Learning to program as empirical inquiry: using a conversation perspective to explore student programming processes

Kristina Litherland & Anders Kluge

To cite this article: Kristina Litherland & Anders Kluge (07 Dec 2023): Learning to program as empirical inquiry: using a conversation perspective to explore student programming processes, Computer Science Education, DOI: 10.1080/08993408.2023.2290410

To link to this article: https://doi.org/10.1080/08993408.2023.2290410

COMPUTER SCIENCE EDUCATION
https://doi.org/10.1080/08993408.2023.2290410

Learning to program as empirical inquiry: using a conversation perspective to explore student programming processes

Kristina Litherland and Anders Kluge

Department of Education, University of Oslo, Oslo, Norway

ABSTRACT
Background and Context: We explore the potential for understanding the processes involved in students' programming based on studying their behaviour and dialogue with each other and "conversations" with their programs.
Objective: Our aim is to explore how a perspective of inquiry can be used as a point of departure for insights into how students learn to program.
Method: We completed a qualitative study situated in elective computer science classes in an upper secondary school in Norway. We collected data by video recording classroom interactions and used screen-recording software.
Findings: Our findings include how we consider programs as both means and ends and reconsider the "error" in trial-and-error strategies, the role of error messages, and how programs are bound to context and particular moments in time.
Implications: Our findings have implications for the ways we understand programs as mediating tools in research and apply them in the field of practice.

ARTICLE HISTORY
Received 5 July 2022
Accepted 29 November 2023

KEYWORDS
Computer science education; empirical inquiry; JavaScript; programming; sociocultural perspective

1. Introduction

The topic of computer programming has gained increasing attention in academic research and educational curricula worldwide (Bocconi et al., 2022), often attributed to Wing's (2006) highly influential paper on the power of learning under the umbrella term of computational thinking. While the body of literature on programming in schools concerns programming as a method for learning other school subjects (Papert, 1980) and general problem-solving skills (Wing, 2006), much of the research in computer science education primarily focuses on higher education, emphasizing programming as a professional skill (e.g. Denning & Tedre, 2021), which may only to a limited extent be transferable to school contexts. Programming is traditionally presented with the goal of writing coded instructions that tell a computer how to perform certain tasks (Denning & Tedre, 2021), and much of programming didactics emphasise the importance of

CONTACT Kristina Litherland kristina.litherland@iped.uio.no Department of Education, University of Oslo, Oslo, Norway

understanding basic programming concepts, such as variables, conditionals, and loops (Lye & Koh, 2014).

In line with the Lye and Koh (2014), and the findings of Luxton-Reilly et al. (2018), there is untapped potential for gaining insights into students' programming processes, e.g. by studying their actions, dialogue with one another and conversations with their computers. Going beyond the concepts of programming in this way is not a new perspective. Soloway (1986) showed that programming students should be taught more than "syntax and semantics" (p. 858). Learning to program is, according to Soloway, "learning to construct mechanisms and explanations" (p. 850). In a similar note, Du Boulay (1989) described computers and programming as "a tool-building tool" (p. 285). Since then, work such as the "The Block Model" (Schulte, 2008) represents a line of research where programming concepts are treated as part of a larger context of code blocks and interrelations that function to reach meaningful goals in a program. Work has also been done on employing the SOLO taxonomy (Lister et al., 2006) where learning to program is framed as moving towards higher levels of abstraction and higher levels of complexity within these different levels of abstraction. Muller et al. (2007) found empirical support that students performed better when instruction was pattern-oriented.

The proliferation of programming in schools since Wing's influential paper has prompted discussions on how curricula should adapt to cater to 21st-century learners and contexts. This has led to the emergence of various approaches to learning and teaching strategies and technologies such as block-based programming (Resnick et al., 2009), "unplugged" programming (Bell et al., 2009), coding puzzles for learning (and assessment) known as Parsons problems (Parsons & Haden, 2006), and the Predict-Run-Investigate-Modify-Make framework (PRIMM) (Sentance et al., 2019).

Our point of departure is taken in a perspective of programming implying more than applying programming concepts in problem solving and more than the result represented by a "finished" program (see for example Bjerknes et al., 1991). To explore learners' programming processes, we adopt a sociocultural perspective on learning, focusing on learning processes in context (Vygotsky, 1980), mediation (Wertsch, 1998), and scaffolding (Andersen et al., 2022). A considerable proportion of computer science education research has favoured cognitive learning approaches (Sentance et al., 2019). By using a sociocultural lens, we apply the perspective that externalization of knowledge through participation in social practices precedes individualized internalized learning. According to Vygotsky (1980) and Wertsch (1998), physical and symbolic tools mediate this process, and the most important learning tool is the symbolic object of talk. We understand the use of tools from a dialogic perspective, where we consider the "sequential unfolding of activities in time" (Arnseth & Ludvigsen, 2006, p. 181). These activities include talk and actions, the latter encompassing both interactions with digital and/or physical tools and the use of gestures. We are inspired by the idea of empirical inquiry (Newell & Simon, 1976), where computing is viewed as a "conversation" between a human and a machine (as described below).

To investigate this phenomenon, we have formulated the following research question: *How can empirical inquiry serve as a perspective for analyzing the activity of students learning to program?*

We utilise the perspective of empirical inquiry as an analytical lens to focus our attention on specific aspects of the programming process and observe the students' actions, dialogue and conversations. We reserve "dialogue" for the exchange between humans, and "conversations" to describe the exchange between humans and computers. Our data consists of students working in pairs and individually.

The remainder of the paper is organised as follows. First, we present literature on empirical inquiry and the field of learning to program. Then, we introduce our method, followed by our results. Finally, we discuss the results and present our conclusions, limitations, and directions for further work.

1.1. Empirical inquiry

In general terms, empirical inquiry is a method for investigating research questions through the use of empirical data, as opposed to exploring such questions theoretically. However, Newell and Simon (1976) presented a perspective on computer science as empirical inquiry, building on the work of Simon (2019, first published in 1970), in which he argued for the "sciences of the artificial". Newell and Simon emphasised the use of computers as empirical sources: "Actually constructing the [computing] machine poses a question to nature; and we listen for the answer by observing the machine in operation and analyzing it by all analytical and measurement means available" (1976, p. 114). By observing and analysing the machines' behaviour, researchers can extract answers and gain knowledge. Therefore, computers ("the machines") are not only solutions to problems; they can also be seen as data from which we may learn. From this point of view, programs are not just a set of instructions but also part of a process of questioning and interpretation.

In our study, we adopt an empirical inquiry perspective to explore the process of learning to program. Our focus is not on defining computer science as a scientific field of inquiry. Instead, we conceptualize student programming as an inquiry process including conversation (von Hausswolff, 2021). We emphasize that programming is not merely a physical tool but also a symbolic tool, serving as an object for conversation. Programming is a process that involves many "hidden" mechanisms, e.g. in our case how JavaScript is read by the computer and returned as a web page with which the students can interact. Inquiry may be a perspective that can show how students learn about the relationships between the code and the results it presents. Rather than viewing programming as a series of one-shot instructions, we highlight how students understand and utilize the feedback they receive from the computer during the programming process. We focus on students' talk, writing, and editing of code, as well as the execution of the code as sources of empirical inquiry, as described in the method section.

Our perspective on empirical inquiry aligns with Dewey's (1938) pioneering work on inquiry as an educational method. Dewey argued that problems cannot be solved remaining detached from them. Instead, interaction with problems is essential for their resolution. According to Schön (1992), Dewey emphasised the "inherently open-ended relationship between the inquirer and the situation" (p. 122). In the context of programming, this relationship can be seen in Papert's (1980) notion of "microworlds" where students interact within specific subject-related situations. Learning and discovery can, metaphorically, be framed as a "conversation with the situation" (Schön, 1992, p. 125), a perspective that relates to the concept of bricolage where the scientist builds knowledge on the interaction with materials (Turkle & Papert, 1990). Pea (1986) claimed that many novice programmers hold a misconception that computers have a "hidden mind" that can understand what the programmer is trying to program, the so-called “superbug”. We do not delve into the debate on the existence of the superbug; rather, we examine how students explore and “converse” with the program code in specific situations. Our perspective is therefore not on how the student conceives that the computer understands them (i.e. Pea’s superbug), but on how the student understands the program by conversing with it.

