Designing Template-Based Examples To Inspire Novices' Creative Code Reuse in Open-Ended Programming

Open-ended programming motivates and engages students by connecting their real-world experience and personal interest with their self-choice programming projects, such as creating apps or games. However, such open-ended programming tasks can be challenging, as they require students to implement features that they may be unfamiliar with. We present Example Aid, which supports students with gallery-based code examples during open-ended programming. The Example Aid is designed to address four learning barriers students encounter when using code examples during open-ended programming: decision, search, and testing barrier. We evaluated Example Aid through a classroom study over 46 students, and found that Example Aid achieved its goals by encouraging iterative prototyping and affording multiple types of strategies.

ACM Reference Format:

. 2021. Designing Template-Based Examples To Inspire Novices' Creative Code Reuse in Open-Ended Programming. 1, 1 (July 2021), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Open-ended programming projects, such as creating apps, games, and stories, encourages students to create projects of their own choice. These projects are widely applied in many introductory programming curricula [4, 5, 7, 17] and informal learning settings [21]. They engage students by allowing them to express ideas creatively [12], and motivate students to keep pursuing CS [9] by tying their authentic, real-world interest with their programming experience [20]. However, open-ended programming can also be challenging for novices [7], as implementing unique and authentic ideas may require knowledge of programming concepts and APIs that they were unfamiliar with [4, 7].

Offering students code examples in the middle of open-ended programming has been shown to help students generate ideas, build code, debug, and confirm their own code [4]. However, novices encounter a number of barriers when using code examples, such as not knowing how to explain an example they want; difficulties to understand and reuse unfamiliar code blocks; and difficulties to modify and integrate examples to their own code [4, 13]. In addition, many students need to modify examples to use them in their own code, but encounter barriers in doing so [4]. This raises questions about how to create examples and build interfaces that most effectively assists these novice students, and to understand how students modify code examples to use them in their own project.

Based on students' needs and barriers, discovered by prior work [4], we present our design process of building Example Aid[1], a system that uses code examples to support creative, open-ended programming. The goal of Example Aid is to support creative, open-ended programming by helping students to modify and reuse examples for their own needs. To understand how Example Aid achieves its goal to support students to reuse code examples, we conducted a study with 46 novice students working with Example Aid in Snap*!*, a block-based programming environment.

[1]Name changed for anonymity.

2 RELATED WORK

Open-Ended Programming. Open-ended programming typically requires students to create an app, game or simulation based on their own choices [7]. Such open-ended projects are important components of project-based learning, which situated the learning process in the middle of students' self-choice problem-solving and artifact-building experience [2], and are therefore commonly seen among introductory programming curricula [5, 7, 17]. However, prior work has shown that novices struggle to create "logically coherent"[18] programming components when building open-ended programming projects, or struggle to apply knowledge of existing concepts (e.g., variables) into code [7], or to implement new concepts or APIs [4].

Exploratory Programming Behaviors. The first step towards learner-centric designs for building tools is to understand students' own needs and practices [8]. Novices' open-ended programming practice is a type of exploratory programming, which is defined as practices of which the goal is "open-ended", and "evolves through the process of programming"[15]. Different from programming tasks with a fixed goal or specification, explorative programming typically includes a number of exploration/ experimentation-based activities, such as bricolage, tinkering, sketching and hacking [1, 15]. In a systematic literature review across various types of exploratory programming practices, Kery and Myers summarized that, different from non-explorative, specification-based programming, in exploratory programming, programmers engage in the following three key types of distinguishing activities [15]: 1) *Opportunistic programming,* where programmers rely heavily on code examples found from online resources, and often use functionalities such as copy-and-paste to patch together example code into their own program [3]. 2) *Debugging into existence:* After directly copying code found from online resources, programmers debug those code until they work correctly in their own program [25]; and 3) *Rapid prototyping,* where programmers iteratively create, test and experiment a prototype at an early stage of the programming process [11, 16]. Based on these key distinguishing activities, Kery and Myers suggested building tools to support exploration and experimentation among exploratory programmers [15].

