Novices' Motivations and Strategies for Using Code Examples in Open-Ended Programming

ACM Reference Format:
. 2021. Novices' Motivations and Strategies for Using Code Examples in Open-Ended Programming. 1, 1 (July 2021), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code examples are one of the primary sources of information that programmers of all skill levels use to acquire programming knowledge and learn language usage patterns [2, 6, 29, 39, 46]. In particular, novice programmers stand to benefit from programming examples, which can introduce new programming concepts [38, 44, 51, 54], and scaffold users to create more complex and interesting programs [24]. However, prior work on systems that support novices' example use have identified a variety of barriers encountered by students, such as difficulties to *understand* the example code, to *integrate* the example code to their own code, and to *modify* the example towards their own goals [13, 25, 54].

These barriers raise questions about how systems can more effectively support novices' example use. To do so, it is important to understand situations in which novices are asking for and using examples. Specifically, we aim to investigate students' **motivations** for using examples, as effective support systems must directly address these motivations [20]. For example, a student who is using examples to *implement a feature* may need very different support from a student using examples to *verify their work* or *generate ideas*. Additionally, we investigate students example use **strategies** because systems should encourage effective strategies, and discourage less effective ones [26].

In this work, we ask the research question: **What are novices' motivations and strategies for using examples when creating open-ended programming projects?** We choose to focus on open-ended projects, because these projects attract students of varying interests by allowing them to pursue goals that feel meaningful to them [21], and are therefore widely used in many introductory programming curricula [13, 15, 18, 36] as well as after-school, informal learning settings [43]. However, students are also known to face a number of barriers to incorporate challenging new programming patterns and APIs in open-ended programming [18], which code examples that demonstrate such knowledge may help to overcome [13, 24].

We conducted our study with 24 high school novice students as they created open-ended programming projects. While making these projects, students were able to search, browse, view and copy code examples from a system called Example Aid[1] [13], an example support system designed for open-ended programming in Snap*!* [37]. We analyzed video, interview, logs, and project submissions, identifying 5 distinct motivations and 4 key strategies that students employ when using examples. We also found that students almost always use *some* strategy, but that when they instead simply copy the example without modification, this rarely leads to successful integration. Students also reported

[1]System name changed for anonymity.

examples being helpful for their performance on current and future tasks, which is supported by student outcomes in our study. Based on these findings, we then propose a set of design recommendations to facilitate students' learning through creative design and planning, active code reconstruction, and comparison-based knowledge integration. Our key contributions are:

- An analysis of novices' motivations and strategies when using code examples in open-ended programming.
- Recommendations of design opportunities for systems to incentivize effective learning from active use of examples.

2 RELATED WORK

2.1 Open-Ended Programming

Much prior work on novices' example use focus on supporting students to complete *closed-ended* programming tasks [38, 44, 51, 54, 58]. By contrast, in open-ended projects students are encouraged to choose their own goals and to pursue projects that feel meaningful to them, such as making apps, games, and simulations [18]. These open-ended, choice-driven projects engage students by allowing them to create projects that connect to their own personal interest [21], and are popular among many introductory programming curricula [13, 15, 18, 36]. However, prior work has shown that students encounter a number of barriers during open-ended programming, such as difficulties to apply knowledge of programming concepts into code implementation [18]; and difficulties to understand and use unfamiliar APIs [13]. These challenges could be effectively addressed by code examples, which can demonstrate concepts and API use [13], suggesting the need for better understandings on novices' example use during open-ended programming. We do so by investigating *why* and *how* students use code examples, discussed below.

2.2 Why do novices ask for examples?

There has been little prior work on novices' own motivations for using code examples in open-ended programming. This is because traditionally in programming education, examples are used in the context of *worked examples* [38, 44, 51, 58]. When learning worked examples, students follow the structured learning practice to first study a step-by-step demonstration of solving a short, closed-ended problem (called a worked example), and next solve a similar problem independently [51]. This context differs fundamentally from our goal, which is to support students self-initiated example use, *in the middle of* programming. We draw inspirations from Wang et al. [54]'s work, which provides novices with code examples upon request, in the middle of completing closed-ended, drawing-based problems. In an interview study with 9 undergraduate novice students, they found that students request examples for 3 main reasons: "find next step; find how to complete a step; and fix a problem in their code" [54]. Going beyond prior work [54], we aim to collect data from multiple sources in a larger-scale study to identify motivations, with a focus of the context of open-ended programming, where students can encounter a different set of barriers [18].

2.3 How do novices use examples?

Much prior work has focused on understanding *experienced* programmers' [4, 47] and professional end users' [5] example reuse behaviors. Example reuse refers to "gluing together" [26] existing code from examples to use in a new context [26]. Such reusing behavior is usually "opportunistic" [6], with the primary goal of saving time [5, 29, 47, 48]. Rosson and Carroll investigated four experienced Smalltalk programmers' example use, and summarized the reusing process as "getting something to work with" by directly copying an example to their own workspace, and then heavily

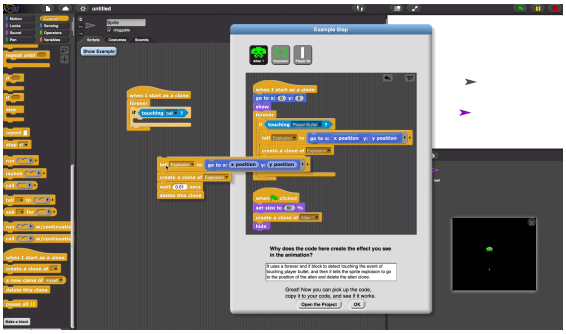


Fig. 1. The Example Aid Interface.

relying on the system debugger to test the example code and "debug into existenceâĂİ. The authors concluded that this strategy allowed programmers to quickly incorporate example functionality into their own code, but also may have caused two programmers to not end up completing the required programming artifact [47].