We build upon these contributions by providing empirical insights within a learning context, utilising a broader understanding of inquiry inspired by Dewey (1938), and extended by Schön (1992), as a framework for our analysis within a sociocultural learning perspective. Our perspective considers computers as parts of conversations, with code and code execution serving as integral components of the process. By adopting this perspective, we enhance our understanding of how learning occurs in specific inquiry situations.

1.2. Learning to program: processes, practices, strategies

In what activities do the students engage when they learn to program? Researchers describe these activities using different terms: practices and strategies (Brennan & Resnick, 2012), processes (Allsop, 2019), and execution of skills (Grover et al., 2015). Lye and Koh (2014) suggested that the field of practices is under-investigated, Robins et al. (2003) discussed the difference between knowledge and strategies in learning to program and called for more work on how novice strategies emerge. Since then, work has been carried out on supporting the development of programming processes of students, including coding puzzles known as Parsons problems (Parsons & Haden, 2006) and the PRIMM framework in which students are – among other activities – asked to read code snippets and predict the results before they start coding on their own (Sentance et al., 2019). The approaches are often based on developing and testing teacher interventions, including tools for assessment (Weintrop et al., 2021). There seems to be less work on the inductive exploration of what practices the students themselves develop and use, such as trial and error strategies (Moskal & Wass, 2019) and debugging (Liu et al., 2017). Robins (2019, p. 361) argues that more knowledge is needed on what differentiates “effective and ineffective novices”.

Trial-and-error is reported as a common strategy among novice programmers, occasionally presented as the act of making seemingly random changes to a program (Moskal & Wass, 2019) and as unwanted behaviour in classrooms (Hao et al., 2022). Some researchers present trial and error as a meta-strategy for developing methods of debugging (Brennan & Resnick, 2012), which is the process of identifying and fixing problems in a program. Debugging may be considered a separate process to that of programming (Liu et al., 2017) and is sometimes framed as “moments of failure” (Dahn & DeLiema, 2020, p. 363). In this study, debugging is seen as an important part of programming as it concerns such questions as how errors in the program occur, where they are found, and how students deal with them (McCauley et al., 2008). For instance, Liu et al. (2017) found that debugging was a much more demanding task than the production of new code. We aim to build on the idea in which the unfolding programming processes are taken as a starting point for the analysis, focusing on trial and error and debugging activities.

1.3. The role of dialogue and conversation in learning to program

In dialogic learning, researchers may focus on features of the exchange, such as the ways participants take on shifting roles of listener and source in the learning process (Furberg & Silseth, 2022).

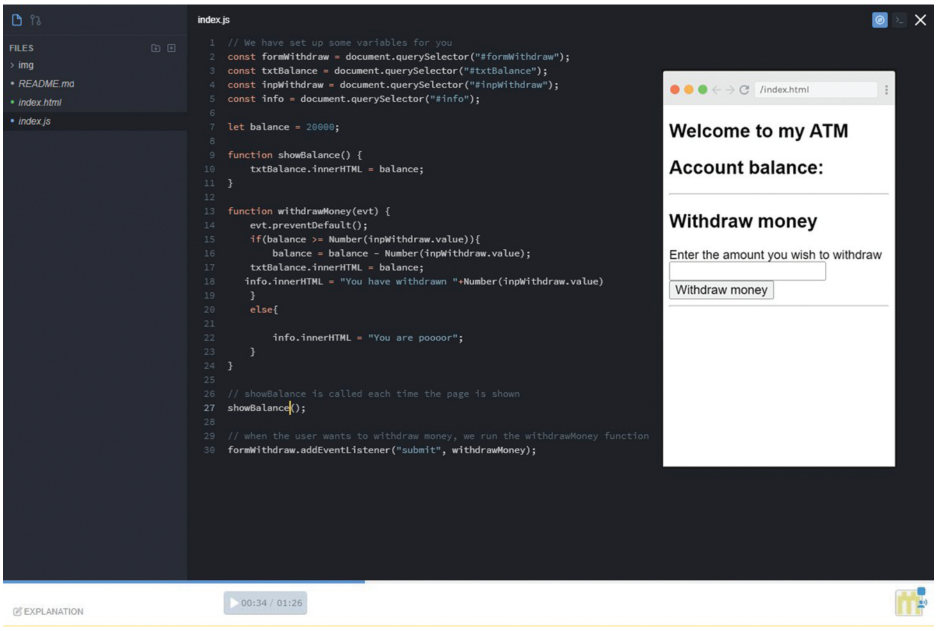
Interactions in the context of learning to program can take place by means of different mediating tools and between different types of actors. Tsan et al. (2018) found that there are great differences in students' dialogues while programming, and Jenkins (2017) found that dialogical approaches are important in programming classrooms. Brennan and Resnick (2012) include questioning as an important aspect of programming. In the view of Brennan and Resnick, questioning implies that students are curious and ask questions about the technologies they use in their lives, such as to "think about how anything is programmed" (p. 11), as opposed to the programming being part of the questioning process. Cutts et al. (2012) argue that natural language, computing talk, and code are all parts of the "language" in programming, suggesting (implicitly) that coding is a conversational process consisting of a language of several dimensions. Perrenet et al. (2005) include program execution as part of this language.

One example of ways computers become part of the programming conversation is through error messages. However, error messages sometimes cause frustration for beginner programmers, as the messages are rarely designed with the aim of assisting beginners (Kohn, 2019). Learners are at the mercy of the message designers and have little or no influence over what kind of information they can receive from predefined messages. Work on specialised error messages for learners is underway, with promising results (Hermans, 2020), but the messages seem currently to remain a representation of the voice of the coding environments' designers, more than being a conversational partner for the (novice) programmer.

From our perspective of empirical inquiry, there is potential to further the understanding of the various processes involved in learning to program, especially with the development of generative artificial intelligence systems although this is outside the scope of this paper.

2. Method

This qualitative study involved the participation of secondary school students (aged 15 to 19) enrolled in elective computer science classes. The paper is based on data from a larger design-based study (Barab & Squire, 2004), but the design-based aspects of the study are outside the scope of this paper. Instead, we frame this paper as a case study (Yin, 2018) where we focus on data from a single secondary school and iteration in the research project. The study centred on the use of a web-based code editor (Figure 1), emphasising its functionality for recording and playing screen recordings of hands-on coding in JavaScript. An important feature of the editor is the presence of the execution window, which makes code executions easily accessible as part of the programming process. We applied a purposeful sampling technique (Patton, 2002) where we focused on the students who worked on one of the several tasks we issued during the project (described below). The data concerning this task consist of 2 hours of video and 12 screen recordings (~2.5 hours). One teacher and nineteen students



```
// We have set up some variables for you
const formWithdraw = document.querySelector("#formWithdraw");
const txtBalance = document.querySelector("#txtBalance");
const inpWithdraw = document.querySelector("#inpWithdraw");
const info = document.querySelector("#info");

let balance = 20000;

function showBalance() {
    txtBalance.innerHTML = balance;
}

function withdrawMoney(evt) {
    evt.preventDefault();
    if(balance >= Number(inpWithdraw.value)){
        balance = balance - Number(inpWithdraw.value);
    txtBalance.innerHTML = balance;
   info.innerHTML = "You have withdrawn "+Number(inpWithdraw.value)
    }
    else{

        info.innerHTML = "You are poooor";
    }
}

// showBalance is called each time the page is shown
showBalance();

// when the user wants to withdraw money, we run the withdrawMoney function
formWithdraw.addEventListener("submit", withdrawMoney);
```

Welcome to my ATM

Account balance:

Withdraw money

Enter the amount you wish to withdraw

Figure 1. Screenshot of the code editor. Left: file directory; centre: code editor; right: output window; bottom: playback timeline for recording and/or viewing screencasts.

participated, some working alone, but most in pairs. We discarded approximately 20 minutes of screencast videos because of missing or bad quality audio. Data were collected in 2020 before CoViD lockdown.

