Exploring Design Choices to Support Novices' Example Use During Creative Open-Ended Programming

Wengran Wang
North Carolina State University
Raleigh, USA

Audrey Le Meur
North Carolina State University
Raleigh, USA

Mahesh Bobbadi
North Carolina State University
Raleigh, USA

Bita Akram
North Carolina State University
Raleigh, USA

Tiffany Barnes
North Carolina State University
Raleigh, USA

Chris Martens
North Carolina State University
Raleigh, NC, USA

Thomas Price
North Carolina State University
Raleigh, USA

ABSTRACT

Open-ended programming engages students by connecting computing with their real-world experience and personal interest. However, such open-ended programming tasks can be challenging, as they require students to implement features that they may be unfamiliar with. Code examples help students to generate ideas and implement program features, but students also encounter many learning barriers when using them. We explore how to design code examples to support novices' effective example use by presenting our experience of building and deploying Example Helper, a system that supports students with a gallery of code examples during open-ended programming. We deployed Example Helper in an undergraduate CS0 classroom to investigate students' example usage experience, finding that students used different strategies to browse, understand, experiment with, and integrate code examples, and that students who make more sophisticated plans also used more examples in their projects.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; • **Social and professional topics → Computing education**.

KEYWORDS

open-ended programming, code examples, novice programming

ACM Reference Format:
Wengran Wang, Audrey Le Meur, Mahesh Bobbadi, Bita Akram, Tiffany Barnes, Chris Martens, and Thomas Price. 2022. Exploring Design Choices to Support Novices' Example Use During Creative 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/10.1145/3478431.3499374

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499374

1 INTRODUCTION

Open-ended programming projects, such as making apps, games, and stories, encourage students to create projects that are aligned with their own motivation and interests [7]. These projects are widely used as activities and assignments in many introductory programming curricula [5, 7, 18] and informal learning settings [23]. 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 [21]. 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 they are unfamiliar with [7].

Code examples are often used by professional programmers to learn and use APIs and code usage patterns [2, 22, 25]. However, novice programmers lack skills such as program tracing [17] and fundamental programming concepts such as variables [13], which may prevent them from using those examples effectively during open-ended programming. In our prior work, we conducted the first known study to systematically analyze the types of barriers students encounter when using code examples during open-ended programming, using a basic example system (which we refer to as Prototype-EH in this paper). We found that students encountered barriers such as not knowing when to use an example (decision barrier); how to find an example they need (search barrier) and how to test and experiment with the examples (testing barrier) [31].

How to design code examples to address students' decision, search, and testing barriers? In this work, we describe our experience designing, building and deploying Example Helper, a fully remodeled example support system based on Prototype-EH. Example Helper supports students' open-ended programming with a gallery of code examples. We explored design choices to encourage students' exploration and experimentation with code examples. We deployed Example Helper in an undergraduate CS0 course, with 46 novice students working on an open-ended programming project in Snap*!*, a block-based programming environment. We analyzed students' programming log data, project plans, and project submissions. We

found that students used many different strategies to browse, understand, experiment with, and integrate code examples into their code. We also found a significant, positive correlation between the complexity of a student's project plans and the number of integrated examples, showing that students who had more ambitious project goals used more code examples. Finally, we discuss to what extent Example Helper addressed the decision, search and testing barriers, and suggest ways to better support students' example use. The contributions of this work are:

(1) A synthesis of design choices for building code example systems to address novices' learning barriers, and for enabling effective example use during open-ended programming.
(2) Example Helper, a system that instantiates the principles for providing code examples to students.
(3) An in-depth analysis of students' example-usage experience, as well as the factors that influenced students' example use, in an authentic, classroom study.

2 RELATED WORK

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, exploratory programming typically includes many 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 program [2]. 2) *Debugging into existence:* After directly copying code found from online resources, programmers debug those code until they work correctly in their program [27]; and 3) *Rapid prototyping,* where programmers iteratively create, test, and experiment with 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. Code example systems for novices mostly support closed-ended programming tasks [6, 30, 32], such as by giving students a correct student's solution when students request help [32], or by separating program completion into different individual steps [30]. By using such tools during the completion of closed-ended tasks, novices were shown to be able to complete tasks faster [32]. However, few prior works have built tools to specifically target students' exploration and experimentation during open-ended programming.