Some prior work that investigated novices' example reuse behaviors has shown that novices may employ other strategies when using examples [25, 54]. Wang et al. found that novices use strategies such as comparison, locating a change, or copying directly to use examples in closed-ended tasks [54]. Ichinco and Kelleher found two strategies novices' employ when working with code examples in closed-ended tasks: 1) code-example comparison, where they compare example and their own code to find what's missing; 2) example-emphasis, where they find important parts of an example from highlights in the system [25]. However, prior work [25, 54] did not analyze how these strategies may have supported more effective example use, and it is unclear whether these strategies would show up consistently in the context of open-ended programming, and whether new strategies would appear. We investigate these questions from a systematic analysis of multiple data sources in this work.

3 STUDY SETUP

Our study setup aims to collect multiple sources of data to record novices' example-usage experience, in an authentic, engaging open-ended programming experience.

3.1 System

We make use of an existing system called Example Aid [13], which adds a "show example" button on the Snap*!* editor, showing a gallery of examples upon request (Figure 1). Example Aid is particularly suitable for our goal to analyze novices' example use, for three key reasons:

3.1.1 ***High-quality examples.*** Example Aid includes a curated gallery of high-quality code examples, collected from a systematic analysis of common game behaviors students make in open-ended programming [13], and refined by expert researchers for the purpose of readability and integration of advanced programming concepts (e.g., lists).

3.1.2 ***Supports for searching, copying, and testing example code.*** Example Aid is designed specifically for supporting open-ended programming [13], where students may need to learn to use and integrate new concepts and code patterns on their own [18]. This process of searching for and learning programming knowledge is described by the COIL model [14], which includes information collection & organization; and solution testing. Students are provided with supports for all these three elements by Example Aid. First, a student can search for an example (*information*

collection): When clicking the "show example" button, students see a gallery of examples [13], where they can browse; search over a search box; or filter examples based on tags. Next, when they find an example and click to open it, they can read or copy code (*information organization*): when reading, the student can navigate through codes on different sprites[2], shown by different tabs in the example interface (Figure 1). They may also copy example code by dragging it to their own code. Last, to test code, they may run copied example code in their own program, or open the example code in a separate window by clicking on the "open the project" button (*solution testing*).

3.1.3 ***Prompts to self-explain.*** When reading an example, the student may answer a self-explanation prompt: "Why does the code here create the effect you see in the animation?". After typing 30 characters, they can copy the example code by dragging blocks to their own workspace. Example Aid encourages this self-explanation process, as it has consistently been shown to aid learning from examples [1, 50, 54].

3.2 Participants & Procedure

We held our study in a summer internship program, which aimed to teach high school students programming, and creating computing-infused projects for middle and high school teachers. The program was held online due to the COVID-19 pandemic. Our participants included 24 high school students in the program, 7 males and 17 females, who self reported as 2 White, 2 African American, 17 Asian, 1 Other, 2 Multiracial. The researchers who conducted the study were not directly involved with the internship program outside of instructing students during the study.

Our study occurred in first 3 days of the second week, described below, before which students completed a one-week coding bootcamp to program in NetsBlox [7]. We designed **a controlled study with alternating conditions**, where the Early group (n = 7 pairs) having access to examples only on Day 2, and the Late group (n = 5 pairs) having access to examples only on Day 3[3]. A researcher demonstrated how to use Example Aid, but students were not specifically prompted to use examples. To ensure an authentic and engaging learning experience, students **pair programmed** in Days 2 & 3, as pair-programming has been shown to promote higher performance for novices during open-ended programming [18], and is a standard practice in many real-world classrooms [31, 34, 52]. We therefore analyzed students in pairs.

3.2.1 ***Day 1: warm-up activity to assign groups & pairs.*** Students did a Snap*!*-based warm-up activity on Day 1, where they programmed 18 short, closed-ended tasks, including drawing shapes and programming multiple-sprite interactions, using loops and conditionals [56]. We ranked students' performance based on the time each student spent to complete the warm-up activity, and used this rank to balance groups, such that each group had a similar average performance. We also assigned students with adjacent ranks into the same pair, which can promote better learning outcomes for the pairs [32].

3.2.2 ***Day 2 & 3: building games.*** On Days 2 & 3, students built games with two different themes – *breakout* and *space-invaders*, respectively. These two themes include features such as the player interacting with a larger group of sprites (e.g. bricks, enemies), or collision causing them to disappear. These themes were suitable *open-ended* tasks, as they required the usage of many concepts (e.g., loops) and APIs (e.g., cloning in Snap*!*). Additionally, they provided flexibility and variability in game design [23] (e.g., adding new actors with different roles, designing levels), allowing students to incorporate their own choices and goals. To foster creativity, we started Days 2 & 3 by introducing a variety

[2] A sprite in Snap*!* is an object, such as an actor in a game.
[3] 28 students attended the study, based on which we assigned 7 pairs in each group. However, 2 students from 2 separate pairs in the Late group did not consent, we therefore excluded the two pairs' data from analysis.

of breakout/space invader games, retrieved from the online Scratch community [35, 53]. We did not require any specific features in games, and encouraged students to make *unique* and *creative* artifacts.

3.2.3 ***Interviews.*** To understand students' own perceptions, at the end of Days 2 & 3, we invited each pair to a 15-minute semi-structured interview, where they discussed their experience by answering questions such as "Describe a scenario where you have requested a code example". When students used vague terms such as "helpful", we encouraged them to describe a concrete example usage scenario they experienced.

4 ANALYSIS

We analyzed our data using the "Case Study Research" [22, 57] method, a systematic approach to research "decisions" – "why they were taken, how they were implemented, and with what result" [49]. Yin proposed that these "why" and "how" questions require tracing over time, and are therefore difficult to be summarized as incidents or frequencies, but rather require analysis from a time-series-based perspective, collecting data from multiple sources to describe phenomena with their own context – "cases" [57].