We chose to focus on these data for two reasons. First, this data material included two full program development processes (two pairs of students' video recorded for one hour each), from reading the task description to finalising their self-produced screencast recordings (explained below), providing us with a full depth view of their programming processes. Second, the 12 screen recordings, where the students themselves presented their code, provided us with a breadth of different student approaches to the task, complementing the more in-depth, full process video material. Some students recorded explanation screencasts at the end of the class, others recorded themselves as they were programming. This gave us broader insights about approaches among most of the students in the class, even when we did not have the infrastructure available to support videotaping each student. The use of screencasts also has an ethical dimension, as more students consented to submit self-recorded screencasts of which they were themselves in control of, than those who consented to being videotaped in class throughout the class. Further ethical aspects of the study are described in section 2.4.

From this dataset, we display four extracts that serve as micro cases or examples, with the intention of showing the diversity of the approaches used (Yin, 2018). The extracts derive from the two pairs of students we videotaped (pair 1: extract 1, pair 2: extracts 2 and 3), and one single student who submitted a screencast (extract 4). We do not suggest that these extracts are representative of all students in introductory programming, but we are confident that they represent the diversity within our case.

2.1. Student task

The data selected for this paper were collected at a single upper secondary school where the students worked on a particular task of creating a virtual automated teller machine (ATM). The students could use any resources or strategies to solve the task, which was important for our research question of understanding the learning process.

The students we recorded were selected at random among the students in the class who had consented to participate, and we had no prior knowledge of their level of programming proficiency.

The ATM task comprised two parts. The first part involved developing a simple, virtual ATM in JavaScript. More specifically, the goal of the task was to develop a script that received input from the user (money withdrawn or deposited) and changed the user's fictional account balance based on this input. We provided the students with draft code from which they were to develop their program. The draft code included a complete HTML form to handle input from the user, a list of predefined JavaScript variables we considered useful, and "empty" functions we expected the students to create (e.g. showBalance and withdrawMoney). The purpose of the draft code was both to reduce the time the students spent writing HTML code, allowing them to spend more time on the JavaScript, and to assist them in designing the ATM code by providing some hints about expected functions and variables. The second part of the task concerned recording an audio-visual screencast (screen recording) where the students explained and demonstrated their codes, which they shared with the teacher for feedback and assessment purposes. We encouraged the teacher to let the students work in pairs as this facilitated collaboration and the use of natural language, which was central to our data collection, but some students requested – and were allowed to – work individually.

2.2. Analysis framework

As mediational means, such as talk and physical tools, are central to understand learning from our sociocultural learning perspective (Vygotsky, 1980), we developed a framework where we consider four aspects of the conversations in the final, in-depth analysis of the data: student talk, student action, code, and execution.

The first aspect we considered was the students' talk, analogous to Cutts et al. (2012) two dimensions of natural and computing language and the social plane of Sentance et al. (2019). Both papers argue for the role of language as central to programming. Other researchers have also emphasised the importance of talk and reflection as part of the learning process in computing (Brennan & Resnick, 2012; Zakaria et al., 2022).

The literature on physical programming activities (sometimes known as makerspaces) includes examples of considering students' embodied actions as part of the learning process (Kajamaa & Kumpulainen, 2020). We therefore included some contextual information, such as physical and digital gestures (actions), as part of the unit of analysis. Our main emphasis was on the use of the body (sometimes with the assistance of physical tools such as computers) to produce symbolic output (e.g. pointing, highlighting) that mediate the learning process, as described by Vygotsky (1980). The things people do may be labelled "actions", but we refer here to specific types of action that are not directly reliant upon oral talk or on writing code.

Third, we considered the code produced by the students, equivalent to the code level of Cutts et al. (2012) and the program level of Perrenet et al. (2005). Typically, the program dimension involves writing code – that is, it is traditionally framed as creating instructions for a computer, which are then executed (Denning & Tedre, 2021), and recording and analysing code changes are important for understanding student learning processes (Guenaga et al., 2021).

The execution of a program when a computer performs automated tasks correctly is typically considered the end goal of the programming process (Denning & Tedre, 2021), but it is also used as a tool for testing code in the search for technical bugs (McCauley et al., 2008). That is, code executions can be part of the programming process that we unpack in our analysis. In line with Perrenet et al. (2005), we refer to this as the execution dimension.V

2.3. Data analysis

In the analysis, we used the perspective of understanding programming as a conversation between a human and a computer, considering the four dimensions just presented. The aim of the analysis was to gain insight into the exchanges involved, including the computer as part of those conversations. In the first stage of analysis, we looked through all the selected data (video and screen recordings) and used thematic coding (Braun & Clarke, 2012) to create a "map" of the data.

After the mapping session, one hour of the most relevant screencasts and one hour of the most relevant video data were transcribed in detail for a more fine-grained interaction analysis (Jordan & Henderson, 1995). In the data extracts presented in the section 3, we use the notion of *events* to divide the unfolding activities into meaningful units (Derry et al., 2010). The extracts include the contributions of five students. For clarity purposes, the talk, text, variable, and function names in the extracts from the original data material were translated to English by the first author. The authors performed the analysis process individually before meeting and discussing the results and arriving at a shared understanding. This process was repeated several times over the course of two years. We arranged a data workshop where we invited external researchers (not affiliated with the study) to view the data, provide their interpretations, and (if relevant) dispute our interpretations, and an early version of this work was presented as a poster paper at an international conference (Litherland et al., 2021).

The analysis process led us to the development of several thematic codes, of which we found four relevant to answering the research question. Codes excluded from this paper include topics such as teacher interventions, the use of subject specific terminology, and software specific functionality (i.e. the recording, re-recording, and playback of screencasts within the code editor). In the results section, we present examples of the following four codes: edit and test loops, observation, problem isolation, and validation. Finally, in the discussion section, where we view our empirical data and codes in relation to former research, we synthesise our findings into five major themes: programming as inquiry, removing the error from trial and error, programming bound to context, error messages as an indirect conversation, and programs as means and ends.

The aim of the analysis process was not to decide the frequency of use of different programming approaches, but to identify and describe the diversity itself (Lemke, 2011).

As such, we do not provide a quantitative analysis, but we did provide context for the reader with short qualitative descriptions of whether we observed approaches more or less frequently.

2.4. Research ethics

All the participants (teachers and students) signed a consent form prior to taking part in the project and data collection. In addition, we applied ongoing consent, asking for permission immediately prior to starting all the recordings and informing the participants that we would stop recording any time at their request. The students who self-recorded screencasts did this without our intervention and were free to decide when to record and the content (within the border of the task). They were free to re-record or decide not to submit the screencast if they wished. The research project was registered with the Norwegian Centre for Research Data, an organisation that analyses research projects that handle human research data and ensures that such data are legally collected and handled. This work received support from the Regional Research Fund Viken under grant number 284,976. The authors report there are no competing interests to declare. The abbreviations used below to refer to individual participants were chosen at random, and the participants shown in snapshots from the video data are drawn as silhouettes to preserve anonymity.

3. Results

In this section, we present four selected episodes from our data material. The extracts serve as examples of ways students converse with their computers through programming, each representing a code we developed through the analysis process: 1) edit and test loops, 2) observation, 3) problem isolation, and 4) validation.

We present the extracts in tables of four columns. The first column holds the event and actor references. The second column includes talk. The third column contains additional information about actions or events from the videos/screen recordings. In the rightmost column of the data extracts, the program code is displayed as light text on a dark background, and the executed code is presented as dark text on a light background. We also included some snapshots from the recordings both within and outside the table extracts to show physical arrangements and actions of the participants, and many of the program and execution snapshots include representations of actions, such as cursor and caret placements. The student actors are marked using uppercase letters (e.g. B, G), and the computer "actors" are represented specifically as (C).

