ABSTRACT

WANG, WENGRAN. The Design and Evaluation of Automated Examples to Support Creative, Open-Ended Programming. (Under the direction of Thomas Price).

Open-ended programming engages students by connecting computing with their real-world experience and personal interests. However, such open-ended programming tasks can be challenging, as they require students to implement features that may require knowledge students are unfamiliar with. Code examples can help students to generate ideas and implement those features, but students often face barriers when searching, learning, and integrating those examples. This work explores what challenges students face when using code examples during open-ended programming, and how to automatically generate personalized code examples, and how to design systems to present examples, in order to improve students' programming performance, sense of ownership, and perceived creativity. In this work, I investigate three research questions: 1) What are novices' motivations, strategies, and barriers when using code examples during open-ended programming? 2) How can we design code examples to address students' decision, search, mapping, understanding, and testing barriers? 3) What is the impact of having access to code examples on students' open-ended programming? I present 5 studies to address these research questions.

In Study 1, I built Example Helper, a tool that offers galleries of code examples for students to search and use. I conducted a lab study with 12 pairs of high school students when using Example Helper to complete open-ended programming tasks. The goal of this study was to systematically analyze and describe students' example reuse during open-ended programming - what are their needs and strategies during example reuse in open-ended programming. I found that students request code examples primarily to explore ideas; understand how to start a step; debug incorrect code; confirm their own code; or avoid re-implementing the same code. I also found 4 different strategies students employ when requesting examples: by integrating one block/feature at a time; by comparing their code with the example code to identify the key differences; by using tinkering to understand an example code; or by implementing a feature after closing the example. I found that some example requests (13.8%) also exhibit a lack of strategy, which is indicated by students copying and replacing the example code with their own code blindly, and using shallow debugging methods, such as making arbitrary changes. These findings suggest the need to build example systems to address these motivations and strategies.

In Study 2, I investigated the challenges students encounter when using code examples

during open-ended programming, in an authentic classroom environment. I found that students encounter 3 types of major barriers when using code examples in open-ended programming: decision barrier - not knowing when to use an example; search barrier - not knowing how to find a needed example; and integration barrier - failing to use the example in their project, caused by difficulties in understanding, testing and modifying the example code. These insights suggest future work to build example support systems, such as supporting experimentation and modification to address integration barriers (Study 3), and offering personalized support to address decision and search barriers (Study 4 and 5).

In Study 3, I improved the Example Helper system by offering search recommendations, example previews, and testing windows. In a quasi-experimental comparison, I found that across students who used the system, these improvements significantly improved example integration compared to its earlier prototype.

In Study 4, I developed and evaluated Pinpoint, a system that helps Snap*!* programmers understand and reuse an existing program by isolating the code responsible for specific events during program execution. Specifically, a user can record the execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred, to view code that is relevant to this event. I conducted a lab study to compare students' program comprehension experience with and without Pinpoint, and found suggestive evidence that Pinpoint helps users understand and reuse a complex program more efficiently.

In Study 5, using the improved version of Example Helper from Study 3, I conducted a controlled study to evaluate the impact of having access to code examples on students' programming and learning outcomes. I conducted the study with 46 local high school students in a full-day coding workshop, where half of the students had full access to 37 code examples using EXAMPLE HELPER, and the other half had 5 standard, tutorial examples. I found that students who had access to all 37 code examples used a significantly larger variety of code APIs, perceived the programming as relatively more creative, but also experienced a higher task load. I also found weak evidence of a better post-assignment performance from the EXAMPLE HELPER group, showing that some students were able to learn and apply the knowledge they learned from examples to a new programming task. My results show that having access to code examples during open-ended programming helped students become more creative, build projects with a larger variety of APIs, and learn new knowledge for future tasks.

The contribution of this thesis includes: 1) A systematic analysis of the goals, strategies, and barriers novices experience or encounter when using code examples during open-

ended programming (Studies 1 and 2); 2) The design and deployment of two interconnected systems, which support just-in-time example extraction (Pinpoint); and testing-centered example integration (Example Helper); each addressing one or more barriers students encounter during example reuse (Studies 3, 4 and 5); 3) The empirical evaluations of the impact of these example support systems on students' programming performance, perceived creativity and learning outcomes(Study 4 and 5).

The Design and Evaluation of Automated Examples to Support Creative, Open-Ended Programming

by
Wengran Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

Tiffany Barnes	Kathryn Stolee
Eric Wiebe	Thomas Price Chair of Advisory Committee

DEDICATION

To mom and dad, for their unconditional love.

BIOGRAPHY

Wengran was born into a family in Hangzhou, China that had a deep love for books and music. Her father, Ronghao, who holds a Bachelor's degree in Computer Science from Zhejiang University, introduced her to the magic of computing at a young age. When Wengran was just eight years old, he gifted her with a personal website where he personally typed and published all of her diaries. The experience of being able to share her thoughts and ideas online through digital publishing left a profound impact on her.

Wengran's mother, Xiya, is a dedicated teacher, doctor, and mother. She spent all her time working hard to support her family and take care of Wengran. She taught Wengran that true beauty in life is not measured by material success or external possessions, but rather by the internal experience of love, perseverance, and passion. Throughout her life, Wengran's parents placed a high priority on her education, valuing it above all else.

Wengran attended Zhejiang University for her undergraduate studies, majoring in Environmental Science and earning a minor degree in French Language and Literature. While working on a research project focused on modeling local Carbon Dioxide emissions, she became intrigued by the fields of Statistics and Computer Science. This led her to pursue a Master's degree in Statistics from NC State University, followed by a Ph.D. in the Computer Science department, where she focused on the area of Computing Education.

Recognizing the impact that computing technology has on our world, Wengran is passionate about broadening participation in Computing Education. She firmly believes that involving individuals from marginalized groups can help build a more equitable future for humanity. Through her research, she hopes to contribute to the goal of creating a more inclusive and diverse computing community.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor, Dr. Thomas Price, for his brilliant ideas, high standards, extensive knowledge, strong support, and sheer sincerity. I feel extremely fortunate and grateful to be able to work with Dr. Price, who has helped me navigate through countless research ideas, build useful systems, run rigorous studies, and write interesting papers. Dr. Price has been a role model for how to effectively engage with and motivate teamwork, how to passionately initiate, pursue, and persist through research projects, and how to mentor students with care, compassion, and kindness.

I would also like to extend my sincerest thanks to the professors and committee members, from whom I have received great support and guidance.

- Many thanks to Dr. Gordon Fraser, for sharing his extensive knowledge about program analysis and maintaining software engineering teamwork, involving me in several wonderful research projects, introducing me to his amazing lab, and giving me thoughtful critiques and insightful guidance.

- Special thanks to Dr. Tiffany Barnes, for sharing her energy, enthusiasm, and innovative ideas in lots of meetings, and for encouraging me to pursue Ph.D. when I was doing my Master's degree.

- Thanks also to Dr. Chris Martens, for sharing with me her experience and knowledge in generative methods and game design.

- I'm also grateful to Dr. Kathryn Stolee, for her brilliant ideas and actionable suggestions in advancing my work.

- Thanks should also go to Dr. Eric Wiebe, for his clear, rigorous, detailed advice for running my student studies.

I am also grateful to my collaborators and mentees, who contributed directly to my work presented here:

- My collaborators, co-authors, and lab mates, who contributed to my work: Ally Limke, for doing extensive, patient, careful, and insightful work in classroom and camp studies, and for teaching me how to develop rapport with students. Samiha Marwan, for sharing with me great insight and ideas about measures, study design, her great paper editing skills, and her loving, optimistic life attitude. Yang Shi, for involving

me in researching AI for education, and for giving me frank, honest critiques. Thank James Skripchuk, and John Marsden, for sharing with me various ideas from different perspectives; John Bacher, for helping me run studies with skills and care; Keith Tran, for helping me conduct thoughtful analysis. Thank Benyamin Tarbarsi, Heidi Reichert, and Sandeep Sthapit, for helping me with research and paper writing, even when it's last-minute.

- My amazing collaborators from Germany. Many thanks to Andreas Stahlbauer, for his extensive knowledge and insight on static program analysis, and software engineering. Thank Patric Feldmeier, for his excellent programming skills and dedication. Thank Sebastian Schweikl, for his various ideas, software knowledge, and kindness. Thank Adina Deiner, for her support and teamwork. And thank Florian Obermüler, for his generous help.

- Many (former) students, who helped me get started: Rui Zhi, for helping me get to learn about the amazing field of Computing Education; Alexander Milliken, for working with me through many camp studies; Nick Lytle, for sharing insights to start my first research paper; Yihuan Dong, who shared with me great ideas and insights on computational thinking. Thank Jennifer Tsan, for sharing with me excellent insights on open-ended programming and pair programming, and for her great support and kindness.

- My amazing mentees and undergraduate students, who gave me support and motivation. Thank Mahesh Bobbadi and Audrey Le Meur, for their dedication and commitment. I felt lucky to have the chance to work closely with them, who gave me lots of inspiration and motivation, and brought novel perspectives to my research. Thank Yudong Rao and Archit Kwatra, whose efforts and hard work went to the prototype systems that informed my research. Thank Neeloy Gomes and Sarah Martin, for providing help during my student studies.

- Thank many instructors who provided me with the resources, opportunities, and extensive support to run my studies and deploy my systems: Bita Akram, Amy Isvik, Veronica Catété, for always being there to run camp and classroom studies with me, and providing support with both teaching and research. Thank Shuyin Jiao, Adam Gaweda, Sterling McLeod, John Marsden, and many other instructors, for offering me many research opportunities with their students.

In the end, I want to express my deepest gratitude to my family. The completion of my dissertation would not have been possible without their love and support. Thank Wai Po (Yunying Lu), for her perseverance and love. She showed me the magic of words and poems when I was just learning to speak. Thank Wai Gong (Yunfa Weng), for his brilliant, strong mind, Wai Gong believed in and loved me, trusting me to achieve the best in my life. Thank Nai Nai and Ye Ye (Zhenying Kong and Lishui Wang), for their silent, constant love. I carry their love and kindness with me.

Thank Mama, Xiya Weng, for teaching me to take strong stands and live a passionate, courageous life.

Thank Baba, Ronghao Wang, for teaching me the value of action, the long-term effect of staying focused, and the power of details.

Thank my husband, Yudong Rao, for being such a joyful presence.

TABLE OF CONTENTS

List of Tables . . . . . **x**

List of Figures . . . . . **xi**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Research Outline 3
1.2.1 Understanding students' needs and challenges during example use 3
1.2.2 Addressing the challenges in two different dimensions. 4
1.2.3 An Evaluation of Having Access to Examples in Open-Ended Programming 5
1.3 Research Questions 5
1.4 Contributions 6

Chapter 2 Related Work **7**
2.1 Open-Ended Programming 7
2.2 Code Examples 9
2.2.1 Code Examples for Closed-Ended Programming Problem 9
2.2.2 Code Example Systems for Open-Ended Programming 10
2.2.3 Code Examples for Informal Learning Settings 12

Chapter 3 Novices' Motivations and Strategies for Using Code Examples in Open-Ended Programming **16**
3.1 Introduction 16
3.2 Example Helper System 18
3.3 Study Setup 19
3.3.1 System 19
3.3.2 Participants & Procedure 21
3.4 Analysis 23
3.5 Results 25
3.5.1 **Motivations: Why do students ask for examples?** 25
3.5.2 **Strategies: How do students learn and use an example?** 26
3.5.3 **An in-depth example** 32
3.5.4 **Outcomes of example use** 33
3.6 Discussion 34
3.6.1 Support design and planning for creating complex projects 35
3.6.2 Encourage effective example learning strategies 35
3.7 Limitations & Conclusion 37

Chapter 4 Novices' Learning Barriers When Using Code Examples in Open-Ended Programming **38**

4.1 Introduction . . . 38
4.2 Example Helper System . . . 39
4.3 Participants & Procedure . . . 41
4.4 Analysis . . . 42
4.5 Results & Discussion . . . 43
4.6 Limitation & Conclusions . . . 50
4.7 Acknowledgements . . . 51

Chapter 5 Exploring Design Choices to Support Novices' Example Use During Creative Open-Ended Programming . . . 52
5.1 Introduction . . . 52
5.2 The EXAMPLE HELPER System . . . 54
5.2.1 Interface Design . . . 54
5.2.2 Example Content Design . . . 55
5.3 Methods . . . 57
5.3.1 Participants & Procedure . . . 57
5.3.2 Data & Analysis . . . 58
5.4 Results & Discussion . . . 59
5.4.1 RQ1: How did students use examples? . . . 59
5.4.2 RQ2: Who used examples? . . . 61
5.4.3 RQ3: To what extent did our design choices address students' learning barriers? . . . 62
5.5 Limitations & Conclusion . . . 64
5.6 Acknowledgements . . . 64