Data Organization. To ensure construct validity [10], we collected and organized data following the 3 principles by Case Study Research: 1) We collected data from "**multiple sources**" [57], including: a) video recordings of students' screen, including transcriptions of pair conversations; b) interview transcriptions; c) logs, including students' code and activities (e.g., each code edit) at every timestamp; and d) students' final submissions. 2) Since we focused on analyzing example usage, we defined each example request as a single "case", and created a "**case study database**" [57] of all 88 example requests. For each request, we compiled a "case profile", including the data from sources (a) – (c). Because we encouraged students to describe concrete scenarios (Section 3), most interview quotes map to specific example requests, though some do not – for those interviews that describe students' general experience, and for data source (d), we 3) established a "**chain of evidence**" [57] by linking interview and submissions to the case profiles of corresponding pairs, to enable tracing back/forward between different data sources and analysis stages [57].

Analysis. We next analyzed data to investigate our research question on novices' motivations and strategies, following the 2 analytic techniques by Case Study Research.

4.0.1 ***Finding "patterns" [57] from logs.*** A"pattern" describes cause, effect, or events that relate to the central phenomenon of interest [57]. As log data captures most precisely students' experience comparing to interviews, which may suffer from response bias and inaccuracies [11, 42], we start our analysis of cases first on their log data, a commonly-used primary data source to analyze programmers' [6], end users' [27] including novices' [25] programming experiences.

We find patterns of **situations** where students ask for examples, based on two types of log data: 1) the code students have to complete a feature demonstrated by an example before asking it (called "starter code"); and 2) the activities students engaged in with examples – whether they attempted reusing the example code, or they immediately closed examples. These two data types have been shown by prior work [54] to describe students' goals for requesting examples. For example, we can infer a "debugging goal" when a student had buggy code and used example to locate changes to make. Based on these two data types, we identified 5 situations where students ask for examples (e.g., "when starting a step").

Similarly, we look for patterns of **strategies** by analyzing students' programming activities in logs. From the case database of 88 requests, we first filtered out 41, where students immediately closed the example after opening. For the remaining 47, we analyzed those repetitive requests of the same example in aggregate, creating a total of 29 sets of

⋆When browsing/exploring	38.6%
When starting a step	29.5%
When debugging incorrect/incomplete code	22.7%
⋆When finished with a step	8.0%
⋆When re-implementing a step	1.1%

Table 1. 5 situations where students ask for code examples and their frequencies among a total of 88 example requests.

example requests. We started with a detailed account of all activities pairs engaged in when using examples, such as the time when the student started programming the relevant feature, their starting code, the programming behaviors they engaged in while using examples (e.g., "copied block x from example code"), with timestamps, students' conversations (capture by the videos), the students' final code after completing (or abandoning) the feature demonstrated by the example (called "final code") and their comments in the interview when available. One researcher coded thoroughly these documents, generating 7 initial patterns of strategies. The researcher next worked with another researcher to merge similar strategies, generating 4 strategies, and created definitions of these strategies. The researcher next re-coded these documents again to confirm these strategies and label each example request with its corresponding strategies.

4.0.2 ***Building "explanations" [57] from conversations, interviews and submissions.*** Based on the found patterns, we build explanations for two goals: 1) To find evidence from students' conversations and interviews to explain motivations and strategies. Towards this goal, we coded the conversations and interview data on each case profile to look for presence of existing patterns, and examine whether new patterns appeared. Based on the situations of *when* students ask examples, we used evidence from conversations and interviews to explain *motivations*. This extra data confirmed our identified situations and strategies, adding students' perceptions of causes and effects of their example usage motivations and strategies. We used this data to re-code all case profiles the third time, confirming that students' discussion were accurate at describing their example reuse scenarios.

2) If students' needs were met and their strategies effectively used, we would find examples not only supporting students' individual requests, but also helping them to create more complex and creative projects. We therefore analyzed all students' project submissions, following the approach by Catete et al. on developing scientific rubrics [8]. Two researchers independently examined all student programs to generate lists of all features created by students in their programs, with each feature to 1) describe a distinct behavior of the game and 2) could be adapted to other game design tasks. This created a total of 23 and 25 features for breakout and space invaders respectively. Examples of such features include "an actor moves with a key press" and "increase score when one actor hits another"[4]. The two researchers then graded each student program by how many features in the list a student program completed. This result is discussed in Section 5.4.

5 RESULTS

5.1 Motivations: Why do students ask for examples?

Table 1 shows 5 distinct situations when students requested examples. Among the 5 situations, 2 were discovered by prior work as students' motivations for using examples in *closed-ended* tasks [54]; others (denoted with ⋆ in the

[4]Such feature-based rubrics are commonly used in grading interactive, visual programs [55].

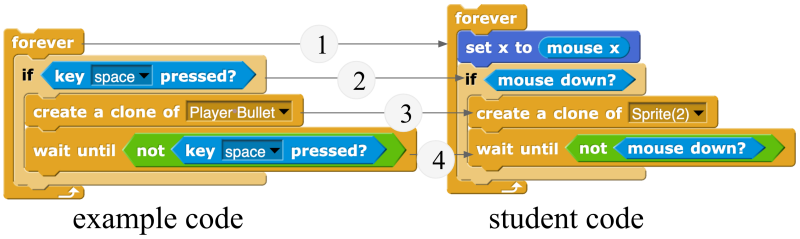


Fig. 2. L3 copied, modified, and tested to integrate the example code to their own code **one at a time**.

table) correspond to new, distinct motivations that arise from our analysis, potentially due to the context of open-ended programming. For each example-use situation we report, we also identify students *motivations*, reported in the interview data, that corresponded with these situations.

Many example requests (68.1%) come from students who opened an example about a new feature *not* implemented in their code. Some students may have a **browsing/exploring** motivation (38.6%), as evidenced by opening and closing the example, without attempting to integrate the example to their own code. Students in interviews described that they "scrolled through the gallery" to "choose our examples", by "click[ing] on it" to open and check "if it looked like something in I would be using"[E6].