3.1. Extract 1: edit and test loops

Context: In the first extract (Table 1), two students (B and G) were working together on the ATM task. They were loosely following the structure of pair programming, sharing a laptop computer with which B performed most of the direct interactions. In general, G was the "co-pilot", but they changed roles at will. Figure 2 shows the physical arrangement of the actors.

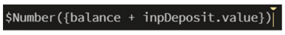
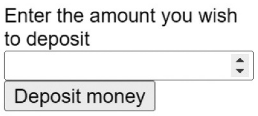
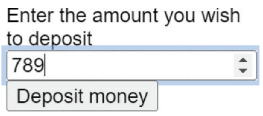
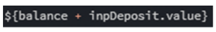
Prior to the episode below, the students tested their deposit function: a function that was supposed to 1) receive input from the user on how much money they were depositing into their fictional bank account and 2) add the input number to the balance variable. Instead of adding the test input 789 to the 20,000 balance (= 20,789), the program output a balance of 20,000,789 as the two values were concatenated. The students agreed that they needed the Number() function to solve the issue, which they correctly identified as a data types issue. Simply put, Number() makes values that are strings (data type) into numbers (data type). Applied to the correct value (inpDeposit.value), it would solve their issue.

This extract was chosen to represent the edit and test loop behaviour we observed in several students (often repeatedly, including the pair represented in the extract), referred to in the literature more broadly as trial and error (Moskal & Wass, 2019).

B asked G about the Number() function (event 1.1), but G replied (1.2) that she did not remember how to use it. In event 1.1, B added the Number() function using the balance and inputDeposit.value as parameters, commenting in event 1.3 that he would "just try something". This change resulted in the line of code being output as pure text (1.6), causing him to laugh (1.7). B deleted the changes (1.8). Later (not included in the transcript), the teacher helped them solve the problem by confirming that the program treated a number as a string, adding the numbers together in sequence. The teacher explained that the values from input fields are always strings. With this clarification, the students solved the issue by using the Number() function on the inpDeposit.value.

In this example, we saw how the students rapidly changed the code and ran it to see the result. The strategy did not solve the problem of the concatenated values. Although the students touched upon changes that were close to a possible solution, they were not

Table 1. Edit and test loops. Includes data previously presented in (Litherland et al., 2021).

Event (Actor)	Talk	Actions/comments	Program/execution
1.1 (B)	Okay. If we put Number() outside here then? How do you do that?	B moves the caret to where the variable balance is the output and types $Number() with the sum of the balance and inpDeposit.value as parameters.	 `$Number(${balance + inpDeposit.value})`
1.2 (G)	I don't remember any of the Number() stuff.		
1.3 (B)	I'll just try something.	Clicks the execution button.	
1.4 (C)		Displays the deposit form.	 Enter the amount you wish to deposit / Deposit money
1.5 (B)		Enters 789 into the field and clicks the Deposit money button.	 Enter the amount you wish to deposit / 789 / Deposit money
1.6 (C)		Displays the result of the deposit function.	$Number({balance + inpDeposit.value})
1.7 (B)	Yes, ha-ha, oops.	Sees the line of code is output as text.	
1.8 (B)		Removes the changes made in 1.1.	 `${balance + inpDeposit.value}`

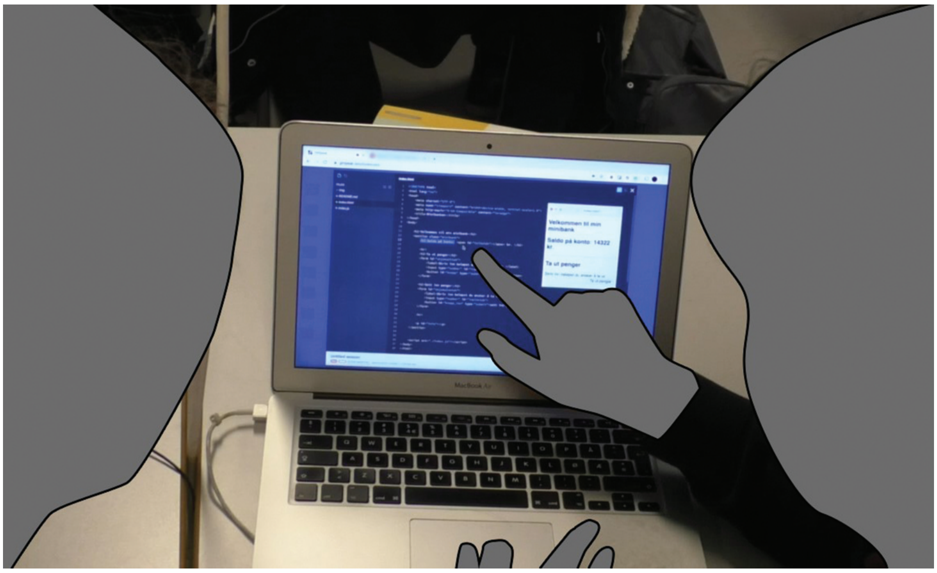


Figure 2. Students G (left) and B (right) focusing their attention on parts of the code using digital and physical gestures. The code is written in JavaScript.

able to identify these changes as meaningful or learn from them beyond them not providing the intended result. Their conceptual knowledge of data types and the Number() function helped them towards a solution, but they were not successful at applying this knowledge to the correct part of the code.

The students showed that they understood how the computer could serve as a form of "conversational" partner in the programming process, as they made changes and sought information from the program about how it would answer to those changes. However, they struggled with formulating questions that would provide them with answers to support the process further. The edits the students made were not "errors" or "mistakes", as asking questions does not imply failure.

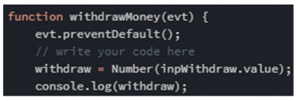
3.2. *Extract 2: observation*

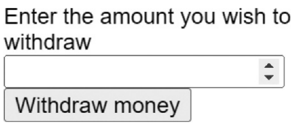
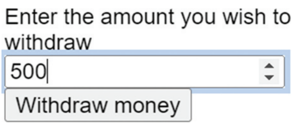
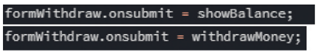
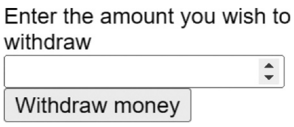
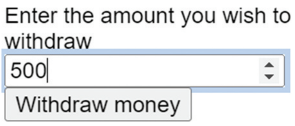
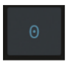
In the next extract (Table 2), students E and P were working on the withdrawal function of their ATM. E was in charge of the typing, while P sat next to him following the code development on his own laptop through the live screen sharing functionality in the code editor. P did not make any code changes through the screen sharing but was passively involved in the process by pointing his cursor, which was visible to E, and through E's many monologues, such as the one below. E experienced issues receiving input from the user and decided to add a console logging function to trace the withdraw variable value. Although E made some verbal utterances in the process, most of the important events consisted of direct interactions with the code and the execution of the program.

We chose this extract to represent the act of tracing values through program execution, a behaviour we found in several students.

Table 2. Observation.

Event (Actor)	Talk	Actions/comments	Program/execution

2.1 (E)		Adds the console.log function to the withdrawMoney function and clicks to execute the code.	 `function withdrawMoney(evt) {`
			`evt.preventDefault();`
			`// write your code here`
			`withdraw = Number(inpWithdraw.value);`
			`console.log(withdraw);`

2.2 (C)		Displays the withdrawal form.	 Enter the amount you wish to withdraw Withdraw money
2.3 (E)		Enters 500 and clicks the Withdraw money button.	 Enter the amount you wish to withdraw 500 Withdraw money
2.4 (C)		Nothing is logged to the console.	[no console log appears]
2.5 (E)	Oh, not even the console log is working now . . .	Observes that nothing is logged to the console. Executes the code several times, receiving the same result (no log). Scrolls through the code.	
2.6 (E)	Ah, it [missing log] is because I've written the wrong name here. Withdraw money . . .	Student points to and explains that the withdraw event calls the showBalance function. Corrects the function name and executes the code.	 `formWithdraw.onsubmit = showBalance;` `formWithdraw.onsubmit = withdrawMoney;`
2.7 (C)		Displays the withdrawal form.	 Enter the amount you wish to withdraw Withdraw money
2.8 (E)		Enters different inputs, such as 50 and 500.	 Enter the amount you wish to withdraw 500 Withdraw money
2.9 (C)		Displays 0 on the console.	 `0`
2.10 (E)	Zero, ok? So now it's zero.		