As a first step, in our prior work, we conducted a pilot study to explore students' learning barriers when using a basic example support system (Prototype-EH). Prototype-EH offers a gallery of code examples, where students can search and browse through gif animations of examples in a gallery. By clicking on a gif, the student can view a non-editable example code window, and can drag to copy the example code into their code, or to move it aside and use it as a reference to build their own code [31]. We investigated students' learning barriers, and found that students can feel a lack of motivation to use an example even when they needed help (decision barrier); they may not know how to explain a needed example, and instead type in text that returns no found results (search barrier). Students may also need, but be unable to, test or experiment with an example immediately when opening it (testing barrier) [31]. These insights led to the design of Example Helper, which is built to address these barriers.

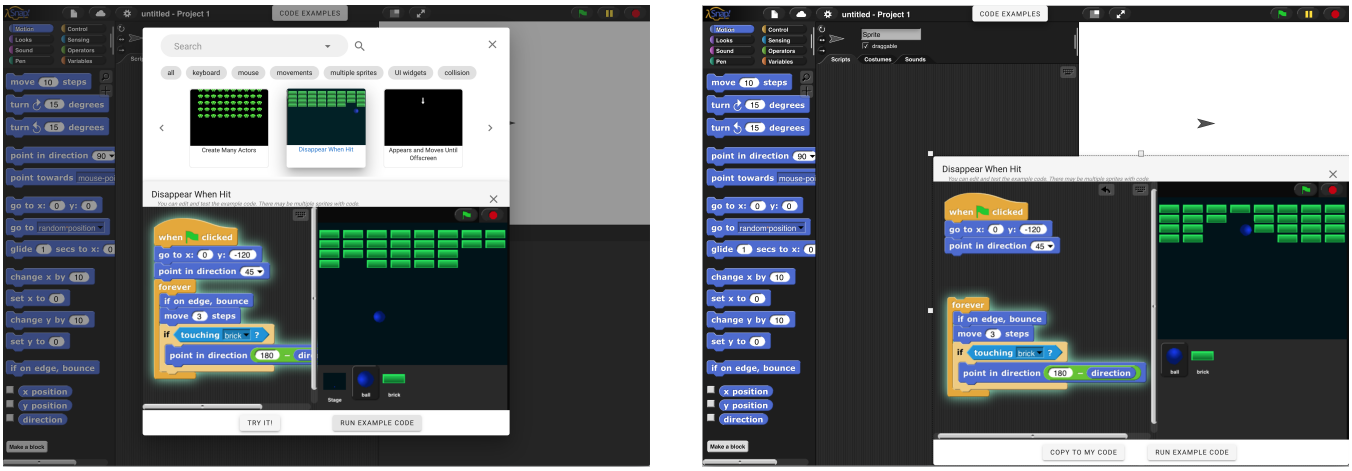
3 THE EXAMPLE HELPER SYSTEM

3.1 Interface Design

Figure 1 shows the interface of Example Helper. 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 (Figure 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 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 on the bottom-left of the interface. After clicking on the "try it" button, the student is prompted with a new "playground" window (Figure 1-b), 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 Helper is informed by the following two design choices, to address the 3 learning barriers from prior work (Table 1) [31].

1) **Incentivise ideation.** A key activity exploratory programmers engage in is exploring and discovering new ideas in the middle of programming [15]. In addition to the support for browsing and viewing gif animations, Example Helper added two more features to support ideation and exploration of examples: 1) Autocompletion suggestions when searching for an example. The search mechanism matches students' search with words in the name of an example, and instantly provides autocomplete suggestions, showing potential items a student needs; 2) Preview window. Whenever a student clicks on an example, they can view edit, test, and run the example in the preview window (shown in Figure 1 left). The goal of this feature is to address decision barriers, as we hypothesized that with easier access to the preview, students would become more willing to view and test an example they need.

2) **Encourage prototyping.** Prior work shows that exploratory programmers experiment with the code to implement and test new ideas [11, 15, 16]. Our prior work showed that students needed immediate, straightforward ways to experiment with the example,



(a) gallery (b) playground

Figure 1: The Example Helper interface, which includes a selection-based gallery (a) and a playground view (b).

and need multiple modifications and test cycles to use examples effectively in their code, but was unable to do so efficiently in Prototype-EH [31]. Example Helper 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.