Chapter 6 Pinpoint : A Record, Replay, & Extract System to Support Code Comprehension and Reuse . . . 65
6.1 Introduction . . . 65
6.2 Related Work . . . 66
6.2.1 Code Reuse . . . 67
6.2.2 Supporting Code Comprehension & Reuse . . . 67
6.3 System Design Goals & Formative Study . . . 68
6.4 The Pinpoint System . . . 69
6.4.1 The Pinpoint Design . . . 69
6.4.2 Pinpoint Implementation . . . 72
6.5 Methods . . . 73
6.5.1 Participants and Study Design . . . 74
6.5.2 Materials: Two Reuse Assignments . . . 75
6.6 Data Collection and Analysis . . . 76
6.6.1 Pretest . . . 77
6.6.2 Task Performance . . . 77
6.6.3 Qualitative Interview Analysis . . . 77
6.7 Results and Discussion . . . 78

6.7.1 RQ1: What was the impact of Pinpoint on students' ability to extract and reuse code from an example? . . . 78
6.7.2 RQ2: What are students' perceptions of their reuse experience? . . . 80
6.8 Conclusions and Future Work . . . 82

Chapter 7 Investigating the Impact of On-Demand Code Examples on Novices' Open-Ended Programming Experience . . . 83
7.1 Introduction . . . 83
7.2 Related Work . . . 84
7.2.1 Open-Ended Programming . . . 84
7.2.2 Code Examples . . . 85
7.2.3 Open-Ended Programming . . . 88
7.2.4 Code Examples . . . 88
7.3 Methods . . . 90
7.3.1 The EXAMPLE HELPER System . . . 90
7.3.2 Participants & Learning Context . . . 92
7.3.3 Procedure . . . 93
7.3.4 Measures . . . 94
7.4 Results . . . 97
7.5 Discussion & Conclusion . . . 101

Chapter 8 Conclusion . . . 103
8.1 Research Questions . . . 103
8.1.1 Research Question 1 . . . 103
8.1.2 Research Question 2 . . . 105
8.1.3 Research Question 3 . . . 106
8.2 Design Principles & Future Work . . . 107
8.3 Contributions . . . 108

LIST OF TABLES

Table 3.1 5 situations where students ask for code examples and their frequencies among a total of 88 example requests. 25
Table 3.2 4 example reuse strategy and the copy-run-debug behavior (i.e., lack of strategy). Strategies started with ⋆ denotes example use strategies not mentioned in prior work [Wan20c; Ich15]. 30

Table 5.1 EXAMPLE HELPER design targets to address the search, decision, testing, and modification barriers students encounter when using code examples during open-ended programming. 56

Table 7.1 Statistics for the control (ctrl.) and example (exp.) groups, with the p-value and effect sizes. * means the data follows the normal distribution. 98
Table 7.2 Statistics for the control (ctrl.) and example (exp.) groups, with the p-value and effect sizes. * means the data follows the normal distribution.100

LIST OF FIGURES

Figure 3.1 The Example Helper Interface. 18
Figure 3.2 The Example Helper Interface. 20
Figure 3.3 L3 copied, modified, and tested to integrate the example code to their own code **one at a time**. 26
Figure 3.4 Using the **comparison** strategy, E7 identified meaningful differences to use in their own context, without discarding un-meaningful differences. 27
Figure 3.5 L4 employed the strategy "**understanding through tinkering**" to understand an unfamiliar block - "set size to". 28
Figure 3.6 Case study: students' implementation of spawn_clones. 31

Figure 4.1 The Example Helper Interface. 40
Figure 4.2 # unfamiliar blocks v.s. integration rate. 47
Figure 4.3 example type v.s. integration rate 48

Figure 5.1 The EXAMPLE HELPER interface, which includes a selection-based gallery (left) and a playground view (right) for students to program while using the example as a reference. 54

Figure 6.1 Pinpoint users can 1) record a program execution (including user input and graphical output), 2) replay a recording and select a time interval where an event has occurred, and 3) inspect an executable code slice relevant to the event, where the code executed inside the selected time interval is highlighted. 70
Figure 6.2 Users can also trace changes to individual variables by selecting "How" questions on different variables and attributes. 72
Figure 6.3 For the 11 students without perfect performance, The Late group (shown in yellow) showed significantly more improvement than the Early group (shown in green). 79

Figure 7.1 The EXAMPLE HELPER interface [Wan22]. Students can browse and search for examples in a gallery (left) interface, and then test and modify them in a sandbox (right), where they may also click to copy the example into their own workspace. 91

CHAPTER

1

INTRODUCTION

1.1 Motivation

Open-ended programming projects, where students make apps, games, and stories that they have designed themselves, are widely-used in many introductory programming curricula [Gar15; McG18; Gro18] and online, informal learning settings [Pep07]. During open-ended programming, students can freely explore, design, and implement a relatively complex programming project, and can express their ideas creatively [Hul15]. As open-ended projects enable learners to freely define their own goals, they allow learners to connect their real-world experiences and interests with their programming projects [Pap80], motivating them to pursue Computer Science [Mar02; Guz05].

Open-ended programming projects are characterized by a number of properties that make them engaging, but which can also be challenging for novice programmers: First, they require students to engage in the complex cognitive processes of generating ideas, designing plans and implementing solutions [Win11; Blu91]. Because the projects are somewhat or fully student-designed, students need to engage in these activities and manage a large and complex project. This can lead to many challenges during both design and

implementation [Ald18; Kok16; Wri07]. Second, open-ended programming encourages students to be ambitious in their designs to create interesting artifacts, which often requires students to combine different programming concepts and APIs, or make use of APIs they are still unfamiliar with. and write larger, more complex projects than typical assignments. Lastly, as students can freely explore among infinite choices to generate ideas and develop solutions, it is difficult for instructors to prepare students with all possible materials before project-making; or offer students personalized feedback or suggestions when they request help during programming. This suggests the need for better ways to support novices working on open ended projects.

Code examples provide a promising option to help students overcome these challenges. Many novices use code examples to explore ideas, learn new programming concepts, and to integrate new API usage patterns into their open-ended projects [Ker17b; Kha19]. Prior work has shown that novices consider code examples as useful learning materials [Lah05], and that they were able to reuse new API usage patterns effectively after learning them from code examples [Ich17]. However, exploratory studies from prior work also show that novices encountered a number of challenges when integrating examples into their own code [Ich15], such as difficulties *understanding* the example code, and *integrating* the example code to their own code [Ich15; Wan20c].

A large body of related work has focused on developing example support and studying students' example learning for *closed-ended* programming problems (i.e., with predefined specifications, rather than student-defined goals) [Ich15; Zhi19; Tra94; Bru01], while little has explored building example support for open-ended programming projects. In particular the open-ended programming projects addressed in this work can be characterized by a number of distinct challenges. First, as open-ended programming includes multiple iterative phases of design, planning and implementation, students may need code examples for a variety of goals, and may use them in different ways. Second, as students freely decide the goals of their projects, we do not know what examples a student will need before project-making. This motivates my work to systematically investigate the phenomenon of example use during open-ended programming (i.e., why and how students use code examples during open-ended programming), to explore the ways in which students may encounter barriers to use code examples during open-ended programming, to design and evaluate systems to address these barriers, and to evaluate how access to examples affect students' open-ended programming experience.

1.2 Research Outline

I present my thesis work in three parts. The first part explores the affordances and challenges of example use during open-ended programming; the second part discusses the design, deployment and evaluation of two complementary and interconnected example support systems, each addressing specific challenges for example reuse; the third part discusses evaluations of each of the three support systems, with implications for pedagogy and tools to support example use during open-ended progamming.

1.2.1 Understanding students' needs and challenges during example use

I begin in Chapter 2 with a literature review on the related work on open-ended programming and code examples, and discuss the design space for building example-based support for open-ended programming.

In Chapters 3 and 4, I discuss a systematic analysis on the phenomenon of example use during open-ended programming. To do that, I first built a prototype version of Example Helper, a gallery-based example support system, and deployed it in a lab study with a group of high school students. I employed the Case Study Research methodology to analyze students' motivations and strategies when using code examples during open-ended programming, using an aggregation of log, think-aloud (Chapter 3). Next, I deployed the Prototype Example Helper system to an introductory programming classroom, where I investigated the specific challenges students experience when using those examples. I found that students encounter the following barriers: 1) many students do not use examples even when they need help (*decision barrier*), perhaps due to the lack of personalization of the examples' content, or due to the lack of awareness or trust towards the code examples. 2) They encounter difficulties in describing which examples they want when searching (*search barrier*). 3) When they find an example they need, they encounter barriers to map a property of the example to the property of their own code, as the example is presented in a different context than the students' own code (*mapping barrier*). 4) When learning an example, they also encounter difficulties understanding the example code (*understanding barrier*). Lastly, 5) they need immediate access to modifying and testing each code examples, which the Prototype Example Helper did not offer (*testing barrier*).

1.2.2 Addressing the challenges in two different dimensions.

In Chapters 5 and 6, I present two new systems that I have built to address the challenges students encounter during example reuse.

- Example Helper (Chapter 5).I completely redesigned Example Helper to address the challenges identified in Chapter 4, such that the system includes immediate search results, autocomplete suggestions, and direct running and modification support in the example browsing and testing interface. I conducted a quasi-experimental study on the usage of Example Helper for creating open-ended programming projects, and found that Example Helper helped students to integrate more examples into their own code compared to its prior prototype.

- Pinpoint (Chapter 6). Pinpoint is a system that helps Snap programmers to understand and reuse an existing complete program by isolating the code responsible for specific events during program execution. Specifically, a user can record an execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred, to view code that is relevant to this event. Students can search and identify a subset of code blocks from a complex example program. I conducted a lab study with 17 students, and found suggestive evidence that Pinpoint improves students' ability to integrate code examples into their own projects. The students explained forming more confident hypothesis about a code segment's runtime behaviors, employing more focused, targeted example learning approach, and connecting different code segments more easily using Pinpoint.

These two systems are complementary and address barriers students encounter at different phases of project-making. 1) To address the *decision barrier*, I increase students' awareness and trust towards the code examples, by including students in the process of example co-creation. Pinpoint extracts code examples based on students' own toggling of an execution trace. 2) Pinpoint also addresses the *search barrier*, by extending the modes of example search from simply typing text in a searchbox, to also include allowing students to find examples by drawing storyboards or searching within larger example projects. 3) To address the *understanding barrier*, Pinpoint uses highlighted code blocks to directly map the executing code to its corresponding output. 4) Lastly, to address the *testing barrier*, Example Helper allows students to quickly access, run and modify examples from within in the example browser.

1.2.3 An Evaluation of Having Access to Examples in Open-Ended Programming

In Chapter 7, I discuss a controlled study, which aims to evaluate the effect of having access to code examples on novices' open-ended programming experience, including their 1) project complexity, 2) self-perceived task load and creativity, and 3) learning outcomes. I ran a controlled study in a full-day coding workshop, with students recruited from local high schools. The examples were provided through EXAMPLE HELPER, on which students can search, select, test and use an example from a gallery of code examples. Going beyond the system evaluation studies discussed in Chapters 5 and 6, this work measures the impact of example use in a longer (3 hours), ecologically valid learning setting (a day camp classroom), where students made a project from start to finish, which they had the freedom to design and plan themselves. Further, I not only investigate the extent of API use from code examples, I also investigate how access to examples affects the complexity of students' projects, their post-task performance, and students' perceptions of task load and creativity.

1.3 Research Questions

This work investigates the following 3 high-level research questions:

- RQ1: *What are novices' motivations, strategies, and barriers when using code examples during open-ended programming?*. This RQ is addressed in Chapers 3 and 4, where I discuss the phenomena of example reuse during open-ended programming, and the key barriers students encounter when using code examples.
- RQ2: *How can we design code examples to address students' decision, search, mapping, understanding, and testing barriers?* This RQ is addressed in Chapters 5, 6, where I discuss the design and deployment of two complementary example support systems to address the above learning barriers.
- RQ3: *What is the impact of having access to code examples on students' open-ended programming?* This RQ is addressed in Chapter 7, where I discuss the evaluation of the Example Helper system, and evaluate its impact on students' project complexity, self-perceived task load and creativity, and learning outcomes.

1.4 Contributions

The contribution of this proposed thesis includes:

- A systematic analysis of the goals, strategies, and barriers novices experience or encounter when using code examples during open-ended programming.
- The design and deployment of two interconnected systems, which support, respectively, just-in-time example extraction (Pinpoint); and testing-centered example integration (Example Helper); each addressing one or more barriers students encounter during example reuse;
- The empirical evaluations of the impact of these example support systems on students' programming performance and perceived creativity;

CHAPTER

2

RELATED WORK

I discuss the Related Work in two parts. First, I discuss the open-ended programming context, which situates my work. Next, I discuss prior work on code examples, and the gaps between prior work and my proposed thesis.