In event 2.1, E placed the console.log() call within the withdrawMoney function. E entered a test input (2.3) and realised that the form was not logging anything. In event 2.6, he explained that "it" (the problem of the missing log) was caused by an incorrect function name. The form referred to the showBalance function when it was supposed to point to the withdrawMoney function. After changing the function name (2.6) and testing the program with various inputs (2.7–2.8), the console log returned 0 (2.9), meaning that the console log code was run but did not receive the correct values from the user. Although not included in the transcript, the student made frequent small changes to the code to try to fix the logging but did not succeed, in a similar manner to that of students G and B in the previous extract.

Student E actively supports the conversation process by making explicit the computer responses using the console. Not only does this imply the recognition of the computer "voice", but also an active choice to make it heard. The aim of the task is not to display values in the console. Instead, the console provides the student with empirical/observable information about the program, which he uses to support the continued programming process. The student practiced conversation with the situation, as he saw that there was something the program could tell him that would help him in solving the problem. We argue that framing this as "trial and error" greatly undermines the sophistication of the process, as the student takes an active role in learning about his own code, using empirical insights to continue the development process.

3.3. Extract 3: problem isolation

The next extract (Table 3) took place about one minute after the previous extract, and E was still struggling to show the log. Running the code, he got an error message and saw that the account balance was not printed in the output window as it had been before. In fact, parts of the HTML form disappeared entirely, implying that some of his small changes had serious consequences. P watched E working.

This extract was chosen to represent the process of temporarily decomposing programs by deletion or commenting, which we observed in some variations among some of the students.

P decided to ask E about the problem. E explained that the balance was not the output (3.3) after his recent attempts at printing the withdrawal value. E decided to highlight and delete three lines of code, which he suspected were the cause of the balance not being the output (3.3). After deleting the code, he executed the program, and the balance was returned to the output window (3.4). He undid the code deletion, having confirmed his hypothesis that the error lay in one of the three code lines he had deleted (3.6). P confirmed the same. P then pointed to the second line that had been deleted and re-entered (3.10). E noticed the misplaced backtick, told P to stop talking, removed the backtick, and ran the code, observing the account balance printed in the output window (3.13) and his input of 50 correctly logged to the console (3.14).

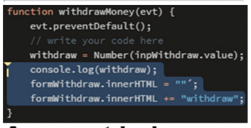
The significance of this extract lies on the way student E gained empirical information about the program by actively deleting parts of it temporarily. This extract shows how student E tried to gain access to the hidden mechanisms that were at play in the program he was creating, employing help from both his partner (student P) and using the program itself as a tool.

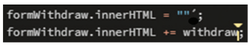
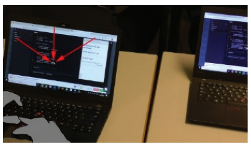
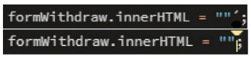
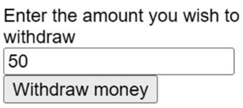
3.4. Extract 4: validation

The following extract (Table 4) is from a self-recorded screencast, where a student (M) recorded his own screen and voice while programming to explain some of his thoughts and actions while developing the ATM. The extract took place 11 minutes into the screencast. Prior to the events described below, the student spent his time continuously writing code without executing it. We included this extract to serve as an example of a student who runs their code sparingly.

Table 3. Problem isolation.

Event (Actor)	Talk	Actions/comments	Program/execution
3.1 (P)	Okay, what is the problem?		
3.2 (C)		Shows no account balance value.	**Account balance:**

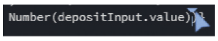
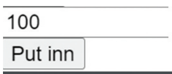
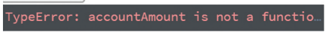
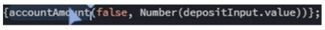
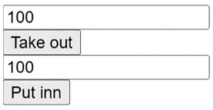
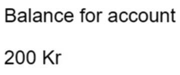
3.3 (E)	Right now, it won't show the balance.	Highlights the code and deletes it. Clicks the button to execute the code.	 `function withdrawMoney(evt) {`
			`    evt.preventDefault();`
			`    // write your code here`
			`    withdraw = Number(inpWithdraw.value);`
			`    console.log(withdraw);`
			`    formWithdraw.innerHTML = ""´;`
			`    formWithdraw.innerHTML += "withdraw";`
			`}`

3.4 (C)		Shows a balance of 20,000.	**Account balance: 20000**
3.5 (P)	But now it shows the balance.	Observes that the balance is shown.	
3.6 (E)		Adds the code back in, and both students look for a potential error. E removes incorrect quotation marks around the withdraw variable reference.	 `formWithdraw.innerHTML = ""´;` `formWithdraw.innerHTML += withdraw;`
3.7 (P)	But . . . You have . . .		
3.8 (E)		Executes the code.	
3.9 (C)		Still does not produce an account balance.	**Account balance:**
3.10 (P)	You know, here.	P points his pointer (left arrow) to the end of the second line that was temporarily deleted, which contains a misplaced backtick. E (with his hands on his own laptop) points to the same area with his own pointer (right arrow) and places the caret (middle arrow) on the line.	
3.11 (E)	Oh yeah, no . . . Stop, stop, stop.	Sees the misplaced backtick P pointed to and tells him to stop pointing and/or talking. Removes the backtick. Clicks the button to execute the code.	 `formWithdraw.innerHTML = ""´;` `formWithdraw.innerHTML = "";`
3.12 (C)		Shows the balance (and the withdrawal form).	**Account balance: 20000**
3.13 (E)	There, okay. 50.	Observes that the balance is shown. Enters an input of 50.	 Enter the amount you wish to withdraw 50 Withdraw money
3.14 (C)		Displays the value 50 in the console log and a corresponding 50 in the transaction log.	50 (console log) **Withdraw money**
3.15 (E)	Okay, so it works.		

This extract was chosen to represent the actions of the many students who used program execution as a final task to validate the accuracy of their programs.

In event 4.1, the student explained that it was time to check the code for what he called "formatting errors". The JavaScript code consisted of about 30 lines at this time. He pointed both his cursor and caret to the same line of code, indicating that he was reading

Table 4. Validation.

Event (Actor)	Talk	Actions/comments	Program/execution
4.1 (M)	Now we need to check whether there are any formatting errors.	Cursor and caret points to the same line of code.	 `Number(depositInput.value)}`;
4.2 (M)	No, there are none.	Confirms there are no errors in the line of code in focus.	
4.3 (M)	Okay, so now it's supposed to work.		
4.4 (M)	Deposit 100 Norwegian kroner.	Tries to deposit 100.	 100 Put inn
4.5 (C)			 `TypeError: accountAmount is not a functio…`
4.6 (M)	Account amount is not a function . . .	Reads the message on the screen	
4.7	Hmm . . . account update, right. Here . . .	Updates the function name from accountAmount to accountUpdate.	 `{accountAmount(false, Number(depositInput.value))};`
4.8 (M)	100 Norwegian kroner . . . Nice . . . and take out 100 kroner.	Deposits 100 several times and withdraws 100 a few times.	 100 Take out 100 Put inn
(C)		The account balance is updated according to the input.	 Balance for account 200 Kr
	There, yes. Okay. So, it works.		

this line. Upon reading this, he claimed the code would work (4.3). M inserted a test input into the deposit form (4.4), and the program returned an error message that M read aloud. The message prompted him to check the names of the functions, resulting in the renaming of one. He ran the code again, entering an amount to deposit, and confirmed that the form now worked.