Others tried to integrate the example into their own code (29.5%), showing example use when **starting a step**. Students' quotes explained two *motivations* that may be met in this situation: 1) they wanted to know what to do next: "[examples] isolate a step that you wanted to do and then [you could] focus on that instead of trying to do everything at once."[L1][5]; or because 2) they know their next step, but wanted to know how to implement it: "We had an idea of how the code would work, but we didn't know the exact way we could all put it together."[E4].

22.7% example requests came from students who were **debugging incorrect or incomplete code**: students explained that they were in the middle of completing a certain feature – "we sort of got it", but don't know "what wasn't working with our [code])"[E1]. 8.0% requested examples were about features the students have already completed in their code. In these example requests, students opened the example code, but did not try integrate the example code to their own code, perhaps due to their code being already completed, showing a motivation of **confirming** their own implementation of a certain feature, although this motivation was not discussed during the interviews.

One student, after spending time learning and using an example, requested the example again in another sprite, and directly copied the example to their own code. While the student did not explain their motivations during the interview, it seems that the student were using the example code for the purpose of **avoiding re-implementing** it on their own.

5.2 Strategies: How do students learn and use an example?

We next discuss the students' example-use *strategies* that reflect their own choice of *how* to learn and use examples.

5.2.1 ***Integrate one block/feature at a time.*** In 37.9% example requests, students integrated the example code to their own code, by **copying, modifying, and testing** the example code one block or one small feature at a time. For example, in Figure 2, L3 separated the process of copying and reusing into 4 sub-steps, each focused on one block,

[5]A quotation from Pair 1 in the Late group. E and L denotes Early and Late group, respectively.

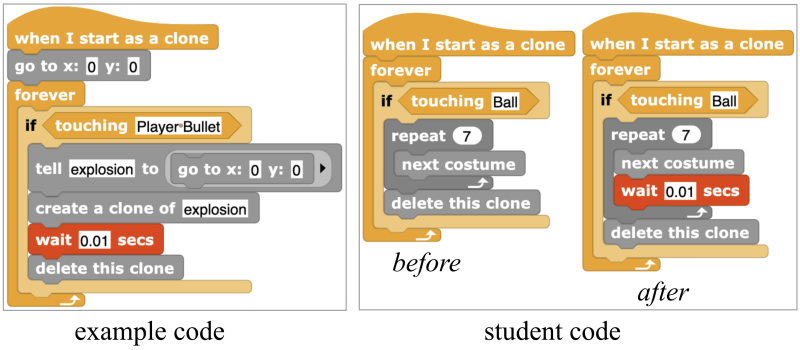


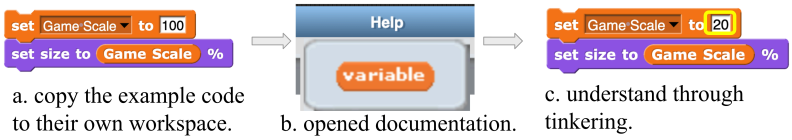
Fig. 3. Using the **comparison** strategy, E7 identified meaningful differences to use in their own context, without discarding unmeaningful differences.

shown by the arrows from the example code to the student's own code[6]. With each sub-step, they modified their code, sometimes testing it (2/4 times), before copying the next code block. During interview, L3 explained that they "individually went into"[L3] the left code palette to copy code, and commented that "doing that allowed me to make my own modifications as I went and I better understood it."[L3].

5.2.2 ***Comparison to identify key differences with example.*** When requesting examples, many students have existing code that completes partially the target feature they need (e.g., when the feature was half-complete but was buggy). However, students' existing code can very different from the example code they requested. In these scenarios, students have described a comparison strategy, where they "looked at [their] code and that [example] code side by side" and "compare[d] it"[L1]. Figure 3 shows how E7 employed the comparison strategy. To clearly illustrate the learning scenario, we recolored the code blocks to differentiate 1) blocks that were *different* from student code to example code (grey), 2) blocks that were the *same* in student code and example code (yellow); 3) block that's *added* after reading the example code (red)[7]. The students' code has many differences comparing to the examples. However, while comparing their own code to the example code, the students identified the meaningful differences between their own code and the example code, and added it onto their own code without discarding the less meaningful differences (e.g., changing costumes).

5.2.3 ***Understanding through tinkering.*** Tinkering refers to "an informal, unguided exploration initiated by features visible in the environment" [3]. After copying examples to students' workspace, we found that some students experimented code blocks by modifying (e.g., changing variables, or by removing a block they do not understand), and testing to find the difference, showing a "test-based tinkering" behavior [12], which aims to understand the example code. We call this strategy "understanding through tinkering", shown in 17.2% of the example requests. For example, in Figure 4, L4 was confused by the code block "set size to Game Scale %": "I do not know what Game Scale is or what it's doing."[L4]. They right clicked on the "Game Scale" variable to open the Help documentation, which only explained the generic usage of a variable block, but not how "Game Scale" is used in the context. So the student then changed the value of the variable from 100 to 20, tested again, realizing that the block changes sizes of a sprite: "so, is it like, if

[6]In specific, such copying is made by using the example as a reference, and moving the code from the left block palette (Figure 1).
[7]Due to space limit, we simplified the example and student code in Figure 3 to only show the feature that the student asked example for (i.e., an explosion effect).



a. copy the example code to their own workspace. b. opened documentation. c. understand through tinkering.

Fig. 4. L4 employed the strategy "**understanding through tinkering**" to understand an unfamiliar block - "set size to".

strategy name	freq	suc	add/e	del/e	e/add	final/e
⋆one at a time	37.9%	1.0	0.53	0.02	0.71	0.38
comparison	34.5%	0.8	0.22	0.01	0.9	0.42
⋆tinkering	17.2%	1.0	0.87	0.01	0.83	0.90
⋆impl. after closing	13.8%	0.75	0.56	0.0	0.74	0.47
copy-run-debug	13.8%	0.25	0.33	0.0	0.84	0.15

Table 2. 4 example reuse strategy and the copy-run-debug behavior (i.e., lack of strategy). Strategies started with ⋆ denotes example use strategies not mentioned in prior work [25, 54].

you make it a larger number it would just get . . . ah I see."[L4] They later integrated the example by deciding on value of "Game Scale" to be 30.