Code Examples. Researchers have built various code example systems to support novices, many towards supporting closed-ended programming tasks [6, 28, 29], such as by giving students a correct student's solution when students request help [29], or by separating program completion into different individual steps [28]. By using such tools during the completion of closed-ended tasks, novices were shown to be able to complete tasks faster [29]. However, few prior work has built tools to specifically target students' exploration and experimentation during open-ended programming.

In our prior work, we had developed the first version of Example Aid that targets supporting open-ended programming, by offering a gallery of code examples. Students can search, browse and select code examples to view through a gallery of gif-animations of the code. By clicking on a gif animation, the student can view a non-editable code window to see the example code, and can drag to copy the example code into their own code, or to move it aside and use it as a reference to build their own code [4]. However, a study among 44 undergraduate novice students has found that students encounter a number of barriers when using these code examples in their own code, such as reluctance towards opening an example; difficulties to find an example they need; and barriers to test and modify the example immediately. Informed by the difficulties identified by our prior work, we designed the current version of Example Aid to specifically address these barriers.

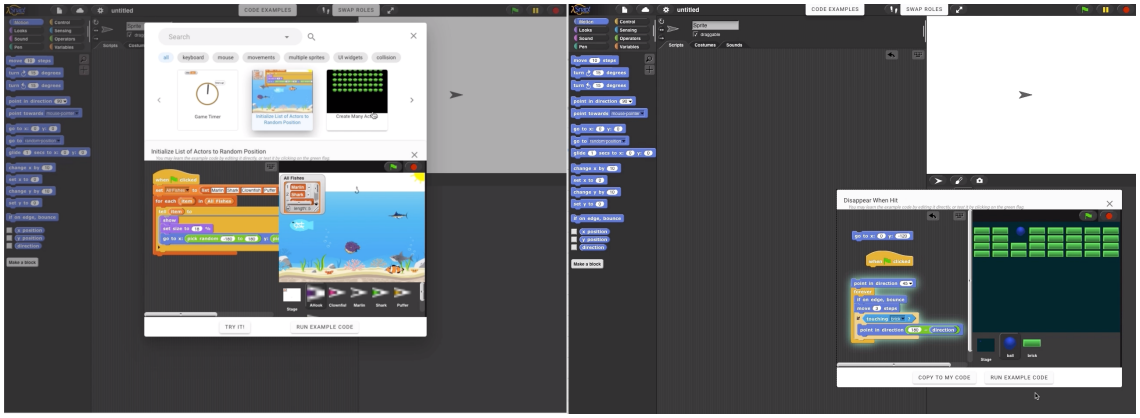

Fig. 1. The Example Aid interface, which includes a selection-based gallery (left) and a "try it" view for students to program while using the example as a reference.

3 THE EXAMPLE AID SYSTEM

3.1 The Example Aid Interface

Fig. 1 shows the interface of Example Aid. When a student needs a new idea, or is stuck on implementing an existing idea, they can click on a show example button on the top-center of the screen to open a gallery of code examples (Fig. 1-a. Inside the gallery, they can use the search-box or the tags to find an example, or click through the left-right arrows to browse through the gif-animations of the output of each code example. When a student clicks on a code example, it opens up a preview window, which shows editable code with its output shown on the right side of the example code. The student can modify the example code, and click on the button "Run Example Code" or the green flag shown on the top right of the example to run and test the example code. If the student wants to use the example in their own code, they can click on the "try it" button shown on the bottom-left of the interface. After clicking on the "try it" button, the student is prompted with a new "playground" window, where they can continue to edit and test the example, or use the example code as a reference to implement their own code. They can also click on the "Copy to my code" button on the bottom-left of the example, which prompts them to copy the example code to their own code. The design of Example Aid is informed by the following 3 design considerations:

1) **Incentivise ideation.** One key activities exploratory programmers engage in is exploring and discovering new in the middle of programming. In our prior work [4], we have found that most example requests (68.1%) in open-ended programming come from students who started with no relevant code about this behavior, and needed ideas about what program behaviors to use in their code, and how to implement it. Example Aid incentivises ideation by showing the animation of code examples at the gallery, with a one-phrase headline of what the example does. It also allows students to search, browse examples easily.

