A Case Study on When and How Novices Use Code Examples in Open-Ended Programming

ABSTRACT

Many students rely on examples when learning to program, but they often face barriers to incorporating these examples into their own code and learning the concepts they present. As a step towards designing effective example interfaces that can support student learning, we investigate novices' needs and strategies when using examples to write code. We conducted a study with 12 pairs of high school students working on open-ended game design projects, using a system that allows students to browse examples based on their output, and to view and copy the example code. We analyzed interviews, screen recordings, and log data, identifying 5 moments when novices request examples, and 4 strategies that arise when students use examples. We synthesize these findings into principles that can inform the design of future example systems to better support students.

ACM Reference Format:

. 2023. A Case Study on When and How Novices Use Code Examples in Open-Ended Programming. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/xxxxxxxxxxx

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 [6]. In particular, novice programmers stand to benefit from programming examples, which can introduce new programming concepts [33, 37, 41, 43], and scaffold users to create more complex and interesting programs [22]. However, prior work on systems that support novices' example use has 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 [12, 21, 43].

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 *when* students request examples, as effective support systems must directly address these needs [17]. 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 [25].

In this work, we ask the research question: **When do novices ask for code examples, and how do they use code 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 [18], and are therefore widely used in many introductory programming curricula [12, 14, 15, 31] as well as after-school, informal learning settings [36]. However, students are also known to face a number of barriers to incorporate challenging new programming patterns and APIs in open-ended programming [15], which code examples that demonstrate such knowledge may help to overcome [12, 22].

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] [12], an example support system designed for open-ended programming in Snap*!* [32]. We analyzed video, interview, logs, and project submissions, identifying 5 distinct moments 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 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' potential needs 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

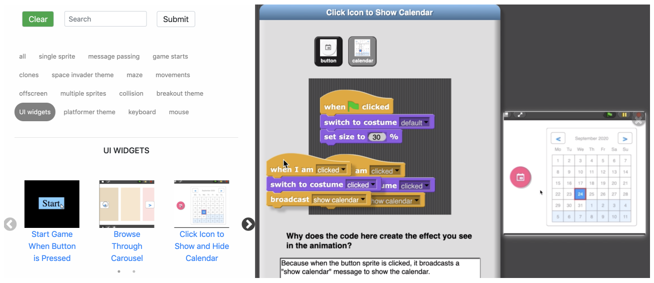
Open-Ended Programming. Much prior work on novices' example use focus on supporting students to complete *closed-ended* programming tasks [33, 37, 41, 43, 46]. By contrast, in open-ended projects students are encouraged to choose their own goals and to pursue projects that feel meaningful to them, such as making apps, games, and simulations [15]. These open-ended, choice-driven projects engage students by allowing them to create projects that connect to their own personal interest [18], and are popular among many introductory programming curricula [12, 14, 15, 31]. 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 [15]; and difficulties in understanding and using unfamiliar

[1] System name changed for anonymity.

APIs [12]. These challenges could be effectively addressed by code examples, which can demonstrate concepts and API use [12], suggesting the need for a better understanding of novices' example use during open-ended programming. We do so by investigating *why* and *how* students use code examples, discussed below.

***When* do novices ask for examples?** Our goal of identifying the moments of example use, is to understand novices' needs when requesting examples. There has been little prior work investigating novices' needs for using code examples in open-ended programming. This is because traditionally in programming education, examples are used in the context of *worked examples* [33, 37, 41, 46]. 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 [41]. Some CS instructors [28] or systems [44] also include examples in Use-Modify-Create pedagogy or practices, where students first learn an example demonstration of a problem (use), then make modification (modify), and then use the concepts they have learned to build a new program (create). This context differs fundamentally from our goal, which is to support students' self-initiated example use, *in the middle of* programming. We draw inspiration from Wang et al. [43]'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" [43]. Going beyond prior work [43], we aim to collect data from multiple sources in a larger-scale study to identify the key moments when novices ask for examples, with a focus on the context of open-ended programming, where students can encounter a different set of barriers [15].