2.1 Open-Ended Programming

The first step towards creating learner-centric tools, such as those presented in this work, is to understand students' own needs and practices [Guz15]. In order to understand how to build tools to support open-ended programming, I first review the benefits of open-ended programming, exploratory programming behaviors, and the challenges they may encounter when making open-ended programming projects.

Open-ended programming allows learners to integrate personal interests into creating an artifact that is meaningful to them. Many efforts to promote open-ended programming draw on the theory of Constructionism [Pap80], which suggests that learners effectively build their own knowledge structure when engaging in creating a programming artifact they feel connected with [Pap80]. In many introductory programming curricula, the process

of completing an open-ended project includes the multiple phases of student-centered activities, where students generate ideas, make designs, discuss plans, and implement their solutions [Mil21].

Prior work also has summarized two key types of challenges novices encounter when making open-ended programming projects. The first is the cognitive challenges towards the self-directed process of designing, planning, and building an open-ended project [Wri07], which requires self-regulatory skills [Win11], focused attention [Wri07] and high levels of engagement [Kok16]. Prior work has shown that students encounter barriers in multiple phases of the self-driven activities of designing and building a programming artifact. For example, Marx et al. conducted four case studies with middle school teachers, and discussed multiple barriers towards a project-based learning approach, such as difficulties to engage students throughout the time-consuming process of goal setting, planning, and implementation [Mar94]. Aldabbus conducted a qualitative survey and interview study with 24 teachers in multiple disciplines on their classroom experience of project-based learning, and found that 3/4 of the teachers were unable to implement project-based learning with their students, and discussed challenges including students seeking for expedient solutions, rather than meaningfully engage in the problem-solving process [Ald18]. This shows that students need external support when completing multiple phases of open-ended programming, including design, planning and code implementation.

The second type of challenge novices face when making open-ended programming projects relates to the difficult process of building a complete, functional programming artifact, which requires organizing and implementing many different programming concepts and API knowledge. Prior work has discussed many challenges when novices build an open-ended programming project. 1) They may struggle to design "logically-coherent" programming components, and may start by putting together all possible code elements that seemed relevant [Mee11]. 2) Their programs may suffer from code smells such as duplicated code [Rob17]. 3) Their final artifacts were shown to be lack of usage of fundamental programming concepts (e.g., variables, operations), from a systematic evaluation of 80 novices' open-ended projects collected from 20 urban middle school classrooms [Gro18]. These show struggles to apply existing concepts into code, or to explore new programming concepts or APIs. Kirschner, Sweller, and Clark summarized through a literature review that open-ended discovery may lead to experiential learning, where learners rely heavily on trial-and-error instead of learning new knowledge [Kir06]. These challenges encountered by novices during open-ended programming are examples of "Play Paradox" [Nos96], which explains that learning activities should strike a balance between creative exploration and

some levels of external support [Nos96]. This shows the need for supporting open-ended programming with information and materials that demonstrate API knowledge and code usage patterns.

2.2 Code Examples

Prior work on code examples for novice programmers focus primarily on supporting students to complete closed-ended programming problems, where they are asked to make a program to complete a set specification [Gro14; Zhi19; Wan20c]. In this discussion, I first summarize the Worked Example effect , and discusses evidence of the effect of code examples to support learning and programming performance. However, little research has focused on the open-ended programming context, which adds specific challenges towards example use. I first review related work on code examples for closed-ended programming tasks, which sets the foundation of my work. I next discuss the literature on using code examples for open-ended programming, and discuss the challenges for code reuse in this context.

2.2.1 Code Examples for Closed-Ended Programming Problem

The Worked Example Effect

Worked Examples (WEs) are a form of instructional support, which give students a demonstration of how to solve the problem [Cla11]. WEs are traditionally offered in lieu of problem solving, usually "before" or "after" a student solves a distinct but related programming task [Bru01; Tra94]. The effectiveness of WEs is primarily grounded in Cognitive Load Theory, which argues that learners have a finite amount of mental resources during problem-solving (called cognitive load), and when problems impose an unnecessary burden on those resources (intrinsic load), the student has fewer resources left for processing and learning the material (germane load) [Swe88]. WEs support learning by providing support for "borrowing" knowledge, reducing the unnecessary intrinsic load [Swe06]. Programming learning environments use WEs widely. For example, WebEx provides web-based self-explaining code examples for students [Bru01]. Such programming WEs could help students learn the problem-solving schema [Gen03] and transfer it to another task [Tra94]. Trafton et al. evaluated 40 undergraduate students' post-test scores after programming in BATBook, a Lisp programming learning environment, and found that those with alternating WE and prob-

lem solving (PS) pairs performed better than those with PS pairs [Tra94]. However, another group, who saw all WE problems, followed by all PS problems (not in pairs), solved problems significantly slower, and achieved significantly lower post-test scores. These results show the promising benefits of using examples to support learning and performance.

Code Example Systems for Closed-Ended Programming Problems

Based on the theory of the Worked Example Effect, prior work discussed several systems that used code examples to support completion of closed-ended programming problem. For example, my own prior work explored offering step-by-step examples with options to immediately run the example code [Wan20c]. Other systems offer an online database of annotated examples [Bru01]. Peer Code Helper offers such step-by-step code examples from the same task, during block-based programming [Zhi19]. An evaluation on 22 high school novice students showed that students using these code examples solved tasks quicker than those without, without hindering their learning [Zhi19]. The FIT Java Tutor [Gro15] provides such step-by-step code examples for Java programming. Investigation on five students' programming experience showed that students occasionally followed the feedback and improved their program over time [Gro14]. However, novices also encountered a number of challenges when understanding code examples. For example, Looking Glass provides students with annotated code examples from a similar task during block-based programming [Ich15]. However, learners had difficulties understanding these examples in Looking Glass, encountering "example comprehension hurdles" while trying to connect example code to their own code [Ich15]. In a study evaluating 23 students' experience with step-by-step code examples offered during Java programming, students barely followed the examples, reporting them being "unspecific and misleading" [Coe17]. Therefore, more work is needed to design new forms of example feedback, to improve students' understandings of code examples, and connect code examples more closely for students' own program [Col88].

2.2.2 Code Example Systems for Open-Ended Programming

Example Use During Open-Ended Programming

I first review the types of example reuse behaviors during open-ended programming. 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"[Ker17b]. 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 [Ber16; Ker17b]. 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 [Ker17b]: 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 [Bra09]. 2) *Debugging into existence:* After directly copying code found from online resources, programmers debug those code until they work correctly in their program [Ros93]; and 3) *Rapid prototyping*, where programmers iteratively create, test, and experiment with a prototype at an early stage of the programming process [Har08; Ker17a]. Based on these key distinguishing activities, Kery and Myers suggested building tools to support exploration and experimentation among exploratory programmers [Ker17b].

Little prior work has developed code example systems to support open-ended programming. To increase awareness of API usage patterns, Ichinco, Hnin, and Kelleher used a set of static rules to automatically check programs and find opportunities to prompt code examples that demonstrates a specific API use (e.g., for how to use a code block). In a study with novice students making open-ended programming projects, the group of students who have access to these static code examples used these suggestions twice as much comparing to the group who used documentations instead of code examples, and included those new API methods from code examples more frequently. This work shows the potential of using code examples to support open-ended programming.

However, related work only discussed authoring static code examples when supporting open-ended programming [Ich17], and is lack of investigation in the following aspects: First, during open-ended programming students may use a wide variety of APIs and features, in various combinations. Therefore, it is difficult to anticipat e all possible choices of examples students may need. This suggests the need for more personalized or automatically generated examples. Second, examples present a different context from the students' own work, which has been shown to cause "example comprehension hurdles", as students encounter challenges to connect example code to their own code [Ich15], which raises the question of how to generate code examples that best suit students' needs and are easier to understand. Finally, related work only focused on direct measures of whether the example were used by the students, but not how it affected the students' project-making outcomes.

2.2.3 Code Examples for Informal Learning Settings

In informal learning and project-making context, code examples are also one of the primary resources students and end-users use to learn programming knowledge and API usage patterns [Bra09]. Such an example usage scenario arises when a programmer feels in need of resources in the middle programming. They search for a code example (e.g., through documentation or forums) [Bra09], and then integrate the example to their project through testing and modification [Bra09]. Prior work has shown that, different from learning traditional Worked Examples [Cla16], where programmers engage in *deliberate learning* of a step-by-step demonstration *before* working on the actual task [Mor15; Tra94; Pir94], learning an example *in the middle of* programming is a type opportunistic learning [Bra09; Gao20], where programmers search, select, and copy code examples to "get something to work with", and then briefly test or modify to integrate examples into their own code [Ros93]. When investigating experienced programmers' opportunistic learning, Rosson and Carroll found that these programmers made effective use of examples to complete functionalities that they were unfamiliar with, but many don't reflect on *how* the example works [Ros93]. They may also struggle to apply or extend examples afterwards [Tha20]. While this explains the experts' opportunistic learning of code examples, and described *how* experts can encounter difficulties in using and applying code examples, it is unclear how this theory will extend to novices.

Code Reuse Behaviors

A key phase of using examples during open-ended programming is *code reuse*, which refers to the process of identifying useful components of example code and integrating them into one's own program [Hol09]. Programmers reuse code examples for different purposes, such as exploring ideas, understanding implementation details, and debugging their own code [Wan20c]. Learning from code examples before or while making a similar program has been shown to help students not only complete the program faster [Zhi19], but also to perform better on a concept-related posttest [Tra94], and to effectively learn how to use APIs later in their own code [Ich17].

Holmes et al. conducted four case studies on programmers' process of code reuse, and characterized the reuse process into two stages: 1) *locating and selecting* and 2) *integrating* [Hol09]. During the *locating and selecting* stage, programmers need to navigate through a complete example program to find relevant areas of interest [Hol09]. This process can be challenging for both experienced and novice programmers. For example, Ko et al. found that

in this selection stage, software developers begin by searching for relevant information, but they often make use of limited and misrepresented cues in the program or the environment, causing failed searches [Ko06]. Similarly, Gross et al. conducted an observational study for 14 novice programmers to identify code responsible for a target functionality, and found that they engage in a cyclic search process of 1) generating assumptions based on a search target in the code or output, and then 2) read and search code to adjust or expand the potential code region relevant to the target functionality. These programmers frequently made false assumptions and failed 59% of the code identification tasks [Gro10a]. These results suggest that programmers need support that helps them make more accurate assumptions when relating functionality to a relevant code segment.

During the *integration* stage of code reuse, programmers need to adapt and integrate the selected code into their own program [Hol09]. During this process, programmers may directly copy a subset of an example code to their own code, or may re-implement a functionality by themselves after reading and learning an example [Wan22]. Prior work has identified many barriers programmers encounter when integrating example code [Wan21]. For example, Wang et al. analyzed 44 novice programmers' example integration process, and found that these programmers encountered barriers in *understanding* how to integrate an unfamiliar code block into their own context, *mapping* the functionality of a part of an example to their own code and *modifying* the example to fit their own needs[Wan21]. Wang et al. also found that students prefer smaller code examples with few or no unfamiliar code blocks [Wan21]. This shows the need to craft examples into smaller, comprehensible code segments, so that students may understand a specific segment before integrating it into their own code.

Remixing in Scratch

To support code reuse, the Scratch online program community is built on the culture of remixing [Das16], where users can reuse another program by making copies of an existing Scratch program and make modifications to build their own code [Kha19]. Remixing allows programmers of diverse background and programming skills to creatively collaborate with one another asynchronously [Mon07], so that they may share ideas, and learn new skills and techniques from one another [Roq16]. In the online communities offered by Scratch and other novice programming environments (e.g. Snap*!*), many programmers start programming by using another project and modifying it to make their own version [Kha19]. As a result, a large portion of projects in Scratch are remixed projects [Mon12].

However, empirical research has found many issues with remixed programs, specifically in the online Scratch community, pointing to potential challenges programmers encounter when remixing. Remixed programs can lack transfer of API knowledge from the original program to the remixed program. For example, Khawas et al. analyzed 8142 Scratch projects remixed from 160 original programs to inspect evidence of learning of Scratch API knowledge (cloning and procedures), and found that the remixed programs failed to use the cloning APIs correctly when needed even when the original program used clones; and the majority (98.6%) of remixed programs did not create new procedures when remixing from an original program that uses procedure [Kha19]. As a result, Hill and Monroy-Hernandez collected and analyzed peer rating data on more than 1 million scratch projects in the online Scratch community, and found that the remixed projects were rated lower by their peers. This confirms that remixers tend to be lower skilled and may need help with program comprehension and integrating their own code. Similarly, Amanullah and Bell conducted an analysis on 9141 Scratch users' programs, and found that even when programmers observed a sophisticated API usage pattern (e.g., Process All Items) when remixing another's program, they generally did not use them later in their own original programs, and that many remixing users copy the original program without understanding it [Ama19]. This suggests the users need to understand a program in order to apply and modify it in their remixing programs, or transfer the usage of programming concepts from the remixing program to their own future programs.