2) **Encourage prototyping.** Prior work on exploratory programming [15] shows that programmers need different types of experimentations to implement and test new ideas [11, 15, 16]. Our prior work showed that students needs quick, easy ways to run and experiment with the example, and need multiple modification and test cycles (i.e., modify-run-adjust) to use examples effectively in their own code, but were unable to do so efficiently in the first version of

Barrier	Definition	Previous Version	Current Version	Design principle
Search Barrier	Students' typed queries sometimes don't return search results	No query recommendations.	Provides immediate search results and autocomplete suggestions.	Incentivise Ideation
Decision Barrier	Students are reluctant to open an example even when knowing it may help	Opening examples always cause the search window to close, causing the example to be less accessible.	Allows previewing and testing the example in the browsing interface.	
Testing Barrier	Students need to experiment with the example when reading it	Example window shows no output, does not allow testing inside the example window	Allows running, modifying and viewing immediate output inside the example window.	Encourage prototyping

Table 1. Example Aid design targets to address the search, decision, and testing barriers students encounter when using code examples during open-ended programming.

Example Aid [4]. Example Aid encourages prototyping by allowing students to experiment and modify the example and view its immediate output on the right output stage, as a single, standalone prototype.

In specific, the current version of targets the four types of barriers students encounter when using the previous version of Example Aid during open-ended programming: decision, search, testing, and modification, listed in Table 1.

3.2 Example Content Design

While programming environments such as Snap*!* [19] and Scratch [22] offers existing programs for students to use, prior work has shown that novices [14] and experienced programmers [23] preferred using "snippet-sized" examples that teach an API usage pattern – how code can be organized to produce a certain behavior[24, 26]. As different students' implementations are drastically different [27], it is challenging to cluster a common implementation that exist across multiple programs. We therefore designed example content through the manual process of decomposing steps towards completing a large programming problem.

To do that, we first collected 27 pieces of CS0 students' project submissions, where students worked on creating open-ended game designs in Snap*!*. We systematically coded all submissions on dimensions such as game mechanics, code quality, and project aesthetics, and listed features that each submission include. We found a total of 37 code usage patterns in student programs. Among them, 7 were shown in at least 2 student programs (e.g., move with keyboard, display and initialize a variable, initialize actor positions), the rest of features were only present in one student programs. Despite being the less requested examples, they were still important programming behaviors that students may benefit from (e.g., implementing a jump in Snap*!* [27]). In addition, we found that students' projects also avoided using advanced code blocks (e.g., lists and clones) that may have been helpful for them to create clean and concise code; and their code sometimes include logic errors. Leveraging the collection of code usage patterns we found from this formative analysis, we built 18 sample programs to cover all behaviors (one program can cover multiple behaviors), with common game themes that students may be familiar with (e.g., a quiz app, or an arcade

game). We next decomposed those programs into examples that students may follow to build their code. This created a total of 52 examples. Prior work showed that examples with multiple unfamiliar blocks or unnecessarily complex can be challenging for students [4], we therefore removed those examples, creating a total of 31 examples.

Example Decomposition We decomposed sample programs code examples that represent distinct program behaviors, where a program behavior is defined as a distinct feature that leads to meaningful behaviors semantically (i.e., can be described in short human language) [27]. Such behaviors can be implemented in a variety of ways, and the number of code blocks that are needed to complete these behaviors range from approximately 10 to 50 blocks of code [27]. For example, a space invader game can be decomposed into the following 6 examples: 1) actor moves with key; 2) creating a spawn of enemies; 3) enemy moves intermittently; 4) shoot actors; 5) an enemy explode when hitting bullet; 6) increases score when bullet hits enemy. While most examples were distinct from one another (i.e., does not involve the same code) (e.g., 1-5), some examples can be built on a similar set of starter code (e.g., 5 & 6).

4 STUDY SETUP

4.1 Participants & Procedure

To understand students' experience using Example Aid, we conducted our study in an undergraduate CS0 classroom, among X non-CS-major novice students, in a research university in Southeast US. The course was held remotely during the COVID-19 pandemic.