***How* do novices use examples?** Much prior work has focused on understanding *experienced* programmers' [4, 38] and professional end users' [5] example reuse behaviors. Example reuse refers to "gluing together" [25] existing code from examples to use in a new context [25]. Such reusing behavior is usually "opportunistic" [6], with the primary goal of saving time [6]. 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 relying on the system debugger to test the example code and "debug into existence". 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 [38]. Prior work that investigated Scratch's "remix" projects also found that novices encounter challenges to reusing or modifying complex API usage patterns (e.g., procedure and cloning) when remixing an example project, and encounter challenges to understanding an example, and apply it in their own projects [1, 23]. Analysis of Scratch's "remixing" projects also found that novices often misuse or ignore sophisticated APIs (e.g., cloning and procedures) when remixing an example project, and experience difficulties to understand example code and connect them with their own goals [1, 23]. Some prior work that investigated novices' example reuse behaviors has shown that novices may employ other



```
when [flag] clicked
switch to costume [default]
set size to (30) %

when I am [clicked]
switch to costume [clicked]
broadcast [show calendar]
```

Why does the code here create the effect you see in the animation?

Because when the button sprite is clicked, it broadcasts a "show calendar" message to show the calendar.

Figure 1: The Example Aid Interface.

strategies when using examples [21, 43]. Wang et al. found that novices use strategies such as comparison, locating a change, or copying directly to use examples in closed-ended tasks [43]. 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 [21]. However, prior work [21, 43] 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 [12], which adds a "show example" button on the Snap*!* editor, showing a gallery of examples upon request (Figure 1). Example Aid is suitable for our goal to analyze novices' example use, for three key reasons:

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 [12], and refined by expert researchers for the purpose of readability and integration of advanced concepts (e.g., lists).

Supports for searching, copying, and testing example code. The Example Aid's design follows the COIL model [13] on programmers' information search and integration process. First, a student can search for an example (*information collection* [13]): When clicking the "show example" button, students see a gallery of examples [12], 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 the example, or copy the code by dragging it to their own code (*information organization*[13]). 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* [13]).

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 [2, 40, 43].

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, and 2 Multiracial.

Our study occurred in the first 3 days of the second week, before which students completed a one-week to program in NetsBlox [7]. We designed a formative study, alternating conditions for when students had access to Example Aid. The Early group (n = 7 pairs) had access to examples only on Day 2, and the Late group (n = 5 pairs) had access to examples only on Day 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 [15], and is a standard practice in many real-world classrooms [26, 29**?**]. We, therefore, analyzed students in pairs.

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, using loops and conditionals [44]. We ranked students' performance based on the time each student spent completing 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 [27].

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*!*), and provided flexibility and variability in game design [20] (e.g., adding new actors with different roles, and designing levels), allowing students to incorporate their own choices and goals. To foster creativity, we started Days 2 & 3 by introducing a variety of breakout/space invader games, retrieved from the online Scratch community [30, 42]. We did not require any specific features in games, and encouraged students to make *unique* and *creative* artifacts.

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" [19, 45] method, a systematic approach to research "decisions" – "why they were taken, how they were implemented, and with what result" [39]. 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" [45].

Data Organization. To ensure construct validity [9], we collected and organized data following the 3 principles by Case Study Research: 1) We collected data from "**multiple sources**" [45], 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**" [45] 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**" [45] by linking interview and submissions to the case profiles of corresponding pairs, to enable tracing back/forward between different data sources and analysis stages [45].

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

1. Finding "patterns" [45] from logs. A pattern" describes the cause, effect, or events that relate to the central phenomenon of interest [45]. As log data capture most precisely students' experience compared to interviews, which may suffer from response bias and inaccuracies [10, 35], we start our analysis of cases first on their log data, a commonly-used primary data source to analyze programmers' [6], end users' [24] including novices' [21] programming experiences.

We searched for 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. Similarly, we looked 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 the repetitive requests of the same example in aggregate, creating a total of 29 sets of 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 starter code, and 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 creating 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.

2. Building "explanations" [45] from conversations, interviews and submissions. In the "building explanations" step, we aim to find evidence from students' conversations and interviews to explain students' needs and strategies. Towards this goal, we coded the conversations and interview data on each case profile to look for the presence of existing patterns, and examine whether new patterns appeared. Based on the situations of *when* students ask for examples, we used evidence from conversations and interviews to explain students' *needs*. This extra data confirmed our identified situations and strategies, adding students' perceptions of the causes and effects of their example usage needs and strategies. We used this data to re-code all case profiles the third time, confirming that students' discussions were accurate at describing their example reuse scenarios.