5.2.4 ***Implement after closing the example.*** In 13.8% of example requests, students closed the example, and tried to implement a needed feature themselves.

5.2.5 ***Lack of strategy: copy-run-debug.*** 13.8% example requests did not include any of the above-mentioned example reuse strategies, but used more expedient, "opportunistic"[6] approaches (called "copy-run-debug"), with two representative behaviors. 1) **copy/replace blindly**: In 2 example requests, the students copied the entire example to their own code, and completely removed their own existing code, although it was partially correct. In these scenarios, the "comparison to identify key differences" strategy would have been useful but was not employed. 2) **shallow debugging**: In 3 example requests, the students tested the copied code from example and found it did not work as expected. However, students' conversations showed they ignored the blocks they were unfamiliar with, but kept modifying other (correct) blocks that they thought produced the error, making arbitrary changes in an effort to resolve the error. In these scenarios, the "understanding through tinkering" strategy would have been helpful for the students to first understand the unfamiliar blocks.

5.2.6 ***How effective are these strategies?*** One way to evaluate effectiveness is to understand how these strategies helped students to overcome their barriers when using examples. We focus on two measures: 1) Was the example successfully integrated in students' code?; 2) To what extent did the student modify the example code?, as integration and modification are two explicit goals that students have when using examples [13, 54]. Table 2 shows 6 statistics for the above 2 measures: 1) freq: the percentage of strategies shown in the 29 example requests. 2) suc: proportion of the example requests that were successfully integrated to students' code. We refer to this number as *success rate*. 3) & 4) add/e & del/e: code that was added/deleted while using the example, divided by the length of example code. For example, in the example request of Figure 3, add/e = 1/10 = 0.1; del/e = 0/10 = 0, averaged across requests. 5) e/add: the proportion of the added code blocks that are in the example code, e.g., e/added of Figure 3 is 1/1 = 1, averaged across

requests. 6) final/e: the proportion of example code kept in students final code. e.g., final/e of Figure 3 is 5/10 = 0.5, averaged across requests. Table 2 presents the following insights:

a) **Students almost always used a strategy of some form, which often led to success and adaptation.** 86.2% (25/29) of example requests included the use of at least one of the 4 key strategies. Each strategy led to at least a 75% successful integration rate. Column e/add shows that the majority of students do not copy blindly, as when adding example code to their own code, students also add code blocks that are not from the example code, showing evidence of modifications.

b) **Different strategies have different use cases and affordances.** Column final/e shows how much similarity there was between the students' final code and the example code. The *tinkering* strategy creates code of the highest similarity, and has the highest add/e rate, showing that students may use tinkering when copying large chunks of example code, without much modification. The *comparison* strategy was employed when adding a small amount of code from example to students' own code, which was usually kept, showing that this strategy was most often employed in scenarios such as Figure 3, where students already had partially complete code blocks, and used the needed block from an example to fix a bug. Finally, *one at a time* and *implement after closing* led to high modifications, shown by the lower e/add and final/e statistics. This shows that these two strategies were more appropriate for students who needed more example adaptation, mirroring students' interview comments presented in Section 5.2.1.

c) **Lack of strategy can lead to failures of integration.** While typical strategies led to at least 75% success rate, in the 4 instances of copy-run-debug behaviors, only one led to successful integration. It also has high kept rate (0.84), showing that students modified less of the example when adding. In addition, many parts of the example were discarded from the students' final code, as they were not able to reuse them successfully, shown by its low add/e and final/e rates.

5.3 An in-depth example

We use the most frequently requested example "spawn_clones" to illustrate how motivations and strategies were reflected in students' experience. "Spawn_clones" demonstrates a commonly-seen feature in breakout and space invaders, which creates groups of bricks/enemies by using clones, an API that creates copies of sprites in Snap*!*. The grey bars in Figure 5 indicates the time duration when students were implementing this feature. They removed durations of activities on creating other features, which are sometimes interleaved in between implementations of "spawn_clones". At the end of each grey bar shows the implementation outcome – successful (tick) or abandoned (cross), as well as total time spent on implementing just "spawn_clones". The colored boxes on the grey bars are the time when students had the example interface opened, where each color represents a specific example use strategy (see Section 5.2). The triangles at the start of these boxes indicate the students' situation/motivation when opening the example (see Section 5.1). Figure 5 presents the following highlights:

5.3.1 ***Students change strategies at different stages of example use.*** Students employed diverse strategies for a variety of reuse scenarios. For example, E1 first opened, read, closed the example, and then *implement after closing* for a minute, till their code was buggy, where they asked for the example again to debug for two more minutes, using a *comparison* strategy. When they completed a correct implementation, they requested the example the third time to confirm (without changing their code). E2 shows **a transition from a lack of strategy to more active strategies**. They first used the *copy-run-debug* behavior to blindly copy the example code. They found the integration to be erroneous, but superficially debugged and were unable to fix the error. They next removed all code and requested example again, this time spending more time to modify and test different blocks in the example code. After they