3.2 Example Content Design

While the Snap*!* website [20] offers galleries of *complete projects* for students to browse, prior work has shown that novices [14] and experienced programmers [25] preferred using "snippet-sized" examples that teach an API usage pattern – how code can be organized to produce a certain behavior[26, 28]. We designed example content through a manual process of decomposing steps towards completing multiple large programming projects.

To do that, we first collected 27 pieces of CS0 students' project submissions, where students did open-ended programming in Snap*!*. We systematically coded all submissions on dimensions such as game mechanics, code quality, and project aesthetics, and listed features that each submission included. We found a total of 37 code usage patterns in student programs, such as moving with the keyboard, displaying and initializing a variable, and initializing actor positions. 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 included 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 include multiple behaviors), with known game themes that students may be familiar with (e.g., a quiz app, or an arcade game).

We next decomposed sample programs into code examples that represent distinct program behaviors, which should be meaningful semantically, and can be described in short human language [29]. 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 a bullet hits an enemy. After constructing those examples, we did multiple passes to break down long examples into smaller sub-components, merged examples that are of similar functionalities, and filtered out examples that include a large number of code blocks and could not break down into sub-components. This creates a total of 31 examples.

4 METHODS

We conducted a student study to understand how students used the Example Helper in a real-world classroom environment. To generate a comprehensive, in-depth understanding of students' experience, we used the following three research questions, each with increasing specificity, to guide our study and analysis.

- **RQ1**: How did students use examples? We aim to identify the types of behaviors and strategies students engaged with when using code examples.
- **RQ2**: What types of students used examples? We aim to look at student-specific factors that may influence students' example use.
- **RQ3**: To what extent did the new features introduced by Example Helper address students' learning barriers? We used data collected from the study to qualitatively evaluate whether the specific features we added are useful in helping students overcome barriers.

4.1 Participants & Procedure

We conducted our study in an undergraduate CS0 classroom, among 46 non-CS-major novice students, in a research university in Southeast US. The course was held remotely during the COVID-19 pandemic. We did not collect students' demographic information. The study happened during the second month of the students' programming course, and includes the following procedure:

Barrier	Definition	Prototype-EH	Example Helper	Design choice
Search Barrier	Students' typed queries sometimes did not 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 stuck and need help.	No preview window.	Allows previewing and testing the example in the browsing interface.	
Testing Barrier	Students need quick, iterative experimentations with the example.	No interactive output. Does not support testing/experimentation inside the example window.	Allows running, modifying, and viewing immediate output inside the example window.	Encourage Prototyping

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

Pre-test. Before the study, students completed a pre-test, which tested students' knowledge on concepts they learned in the first month before the study: variables, lists, loops, and Snap*!* APIs.

Project pitch. To guide students towards designing a free-choice, open-ended project, the instructor introduced students to the engineering design process [10]. They were asked to design their projects to solve a real-world problem with creative ideas, and to publish a project pitch in the online class discussion platform, which allows for 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 on a project idea that they were both interested in. They could also choose to work independently. This led to 36 student groups, among which, 10 were pairs and 26 were students who worked independently[1]. After forming groups, students started with a week of planning in the PlanIT digital planning system [19], where they listed the features they wanted to complete in their project (e.g., "once the snake crashes into itself the game is over"), as well as a project description, and then worked on their projects for two weeks. To allow collaborative programming, we instrumented the Snap*!* interface with a "save/load" button, on which students could click to save/load their/their pair's project. We encouraged pair programming, as prior work has shown that students achieved significantly higher performance in pair projects when creating open-ended projects [7].