To directly compare the effectiveness of Example Aid in comparison to its previous deployments, the study applied the same procedure as in our prior work [4], in the same CS0 course during a subsequent semester. The study happened during the second month of students' programming course, and includes the following steps:

Project pitch. To guide students towards designing a free-choice, open-ended programming project, we introduced students to the engineering design process [10]. They were asked to design their products based following this design process, to solve a real-world problem with creative ideas, and publish a project pitch in the online class discussion platform, which allows follow-up discussions of each pitch.

Pair planning and programming After the project pitches and follow-up discussions, students had the choice to form a two-person team to on a project idea that they were both interested in. They could also choose to work independently on their own project. This creates 36 student groups, among which, 10 were pairs and 26 were students who work independently [2]. After forming groups, students started with a week of planning, and then worked on their projects for two weeks. To allow a pair to collaboratively program, we instrumented the Snap*!* interface with a "save/load" button, on which students can click to save/load their/their pair's project. We encouraged pair programming because prior work has shown that students achieved significantly higher scores in pair projects when creating open-ended programming projects [7].

4.2 Data & Analysis

We focus our analysis based on the log data collected during students' interaction with code examples, such as open, search, and code editing. To begin with, Three researchers manually inspected studentsâĂŹ logs from 16 example requests on one randomly-selected student group, to describe actions students take while using the example. Next, one researcher collected all events summarized in the manual inspection stage, and created a shortlist of all types of events with definitions. One researcher trained another researcher to conduct the log analysis. They next each did

[2]We use the term "group" to refer to single-student or pairs, who worked on a single project

independent coding on 10% of the data (7 combined requests), achieving initial inter-rater reliability of 82.8%. The two researchers then discussed to resolve conflicts and refine definitions, achieving final inter-rater reliability of 100%. After resolving differences, the second researcher conducted the rest of the log analysis based on the definition. The number of example requests in the manual log analysis was then compared to the number of example requests in the actual log, and found 24 example requests were inconsistent between the actual log. One researcher re-coded the 24 example requests to finalize the results. We present the collected event sequences in Section 5

5 RESULTS & DISCUSSION

The goal of Example Aid was to address the three types of learning challenges students encounter when using code examples provided by its previous version, during open-ended programming, by offering supports for ideation and prototyping. We next discuss to what extent the Example Aid achieved its design goals, focusing on the following research question: to what extent does Example Aid address the types of barriers students encounter when using code examples during open-ended programming? Example Aid was designed to address 3 types of barriers: decision, search, and testing (Fig. 1). We discuss each below.

Among the 36 student groups, 17 groups have clicked on the "show example" button to open the gallery and use the Example Aid, which is lower than 27/31 from the previous semester [4]. This is because during this semester, the instructor did not introduce the interface to the students. Many students may be unaware of the existence of the interface. While this shows that lack of salience of the interface introduces hurdles towards students' use of examples, we next discuss to what extent the different components inside the interface may have reduced the Search, Decision and Integration Barriers, only among students who actually used the Example Aid.

Search barrier. In the previous version of Example Aid, only 48% percent of student search queries returned results. In the current version, 85.2% (29/34) search returned results, as auto-complete suggestions show students search findings when typing, and therefore prompt students to complete with queries that lead to found examples.

Decision barrier. In our prior work, we found that students may decide to not open an example even when they are stuck and need help, as the example were less accessible to the students – students cannot briefly preview the example, to view an example, they had to close the gallery and view it in a separate window. We found students who used Example Aid's previous version from prior semester opened the gallery with an average of 5.67 times (153/27). However, in this version of Example Aid, each student who used Example Aid opened the gallery an average of 16.8 (286/17) times, showing that students were more willing to view an example when being supported with an accessible preview window.

Testing barrier. One barrier students encountered with previous version of Example Aid was difficulties to test and experiment the example, as the previous version only allows students to view an example without tinkering it inside the interface. Table 2, on the other hand, shows the variety of approaches students adopted to experiment with the example: the "#Requests" column shows the appearance of each event inside each example request. For example, among the total of 214 example requests, 54 requests included opening the example in the playground window, 61 included running the example at least once. Similarly, "#Students" column revealed the number of students who did this event. Table 2 shows that students engage in multiple types of tinkering activities when opening an example, such as running (85%, 12/14), modifying the example (57%, 8/14), and open documentation to learn certain blocks (28.6, 4/14).