Supporting Code Comprehension

Code comprehension refers to the process of programmers building a mental model of how a piece of code works [Von95; Gro10a]. Von Mayrhauser defined that a key cognitive process during code comprehension is generating a hypothesis of the causal effect from a code segment to its output [Von95]. Programmers of different levels may all form an incorrect hypothesis, but experts discard questionable hypotheses and form correct ones more quickly than novices [Von95].

Prior work has developed tools to support program comprehension for programming education and end users. For example, Python Tutor visualizes stack traces for students to see internal data representations of the program state [Guo13]. However, it is not designed for complex user inputs and graphical output of games and apps. Whyline in Alice[Coo00] helps users to ask why and why not questions for debugging their own code [Ko04a]. However, it can only answer object-specific questions such as "Why did Pacman resize .5?", but

not “object-relative” [Ko04a] questions such as “Why did Pacman resize after the Ghost moved”, which were frequently asked by Alice programmers [Ko04a].

Some prior work applied record/replay systems to help users understand or debug programs [Gro10b; Bur13]. Timelapse is a record/replay-based tool for debugging web applications, which points to the users the lines of code responsible for a point of interest during the recorded trace [Bur13]. Similarly, Gross et al. developed a record/replay tool to help users in Looking Glass to record and select the timeframe of interest during the playback. The system then highlights the code responsible for the timeframe [Gro10b]. However, both interfaces only highlight the lines of code responsible for the selected time frame in the output, but do not extract an *executable* code slice from the program.

CHAPTER

3

NOVICES' MOTIVATIONS AND STRATEGIES FOR USING CODE EXAMPLES IN OPEN-ENDED PROGRAMMING

3.1 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 [Rob09; Bra09; Par11; Lan89; Bai20]. In particular, novice programmers stand to benefit from programming examples, which can introduce new programming concepts [Mor15; Tra94; Pir94; Wan20c], and scaffold users to create more complex and interesting programs [Ich17]. However, prior work on systems that support novices' example use have identified a variety of barriers encountered by students, such as difficulties to *understand* the example code, to *integrate* the example code to their own code, and to *modify* the example towards

their own goals [Ich15; Wan20c].

These barriers raise questions about how systems can more effectively support novices' example use. To do so, it is important to understand situations in which novices are asking for and using examples. Specifically, we aim to investigate students' **motivations** for using examples, as effective support systems must directly address these motivations [Guz15]. 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 [Ko11].

In this work, we ask the research question: **What are novices' motivations and strategies for using examples when creating open-ended programming projects?** We choose to focus on open-ended projects, because these projects attract students of varying interests by allowing them to pursue goals that feel meaningful to them [Guz05], and are therefore widely used in many introductory programming curricula [Gar15; McG18; Gro18] as well as after-school, informal learning settings [Pep07]. However, students are also known to face a number of barriers to incorporate challenging new programming patterns and APIs in open-ended programming [Gro18], which code examples that demonstrate such knowledge may help to overcome [Ich17].

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 Helpe [Wan21], an example support system designed for open-ended programming in Snap*!* [Moe12]. We analyzed video, interview, logs, and project submissions, identifying 5 distinct motivations and 4 key strategies that students employ when using examples. We also found that students almost always use *some* strategy, but that when they instead simply copy the example without modification, this rarely leads to successful integration. Students also reported examples being helpful for their performance on current and future tasks, which is supported by student outcomes in our study. Based on these findings, we then propose a set of design recommendations to facilitate students' learning through creative design and planning, active code reconstruction, and comparison-based knowledge integration. Our key contributions are:

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

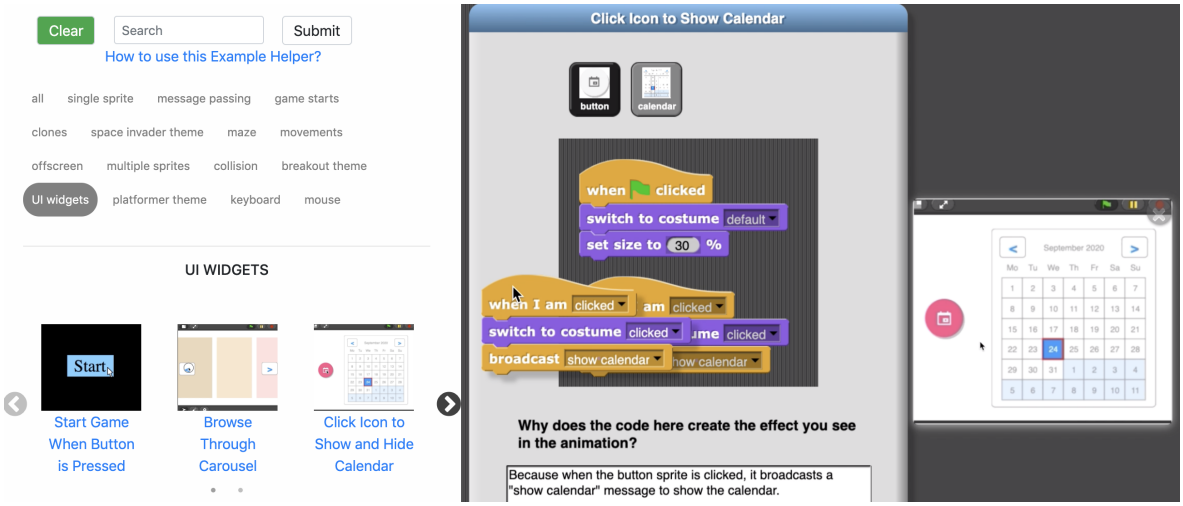


Figure 3.1: The Example Helper Interface.

ing from active use of examples.

3.2 Example Helper System

The design goal of Example Helper is to allow students to view and incorporate existing programming patterns into their own code through effective use of code examples. To lower the barrier for making these programming projects [Mor11], the system is incorporated into Snap*!* [Moe12], a novice programming environment. Similar to other novice programming environments (e.g., Scratch[Res09]), Snap*!* already offers open-source galleries of complete programming artifacts from other programmers, but these are complete projects which demonstrate many related programming features. By contrast, Example Helper offers small snippets of code examples [Rob09] that demonstrate specific functionalities, collected in a curated, browsable gallery. We developed this curated set of examples through an analysis of students' programs from prior semester, extracting key program features that were shared across students, and built these as examples. Many of these key features include usage of multiple sprite interactions[1] (e.g., in a collision event), we therefore also included examples that include usage of multiple sprites. Two experts then reconstructed examples from this repository to include cleaner and higher-quality code. When a student needs an example during programming, they can click on a "show example" button within the scripting area

[1]A sprite in Snap*!* is an object (i.e., in object-oriented programming) that has its own code (scripts), costumes (e.g., a button), and variables.

of Snap*!* to open a gallery of code examples. The student then follow two steps to select and use an example within their own source code:

Step 1: Search for an example. The student can find an example by: browsing through the gallery; or filtering and search for examples by clicking on a tag, or querying in a search box. The search box finds a set of examples the student need by looking for words that overlapped in the examples' names. To visually understand the functionality of the example, the student may also hover on the example to look at the gif animation of the code's output.

Step 2: Use an example. After finding a needed example, the student can click on the gif animation, and learn the example using the following steps:

Read the code in relation to the output. The student may click on different sprites to look at the example code for each sprite (shown in Figure 3.1). They may also look at the animation of the output next to the example code, since reading code in relation to output has been shown to trigger students to reflect on how the example code works [Wan20c]. The student can also click on the "Open the Project" button to view the example in a separate window and experiment with it.

Write a self-explanation. The student can reflect on the example by writing down a self-explanations: "What in the code here creates the effect that you see in the animation?". We designed self-explanation prompt because self-explanation is a critical step towards learning from an example [Tho20; Atk03], since it promotes students to stop and think deeper about the code example [Ale16; Ger04].

Copy the example code. To allow students to test and modify the example easily, after writing their self-explanation of the example, the student may then drag and copy the example to their own code. To discourage students from immediately copying the code without thinking about it, we restricted the length of the self-explanation answer to be at least 30 characters.

3.3 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.

3.3.1 System

We built a system called Example Helper, which adds a "show example" button on the Snap*!* editor, showing a gallery of examples upon request (Figure 3.2). Example Helper is

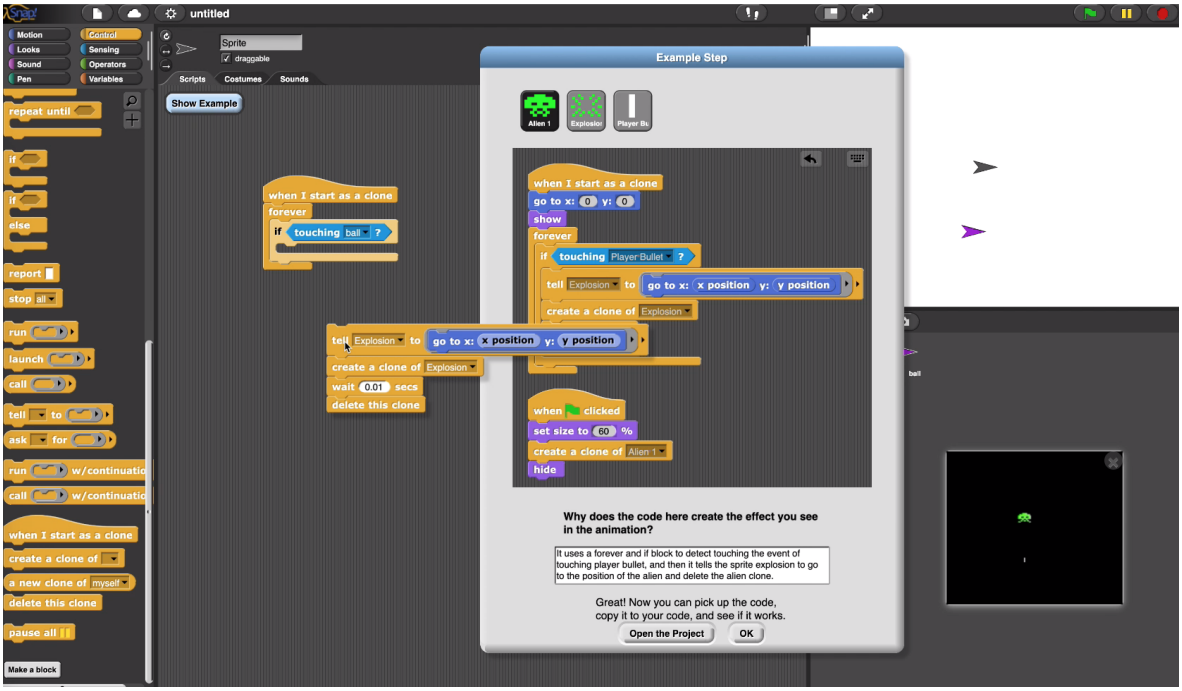


Figure 3.2: The Example Helper Interface.

particularly suitable for our goal to analyze novices' example use, for three key reasons:

High-quality examples

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

Supports for searching, copying, and testing example code

Example Helper is designed specifically for supporting open-ended programming, where students may need to learn to use and integrate new concepts and code patterns on their own [Gro18]. This process of searching for and learning programming knowledge is described by the COIL model [Gao20], which includes information collection & organization; and solution testing. Students are provided with supports for all these three elements by Example Helper. First, a student can search for an example (*information collection*): When clicking the "show example" button, students see a gallery of examples, where they can

browse; search over a search box; or filter examples based on tags. Next, when they find an example and click to open it, they can read or copy code (*information organization*): when reading, the student can navigate through codes on different sprites[2], shown by different tabs in the example interface (Figure 3.2). They may also copy example code by dragging it to their own code. Last, to test code, they may run copied example code in their own program, or open the example code in a separate window by clicking on the "open the project" button (*solution testing*).

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 Helper encourages this self-explanation process, as it has consistently been shown to aid learning from examples [Shi08; Atk03; Wan20c].

3.3.2 Participants & Procedure

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

Our study occurred in the first 3 days of the second week, described below, before which students completed a one-week coding bootcamp to program in NetsBlox [Bro17]. We designed a controlled study with alternating conditions, where the Early group (n = 7 pairs) having access to examples only on Day 2, and the Late group (n = 5 pairs) having access to examples only on Day 3[3]. A researcher demonstrated how to use Example Helper, 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

[2] A sprite in Snap*!* is an object, such as an actor in a game.