4.2 Data & Analysis

We conducted the following three types of data collection & analysis to investigate our research questions:

Interaction with code examples. Example Helper logs all students' interaction data with the system, as well as their code snapshots at every individual timestamp. To investigate RQ1 and RQ3 on students' experience using Example Helper, we conducted a qualitative coding of the log data to generate patterns of interaction behaviors students engaged in when using examples [4]. To begin with, three researchers manually inspected students' logs from 16 example requests[2] on one randomly-selected student group, to describe actions students take while using the example, creating 3 note documents on example-related activities, such as running the example code or modifying the code in the playground. Next, one researcher developed an initial code book, which includes a list of example interaction events that took place, with definitions. Based on the code book, two researchers coded all students' log data to confirm and collect counts on those events. They first each did independent coding on 10% of the data based on the initial list of events, achieving an inter-rater agreement of 82.8%. They next discussed to resolve conflicts and refined the code book, achieving a final inter-rater agreement of 100%. Based on the new refined definition, the second researcher conducted the rest of the log analysis. At the end of the log analysis, the two researchers then inspected the events, merged events that describe similar usage behaviors (e.g., running example in the preview and running in the playground), and grouping codes into themes. This produced 3 high-level themes and 8 example-usage events. We present them in Section 5.1.

Pretest, planned & completed features. To investigate RQ2, we hypothesized that students' programming knowledge, or the complexity of their plans may be related to students' example use. Therefore, we collected students' pre-test scores as an indicator of students' programming prior knowledge. We also collected students' planning data by collecting the list of features they planned in the digital planning system [19]. Some students included extra features in the project description text field. For those student groups, we added from the project description each sentence that describes an extra planned feature into the planned feature list. We used the number of features students included in their plans to indicate the complexity of their plans, and rated students' project submissions based on the number of planned features students ended up completing in their projects. If a student slightly changed a feature's implementation (e.g., by changing variable names), we also marked those features as completed.

Example integration. To understand the outcome of using examples, for each example a group has requested, we also inspected the corresponding log data to check whether the example was successfully integrated to a student's code. We define "integration" as when a student used an example in their projects and kept it in their projects for submissions. To inspect how students modified the examples during integration, two researchers collectively rated the level of modifications students used when integrating an example to their project, based on the following three different levels of adaptations: 1) full copy, where students copied the entire example with no modifications; 2) slight modification, where students only modified the examples slightly, such as changing variable names and initialization logic. 3) structural modification, where students made bigger changes to the events, either deleting many blocks

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

[2] An example request includes all log data when a student opened, tested, closed or used an example.

they did not need, or modifying many blocks to use them in their projects. We next rated students' integrated examples according to the level of modifications, and present the result in Section 5.3.

5 RESULTS & DISCUSSION

5.1 RQ1: How did students use examples?

Our analysis revealed 3 high-level themes of students' example interaction behaviors: experimentation, integration, and other general example usage behaviors. General example usage behaviors described generic example usage events, including opening an example (14 students), clicking on the "try it" button to open playground (7 students), and opening documentation to learn unfamiliar code blocks in an example (4 students). This shows that some students could not understand code blocks in the example, but used the documentations to learn instead. We next present students' experimentation and integration behaviors when using examples:

5.1.1 Experimentation behaviors. Experimentation described how students test, tinker, or modify the example inside the preview or playground window. We found that among the 14 groups who opened an example, most groups (85%, 12/14) tested the example code in the example window, and over half of the groups (57%, 8/14) modified the example inside the preview or playground window to test. This shows that many students made use of from the "immediate test and experimentation" features.

5.1.2 Integration behaviors. Integration behavior describes how students applied and used the example in their workspace. Our log analysis found three key integration behaviors: using the example as a reference and building code themselves (reference; 14%, 2/14); clicking on the "copy to my code" button to copy code directly (copy; 50%, 7/14); or closing the example and then implementing the example code on their own (re-implement; 64%, 9/14).

5.1.3 Use cases. We illustrate below how students used the experimentation and integration strategies to understand and reuse an example. We demonstrate how three different students used the "Move when the key is pressed" (*keymove*) example. *Keymove* demonstrates how to move actors in response to a user's key presses. The example code uses a forever loop to listen to user inputs (i.e., left and right keys) and move the actor position accordingly.

Copy-run-modify. After failing to implement the example themselves, Bo [3] copied the example directly to their code by clicking on the "copy to my code" button. They then ran the example code 4 times and modified the example by adding up and down movement on their respective keys. The student ran their code four more times to test the added behavior.