5 RESULTS

5.1 When do students ask for examples?

We found 5 distinct situations when students requested examples: when browsing/exploring, when starting a step, when debugging incorrect/incomplete code, when finished with a step, and when re-implementing a step. We found that many example requests (68.1%, 60/88) come from students who opened an example about a new feature *not* implemented in their code, including when browsing/exploring (38.6%, 34/88), and when starting a step (29.5%, 26/88): "Browsing/exploring" refers to when students opened and closed the example, without attempting to integrate the example into their own code. A student described in the interviews 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[2]]. "Starting a step" refers to situations where students attempted to integrate the example into their own code. Students' quotes explained their needs as to understand an implementation detail: "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% (20/88) example requests also 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% (7/88) requested examples were about features the students have already completed in their code. In these example requests, students opened the example, but did not try to integrate the example code into their own code, perhaps due to the need to confirm an already-completed step. 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 was using the example code for the purpose of **avoiding re-implementing** it on their own.

[2]A quotation from Pair 6 in the Early group. E and L denote the Early and Late groups, respectively.

5.2 How do students use examples?

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

Strategy 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, L3 separated the process of copying and reusing into 4 sub-steps, each focused on one block, shown by the arrows from the example code to the student's own code[3]. With each sub-step, they modified their code, sometimes testing it (2/4 times), before copying the next code block. During the interview, L3 explained that they "individually went into"[L3] the left code palette to copy code, and explained that "doing that allowed me to make my own modifications as I went and I better understood it."[L3].

Strategy 2: Comparison to identify key differences with the 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 be 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]. For example, E7's code attempted a feature, where the actor will change costumes one at a time. However, the costumes change instantly, so the change effect could not be observed. E7 requested an example, where the actor creates clones one at a time, and found that the example included a "wait 0.01 secs" block. E7 then added the "wait 0.01 secs" block into their own code. In this scenario, the students' code has many differences compared to the example. 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 to their own code without discarding the less meaningful differences (e.g., changing costumes).

Strategy 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 with 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 [11], which aimed to understand the example code. We call this strategy "understanding through tinkering", shown in 17.2% of the example requests. For example, L4 was confused by the code block "set size to Game Scale %", explaining in pair conversation "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 the "Game Scale"

[3]In specific, such copying is made by using the example as a reference, and moving the code from the left block palette (Figure 1).

is used in the context. So the student then changed the value of the variable from 100 to 20, and tested again, realizing that the block changes sizes of a sprite: "so, is it like, if you make it a larger number it would just get . . . ah I see."[L4] They later integrated the example by deciding on the value of "Game Scale" to be 30.

Strategy 4: Implement after closing the example. In 13.8% of example requests, students closed the example, and tried to implement a needed feature themselves. However, students did not discuss this strategy during interviews.

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 the 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.

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? We focus on evaluating integration and modification, as they demonstrate how examples were successfully and meaningfully reused [12, 43]. We calculated 6 statistics to investigate integration and modification, presented in Table 1: 1) The *frequency (freq)*, showing the percentage of strategies shown in the 29 example requests. 2) The *success (succ.) rate*, indicating the proportion of the example requests that were successfully integrated to students' code. We refer to this number as *success rate.* 3) The *addition (add.) rate*, showing the number of blocks students added while using the code example, divided by the number of blocks in the example code. 4) The *deletion (del.) rate*, showing the number of blocks students added while using the code example, divided by the number of blocks in the students' own code. 5) The *adaptation (adapt) rate*, showing the proportion of the added code blocks that are in the example code. 6) The *kept rate*, which defines the percentage of a student's final implementation of a certain feature, that comes from the example code. For addition, deletion, adaptation, and kept rates, the number presented in Table 1 is averaged across the 29 requests. We discuss the following findings based on Table 1:

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. The adaptation rate shows that the majority of students did 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.

Table 1: 4 example reuse strategies and the copy-run-debug behavior.

strategy name	freq.	succ. rate	add. rate	del. rate	adapt. rate	kept rate
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

b) **Different strategies have different use cases and affordances.** The kept rate shows how many similarities there was between the students' final code and the example code. The *tinkering* strategy creates code of the highest similarity, and has the highest addition 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 the example to students' own code, which was usually kept, showing that this strategy was most often employed in scenarios such as Figure **??**, 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 adaptation and kept rates. 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.