[3] 28 students attended the study, based on which we assigned 7 pairs in each group. However, 2 students from 2 separate pairs in the Late group did not consent, we therefore excluded the two pairs' data from analysis.

been shown to promote higher performance for novices during open-ended programming [Gro18], and is a standard practice in many real-world classrooms [Lew11; Lyt20; Tsa21]. 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, closed-ended tasks, including drawing shapes and programming multiple-sprite interactions, using loops and conditionals [Wan20b]. 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 [Lew15].

Day 2 & 3: building games

On Days 2 & 3, students built games with two different themes – *breakout* and *space-invaders*, respectively. These two themes include features such as the player interacting with a larger group of sprites (e.g. bricks, enemies), or collision causing them to disappear. These themes were suitable *open-ended* tasks, as they required the usage of many concepts (e.g., loops) and APIs (e.g., cloning in Snap*!*). Additionally, they provided flexibility and variability in game design [Hun04] (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 [Mal10; Wan20a]. 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.

3.4 Analysis

We analyzed our data using the "Case Study Research" [Yin17; Ham12] method, a systematic approach to research "decisions" – "why they were taken, how they were implemented, and with what result" [Sch71]. 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" [Yin17].

Data Organization. To ensure construct validity [Cro55], we collected and organized data following the 3 principles by Case Study Research: 1) We collected data from "**multiple sources**" [Yin17], 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**" [Yin17] 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 3.3), 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**" [Yin17] by linking interview and submissions to the case profiles of corresponding pairs, to enable tracing back/forward between different data sources and analysis stages [Yin17].

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

Finding "patterns" [Yin17] from logs

A"pattern" describes cause, effect, or events that relate to the central phenomenon of interest [Yin17]. As log data captures most precisely students' experience comparing to interviews, which may suffer from response bias and inaccuracies [Del12; Pau91], we start our analysis of cases first on their log data, a commonly-used primary data source to analyze programmers' [Bra09], end users' [Ko04b] including novices' [Ich15] programming experiences.

We find patterns of **situations** where students ask for examples, based on two types of log data: 1) the code students have to complete a feature demonstrated by an example before asking it (called "starter code"); and 2) the activities students engaged in with examples –

whether they attempted reusing the example code, or they immediately closed examples. These two data types have been shown by prior work [Wan20c] to describe students' goals for requesting examples. For example, we can infer a "debugging goal" when a student had buggy code and used example to locate changes to make. Based on these two data types, we identified 5 situations where students ask for examples (e.g., "when starting a step").

Similarly, we look for patterns of **strategies** by analyzing students' programming activities in logs. From the case database of 88 requests, we first filtered out 41, where students immediately closed the example after opening. For the remaining 47, we analyzed those repetitive requests of the same example in aggregate, creating a total of 29 sets of example requests. We started with a detailed account of all activities pairs engaged in when using examples, such as the time when the student started programming the relevant feature, their starting code, the programming behaviors they engaged in while using examples (e.g., "copied block x from example code"), with timestamps, students' conversations (capture by the videos), the students' final code after completing (or abandoning) the feature demonstrated by the example (called "final code") and their comments in the interview when available. One researcher coded thoroughly these documents, generating 7 initial patterns of strategies. The researcher next worked with another researcher to merge similar strategies, generating 4 strategies, and created definitions of these strategies. The researcher next re-coded these documents again to confirm these strategies and label each example request with its corresponding strategies.

Building "explanations" [Yin17] from conversations, interviews and submissions

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

2) If students' needs were met and their strategies effectively used, we would find examples not only supporting students' individual requests, but also helping them to create more complex and creative projects. We therefore analyzed all students' project submis-

Table 3.1: 5 situations where students ask for code examples and their frequencies among a total of 88 example requests.

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

sions, following the approach by Catete et al. on developing scientific rubrics [Cat18]. Two researchers independently examined all student programs to generate lists of all features created by students in their programs, with each feature to 1) describe a distinct behavior of the game and 2) could be adapted to other game design tasks. This created a total of 23 and 25 features for breakout and space invaders respectively. Examples of such features include "an actor moves with a key pressed" and "increase score when one actor hits another". The two researchers then graded each student program by how many features in the list a student program completed. This result is discussed in Section 3.5.4.

3.5 Results

3.5.1 Motivations: Why do students ask for examples?

Table **??** shows 5 distinct situations when students requested examples. Among the 5 situations, 2 were discovered by prior work as students' motivations for using examples in *closed-ended* tasks [Wan20c]; others (denoted with ⋆ in the table) correspond to new, distinct motivations that arise from our analysis, potentially due to the context of open-ended programming. For each example-use situation we report, we also identify students *motivations*, reported in the interview data, that corresponded with these situations.

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

Others tried to integrate the example into their own code (29.5%), showing example

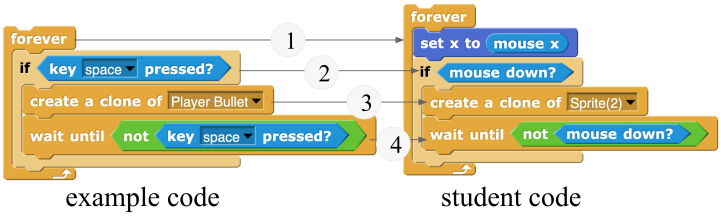


Figure 3.3: L3 copied, modified, and tested to integrate the example code to their own code **one at a time**.

use when **starting a step**. Students' quotes explained two *motivations* that may be met in this situation: 1) they wanted to know what to do next: "*[examples] isolate a step that you wanted to do and then [you could] focus on that instead of trying to do everything at once.*"[L1][4]; or because 2) they know their next step, but wanted to know how to implement it: "*We had an idea of how the code would work, but we didn't know the exact way we could all put it together.*"[E4].

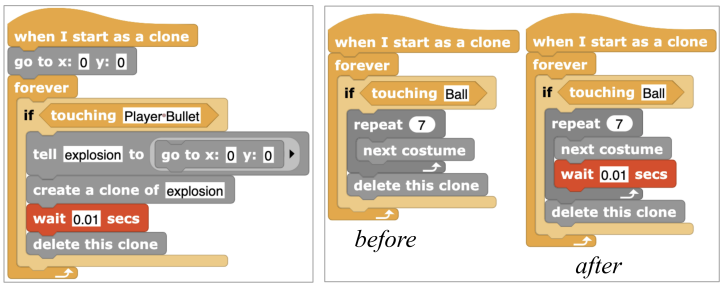
22.7% example requests came from students who were **debugging incorrect or incomplete code**: students explained that they were in the middle of completing a certain feature – "we sort of got it", but don't know "*what wasn't working with our [code])*"[E1]. 8.0% requested examples were about features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

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

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

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

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



example code student code

Figure 3.4: Using the **comparison** strategy, E7 identified meaningful differences to use in their own context, without discarding un-meaningful differences.

Integrate one block/feature at a time

In 37.9% example requests, students integrated the example code to their own code, by **copying, modifying, and testing** the example code one block or one small feature at a time. For example, in Figure 3.3, 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[5]. With each sub-step, they modified their code, sometimes testing it (2/4 times), before copying the next code block. During interview, L3 explained that they "*individually went into*"[L3] the left code palette to copy code, and commented that "*doing that allowed me to make my own modifications as I went and I better understood it.*"[L3].

Comparison to identify key differences with example

When requesting examples, many students have existing code that completes partially the target feature they need (e.g., when the feature was half-complete but was buggy). However, students' existing code can very different from the example code they requested. In these scenarios, students have described a comparison strategy, where they "looked at [their] code and that [example] code side by side" and "*compare[d] it*"[L1]. Figure 3.4 shows how E7 employed the comparison strategy. To clearly illustrate the learning scenario, we recolored the code blocks to differentiate 1) blocks that were *different* from student code to example code (grey), 2) blocks that were the *same* in student code and example code (yellow); 3) block that's *added* after reading the example code (red)[6]. The students' code

[5]In specific, such copying is made by using the example as a reference, and moving the code from the left block palette (Figure 3.2).

[6]Due to space limit, we simplified the example and student code in Figure 3.4 to only show the feature that the student asked example for (i.e., an explosion effect).

a. copy the example code to their own workspace.

b. opened documentation.

c. understand through tinkering.

Figure 3.5: L4 employed the strategy "**understanding through tinkering**" to understand an unfamiliar block - "set size to".

has many differences comparing to the examples. However, while comparing their own code to the example code, the students identified the meaningful differences between their own code and the example code, and added it onto their own code without discarding the less meaningful differences (e.g., changing costumes).

Understanding through tinkering

Tinkering refers to "an informal, unguided exploration initiated by features visible in the environment" [Bec06]. After copying examples to students' workspace, we found that some students experimented code blocks by modifying (e.g., changing variables, or by removing a block they do not understand), and testing to find the difference, showing a "test-based tinkering" behavior [Don19], which aims to understand the example code. We call this strategy "understanding through tinkering", shown in 17.2% of the example requests. For example, in Figure 3.5, L4 was confused by the code block "set size to Game Scale %": "*I do not know what Game Scale is or what it's doing.*"[L4]. They right clicked on the "Game Scale" variable to open the Help documentation, which only explained the generic usage of a variable block, but not how "Game Scale" is used in the context. So the student then changed the value of the variable from 100 to 20, tested again, realizing that the block changes sizes of a sprite: "*so, is it like, if you make it a larger number it would just get . . . ah I see.*"[L4] They later integrated the example by deciding on value of "Game Scale" to be 30.

Implement after closing the example

In 13.8% of example requests, students closed the example, and tried to implement a needed feature themselves.

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"[Bra09] approaches (called "copy-run-debug"), with two representative behaviors. 1) **copy/replace blindly**: In 2 example requests, the students copied the entire example to their own code, and completely removed their own existing code, although it was partially correct. In these scenarios, the "comparison to identify key differences" strategy would have been useful but was not employed. 2) **shallow debugging**: In 3 example requests, the students tested the copied code from example and found it did not work as expected. However, students' conversations showed they ignored the blocks they were unfamiliar with, but kept modifying other (correct) blocks that they thought produced the error, making arbitrary changes in an effort to resolve the error. In these scenarios, the "understanding through tinkering" strategy would have been helpful for the students to first understand the unfamiliar blocks.

How effective are these strategies?

One way to evaluate effectiveness is to understand how these strategies helped students to overcome their barriers when using examples. We focus on two measures: 1) Was the example successfully integrated in students' code?; 2) To what extent did the student modify the example code?, as integration and modification are two explicit goals that students have when using examples [Wan20c]. Table 3.2 shows 6 statistics for the above 2 measures:

1. *Frequency*. the percentage of strategies shown in the 29 example requests.
2. *Success rate*. proportion of the example requests that were successfully integrated to students' code. We refer to this number as *success rate*.
3. *Addition rate*. # blocks students added while using the code example / # blocks in the example code. For example, in the example request of Figure 3.4, the addition rate is $1/10 = 0.1$, as the student added one code block ("wait 0.01 seconds") among the 10 blocks of code examples.
4. *Deletion rate*. # blocks that are deleted from the example / # blocks in the example code. For example, in the example request of Figure 3.4, the deletion rate is $0/10 = 0$, as the students did not delete any code blocks while using the example.
5. *Adaptation rate*. Adaptation rate is defined by the proportion of the added code blocks that are in the example code, For example, the adaptation rate of Figure 3.4 is $1/1 = 1$,

Table 3.2: 4 example reuse strategy and the copy-run-debug behavior (i.e., lack of strategy). Strategies started with ⋆ denotes example use strategies not mentioned in prior work [Wan20c; Ich15].

strategy name	frequency	success rate	addition rate
⋆one at a time	37.9%	1.0	0.53
comparison	34.5%	0.8	0.22
⋆tinkering	17.2%	1.0	0.87
⋆impl. after closing	13.8%	0.75	0.56
copy-run-debug	13.8%	0.25	0.33
strategy name	deletion rate	adaptation rate	kept rate
⋆one at a time	0.02	0.71	0.38
comparison	0.01	0.9	0.42
⋆tinkering	0.01	0.83	0.90
⋆impl. after closing	0.0	0.74	0.47
copy-run-debug	0.0	0.84	0.15

as the one block the student added (i.e., "wait 0.01 seconds") comes from the code example.

6. *Kept rate*. The proportion of example code kept in students final code. e.g., the Kept Rate of Figure 3.4 is $5/10 = 0.5$, as the student's final code has 50% code blocks that are the same as the ones in the example code.