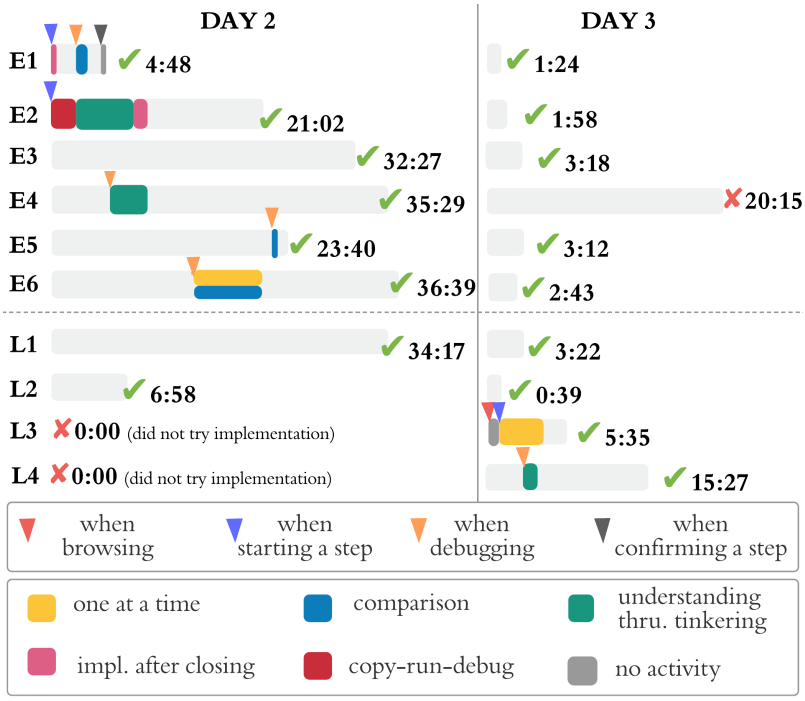


Fig. 5. Case study: students' implementation of spawn_clones.

understand the example through tinkering, they removed all code again started over again, this time led to success integration. This shows that strategies are not used in isolation, and students may start with more expedient strategies, and move on to more time-consuming (and effective) strategies.

5.3.2 ***Integrating example code takes time.*** When implementing "spawn_clones" for the first time (on Day 2), all from the Early group succeeded; half from the Late group, who do not have access to examples, did not attempt to create this feature at all. This shows that **the existence of examples create possibilities to implement certain features**. (e.g. by letting students know it was possible, or by helping them implement it). However, those who had access to examples on Day 2 did not spend less time programming the feature (Early group mean = 25:41), compared to those who did not (Late group mean = 20:38). While our data is for a single challenging example with a small population and cannot support strong claims, it does show that most students spent longer than 15 minutes on this feature when implementing it for the first time, even with the existence of the examples. This shows that students need time to integrate and use some examples.

5.3.3 ***Program a feature the second time is easier.*** When students succeeded in creating the feature on Day 2, they were able to spend about 10 times less time to program "spawn_clones" on Day 3, regardless of whether example was used. Except E4, who implemented the feature on Day 2 with help from instructors, the rest of the students from the Early group were all able to efficiently create the feature on Day 3.

5.4 Outcomes of example use

Responding to motivations found in Section 5.1, students commented that examples "gave [them] ideas"[E6], to "break down a task into parts"[L1], or "with debugging"[E6]; expressing examples have met their needs. Connecting with strategies found in Section 5.2, we found students used diverse strategies to integrate example code into their own code, with high success rate. These shows that example helped students to implement features in *individual requests*. But **what are the general outcomes of using examples in open-ended programming?** Our analysis on 10 pairs[8] was insufficient for any strong claims, but we found suggestive insights from students' submissions and interview data, discussed below.

5.4.1 ***Examples helped students create more complex programs.*** During Day 2, the Early group who had access to examples created an average of 14.2 features, (n = 6, SD = 4.4), over 50% more features than the Late group, who did not use examples and created an average of 9.25 features (n = 4, SD = 6.9). As the two groups had no statistically significant difference in their warm-up activity performance, this finding shows suggestive evidence that examples helped students create complex programs.

5.4.2 ***Students keep creating complex and interesting programs after removed access to examples.*** In the Early group submissions of Day 3, the students created an average of 15.2 features, (n = 6, SD = 5.1); higher than the average of 14.2 features on Day 2, showing that the Early group kept creating many features without access to examples on Day 3. On the other hand, the Late group, who had access to examples, created an average of 10.75 features (n = 4, SD = 3.1) on Day 3, higher than the average of 9.25 features on Day 2.

The Early group's higher performance than the Late group on Day 3 can be likely due to the fact that *finishing* a feature, whether or not using an example, seemed to have a clear impact to speed up implementing the feature on the next day (Figure 5). Therefore, the fact that the Late group only created an average of 9.25 features on Day 2 may have caused them to encounter creating many new features on Day 3 (e.g., L3 & L4 in Figure 5). However, having examples on Day 3 did not necessarily reduce the time spent for integration (Section 5.3. Therefore causing the Late group to create features less than the Early group, as the Early group may have already learned how to program those features from Day 2.

4 pairs from the Early group expressed that learning the example prior to the current task helped them create features independently, explaining that the process of making project was "a lot easier today even though the games are different"[E7], and that "since we learned from the one ([i.e., example]) from yesterday, we figured out how to apply to this one."[E3]. This shows that many students learned or memorized how to create a feature the example demonstrated while using the example, a type of learning event (called "memory and fluency building") by prior work [28].

6 DISCUSSION

We discuss *design implications* for designing systems with code examples to support novices' open-ended programming.

[8] E7 and L5 were excluded from this analysis, as E7 had access to examples on both Days 2 & 3, while L5 programmed on the Snap*!* server that does not have logging features on Day 2; therefore we were unable to retrieve their projects and logs.

6.1 Support design and planning for creating complex projects

Going beyond prior work on students' goals of using examples in *closed-ended tasks* [54], we identified 3 distinct and novel motivations: browsing, confirming, and re-implementing. In particular, the most frequent motivations is for browsing (38.6%), where students needed to find features that they wanted to create. This motivation corresponds to a **design barrier** identified by Ko et al. [27], where end users needs support to identify "what I want the computer to do" [27], and was novel and distinct from prior work, which focus on identifying motivations towards **implementations** [54]. This motivation for design ties in strongly with the potential benefits of open-ended programming, which aims to empower students to create projects that they personally connect to, and engage them in the process of building an artifact of their own choice [18, 21]. Examples should therefore, support students to create designs that feel challenging and meaningful to them.