Run-understand-reference. Mo already had incomplete code for a *keymove* behavior before looking for examples. They browsed several examples and then opened *keymove*. After running the example several times in the gallery, they then opened the playground. Instead of copying the code directly, they used the example code as a reference and built the example one block at a time in their workspace.

Run-close-reimplement. Jo requested the *keymove* example, ran it once, then closed the example. They then re-implemented a modified version of the example which allowed users to use either the up arrow or the w key and controlled the sprite's direction rather than position.

5.1.4 Discussion. Our use case shows three different types of *opportunistic programming* strategies, summarized by prior work [2, 15]. Among them, *Copy-run-modify* is similar to the *debugging into existence* behavior [15, 27], where students engaged in iterative test and modification to update an existing program. This shows the potential for the Example Helper to address the testing barriers encountered by students from our prior work [31].

5.2 RQ2: Who used examples?

We found that only 22% (8/36) students integrated at least one example into their project. Many (61.1%, 22/36) did not view any examples. Therefore, we investigate what types of students were more likely to use Example Helper to integrate examples into their projects. We hypothesized that students' programming knowledge, or the complexity of their plans would affect their example use, and conducted a Spearman's rank correlation test to investigate the relationship between students' pretest scores, their planned events, and their example use.

Is programming knowledge predictive of successful example use? We found no observable correlation between students' pretest scores and their number of integrated examples ($r = -0.07$, $p = 0.71$). This indicates that both low and high-performing students integrated examples into their project, and that *a student's previous programming knowledge does not predict whether a student will successfully integrate examples or not.*

Is project planning predictive of successful example use? We found a significant, moderate correlation between students' number of planned features and their number of integrated examples ($r = 0.40$, $p = 0.02$). This shows that *students who make more ambitious plans integrated more examples into their projects.*

In addition, we also found a significant, moderate correlation between the number of *completed* planned events with the number of integrated examples ($r = 0.44$, $p = 0.01$). The number of completed events, on the other hand, is also strongly correlated with the number of planned features ($r = 0.65$, $p < 0.001$). Because all three numbers (number of planned features, number of integrated examples, and number of completed features) were significantly correlated, we are unable to infer causal relationships, but can only hypothesize that the students who made more ambitious plans integrated more examples, and (perhaps as a result) also completed more complex projects. None of these three variables, on the other hand, had a significant correlation with students' pre-test scores, showing that pre-test scores likely didn't affect how well students make plans and build their projects.

Discussion. Unlike prior work, which found that students with lower prior knowledge may request more code examples during closed-ended programming [30], our results on students' open-ended programming shows that students' prior knowledge was unrelated to whether they can successfully integrate examples. However, the complexity of students' plans – which shows how invested students are in their projects – does have a positive association with how many examples students end up integrating

[3] A group's pseudonym

into their projects. This suggests that in future work, we may help students ideate more features for their project in the planning phase (e.g., through detailed instructions or adaptive support during planning), which may lead to more example use, and potentially towards making better projects.

5.3 RQ3: To what extent did our design choices address students' learning barriers?

We next investigate whether the new features we included in Example Helper were able to address students' decision, search, and testing barriers, found in our prior work [31]. To better interpret results, we use our prior work [31] as a baseline for reference. Although this work and our prior work happened in the same CS0 course with the same curriculum, the two studies happened in different semesters with different instructors. Our analysis, therefore, does not aim to provide strong claims on the benefits of the system (i.e., as in a quasi-experimental comparison), but rather to inform hypotheses on how our design choices may have addressed the learning barriers.

Search barrier. We found a total of 34 search queries across students. 85.2% (29/34) returned results, as auto-complete suggestions showed students search findings when typing, and prompted students to use queries that returned results. This is about twice the percentage of student search queries that returned results from Prototype-EH [31], showing that providing students with auto-complete suggestions during searching has the potential to address students' search barriers.

Decision barrier. The Example Helper used a preview window for students to browse and test the interface. With this feature, we found that students who used Example Helper opened the gallery an average of 16.8 (286/17) times[4], which is two times higher than the average of 5.67 times from Prototype-EH. However, about half of the students also did not click on the "show example" button at all, a barrier Example Helper did not directly address. This shows that the preview window only addressed to some extent the decision barriers among those students who opened the example gallery at least once.