We would like to highlight several parts of this process. First, although not explicitly part of the extract, it is important to note that this student (and several others) rarely employed running code as a development or learning method. As expressed by M, the code was first run when it was "supposed to work" (4.3). M employed the same pattern when he later developed a transaction log for the ATM (an optional "bonus" task). Second, we point to the student's use of the error message. The student understood the English error message, which directed him to investigate specific parts of the code, suggesting that the process was a form of "conversation" (Schön, 1992).

4. Discussion

In this section, we will present discuss and attempt to generalise our findings by comparing them with the findings and conceptual distinction reported in the literature. The discussion is divided into separate subsections, each of which refers to a particular finding. All the findings relate to the research question we proposed in section one: *How can* *empirical inquiry serve as a perspective for analyzing the activity of students learning to program individually and in pair programming?* The extracts are activated and discussed across the findings.

An overall finding is that all the students solved the ATM task adequately (or better). While their final products differed in structure and, to some degree, in functionality, their processes were more divergent. This is an important preliminary finding on which we base the further discussion.

4.1. Programming as inquiry

One way to represent the various processes is to explicitly frame how the students and converse with their programs through inquiry. Based on our empirical observations, we find that this process takes place through four phases: 1) formulate a question, 2) ask the question, 3) receive an answer, and 4) evaluate and give a response (formulate a new question, i.e. return to phase 1). The questions, answers, and responses are interpretations of human and computer actions, and is inspired byNewell and Simon's (1976) ideas of computer science as engaging in a conversation with a machine, in an analytical process.

We define the first step as the formulation of a question the student wants answered. The student converse with the code in different ways, making modifications, such as adding, removing, editing, or moving code. Each change in the program is a new construction of a question, and the student may run the program to "ask the question" and observe the result. Sometimes, this step requires the student to add a test input, that is, additional information needed to answer the question. The asking of the question produces an output (answer), which the student may observe, reflect on, and respond to or ignore.

In Table 5, we present an overview of some of the events in the extracts conceptualised as a process of inquiry.

We identified two main types of questions: 1) confirmation and 2) information gathering. Student B asked confirmation-type questions, of which we provide one example. The goal of confirmation questions is to confirm whether code changes are "correct" or "incorrect" in relation to the problem at hand. In computing terminology, the answers are Boolean (true or false), and possible responses involve either keeping the code changes (when the answer is "correct") or undoing them (when the answer is "incorrect"). One characteristic of a confirmation question is that the answers do not lead to code changes beyond the question itself, contrary to information-gathering questions. Since student B's question in event 1.5 received a negative answer – which the student evaluated as an "oops" (1.7) – the student responded by rejecting the code change.

In the last extract, student M also used the execution of the program as a confirmation process, but in a different manner. While he was not concerned with gathering information about how to fix an issue in the program, the purpose of the execution was to confirm that the program worked as he expected. He received an answer from the computer in the form of an error message, which made him evaluate a specific function name. This is therefore a different type of confirmation, where the student made an explicit prediction about what would happen when the program was executed. As the predictions were not true, the student realised changes had to be made.

Table 5. Programming as inquiry.

Extract	Question (our interpretation)	Type of question	Answer	Type of answer	Evaluation	Type of evaluation	Response	Type of response
1	Does the computer add the numbers correctly when the sum of the balance and inpDeposit.value are made into a number?	Confirmation	No	Boolean	Does not add the numbers correctly	Confirmation	Remove the change	Roll back
2	What is the value of the withdrawal when logged to the console?	Information gathering	The value is not an output in the console	Qualitative	The console log is never called	Problem identification	Correct the function name	Fix bug
2	What is the value of the withdrawal when logged to the console?	Information gathering	0	Qualitative	Nothing is logged	Problem identification	Make small changes to the variable names	Trial and error
3	Is there an error somewhere in these particular three lines of code causing my HTML form to disappear?	Information gathering	Yes	Boolean	Search for errors in these three lines	Problem isolation	Undo the deletion and study the three lines of code	Focusing attention
4	Does this function work?	Confirmation	Error message	Mediated	Check the function name	Problem identification	Correct the function name	Fix bug

Student E asked information-gathering questions. The purpose of these information-gathering questions was not to produce "correct" code but to investigate the code itself. The answer to the information-gathering question in event 3.3 was Boolean (like those of the confirmation questions), but the response was different. The answer ("correct") prompted the student to undo the code change, which represents an opposite response to that of positive answers to confirmation questions, which always involve keeping the code changes. Student E got different answers to the same question asked in events 2.3 and 2.8. The answers were not Boolean but were of different qualities. The first answer, which, in a sense, was no answer at all (nothing was logged in the console), provided E with information. The missing log represented a meaningful answer that prompted the student to look for and fix an error in a different part of the code. In event 2.8, however, the student was not able to interpret and evaluate the answer given by the computer, which caused him to enter a process of trial and error. The frequent changes differed between being both useful and destructive, and the student was not able to distinguish between them.

However, in the case of E, we saw that code changes were not only instructions for the computer but also served an information seeking purpose for the student and were not intended as part of the final instructions for the computer to follow. In professional settings where the goal of programming is a digital product, temporary code changes do not represent the same inherent value. Views of programming that emphasise formulating instructions for a computer (Denning & Tedre, 2021) do not capture the aspect of the development process where programs are a temporary learning tool at a specific point in time. Neither do approaches to research or assessment that rely on the analysis of finished programs (Moreno-León et al., 2015), where such temporary code changes are no longer present. The students we observed only occasionally made explicit predictions (one example is found in event 4.3: "now it's supposed to work".), but it is possible that predictions or hypotheses were made without being expressed. We find that the students participated in a "low level" empirical inquiry, which is not a "capital S" Scientific process, but an inquiry-as-learning process involving investigations of gathering information and testing propositions of various degrees of refinement.

4.2. Removing the error from trial and error

Trial and error is a well-known strategy among novice programmers (Moskal & Wass, 2019). Using an empirical inquiry perspective allows us to divide trial and error into several different, but related, processes based on the types of trials performed, the context of the trials, and how the students interpret the results. Through empirical inquiry, some trial and error-like activities can instead be framed as information seeking activities, as opposed to a trial activity or (even more negatively framed) an "error making activity".

For instance, we frame extract 1 as an "edit and test loop" and avoid negative terms, such as "error". From our perspective, the outputs students B and G received from the computer were not "errors" but valid information about the state of the program. Rather than performing random activities, they expressed ideas about what caused the concatenation issue and applied these ideas to various relevant parts of the code. Similarly, student E's temporary code changes do not represent "errors", but a process of exploring the program.

4.3. *Context bound programming*

The process we exemplify in extract 2, where the student gives the computer a voice by providing it with the means to "answer" through the console, may be understood as a two-way conversation, where the student must provide information (e.g. which values should be tracked when and where), receive information (e.g. understand which values are returned when), and know how to act upon this information. The same applies to extract 3. While we do not have access to the student's perceived expectations, we derive from his actions that the aim of running the code after deleting several lines of it was not to see a finished, functional program. The aim was to receive information about the program in a particular and temporary state (runtime situation).

Throughout our dataset, we found examples of students using temporary code changes as a coding strategy, such as commenting out or removing code to isolate problems (extract 3) and printing/logging variables (extract 2). Temporary code changes may be counter-intuitive for a novice programmer. For instance, some temporary code changes, such as commenting out code, may make the code less functional. From a dialogic perspective, however, these actions are meaningfully bound to the specific points in time and context. Viewing programming as a "conversation with the situation" (Schön, 1992) may open up a conceptual space for students, where they can have these forms of momentary conversations with their programs that make sense in a way that is lost in a pure end-product-based perspective. Two channels of exchange, one towards humans and the other towards the computer's runtime environment extend the symbolic metaphor of communication proposed by Newell and Simon (1976) toward symbols and materials (Schön, 1992).

4.4. *Error messages as an indirect relationship*

Student M received and reacted to a particular type of information about the program, namely the error message received in event 4.5. The error message assisted the student in completing the subtask of designing the deposit function, which malfunctioned because of a naming error.