For addition, deletion, adaptation, and kept rates, the number presented in Table 3.2 is averaged across the 29 requests. Table 3.2 presents the following insights:

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

b) **Different strategies have different use cases and affordances.** The kept rate shows how much similarity there was between the students' final code and the example code. The *tinkering* strategy creates code of the highest similarity, and has the highest 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

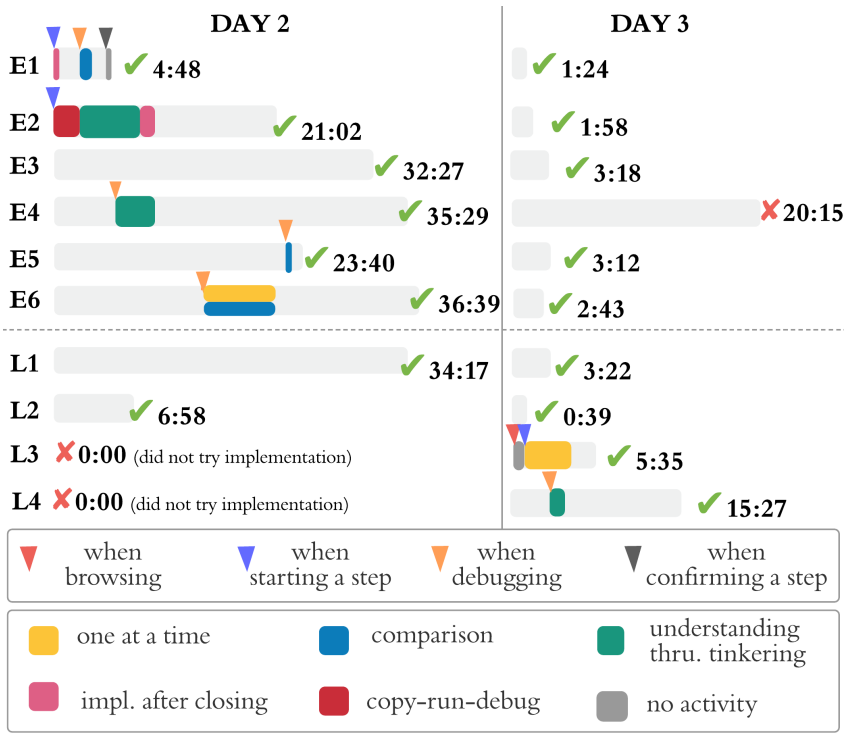


Figure 3.6: Case study: students' implementation of spawn_clones.

amount of code from example to students' own code, which was usually kept, showing that this strategy was most often employed in scenarios such as Figure 3.4, 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 3.5.2.

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 has high kept rate (0.84), showing that students modified less of the example when adding. In addition, many parts of the example were discarded from the students' final code, as they were not able to reuse them successfully, shown by its low addition and kept rates.

3.5.3 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 3.6 indicates the time duration when students were implementing this feature. They removed durations of activities on creating other features, which are sometimes interleaved in between implementations of "spawn_clones". At the end of each grey bar shows the implementation outcome – successful (tick) or abandoned (cross), as well as total time spent on implementing just "spawn_clones". The colored boxes on the grey bars are the time when students had the example interface opened, where each color represents a specific example use strategy (see Section 3.5.2). The triangles at the start of these boxes indicate the students' situation/motivation when opening the example (see Section 3.5.1). Figure 3.6 presents the following highlights:

Students change strategies at different stages of example use

Students employed diverse strategies for a variety of reuse scenarios. For example, E1 first opened, read, closed the example, and then *implement after closing* for a minute, till their code was buggy, where they asked for the example again to debug for two more minutes, using a *comparison* strategy. When they completed a correct implementation, they requested the example the third time to confirm (without changing their code). E2 shows **a transition from a lack of strategy to more active strategies**. They first used the *copy-run-debug* behavior to blindly copy the example code. They found the integration to be erroneous, but superficially debugged and were unable to fix the error. They next removed all code and requested example again, this time spending more time to modify and test different blocks in the example code. After they *understand the example through tinkering*, they removed all code again started over again, this time led to success integration. This shows that strategies are not used in isolation, and students may start with more expedient strategies, and move on to more time-consuming (and effective) strategies.

Integrating example code takes time

When implementing "spawn_clones" for the first time (on Day 2), all from the Early group succeeded; half from the Late group, who do not have access to examples, did not attempt

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

Program a feature the second time is easier

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

3.5.4 Outcomes of example use

Responding to motivations found in Section 3.5.1, students commented that examples "*gave [them] ideas*"[E6], to "*break down a task into parts*"[L1], or "*with debugging*"[E6]; expressing examples have met their needs. Connecting with strategies found in Section 3.5.2, we found students used diverse strategies to integrate example code into their own code, with high success rate. These shows that example helped students to implement features in *individual requests*. But **what are the general outcomes of using examples in open-ended programming?** Our analysis on 10 pairs[7] was insufficient for any strong claims, but we found suggestive insights from students' submissions and interview data, discussed below.

Examples helped students create more complex programs

During Day 2, the Early group who had access to examples created an average of 14.2 features, (n = 6, SD = 4.4), over 50% more features than the Late group, who did not use examples and created an average of 9.25 features (n = 4, SD = 6.9). As the two groups had

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

no statistically significant difference in their warm-up activity performance, this finding shows suggestive evidence that examples helped students create complex programs.

Students keep creating complex and interesting programs after removed access to examples

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

The Early group's higher performance than the Late group on Day 3 can be likely due to the fact that *finishing* a feature, whether or not using an example, seemed to have a clear impact to speed up implementing the feature on the next day (Figure 3.6). Therefore, the fact that the Late group only created an average of 9.25 features on Day 2 may have caused them to encounter creating many new features on Day 3 (e.g., L3 & L4 in Figure 3.6). However, having examples on Day 3 did not necessarily reduce the time spent for integration (Section 3.5.3. Therefore causing the Late group to create features less than the Early group, as the Early group may have already learned how to program those features from Day 2.

4 pairs from the Early group expressed that learning the example prior to the current task helped them create features independently, explaining that the process of making project was "*a lot easier today even though the games are different*"[E7], and that "*since we learned from the one ([i.e., example]) from yesterday, we figured out how to apply to this one.*"[E3]. This shows that many students learned or memorized how to create a feature the example demonstrated while using the example, a type of learning event (called "memory and fluency building") by prior work [Koe12].

3.6 Discussion

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

3.6.1 Support design and planning for creating complex projects

Going beyond prior work on students' goals of using examples in *closed-ended tasks* [Wan20c], we identified 3 distinct and novel motivations: browsing, confirming, and re-implementing. In particular, the most frequent motivations is for browsing (38.6%), where students needed to find features that they wanted to create. This motivation corresponds to a **design barrier** identified by Ko et al. [Ko04b], where end users needs support to identify "what I want the computer to do" [Ko04b], and was novel and distinct from prior work, which focus on identifying motivations towards **implementations** [Wan20c]. This motivation for design ties in strongly with the potential benefits of open-ended programming, which aims to empower students to create projects that they personally connect to, and engage them in the process of building an artifact of their own choice [Gro18; Guz05]. Examples should therefore, support students to create designs that feel challenging and meaningful to them.

3.6.2 Encourage effective example learning strategies

The primary highlight on our findings of students' strategies is the **diversity** of effective strategies, where we also discover novel findings on 3 strategies that were not discussed from prior work [Wan20c; Ich15]. This also contrasts with end users' example reuse strategies, discussed by prior work, which shows that they generally rely on opportunistic strategies ("getting something to work with") [Ros96]. Section 3.5.3 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 [Wan21], this leads to students' choice of strategies to overcome different types of barriers [Wan21] – for example, to overcome **understanding barrier** (i.e., "how to use an unfamiliar code block in the example?" [Wan21]), students use the strategy "understanding through tinkering"; to overcome **mapping barrier** (i.e., "how do I map a property of the example code to my own code?" [Wan21]), students employ the comparison strategy.

This diversity of effective strategies leads to important design implications – prior work has shown that one way systems could do to support users is by encouraging them to "*work in the way they are used to working, but inject good design decisions into their existing practices*" [Ko11]. Based on the ICAP hypothesis [Chi14] and the evidence from our study, we hypothesize that design decisions that encourage more *active* and *constructive* modes of interaction with the system opens up more choices for students to effectively engage

with the examples. We therefore summarize two different ways we may build systems to encourage more active use of examples, discussed below.

Transform passive code copying towards active code reconstruction

We found students who copied the example one block/feature at a time engaged in the process of reflecting and modifying the process, causing them to adapt the example effectively in their own task. This strategy corresponds to an *active* learning behavior suggested by the ICAP hypothesis, and may cause the learner to more efficiently integrate the new information [Chi14]. This spontaneous reading, using, and modifying behavior resembles the Use-Modify-Create practice [Lee11; Lyt19], which encourages learners to complete a series of activities from re-using programming code examples (*use*), to making small modifications (*modify*), to taking full ownership of learners' program by creating program from scratch (*create*). We envision that following a similar progression, an opportunity for students to progress from *modify* to *create* would yield higher learning outcomes and increased level of engagement. Such transformations can be made by programmers reconstructing blocks of examples by solving it as a Parsons problem [Par06], which breaks a correct solution into code pieces and asks students to rearrange the code to the original solution.

Foster knowledge integration through comparison of executable examples

Comparison is a classic constructive strategy to foster knowledge integration [Chi14]. In our study, we found students effectively employed a comparison strategy to locate the relevant part of an example that they could use to correct their code. Example Helpers facilitated comparison by allowing students to read their code and example code in *a parallel view*, which has been shown to enable higher transfer to solve similar, new problems, than a traditional, sequential view of different solutions [Pat13]. In addition to the parallel view to compare *code*, students were also able to run the example code to compare *outputs*, which has been shown to help students identify key differences and how they relate to changes in output [Wan20c]. Our findings that some students were able to infer the important part of the example code to use, and keep their own part of the correct solution, is consistent with the findings by prior work on comparison and transfer, indicating that the comparison allowed students to reason and differentiate the important part of the problem-solving schema [Rit07; Pat13; Gen03]. Although our system only facilitates comparison through an convenient, embedded example support interface, we envision future example support

systems to offer runnable code examples in parallel, such as different approaches to solve the same problem (similar to [Pat13]). We envision such an interface to offer support for constructive learning and generating problem-solving schema [Gic83].

3.7 Limitations & Conclusion

Limitation: we conducted our study with 24 students in just one system — we used multiple sources and prior work to triangulate our findings, and exercised rigor and reflexivity [Gui04] when collecting, organizing, and analyzing data to find valid, grounded evidence. However, we need future work to find statistical evidences (e.g., whether the strategies we found were significantly more effective than others) among larger-scaled populations.

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

CHAPTER

4

NOVICES' LEARNING BARRIERS WHEN USING CODE EXAMPLES IN OPEN-ENDED PROGRAMMING

4.1 Introduction

Creative, open-ended programming projects, such as making student-designed apps, games and simulations, are widely used in many introductory programming courses (e.g. [Gar15]). They encourage novices to pursue projects that feel authentic to them, and to express their ideas creatively, motivating them to keep pursuing CS [Guz05]. In addition, through open-ended programming, novices also learn to use computational thinking strategies (e.g., abstraction, decomposition), and may further apply them in other areas, such as math and engineering [Urb00; Guz03].

Open-ended projects can prove very difficult for novices [Gro18], in part because they encourage novices to design unique programs that address their interests and goals, which may require them to use new programming features, or accomplish new tasks, beyond

what they have already learned. Novices can also struggle to combine the *individual* programming concepts they have learned (e.g. loops, variables, etc.) into a complete program [Gro18], and they may lack experience making use of code blocks or libraries offered by the language (i.e. APIs [Gao20]).

Code examples are a common way for programmers to learn new APIs and coding patterns [Bra09], and are also considered one of the most useful learning material for novices [Lah05]. For example, research in laboratory settings suggests that novices learned to use code blocks more effectively after seeing them from code examples [Ich17]. However, novices can also face challenges learning from examples, and integrating examples to their own code [Ich15], and these challenges may be exacerbated by the challenges of open-ended programming [Ich19]. In addition, there have been few real-world deployments of code examples for supporting open-ended programming. To design example systems that better support novices' open-ended programming, a key step is to uncover their own barriers and frustrations [Guz15]. This suggests the need to explore how novices use code examples in practice, especially in a *classroom* setting, with authentic population and learning activities.

In this work, we ask the research question: **What are the learning barriers that novices face when using examples during open-ended programming?**. To answer this, We designed a system called Example Helper to support open-ended programming with a gallery of code examples. Our analysis of log and interview data found that novices encounter three types high-level barriers: decision, search and integration barriers. Based on these findings, we discuss implications and design opportunities for better supporting novices' open-ended programming with examples. The primary contributions of this work are: 1) The Example Helper system that offers a variety of learning support to novices during open-ended programming. 2) An analysis of learning barriers novices encounter when using code examples in open-ended programming, in an authentic, classroom context. 3) Identification of design opportunities to provide better example-based support to novices.