Testing barrier. Section 5.1 shows that students actively interleave experimentation behaviors such as running the example, and modifying to test different aspects of the example when reading and integrating code examples into their code. While none of these experimentation behaviors were supported by Prototype-EH, the high percentage of students who ran (85%) and modified (57%) examples suggests that the editable example windows in Example Helper addressed students' testing barriers to some extent.

Outcomes. We also inspected students' integrated examples (Section 4.2) to check whether students blindly copied examples. We found that among the 27 examples that are integrated by 8 student groups, only 7.4% (2/27) were completely copied with no modifications (full copy); in about half (55.6%, 15/27) of the copied examples, students only modified slightly; for the rest (37%, 8/27), students did structural modifications (defined in Section 4.2), making bigger changes to the example. This shows that Example Helper encouraged students to meaningfully integrate examples into their own code by making necessary modifications – not copying them blindly.

Discussion. Our results show the autocomplete searches, as well as the accessible, editable preview and playground features lead to relatively low incidents of search, decision, and testing barriers. This shows that the design choices we made have the potential to be successful in addressing students' learning barriers and lead to effective example use. Since Example Helper is built as an extension in Snap*!*, instructors can directly use Example Helper in introductory programming classrooms, as support features that students can use to request examples.

6 LIMITATIONS & CONCLUSION

Limitation: Students may still encounter search barriers. In our prior work, when students did not receive query recommendations from Prototype-EH, we found that about half of the searched queries were theme/asset-based, such as "people", "airplane" and "dining room" using Prototype-EH [31]. However, because Example Helper directed students to complete phrases that would only return results, students did not end up making search queries that were theme/asset-based. This shows that auto-complete suggestions may have limited students' choices to express ideas. In addition, the total amount of 31 examples limited the students' ability to find their own, personalized examples. For future work, we should support more diverse ways of searching, such as inferring the possible behavior or interactions students may need through their descriptions of game themes and assets.

Limitation: Some students still encountered decision barriers. Our work found that about half of students did not use the Example Helper system at all, showing that they may still encountered decision barriers. Prior work suggests that students may avoid seeking help for many reasons, such as viewing requesting help as a threat to their independence and competence [3], or forgetting about the choice to ask for help [24]. The design choices we made for Example Helper assumed that students would open the interface at least once to use and benefit from its features, but none supported those who never used the system at all. In future work, to encourage first-time usage, we may remind students to read or learn relevant examples earlier in the programming process (e.g., before or just when they started programming).

Conclusion. In this work, we presented our design process for building Example Helper, a system that supports students with gallery-based code examples during open-ended programming in Snap*!*. We found that Example Helper supports a variety of exploration and integration strategies, and that students' engagement with the planning process significantly affected students' use of code examples. We found suggestive evidence that Example Helper addressed search, decision, and testing barriers students encounter when using code examples in open-ended programming, pointing to insights that designers can take to build code examples to support effective example use.

7 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

[4] Among the 36 student groups, 17 have clicked on the "show example" button to open the gallery and use the Example Helper.

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] 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.

[3] Ruth Butler. Determinants of help seeking: Relations between perceived reasons for classroom help-avoidance and help-seeking behaviors in an experimental context. *Journal of Educational Psychology*, 90(4):630, 1998.

[4] 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.

[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 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.

[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] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, 2004.

[18] 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.

[19] Alexandra Milliken, Wengran Wang, Veronica Cateté, Sarah Martin, Neeloy Gomes, Yihuan Dong, Rachel Harred, Amy Isvik, Tiffany Barnes, Thomas Price, and Chris Martens. Planit! a new integrated tool to help novices design for open-ended projects. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 232–238, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432552. URL https://doi.org/10.1145/3408877.3432552.

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

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

[22] Chris Parnin and Christoph 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.

[23] 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.

[24] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. Factors influencing students' help-seeking behavior while programming with human and computer tutors. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 127–135. ACM, 2017.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] Wengran Wang, Archit Kwatra, James Skripchuk, Neeloy Gomes, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas Price. Novices' learning barriers when using code examples in open-ended programming. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 394–400, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456370. URL https://doi.org/10.1145/3430665.3456370.

[32] 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.