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 had 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 addition and kept 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 2 indicate the time duration when students were implementing this feature. We 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 the 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 2 presents the following highlights:

Students change strategies at different stages of example use. Students employed diverse strategies for a variety of reuse

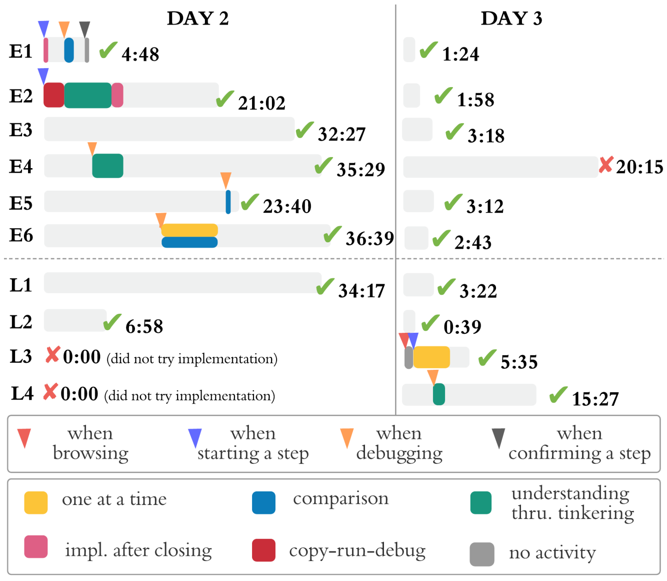


Figure 2: Case study: students' implementation of spawn_clones.

scenarios. For example, E2 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 examples again, this time spending more time modifying and testing different blocks in the example code. After they attempted to *understand the example through tinkering*, they removed all code again and started over again, this time leading to successful 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.

Program a feature the second time can be 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 the example was used. Except for 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.

6 DISCUSSION

We discuss *design implications* for designing systems with code examples to support novices' open-ended programming.
Support design and planning for creating complex projects. We found that one of the most frequent situations, when students request examples, is when they were browsing (38.6%), where students needed to find features that they wanted to create. This is consistent with prior work [24], which shows that end users needs support to identify "what I want the computer to do" [24]. This need 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 [15, 18]. Examples should, therefore, support students to create designs that feel challenging and meaningful to them.
Encourage effective example learning strategies. The primary highlight of our findings of students' strategies is the **diversity** of effective strategies. 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 [12], this leads to students' choice of strategies to overcome different types of barriers [12] – for example, to overcome **understanding barrier** (i.e., "how to use an unfamiliar code block in the example?" [12]), students may 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?" [12]), students may 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*" [25]. For example, one reason for the *integrating one block/feature at a time* strategy to be effective, is that it transforms passive code copying towards active code reconstruction, which can improve learning [8]. Such transformations can be made by programmers reconstructing blocks of examples by solving it as a Parsons problem [34], which breaks a correct solution into code pieces and asks students to rearrange the code to the original solution; or to investigate automated methods to separate examples into different sub-components, where students are required to copy, experiment, and integrate one sub-component at a time, when integrating a relatively complex example.

7 LIMITATIONS & CONCLUSION

One limitation of the work is that, we conducted our study with only 12 groups of students — we used multiple sources and prior work to triangulate our findings, and exercised rigor and reflexivity [16] when collecting and analyzing data to find valid, grounded evidence. However, we need future work to find statistical evidence (e.g., whether the strategies we found were significantly more effective than others) among larger-scaled populations.

In conclusion, in this work, we conducted a case study with 12 pairs of high school students working on open-ended game design projects, while having access to Example Aid, an example support system in Snap*!*. We identified distinct, and diverse motivations and strategies novices employ when using code examples in open-ended programming. We also found that students used a diversity of strategies to reuse and modify examples.

REFERENCES

[1] Kashif Amanullah and Tim Bell. Evaluating the use of remixing in scratch projects based on repertoire, lines of code (loc), and elementary patterns. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2019.
[2] Robert K Atkinson, Alexander Renkl, and Mary Margaret 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.
[3] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis 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] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore 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, 2002.

