Moreover, students approve the fact that static code examples offer a well-structured resource for them to review. For live coding, students appreciate the iterative development process, which helps breakdown complicated concepts and demonstrates the debugging process. We believe our work contributes toward better understanding of the code presentation techniques. Moreover, our work can give instructors a better sense of which technique(s) to choose while presenting code and what to focus on when using such techniques.

Acknowledgements

We thank the TA for this course Ancy Philip and the students for providing detailed responses to the feedback survey. We are also grateful to our anonymous reviewers for their invaluable feedback and suggestions.

References

[1] S Douglas Beets and Patricia G Lobingier. Cyber dimensions: pedagogical techniques: student performance and preferences. *Journal of education for business*, 76(4):231–235, 2001.

[2] Jens Bennedsen and Michael E Caspersen. Revealing the programming process. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 186–190, 2005.

[3] H Russell Bernard and Harvey Russell Bernard. *Social research methods: Qualitative and quantitative approaches*. Sage, 2013.

[4] Wim Blokzijl and Roos Naeff. The instructor as stagehand: Dutch student responses to powerpoint. *Business communication quarterly*, 67(1):70–77, 2004.

[5] Teresa Busjahn and Carsten Schulte. The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling international conference on computing education research*, pages 3–11, 2013.

[6] Jennifer Clark. Powerpoint and pedagogy: Maintaining student interest in university lectures. *College teaching*, 56(1):39–44, 2008.

[7] Catherine Crouch, Adam P Fagen, J Paul Callan, and Eric Mazur. Classroom demonstrations: Learning tools or entertainment? *American journal of physics*, 72(6):835–838, 2004.

[8] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 286–28610. IEEE, 2018.

[9] Rex E Gantenbein. Programming as process: a "novel" approach to teaching programming. *ACM SIGCSE Bulletin*, 21(1):22–26, 1989.

[10] Andrea Garcez, Rosalia Duarte, and Zena Eisenberg. Production and analysis of video recordings in qualitative research. *Educação e Pesquisa*, 37(2):249–261, 2011.

[11] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, 2006.

[12] Alessio Gaspar and Sarah Langevin. Active learning in introductory programming courses through student-led "live coding" and test-driven pair programming. In *International Conference on Education and Information Systems, Technologies and Applications, Orlando, FL*, 2007.

[13] Alessio Gaspar and Sarah Langevin. Restoring" coding with intention" in introductory programming courses. In *Proceedings of the 8th ACM SIGITE conference on Information technology education*, pages 91–98, 2007.

[14] Brian Harrington and Nick Cheng. Tracing vs. writing code: beyond the learning hierarchy. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 423–428, 2018.

[15] Randy Hodson. *Analyzing documentary accounts*. Number 128. Sage, 1999.

[16] Daniel J Hruschka, Deborah Schwartz, Daphne Cobb St. John, Erin Picone-Decaro, Richard A Jenkins, and James W Carey. Reliability in coding open-ended data: Lessons learned from hiv behavioral research. *Field methods*, 16(3):307–331, 2004.

[17] Gurpreet Kaur. Study and analysis of lecture model of teaching. *International Journal of Educational Planning & Administration*, 1(1):9–13, 2011.

[18] Klaus Krippendorff. On the reliability of unitizing continuous data. *Sociological Methodology*, pages 47–76, 1995.

[19] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[20] Amruth N Kumar. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 183–188, 2013.

[21] Karen S Kurasaki. Intercoder reliability for validating conclusions drawn from open-ended interview data. *Field methods*, 12(3):179–194, 2000.

[22] David G Levasseur and J Kanan Sawyer. Pedagogy meets powerpoint: A research review of the effects of computer-generated slides in the classroom. *The Review of Communication*, 6(1-2):101–123, 2006.

[23] 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*, pages 101–112, 2008.

[24] Mattheu B Miles and A Michael Huberman. Qualitative data analysis: A sourcebook of new methods. In *Qualitative data analysis: a sourcebook of new methods*. Sage publications, 1984.

[25] Marina Milner-Bolotin, Andrzej Kotlicki, and Georg Rieger. Can students learn from lecture demonstrations. *Journal of College Science Teaching*, 36(4):45–49, 2007.

[26] Elizabeth R Morrissey. Sources of error in the coding of questionnaire data. *Sociological Methods & Research*, 3(2):209–232, 1974.

[27] Lex Nederbragt. A video introduction to live coding part 1, 2016. Live coding done poorly. Accessed: August 3, 2020.

[28] Lex Nederbragt. A video introduction to live coding part 2, 2016. Live coding done well. Accessed: August 3, 2020.

[29] Paul W Oman and Curtis R Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 244–250, 1990.

[30] John Paxton. Live programming as a lecture technique. *Journal of Computing Sciences in Colleges*, 18(2):51–56, 2002.

[31] L. Porter, C. Bailey-Lee, B. Simon, Q. Cutts, and D Zingaro. Experience report: A multi-classroom report on the value of peer instruction. *ITiCSE*, 2011.

[32] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. A multi-institutional study of peer instruction in introductory computing. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 358–363, 2016.

[33] Marc J Rubin. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 651–656, 2013.

[34] Amy Shannon and Valerie Summet. Live coding in introductory computer science courses. *Journal of Computing Sciences in Colleges*, 31(2):158–164, 2015.

[35] Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. Experience report: Peer instruction in introductory computing. In *SIGCSE*, 2010.

[36] Beth Simon, Julian Parris, and Jaime Spacco. How we teach impacts learning: peer instruction vs. lecture in CS0. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 2013.

[37] Adalbert Gerald Soosai Raj, Pan Gu, Eda Zhang, 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*, pages 152–159, 2020.

[38] Adalbert Gerald Soosai Raj, Jignesh Patel, and Richard Halverson. Is more active always better for teaching introductory programming? In *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 103–109, 2018.

[39] Adalbert Gerald Soosai Raj, Jignesh M Patel, Richard Halverson, and Erica Rosenfeld Halverson. Role of live-coding in learning introductory programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, pages 1–8, 2018.

[40] Edward R Tufte. The cognitive style of powerpoint. 2003.

[41] Robert Philip Weber. *Basic content analysis*. Number 49. Sage, 1990.

[42] Herbert J Weinraub. Using multimedia authoring software: The effects on student learning perceptions and performance. *Financial Practice & Education*, 8(2):88–92, 1998.

[43] Greg Wilson. Teaching tech together. Accessed: December 13, 2020.

[44] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223, 1981.