6.2 Encourage effective example learning strategies

The primary highlight on our findings of students' strategies is the **diversity** of effective strategies, where we also discover novel findings on 3 strategies that were not discussed from prior work [25, 54]. This also contrasts with end users' example reuse strategies, discussed by prior work, which shows that they generally rely on opportunistic strategies ("getting something to work with") [48]. Section 5.3 shows a potential reason for the diversity of strategies among novices: while examples helped students to create features they were otherwise unable to, integrating example code to their own code is still *difficult*, mirroring findings from our prior work on barriers students encounter when reusing code examples [13], this leads to students' choice of strategies to overcome different types of barriers [13] – for example, to overcome **understanding barrier** (i.e., "how to use an unfamiliar code block in the example?" [13]), students use the strategy "understanding through tinkering"; to overcome **mapping barrier** (i.e., "how do I map a property of the example code to my own code?" [13]), students employ the comparison strategy.

This diversity of effective strategies leads to important design implications – prior work has shown that one way systems could do to support users is by encouraging them to "*work in the way they are used to working, but inject good design decisions into their existing practices*" [26]. Based on the ICAP hypothesis [9] and the evidence from our study, we hypothesize that design decisions that encourage more *active* and *constructive* modes of interaction with the system opens up more choices for students to effectively engage with the examples. We therefore summarize two different ways we may build systems to encourage more active use of examples, discussed below.

6.2.1 ***Transform passive code copying towards active code reconstruction.*** We found students who copied the example one block/feature at a time engaged in the process of reflecting and modifying the process, causing them to adapt the example effectively in their own task. This strategy corresponds to an *active* learning behavior suggested by the ICAP hypothesis, and may cause the learner to more efficiently integrate the new information [9]. This spontaneous reading, using, and modifying behavior resembles the Use-Modify-Create practice [30, 33], which encourages learners to complete a series of activities from re-using programming code examples (*use*), to making small modifications (*modify*), to taking full ownership of learners' program by creating program from scratch (*create*). We envision that following a similar progression, an opportunity for students to progress from *modify* to *create* would yield higher learning outcomes and increased level of engagement. Such transformations can be made by programmers reconstructing blocks of examples by solving it as a Parsons problem [40], which breaks a correct solution into code pieces and asks students to rearrange the code to the original solution.

6.2.2 ***Foster knowledge integration through comparison of executable examples.*** Comparison is a classic constructive strategy to foster knowledge integration [9]. In our study, we found students effectively employed a comparison strategy to locate the relevant part of an example that they could use to correct their code. Example Aids facilitated comparison by allowing students to read their code and example code in *a parallel view*, which has been shown to enable higher transfer to solve similar, new problems, than a traditional, sequential view of different solutions [41]. In addition to the parallel view to compare *code*, students were also able to run the example code to compare *outputs*, which has been shown to help students identify key differences and how they relate to changes in output [54]. Our findings that some students were able to infer the important part of the example code to use, and keep their own part of the correct solution, is consistent with the findings by prior work on comparison and transfer, indicating that the comparison allowed students to reason and differentiate the important part of the problem-solving schema [16, 41, 45]. Although our system only facilitates comparison through an convenient, embedded example support interface, we envision future example support systems to offer runnable code examples in parallel, such as different approaches to solve the same problem (similar to [41]). We envision such an interface to offer support for constructive learning and generating problem-solving schema [17].

7 LIMITATIONS & CONCLUSION

Limitation: we conducted our study with 24 students in just one system — we used multiple sources and prior work to triangulate our findings, and exercised rigor and reflexivity [19] when collecting, organizing, and analyzing data to find valid, grounded evidence. However, we need future work to find statistical evidences (e.g., whether the strategies we found were significantly more effective than others) among larger-scaled populations.

In conclusion, our work identified novel, distinct, and diverse motivations and strategies novices employ when using code examples in open-ended programming. We found that many students use effective strategies to reuse and modify examples, potentially lead to creation of more complex, interesting programs. We proposed concrete recommendations for future example designs to incentivize learning through code comparison and reconstructions.

REFERENCES

[1] R. K. Atkinson, A. Renkl, and M. M. Merrill. Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Journal of educational psychology*, 95(4):774, 2003.

[2] G. R. Bai, J. Kayani, and K. T. Stolee. How graduate computing students search when using an unfamiliar programming language. In *International Conference on Program Comprehension (ICPC)*, 2020.

[3] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240, 2006.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[5] A. F. Blackwell. First steps in programming: A rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10. IEEE, 2002.

[6] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[7] B. Broll, A. Lédeczi, P. Volgyesi, J. Sallai, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz, and M. Lu. A visual programming environment for learning distributed programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 81–86, 2017.

[8] V. Cateté, N. Lytle, and T. Barnes. Creation and validation of low-stakes rubrics for k-12 computer science. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018, page 63âĂŞ68, New York, NY, USA, 2018. Association for Computing Machinery.

[9] M. T. Chi and R. Wylie. The icap framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist*, 49(4):219–243, 2014.

[10] L. J. Cronbach and P. E. Meehl. Construct validity in psychological tests. *Psychological bulletin*, 52(4):281, 1955.

[11] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies. " yours is better!" participant response bias in hci. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 1321–1330, 2012.

[12] Y. Dong, S. Marwan, V. Catete, T. Price, and T. Barnes. Defining tinkering behavior in open-ended block-based programming assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 1204–1210, 2019.

[13] B. for review. Blinded for review. Blinded for reivew.

[14] G. Gao, F. Voichick, M. Ichinco, and C. Kelleher. Exploring programmers' API learning processes: Collecting web resources as external memory. In M. Homer, F. Hermans, S. L. Tanimoto, and C. Anslow, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2020, Dunedin, New Zealand, August 10-14, 2020*, pages 1–10. IEEE, 2020.