[6] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott 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] Brian Broll, Akos Lédeczi, Peter Volgyesi, Janos Sallai, Miklos Maroti, Alexia Carrillo, Stephanie L Weeden-Wright, Chris Vanags, Joshua D Swartz, and Melvin 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] Michelene TH Chi and Ruth Wylie. The icap framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist*, 49(4):219–243, 2014.

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

[10] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William 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.

[11] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany 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.

[12] Blinded for review. Blinded for review. Blinded for review.

[13] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. Exploring programmers' API learning processes: Collecting web resources as external memory. In Michael Homer, Felienne Hermans, Steven L. Tanimoto, and Craig 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. doi: 10.1109/VL/HCC50065.2020.9127274. URL https://doi.org/10.1109/VL/HCC50065.2020.9127274.

[14] Dan Garcia, Brian Harvey, and Tiffany Barnes. The beauty and joy of computing. *ACM Inroads*, 6(4):71–79, 2015.

[15] Shuchi Grover, Satabdi Basu, and Patricia 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 999–1004, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159522. URL https://doi.org/10.1145/3159450.3159522.

[16] Marilys Guillemin and Lynn Gillam. Ethics, reflexivity, and "ethically important moments" in research. *Qualitative inquiry*, 10(2):261–280, 2004.

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

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

[19] Lorna Hamilton and Connie Corbett-Whittier. *Using case study in education research*. Sage, 2012.

[20] Robin Hunicke, Marc LeBlanc, and Robert 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.

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

[22] Michelle Ichinco, Wint Yee Hnin, and Caitlin 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.

[23] Prapti Khawas, Peeratham Techapalokul, and Eli Tilevich. Unmixing remixes: The how and why of not starting projects from scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 169–173. IEEE, 2019.

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

[25] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

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

[27] Colleen M Lewis and Niral 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.

[28] Nicholas Lytle, Veronica Cateté, Danielle Boulden, Yihuan Dong, Jennifer Houchins, Alexandra Milliken, Amy Isvik, Dolly Bounajim, Eric Wiebe, and Tiffany 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.

[29] Nicholas Lytle, Alexandra Milliken, Veronica Cateté, and Tiffany 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 832–838, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366943. URL https://doi-org.prox.lib.ncsu.edu/10.1145/3328778.3366943.

[30] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[31] Steven McGee, Randi McGee-Tekula, Jennifer Duck, Catherine McGee, Lucia Dettori, Ronald I. Greenberg, Eric Snow, Daisy Rutstein, Dale Reed, Brenda Wilkerson, Don Yanek, Andrew M. Rasmussen, and Dennis 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 50–55, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159529. URL https://doi.org/10.1145/3159450.3159529.

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

[33] Briana B Morrison, Lauren E Margulieux, and Mark 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.

[34] Dale Parsons and Patricia 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.

[35] Delroy L Paulhus. Measurement and control of response bias. 1991.

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

[37] Peter Pirolli and Mimi Recker. Learning strategies and transfer in the domain of programming. *ITLS Faculty Publications*, 12, 09 1994. doi: 10.1207/s1532690xci1203_2.

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

[39] Wilbur Schramm. Notes on case studies of instructional media projects. 1971.

[40] Benjamin Shih, Kenneth Koedinger, and Richard Scheines. A response-time model for bottom-out hints as worked examples. pages 117–126, 01 2008. doi: 10.1201/b10274-17.

[41] John Gregory Trafton and Brian J Reiser. *The contributions of studying examples and solving problems to skill acquisition*. PhD thesis, Citeseer, 1994.

[42] Wengran Wang, Yudong Rao, Yang Shi, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas W. Price. Comparing feature engineering approaches to predict complex programming behaviors. *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20*, 2020.

[43] Wengran Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas Price. The step tutor: Supporting students through step-by-step example-based feedback. *ITiCSE'20 - Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, To be published*, pages 391–397, 2020.

[44] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W. Price. Crescendo: Engaging students to self-paced programming practices. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 859–865, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366919. URL https://doi.org/10.1145/3328778.3366919.

[45] K Yin Robert. Case study research and applications: design and methods, 2017.

[46] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min 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.