4.2 Example Helper System

The design goal of Example Helper is to allow students to view and incorporate existing programming patterns into their own code through effective use of code examples. To lower the barrier for making these programming projects [Mor11], the system is incorporated into Snap*!* [Moe12], a novice programming environment. Similar to other novice programming

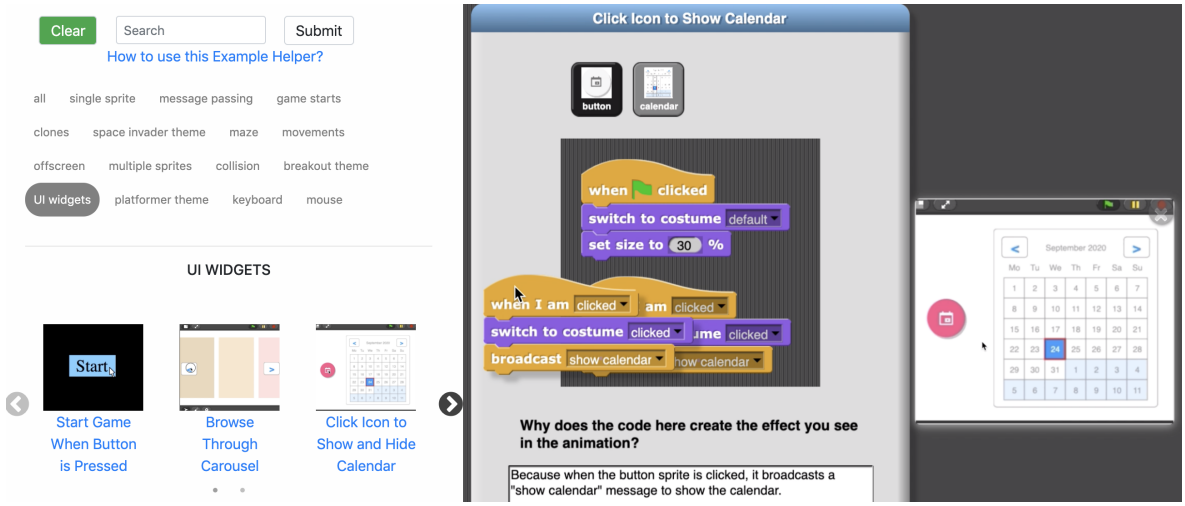


Figure 4.1: The Example Helper Interface.

environments (e.g., Scratch[Res09]), Snap*!* already offers open-source galleries of complete programming artifacts from other programmers, but these are complete projects which demonstrate many related programming features. By contrast, Example Helper offers small snippets of code examples [Rob09] that demonstrate specific functionalities, collected in a curated, browsable gallery. We developed this curated set of examples through an analysis of students' programs from prior semester, extracting key program features that were shared across students, and built these as examples. Many of these key features include usage of multiple sprite interactions[1] (e.g., in a collision event), we therefore also included examples that include usage of multiple sprites. Two experts then reconstructed examples from this repository to include cleaner and higher-quality code. When a student needs an example during programming, they can click on a "show example" button within the scripting area of Snap*!* to open a gallery of code examples. The student then follow two steps to select and use an example within their own source code:

Step 1: Search for an example. The student can find an example by: browsing through the gallery; or filtering and search for examples by clicking on a tag, or querying in a search box. The search box finds a set of examples the student need by looking for words that overlapped in the examples' names. To visually understand the functionality of the example, the student may also hover on the example to look at the gif animation of the code's output.

Step 2: Use an example. After finding a needed example, the student can click on the gif

[1]A sprite in Snap*!* is an object (i.e., in object-oriented programming) that has its own code (scripts), costumes (e.g., a button), and variables.

animation, and learn the example using the following steps:

Read the code in relation to the output. The student may click on different sprites to look at the example code for each sprite (shown in Figure 4.1). They may also look at the animation of the output next to the example code, since reading code in relation to output has been shown to trigger students to reflect on how the example code works [Wan20c]. The student can also click on the "Open the Project" button to view the example in a separate window and experiment with it.

Write a self-explanation. The student can reflect on the example by writing down a self-explanations: "What in the code here creates the effect that you see in the animation?". We designed self-explanation prompt because self-explanation is a critical step towards learning from an example [Tho20; Atk03], since it promotes students to stop and think deeper about the code example [Ale16; Ger04].

Copy the example code. To allow students to test and modify the example easily, after writing their self-explanation of the example, the student may then drag and copy the example to their own code. To discourage students from immediately copying the code without thinking about it, we restricted the length of the self-explanation answer to be at least 30 characters.

4.3 Participants & Procedure

We conducted our study in an undergraduate CS0 classroom for non-CS-majors with no prior programming experience, with 44 consented novice students, in a research university in Southeast US. The course was held online due to the COVID-19 pandemic. To create an authentic learning experience for the students, we did not collect their demographic information.

Students created open-ended projects over 3 weeks, starting from the 7th week of the course. Prior to that, they have learned the usage of fundamental programming concepts in Snap*!* , including loops, conditionals, procedures, and lists. During the first week of project-making, students were introduced to the engineering design process [Hai18], and were asked to make project pitches that may solve a real-world problem, including innovative ideas and user experience considerations.

Pair planning and programming. Students discussed their project pitches online, and then may optionally form a 2-person group if they had a similar project of interests. 18 students

chose to work individually, while the rest (26) worked in pairs, creating 31 student groups[2]. Students then planned their project design in a digital planner [Mil21]. Before students started programming, one researcher came to the Zoom classroom and introduced the Example Helper. We also instrumented Snap*!* to allow student pairs to easily transfer files through saving and loading, and encouraged them to use Zoom's screen share to collaboratively program. We encouraged students to collaboratively program because prior work has shown that in making open-ended programming projects, students achieved significantly higher performance in pair-projects than individual projects [Gro18].

Interviews. During the second week of project-making, we recruited 5 students to attend individual interview sessions with two researchers, where we recorded audio and students' screens. During these interviews, we asked students: "Is there anything you want to program, where you think an example might help you?", and encouraged them to use an example and complete the feature during the interview. During this programming process, we asked students to think aloud [Gre18]. When they asked questions, we first encouraged them to think independently, and then offered them some possible next steps if needed. After completing the feature they wanted using the example, we asked about their experience using the examples, both during the interview and in their project-making experience, such as: "Did you experience any difficulties using the examples?".

4.4 Analysis

Qualitative Interview Data Analysis. To investigate our research question about students' barriers using code examples, we began by analyzing the interview data using thematic analysis [Bra12]. Two researchers each read thoroughly all interview data, and then individually conducted line-by-line inductive open coding on the five pieces of interview, to take note of any quotes or students' programming activities, that reflects their example-usage experience and their perceptions of it. While doing the inductive coding, the two researchers used each sentence as a segment, allowing 0 or more codes per segment. To obtain accurate understanding of students' experience during open-coding, they also used the screen-recording when students did the programming portion of the interview. The two researchers then discussed and resolved discrepancies. This created a merged set of 103 initial codes. The two researchers then investigated the 103 codes to identify ones that described students' learning barriers, and combined codes that described similar incidents

[2]Since some students worked alone and some in pairs, we use the term "group" to refer to the student or students who worked on a single project.

of a type of barrier. This created 7 initial themes of learning barriers. They then discussed and sorted themes that may belong to a higher-level category, which created 3 high-level themes that described students' learning barriers, including 4 sub-themes.

Log Data Preparation. Based on the inductive and in-depth analysis on interview data, we discovered potential learning barriers among a small set of students. We then used log data to validate how these learning barriers are reflected across all participants, throughout their entire project-making classroom experience. Our log data included a total of more than 200 hours of programming activities (e.g., grabbing or destroying blocks), and students' code snapshots at every timestamp when they made a change to their code. To elicit clean data that may be analyzed further to uncover novices' example-usage barriers, we performed a pre-filtering and prepared the following three types of the log data:

Search queries. We collected all search queries that students have typed in the search box to look for an example.

Opened examples. We manually investigated and then built a profile of each incident when a student opened an example, including: 1) What examples were opened. 2) How the students found the example (e.g., whether they opened the best matches found by their search query). 3) What (if any) they did to integrate the example code to their own code (e.g., how they built, modified, or tested the example code).

Project submissions. We analyzed students' final submissions to determine: 1) whether their project submissions included functionality demonstrated by the examples, and 2) whether the functionality came from their integration of a opened example, or from students' implementing the behavior independently.

Using the above filtered data, we further conducted deductive log data analysis based on the 7 themes collected from the interview, to find evidence of how these 7 learning barriers occurred in log data of all students, described in Section 4.5.

4.5 Results & Discussion

Our thematic analysis of the interview data revealed 7 barriers that students encountered when using code examples during open-ended programming, including 3 high-level categories: *decision*, *search*, and *integration* barriers. For each barrier, we report our data by presenting the results from thematic analysis, and then the log analysis we conducted that may explain *how* this barrier occurred in all 44 students. At the end of each barrier, we briefly discuss how this barrier relates to prior work, as well as its design implications.

Decision Barrier: *Should I ask for an example?*

Our thematic analysis revealed that students encountered **decision barriers**, which occurred when students did not recognize their need or ability to ask for an example, even when they were stuck at implementing a programming behavior. For example, students may not consider asking for an example as an option: *"[My partner] hadn't figured out how to implement a timer. I don't know why we didn't think about doing examples, but we didn't."* (P3).

Among 31 student groups, 27 (87.1%) clicked on the "show example" button at least once to browse or search for an example, suggesting almost all were at least aware of the examples. However, we also found that 22% of these students (6/27) opened the example interface only 1-2 times. This may suggest that students forgot about examples once they got started with their work, or the examples were not salient as they worked.

Another explanation could be that students judged the examples to be unhelpful after viewing the interface. While this may be the case for some students, we found that those who *did* open the interface more than 2 times did so in an average of 11.38 times (up to 11 times for one group), suggesting that many students found it useful. We also found that 3 groups who did not use examples implemented functionality demonstrated by an example, totaling 7 times, suggesting examples would have been useful.

Discussion. Prior work on novices' help-seeking behaviors has shown that knowing the need to seek help is an important but challenging self-regulatory skill that requires cognitive competencies [Kar13]. Avoiding to seek help when stuck is a maladaptive learning strategy that can lead to reduced learning outcomes [Ale06]. This can be a particular challenge in programming, where students may have a strong desire to work independently, or get absorbed in their work and forget about asking for help [Pri17]. Our results suggest that this help-avoidance behavior also applies to novices' example use during open-ended programming. One possible way to address the problem of help avoidance in example systems like Example Helper is to offer help automatically (e.g. with a pop-up), which can reduce help avoidance [Mar20], especially if the system can detect when students are stuck.

Search Barrier: *How do I explain the example I want?*

We found that only 63.0% (17/27) of groups who clicked on the "show example" button ended up *opening* a code example to view. Our thematic analysis suggests that this may have been the result of **search barriers**, where students sought an example but were unable to find or articulate what they were looking for. For example, *"I think we had tried to look for a background that was like a sky or like a stage... and I don't believe we found one of those."*

(P2)

The log data reveals *how* search barriers occurred in students' search queries. We found 63 distinct searches across the 15/27 groups (48%) who used the search box to find examples(merging consecutive, identical queries). Two researchers conducted two rounds of coding on the queries to: 1) identify candidate themes that describe at least 10% of the data, discuss to resolve conflicts; and 2) count the number of occurrences of each theme. We found three primary themes: 1) interactions between multiple sprites, such as "lose a point when touching" or 'shoot" (14.3%, 9/63); 2) sprite movement such as "bounce", or "wrap around the screen" (30.2%, 19/63); 3) queries for how a sprite should look (rather than what it should do), such as "dining room", "airplane", and "people" (47.6%, 30/63). While almost half of searches were in this category of how the sprite should *look*, the examples were designed to show *functionality*, so these searches returned no results – such that only 39.7% (25/63) of *all* example searches yielded results. This shows a disconnect between how students *articulated* the example they were looking for, based on aesthetic properties, and how examples are typically organized – leading to search barriers.

Encountering a search barrier may also deter students from looking for examples in the future. Students who found and opened at least one example ($n = 17$) used the "show example" button over 5 time more (avg = 13.2; SD = 10.5) than those who did not (avg = 2.6; SD = 0.91).

In the interview, students discussed that they avoided requesting for help because of expectations that they won't find an example they needed: *"When I didn't find [a needed example], I kinda just steered away from [requesting examples]."* (P1),

Discussion. Prior work on end-users' example search behaviors showed that they may not know how to articulate what it is they want to see in examples [Dor13]. Our analysis found similar results, that novices may also encounter difficulties expressing the *functionality* they need in an example, and instead search for items that they associate with that functionality (e.g. I want a sprite to fly, so I search for "airplane"). To help novices find an example based on these aesthetic properties, we might tag examples with relevant aesthetic tags, so that novices can find examples that include an airplane (or other flying object) when they search for it. We could also try to give feedback on search queries, e.g. "Try searching for a verb – what do you want the sprite(s) to do?".