Although error messages may be important for the development process, as exemplified by student M, the messages can also be considered a mediated relationship between the student and their program. The developers of code editors and/or programming languages curate error messages. Thus, the potential of this type of relationship relies on external actors. Error messages typically focus on syntactic errors, but as demonstrated in extracts 1 and 2, there are many cases where students can benefit from information from the computer without the presence of syntactic errors.

4.5. *Programs as means and ends*

To various degrees, students allow their computers to take on the role of a source in the programming process (Furberg & Silseth, 2022), and this is especially clear in extracts 2 and 3. Brennan and Resnick's (2012) perspective of questioning "how things work" is different to our perspective of questioning in empirical inquiry; programs are not problems to be solved or figured out (a perspective with a stronger presence in students such as student M), but sources of information as part of the learning process. Questioning therefore becomes a process of conversing with programs in ways that provide valuable insights. Instead of asking questions such as "How is this artefact programmed?" in introductory programming in schools, we may shift to a focus on "What new insights can we make from interacting with this program?" These types of questions can be seen from a bricolage perspective (Turkle & Papert, 1990), where a painter does not necessarily ask how a piece of art is to be created, but instead ask what will happen when they start adding layers of paint on a canvas.

While students B and G expressed knowledge of the Number() function and demonstrated a willingness to explore the program empirically, they were not able to link the semantics of this specific function to the particular situation they were in. Nevertheless, the situation and activities they took part in may still have provided meaningful learning outcomes, both concerning the programming process itself and about specific functions.

We suggest treating programs as both means and ends. From a sociocultural learning perspective, the program – while still a goal – may also function as a mediating artefact that supports learning. We argue that this may represent a possible explanation for why many students experience programming as difficult (Luxton-Reilly et al., 2018), as learning how to program may rely on skills concerning how to "converse" with programs. According to Dewey, the inquiry process does not lead to a final point of resolution, but to new problems. Focus on the final program and concepts of programming (Robins, 2019; Xie et al., 2019) may be important for the skill of programming, yet this study seeks to extend the value of programming activities into a more comprehensive learning process. Papert (1980) emphasised programming as a learning method – not a learning outcome. To him, programming in schools was a means of learning about maths and physics, not an end. We ask, why not both?

5. Conclusions

The perspective of empirical inquiry focuses on the *process* of programming. We identified various patterns of programming that show how students display different ways of interacting with their programs. Through activities of empirical inquiry, programming becomes a process of learning with the aid of a computer and by acting, observing, and analysing, extending the idea of programming as producing expected results based on tasks provided by the teacher. Students display various approaches to these activities. This has implications for our understanding of common strategies, such as trial-and-error and assessment. We argue there may be a need to explore how programming practices can be made an explicit part of curricula by treating them with the same care as domain specific concepts, such as variables and loops.

We studied how the relationship between students and their programs is developed through the interplay of multiple interactional levels: the students' talk, actions, the program code, and the executed programs. For instance, the students' completed programs do not provide a comprehensive picture of their learning processes, which may be most important. We suggest that treating programming processes as a complex system may extend our understanding of programming as a learning process, which we have tried to demonstrate by example in this article.

A general view of programming in context may be translated into a micro perspective of each student's own programs; there is potential to encourage students to consider how their programs play a part in the further program development. The "conversation" between human and machine goes beyond "telling the computer what to do", as it requires back-and-forth interaction of both passive and active exchange. A programmer "talks" to computers but must be susceptible to what the computer returns.

5.1. Limitations and directions for future work

As the students participating in this study were part of elective courses in computing, they do not represent the broader population of students now expected to learn programming in schools as part of the movement started by Wing (2006). Our sample may display different attitudes towards learning to program than students in general. However, we do not suggest that this qualitative study is generalisable to this population. We wish to make clear that the various inquiry patterns we present and discuss are suggestive, and more work is needed before claims can be made regarding the spread and frequency of inquiry types, best practices, or other normative conclusions.

We suggest continuing the effort to study programming as a process of inquiry in more diverse empirical settings, for instance, by employing learning analytics tools and other quantitative measures of student conversation, and combining such quantitative methods with qualitative approaches (so-called mixed methods) to retain the student voice. With the advance of physical computing in schools (Kajamaa & Kumpulainen, 2020), there is a need to further our understanding of the role of physical objects and space in such interaction. Furthermore, our observation methods did not make possible the empirical study of students' thoughts about what they were doing beyond what they themselves found natural to share in the moment. Methods such as think-aloud protocols (Fonteyn et al., 1993) could be one way to address this, or by the use of more structured pedagogic approaches such as PRIMM (Sentance et al., 2019).

On a conceptual level, we suggest that more work is needed on viewing our findings in relation to the contemporary discussions in the literature on computational thinking (e.g. Denning and Tedre (2021)). Furthermore, while our work focused on the "conversation" between individual students and their programs, there is a need for more work that considers the complicated inquiry patterns that involve several people or computers, for instance with the development of so-called chatbots and AI (OpenAI, 2022).

We argue for the need for strategies to take advantage of the opportunities and address the challenges of using the perspective of empirical inquiry within the field of practice. One direction could be to investigate how an empirical inquiry perspective applies as a frame of reference for teaching and/or learning to program. This may raise questions of how such a frame may influence pedagogical concerns, such as the design of learning activities, goals, and assessments, including how to develop tasks and teaching strategies that support students in utilising computers as a partner in the programming process.

Acknowledgments

The authors would like to thank Anders Mørch for feedback throughout the writing of this paper and all the participating students and teachers.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by the Regional Research Fund Viken [284976].

Notes on contributors

Kristina Litherland (she/her) is a doctoral research fellow at the Department of Education, University of Oslo. Her research interests include computer science education in various contexts, both formal and informal. She has worked for several years in web development, holding a bachelor's degree in information technology and innovation, and has worked as a teacher educator at Oslo Metropolitan University. She holds a master's degree in pedagogy from the University of Oslo.

Anders Kluge (he/him) is a researcher in technology-enhanced learning at the Department of Education, University of Oslo. His research is directed towards how technology can be designed and used to stimulate productive learning processes. In particular, he is engaged in how to strike the balance between user activity and theory presentation in rich digital representations and co-creative processes. He holds a PhD from the Department of Informatics, University of Oslo.

ORCID

Kristina Litherland http://orcid.org/0000-0001-9694-1291

References

Allsop, Y. (2019). Assessing computational thinking process using a multiple evaluation approach. *International Journal of Child-Computer Interaction*, *19*, 30–55. https://doi.org/10.1016/j.ijcci.2018.10.004

Andersen, R., Mørch, A. I., & Litherland, K. T. (2022). Collaborative learning with block-based programming: Investigating human-centered artificial intelligence in education. *Behaviour & Information Technology*, *41*(9), 1830–1847. https://doi.org/10.1080/0144929X.2022.2083981

Arnseth, H. C., & Ludvigsen, S. (2006). Approaching institutional contexts: Systemic versus dialogic research in CSCL. *International Journal of Computer-Supported Collaborative Learning*, *1*(2), 167–185. https://doi.org/10.1007/s11412-006-8874-3

Barab, S., & Squire, K. (2004). Design-based research: Putting a stake in the ground. *Journal of the Learning Sciences*, *13*(1), 1–14. https://doi.org/10.1207/s15327809jls1301_1

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology*, *13*(1), 20–29.

Bjerknes, G., Bratteteig, T., & Espeseth, T. (1991). Evolution of finished computer systems. *Scandinavian Journal of Information Systems*, *3*(1), 25–45.

Bocconi, S., Chioccariello, A., Kampylis, P., Dagienė, V., Wastiau, P., Engelhardt, K. & Stupurienė, G. (2022). *Reviewing computational thinking in compulsory education: State of play and practices from computing education*. Publications Office of the European Union.