Students' experimentation with the interface also allowed them to integrate the example to their own code more successfully. After experimenting with the interface, we found students used multiple different approaches to copy the

Event Name	Definition	#Requests	#Students
openExample	When students click to open an example inside the gallery	214	14
openPlayground	When students click the "try it" button to open the playground window	54	7
openDocumentation	When students open a "help" documentation on certain blocks	4	4
runExample	When students run the code inside the example window, either in gallery preview or in the playground	61	12
modifyExample	When students modify the example code, either in gallery preview or in the playground	13	8
copyLineByLine	When students use the example playground code as a reference and build code line by line	3	2
clickCopying	When students click on the button "copy to my code" to copy the example code to their own code	39	7
closeAndImplement	When students close the example window and build code from scratch themselves	32	9

Table 2. The number of occurrence of each event when students interact with the example.

example code into their own code, such as by using the example as a reference and build code line by line (14%, 2/14), by clicking on the "copy to my code" button to copy the example code to their own code (50%, 7/14). In addition, many students (64%, 9/14) closed the example and then implemented the example code on their own. This led to an average of 1.65 (28/17) examples were integrated by each student who used the Example Aid in this current semester, much higher than the average of 0.7 examples finally integrated into student programs (19/27) from the previous semester, showing that Example Aid not only helped students to experiment with the interface to test, modify an example, but also helped them to integrate an example into their own code.

————————————————————————————————————limitation is just an outline now.

6 LIMITATIONS

Our breakdown and template-based examples allowed opportunities for a variety of reuse scenarios (otherwise, students who use a full example will be difficult to identify where to reuse) Simple browse & search interface caused students to unable to find code examples that are project or theme-specific. (students only look for generic examples).

Many students still did not use the example to begin with Our search autocompletion did not solve students' real search barrier.

REFERENCES

[1] Ilias Bergström and Alan F Blackwell. The practices of programming. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 190–198. IEEE, 2016.

[2] Phyllis C Blumenfeld, Elliot Soloway, Ronald W Marx, Joseph S Krajcik, Mark Guzdial, and Annemarie Palincsar. Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational psychologist*, 26(3-4):369–398, 1991.

[3] 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.

[4] Blinded for review. Blinded for review. Blinded for review.

[5] Dan Garcia, Brian Harvey, and Tiffany Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[6] Sebastian Gross, Bassam Mokbel, Benjamin Paassen, Barbara Hammer, and Niels Pinkwart. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10*, 9(3):248–280, 2014.

[7] 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 999âĂŞ1004, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159522. URL https://doi.org/10.1145/3159450.3159522.

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

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

[10] Yousef Haik, Sangarappillai Sivaloganathan, and Tamer Shahin. *Engineering design process*. Nelson Education, 2018.

[11] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 91–100, 2008.

[12] Carol Hulls, Chris Rennick, Sanjeev Bedi, Mary Robinson, and William Melek. The use of an open-ended project to improve the student experience in first year programming. *Proceedings of the Canadian Engineering Education Association (CEEA)*, 2015.

[13] 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.

[14] 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.

[15] Mary Beth Kery and Brad A Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29. IEEE, 2017.

[16] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3025453–3025626, 2017.

[17] 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 50âĂŞ55, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351034. doi: 10.1145/3159450.3159529. URL https://doi.org/10.1145/3159450.3159529.

[18] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 168–172, 2011.

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

[20] Seymour Papert. Mindstorms: Computers, children, and powerful ideas. *NY: Basic Books*, page 255, 1980.

[21] 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.

[22] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

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

[24] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2012.

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

[26] Kyle Thayer, Sarah E Chasins, and Amy J Ko. A theory of robust api knowledge. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–32, 2021.

[27] 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. 2020.

[28] Wengran Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas W Price. Step tutor: Supporting students through step-by-step example-based feedback. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 391–397, 2020.

[29] 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, 2019.