[15] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[16] D. Gentner, J. Loewenstein, and L. Thompson. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology*, 95(2):393, 2003.

[17] M. L. Gick and K. J. Holyoak. Schema induction and analogical transfer. *Cognitive psychology*, 15(1):1–38, 1983.

[18] S. Grover, S. Basu, and P. Schank. What we can learn about student learning from open-ended programming projects in middle school computer science. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 999âĂŞ1004, New York, NY, USA, 2018. Association for Computing Machinery.

[19] M. Guillemin and L. Gillam. Ethics, reflexivity, and âĂIJethically important momentsâĂİ in research. *Qualitative inquiry*, 10(2):261–280, 2004.

[20] M. Guzdial. Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, 2015.

[21] M. Guzdial and A. Forte. Design process for a non-majors computing course. *ACM SIGCSE Bulletin*, 37(1):361–365, 2005.

[22] L. Hamilton and C. Corbett-Whittier. *Using case study in education research*. Sage, 2012.

[23] R. Hunicke, M. LeBlanc, and R. Zubek. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, page 1722, 2004.

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

[25] M. Ichinco and C. Kelleher. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 63–71. IEEE, 2015.

[26] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3), Apr. 2011.

[27] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 199–206. IEEE, 2004.

[28] K. R. Koedinger, A. T. Corbett, and C. Perfetti. The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science*, 36(5):757–798, 2012.

[29] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, page 69âĂŞ73, New York, NY, USA, 1989. Association for Computing Machinery.

[30] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith, and L. Werner. Computational thinking for youth in practice. *Acm Inroads*, 2(1):32–37, 2011.

[31] C. M. Lewis. Is pair programming more effective than other forms of collaboration for young students? *Computer Science Education*, 21:105 – 134, 2011.

[32] C. M. Lewis and N. Shah. How equity and inequity can emerge in pair programming. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 41–50, 2015.

[33] N. Lytle, V. Cateté, D. Boulden, Y. Dong, J. Houchins, A. Milliken, A. Isvik, D. Bounajim, E. Wiebe, and T. Barnes. Use, modify, create: Comparing computational thinking lesson progressions for stem classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 395–401. ACM, 2019.

[34] N. Lytle, A. Milliken, V. Cateté, and T. Barnes. Investigating different assignment designs to promote collaboration in block-based environments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 832âĂŞ838, New York, NY, USA, 2020. Association for Computing Machinery.

[35] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[36] S. McGee, R. McGee-Tekula, J. Duck, C. McGee, L. Dettori, R. I. Greenberg, E. Snow, D. Rutstein, D. Reed, B. Wilkerson, D. Yanek, A. M. Rasmussen, and D. Brylow. Equal outcomes 4 all: A study of student learning in ecs. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 50âĂŞ55, New York, NY, USA, 2018. Association for Computing Machinery.

[37] J. Moenig and B. Harvey. Byob build your own blocks (a/k/a snap!). *URL: http://byob. berkeley. edu/, accessed Aug*, 2012.

[38] B. B. Morrison, L. E. Margulieux, and M. Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 21–29. ACM, 2015.

[39] C. Parnin and C. Treude. Measuring api documentation on the web. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pages 25–30, 2011.

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

[41] E. Patitsas, M. Craig, and S. Easterbrook. Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 145–152. ACM, 2013.

[42] D. L. Paulhus. Measurement and control of response bias. 1991.

[43] K. A. Peppler and Y. B. Kafai. From supergoo to scratch: Exploring creative digital media production in informal learning. *Learning, media and technology*, 32(2):149–166, 2007.

[44] P. Pirolli and M. Recker. Learning strategies and transfer in the domain of programming. *ITLS Faculty Publications*, 12, 09 1994.

[45] B. Rittle-Johnson and J. R. Star. Does comparing solution methods facilitate conceptual and procedural knowledge? an experimental study on learning to solve equations. *Journal of Educational Psychology*, 99(3):561, 2007.

[46] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[47] M. B. Rosson and J. M. Carroll. Active programming strategies in reuse. In *European Conference on Object-Oriented Programming*, pages 4–20. Springer, 1993.

[48] M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[49] W. Schramm. Notes on case studies of instructional media projects. 1971.

[50] B. Shih, K. Koedinger, and R. Scheines. A response-time model for bottom-out hints as worked examples. pages 117–126, 01 2008.

[51] J. G. Trafton and B. J. Reiser. *The contributions of studying examples and solving problems to skill acquisition.* PhD thesis, Citeseer, 1994.

[52] J. Tsan, J. Vandenberg, Z. Zakaria, D. C. Boulden, C. Lynch, E. Wiebe, and K. E. Boyer. Collaborative dialogue and types of conflict: An analysis of pair programming interactions between upper elementary students. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1184âĂŞ1190, New York, NY, USA, 2021. Association for Computing Machinery.

[53] W. Wang, Y. Rao, Y. Shi, A. Milliken, C. Martens, T. Barnes, and T. W. Price. Comparing feature engineering approaches to predict complex programming behaviors. *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20*, 2020.

[54] W. Wang, Y. Rao, R. Zhi, S. Marwan, G. Gao, and T. Price. Step tutor: Supporting students through step-by-step example-based feedback. *ITiCSEâĂŹ20 - Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, To be published*, 2020.

[55] W. Wang, C. Zhang, A. Stahlbauer, G. Fraser, and T. Price. Snapcheck: Automated testing for snap programs. ITiCSE '21. Association for Computing Machinery, 2021.

[56] W. Wang, R. Zhi, A. Milliken, N. Lytle, and T. W. Price. Crescendo : Engaging Students to Self-Paced Programming Practices. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2020.

[57] R. K. Yin. *Case study research and applications: Design and methods.* Sage publications, 2017.

[58] R. Zhi, T. W. Price, S. Marwan, A. Milliken, T. Barnes, and M. Chi. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 98–104. ACM, 2019.