Braun, V., & Clarke, V. (2012). Thematic analysis. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), *APA handbooks in psychology®. APA handbook of research methods in psychology* (Vol. 2, pp. 57–71). American Psychological Association.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and Assessing the development of computational thinking. Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vol. 1, Vancouver, 13-17 April 2012, 25 p. http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf

Cutts, Q., Esper, S., Fecho, M., Foster, S. R., & Simon, B. (2012). The abstraction transition taxonomy: Developing desired learning outcomes through the lens of situated cognition. In Proceedings of the ninth annual international conference on International computing education research (ICER '12). Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/2361276.2361290

Dahn, M., & DeLiema, D. (2020). Dynamics of emotion, problem solving, and identity: Portraits of three girl coders. *Computer Science Education*, *30*(3), 362–389. https://doi.org/10.1080/08993408.2020.1805286

Denning, P. J., & Tedre, M. (2021). Computational thinking: A disciplinary perspective. *Informatics in Education*, *20*(3), 361–390. https://doi.org/10.15388/infedu.2021.21

Derry, S. J., Pea, R. D., Barron, B., Engle, R. A., Erickson, F., Goldman, R., Hall, R., Koschmann, T., Lemke, J. L., Sherin, M. G., & Sherin, B. L. (2010). Conducting video research in the learning sciences: Guidance on selection, analysis, technology, and ethics. *Journal of the Learning Sciences*, *19*(1), 3–53. https://doi.org/10.1080/10508400903452884

Dewey, J. (1938). *Logic: The theory of inquiry*. Henry Holt and Company.

Du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice Programmer* (pp. 283–300). Psychology Press.

Fonteyn, M. E., Kuipers, B., & Grobe, S. J. (1993). A description of think aloud method and protocol analysis. *Qualitative Health Research*, *3*(4), 430–441. https://doi.org/10.1177/104973239300300403

Furberg, A., & Silseth, K. (2022). Invoking student resources in whole-class conversations in science education: A sociocultural perspective. *Journal of the Learning Sciences*, *31*(2), 278–316. https://doi.org/10.1080/10508406.2021.1954521

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, *25*(2), 199–237. https://doi.org/10.1080/08993408.2015.1033142

Guenaga, M., Eguíluz, A., Garaizar, P., & Gibaja, J. (2021). How do students develop computational thinking? Assessing early programmers in a maze-based online game. *Computer Science Education*, *31*(2), 259–289. https://doi.org/10.1080/08993408.2021.1903248

Hao, Q., Smith Iv, D. H., Ding, L., Ko, A., Ottaway, C., Wilson, J., Arakawa, K. H., Turcan, A., Poehlman, T., & Greer, T. (2022). Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education*, *32*(1), 105–127. https://doi.org/10.1080/08993408.2020.1860408

Hermans, F. (2020). Hedy: A gradual language for programming Education. In Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20). Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262

Jenkins, C. W. (2017). Classroom talk and computational thinking. *International Journal of Computer Science Education in Schools*, *1*(4), 3–13. https://doi.org/10.21585/ijcses.v1i4.15

Jordan, B., & Henderson, A. (1995). Interaction analysis: Foundations and practice. *Journal of the Learning Sciences*, *4*(1), 39–103. https://doi.org/10.1207/s15327809jls0401_2

Kajamaa, A., & Kumpulainen, K. (2020). Students' multimodal knowledge practices in a makerspace learning environment. *International Journal of Computer-Supported Collaborative Learning*, *15*(4), 411–444. https://doi.org/10.1007/s11412-020-09337-z

Kohn, T. (2019). The Error Behind The Message: Finding the Cause of Error Messages in Python. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). Association for Computing Machinery, New York, NY, USA, 524–530. https://doi.org/10.1145/3287324.3287381

Lemke, J. L. (2011). Analyzing verbal data: Principles, methods, and problems. In B. Fraser, K. Tobin, & C. McRobbie (Eds.), *Second international handbook of science education* (Vol. 24). Springer International Handbooks of Education. https://doi.org/10.1007/978-1-4020-9041-7_94.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, *38*(3), 118–122. https://doi.org/10.1145/1140123.1140157

Litherland, K., Kluge, A., & Mørch, A. I. (2021, September). Interactive screencasts as learning tools in introductory programming. *In European Conference on Technology Enhanced Learning* (pp. 342–346). Cham: Springer International Publishing.

Liu, Z., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, *27*(1), 1–29. https://doi.org/10.1080/08993408.2017.1308651

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., & Szabo, C. (2018). Introductory programming: A systematic literature review. In Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion). Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, *41*, 51–61. https://doi.org/10.1016/j.chb.2014.09.012

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Moreno-León, J., Robles, G. & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, (46), 1–23.

Moskal, A. C. M., & Wass, R. (2019). Interpersonal process recall: A novel approach to illuminating students' software development processes. *Computer Science Education*, *29*(1), 5–22. https://doi.org/10.1080/08993408.2018.1542190

Muller, O., Ginat, D., & Haberman, B. (2007, June). Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, Dundee, Scotland (pp. 151–155).

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, *19*(3), 113–126. https://doi.org/10.1145/360018.360022

OpenAI. (2022). *ChatGpt: Optimizing Language Models for Dialogue*. https://openai.com/blog/chatgpt/

Papert, S. A. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic books.

Parsons, D., & Haden, P. (2006, January). Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education*, Hobart, Australia (Vol. 52, pp. 157–163).

Patton, M. Q. (2002). Two decades of developments in qualitative inquiry: A personal, experiential perspective. *Qualitative Social Work*, *1*(3), 261–283. https://doi.org/10.1177/1473325002001003636

Pea, R. D. (1986). Language-Independent conceptual "Bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Perrenet, J., Groote, J. F., & Kaasenbrood, E. (2005). Exploring students' understanding of the concept of algorithm: Levels of abstraction. *ACM SIGCSE Bulletin*, *37*(3), 64–68. https://doi.org/10.1145/1151954.1067467

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779

Robins, A. (2019). Novice programmers and introductory programming. In S. Fincher & A. Robins (Eds.), *The Cambridge Handbook of Computing Education Research (Cambridge Handbooks in psychology* (pp. 327–376). Cambridge University Press. https://doi.org/10.1017/9781108654555.013

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Schön, D. A. (1992). The theory of inquiry: Dewey's legacy to Education. *Curriculum Inquiry*, *22*(2), 119–139. https://doi.org/10.2307/1180029

Schulte, C. (2008). 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*, Sydney, Australia (149–160).

Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education*, *29*(2–3), 136–176. https://doi.org/10.1080/08993408.2019.1608781

Simon, H. A. (2019). *The Sciences of the artificial, reissue of the third edition with a new introduction by John Laird*. MIT press.

Soloway, E. (1986). Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. https://doi.org/10.1145/6592.6594

Tsan, J., Lynch, C. F., & Boyer, K. E. (2018). "Alright, what do we need?": A study of young coders' collaborative dialogue. *International Journal of Child-Computer Interaction*, *17*, 61–71. https://doi.org/10.1016/j.ijcci.2018.03.001

Turkle, S., & Papert, S. (1990). Epistemological pluralism: Styles and voices within the computer culture. *Signs: Journal of Women in Culture and Society*, *16*(1), 128–157. https://doi.org/10.1086/494648

von Hausswolff, K. (2021). Practical thinking while learning to program – novices' experiences and hands-on encounters. *Computer Science Education*, *32*(1), 128–152. https://doi.org/10.1080/08993408.2021.1953295

Vygotsky, L. S. (1980). *Mind in society: The development of higher psychological processes*. Harvard university press.

Weintrop, D., Wise Rutstein, D., Bienkowski, M., & McGee, S. (2021). Assessing computational thinking: An overview of the field. *Computer Science Education*, *31*(2), 113–116. https://doi.org/10.1080/08993408.2021.1918380

Wertsch, J. V. (1998). *Mind as action*. Oxford university press.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*(2–3), 205–253. https://doi.org/10.1080/08993408.2019.1565235

Yin, R. K. (2018). *Case study research and applications: Design and methods* (6th ed.). Sage.Zakaria.

Zakaria Z, Vandenberg J, Tsan J, Boulden D Cadieux, Lynch C F, Boyer K Elizabeth and Wiebe E N. (2022). Two-Computer Pair Programming: Exploring a Feedback Intervention to improve Collaborative Talk in Elementary Students. Computer Science Education, 32(1), 3–29. 10.1080/08993408.2021.1877987