Integration Barrier: *How do I integrate the example code into my own code?*

Our interview analysis identified **integration barriers** as the challenges students face when trying to integrate an example into their own code, after finding and opening it. For

example, students noted differences between the example and their own code: *"I may have looked at the 'increase score' [example]. But I don't think I used that because I don't think we could have made it work... It wasn't a part of like our code.."* (P5). This difficulty integrating examples may be especially difficult in open-ended projects, where the students received examples that were distinct from the tasks they were trying to solve (i.e. it "wasn't a part of" their own code).

Low integration rate. To understand how many examples students actually integrated into their projects, we investigated the 153 instances of opened example performed by the 17 students who opened examples. We treated an example that was revisited multiple times by a group as one distinct opened example, creating 77 distinct opened examples, covering all of the 48 examples we designed. We define **"integrated examples"** as the distinct opened example where students managed to use code from the example and integrate it into their program to create working code[3]. We also define the **"integration rate"** as a measure of the proportion of opened examples that were ultimately integrated to students' own code (i.e. # integrated examples over # opened examples). We found the integration rate over all opened examples to be (24.7%) 19/77. This includes 9 times where students filled out the self-explanation prompt and copied the code to their own code, modifying it when needed. This number excludes 10 times where students attempted to integrate code but were unsuccessful.

We would not expect *all* opened examples to be integrated into students code. For example, sometimes students *browsed* examples, repeatedly opening examples in search of one they wanted. However, even when students searched for an example and found a relevant match, they did not often integrate it. For the 25 searched items that ended up retrieving at least one matched example, we found all of the top matched examples have been opened, but only 12% (3/25) of them were later integrated to students projects. This suggests that students were encountering barriers to integration. Our thematic analysis revealed 4 specific *types* of integration barriers that described how these difficulties occurred: *understanding, mapping, modification*, and *testing* barriers, discussed below.

Understanding Barrier: *How do I use an unfamiliar code block?*

Understanding barriers occurs when students encounter unfamiliar code blocks in an example, e.g., P2 found a "glide" block that they were unfamiliar with and asked *"What is the glide [block]?"* (P2). In addition to not understanding a new API, students may also experience doubts about the usage of the API in the context of the example: *"I don't know*

[3]This includes 2 examples that were successfully integrated and later deleted.

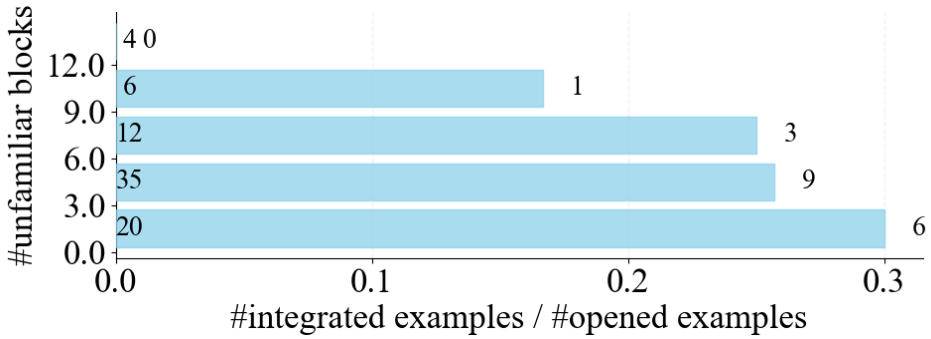


Figure 4.2: # unfamiliar blocks v.s. integration rate.

how this will work with the broadcast start timer" (P3), where "broadcast" is a code block that received "start timer" as its message.

The log data shows that the number of unfamiliar blocks indeed influenced students' ability to integrate an example into their own code. We used the number of distinct unfamiliar code blocks in each code example as the measure of unfamiliar blocks, and calculated it in the following way: 1) We took the set of blocks that appeared in at least 80%[4] student submissions in at least 1 of the 11 programming assignments prior to this open-ended project as a set of *familiar blocks*. 2) In each example, A distinct code block that doesn't belong to the set of familiar block is an *unfamiliar block*. Figure 4.2 shows that the example integration rate continuously decreased from 30% (6/20[5]) to 0 (0/4), as the number of distinct unfamiliar blocks increased from 0-3 to 12-15. This shows that some students were unable to overcome the barrier of using unfamiliar blocks when the number of unfamiliar blocks increased in an example.

Discussion. During open-ended programming, students can benefit from examples that demonstrate how to use features (e.g. blocks, APIs) that are unfamiliar, so it is important not to *eliminate* unfamiliar code. Instead, our results show that students may find it difficult to understand examples when there are *too many* new features (blocks) at once. Therefore, code example systems for novices may benefit from limiting the number of unfamiliar concepts to a certain threshold (e.g., 0-3 blocks). A system could also proactively show or link to documentation on unfamiliar concepts that students have likely not encountered before.

Mapping Barrier: *How do I map a property of the example code to my own code?*

Students encountered **mapping barriers** when trying to understand which parts of the

[4]Other thresholds produced similar results.

[5]6 integrated among 20 opened, shown by the right and left number in each bar

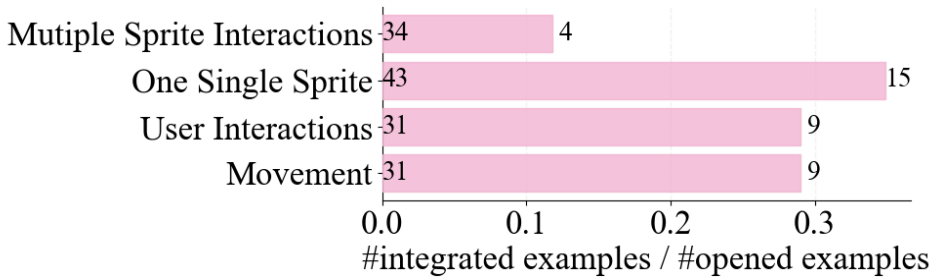


Figure 4.3: example type v.s. integration rate

example corresponded to existing parts of their own code, such as sprites — P3 explained not knowing whether the example code should go into their current sprite or a new sprite: "I don't know if creating another sprite is necessary."; P5, when working with a multi-sprite example, had mistakenly copied example code to the wrong sprite, and later acknowledged that *"Any difficulties that I might have had were... taking some time to understand how to change sprites to fit my project."* (P5).

Mapping barrier is also shown in students' challenges to integrate examples with multiple sprites. The 77 instances of opened examples included 4 primary categories: multiple-sprite interactions, single-sprite examples, user interactions, and movement. One example may belong to multiple categories. We calculated the integration rate for each category (shown in Figure 4.3), and found that multiple-sprite interaction examples, despite being the second-most popular category (with 34 distinct opens), had only an 11.8% rate of successful integration, the *lowest* among all other types of examples. This finding may be explained by the mapping barrier, since when integrating multiple-sprite examples, students face the two-fold barrier of finding "which part of the example code completes my needed behavior", as well as finding "where in my code does the example go?"

One might argue that perhaps the challenge of integrating multiple-sprite examples may also be due to them being longer. We investigated students' ability to integrate examples as the size of the example grows. We divided examples into size bins (i.e. 1-25 blocks, 25-50 blocks, etc.) and compared multiple-sprites examples to other examples within each bin. We found the integration rate was always lower for multiple-sprite examples. For example, for examples of size 1-25 blocks, the integration rate was 13% (3/23) for multiple-sprite examples v.s. 34% (9/26) for others. This shows that students struggle to integrate multiple-sprite examples to their code even when their sizes were small. However, students were still able to integrate some *large* (50-75 block) single-sprite examples (60%, 3/5). This suggests that mapping barriers with multiple-sprites, rather than an example's size, may explain

students' challenges with integration.

Discussion. This difficulty in mapping an example's property to one's own code suggests that students need support to understand the example in the context of *their own code*, e.g., potentially through adapting examples to match the student's current program. For Example Helper, this might mean changing the sprites in the example to match the student's, based on code similarity, or annotating when an example requires creating a new sprite.

Modification Barrier: *How to modify the example code to fit my own needs?*

Modification barriers occurred when students were in the middle of or have completed integrating an example code into their own code, but encountered difficulties in modifying the example to what they actually needed. For example, P2 asked for an example to implement a bounce behavior. However, the example demonstrated how to bounce when hitting a sprite *vertically*, while the students wanted to bounce after a *horizontal* collision. The student gave up using the example because they were unable to modify the example to turn the correct number of degrees: *"We were going to stick to what the code said, but the ball keeps falling off the paddle and we didn't know how to fix that, so I'm trying new stuff."* (P2)

In our log data, we found 19 instances of example modification, which followed two distinct strategies: 1) *build, test, modify, test* ($n = 15$): students started by making code blocks based on example code, then tested and modified the examples by changing blocks. Among these students, 11 succeeded and kept the new code, while 4 were unsuccessful and removed the example code entirely. 2) *modify while building* ($n = 4$): students directly modified the example code as they constructed it (2/4 succeeded). Although with relatively high success rate (68 %), some students who attempted to modify examples have been shown to have failed in doing so.

Discussion. Students' needs to modify the example show an active learning strategy [Chi14], which may cause the learner to mentally integrate the new information with their activated prior knowledge [Chi14]. Because example code introduce a different context, and therefore not work correctly, students need *debug* the examples through modification, which can be challenging [Sor07]. We may therefore include options to toggle the example, or to encourage modification of a specific part of an example after they have used it in their program, which is also supported by the *Use-Modify-Create* practice [Lee11].

Testing Barrier: *How to test the example code?*

Testing barriers occurred when students were expecting to test the example *quickly*, but encounter difficulties in doing so. During our interview, two students asked the interviewer about how to test the example code *immediately* after the student has opened the example

(e.g., *"Is there anywhere to see how the code actually works in the example?"* (P1)). In our log data, all 19 integrated examples were immediately tested once the students have completed making it. In addition, 9 opened example were tested in short time intervals, marked by at least two writing - testing cycles. This showed that students who managed to integrate the example code to their own code may have overcome the barrier of finding how to test the example code. However, our interview showed that some who were able to test the example code were still expecting quicker testing than what they experienced, and it's possible that students who did not integrate the examples successfully to their own code were discouraged by the difficulties of testing immediately.

Discussion. Although our log data showed that all students who integrated examples to their own code have tried testing the example code by running it, students in the interviews were unsatisfied with the expectation that they have to first reconstruct the example in order to run it. Prior work has shown that when learning code examples, actually running the code and see how the code executes may lead to further reflections of the code itself [Wan20c]. Our findings shows that just allowing students to view the animation next to the example code is insufficient; we should allow students easy access to run and test the example program directly.

Summary & Discussion. We found evidence that students encounter decision, search and integration barriers, leading to lower levels of exploring, opening and using examples. Despite our focus on *barriers*, our results still suggest that code examples have strong potential to support open-ended programming, as many students *were* able to successfully integrate examples into their code. Our results on learning barriers also show a strong connection between the challenges faced using examples, and more general programming skills, such as appropriate help-seeking [Mar20], articulating what code does [Dor13], and modifying code [Lee11]. In addition to design opportunities discussed above, our results also have implication for instructors, who often integrate examples into lectures and debugging sessions [Sor07], where students may still face each of the integration barriers we discussed.

4.6 Limitation & Conclusions

This work includes several limitations. With only five interviewees, our interview data may not generalize to other student groups. However, we validated our interview data with evidence from log data, showing some generalisability of these barriers. Additionally, some students programmed in pairs, others alone. We treated them equally as one unit of analysis, although they engage in different modes of programming. However, since the majority of

our log analysis focused on each unit of example requests, and therefore the unit of analysis does not affect the validity of the data we reported.

In conclusion, in this work, we presented the Example Helper system, which supports students' open-ended programming using code examples. We also identified students' learning barriers while using examples in open-ended programming, leading to design opportunities that may better support students.

4.7 Acknowledgements

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

CHAPTER

5

EXPLORING DESIGN CHOICES TO SUPPORT NOVICES' EXAMPLE USE DURING CREATIVE OPEN-ENDED PROGRAMMING

5.1 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 [Gro18]. These projects are widely used as activities and assignments in many introductory programming curricula [Gar15; McG18; Gro18] and informal learning settings [Pep07]. They engage students by allowing them to express ideas creatively [Hul15], and motivate students to keep pursuing CS [Guz05] by tying their authentic, real-world interest with their programming experience [Pap80]. However, open-ended programming can also be challenging for novices [Gro18], as implementing unique and authentic ideas may require knowledge of

programming concepts and APIs they are unfamiliar with [Gro18].

Code examples are often used by professional programmers to learn and use APIs and code usage patterns [Rob09; Bra09; Par11]. However, novice programmers lack skills such as program tracing [Lis04] and fundamental programming concepts such as variables [Ich15], 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) [Wan21].

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. To design EXAMPLE HELPER, We explored additional 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

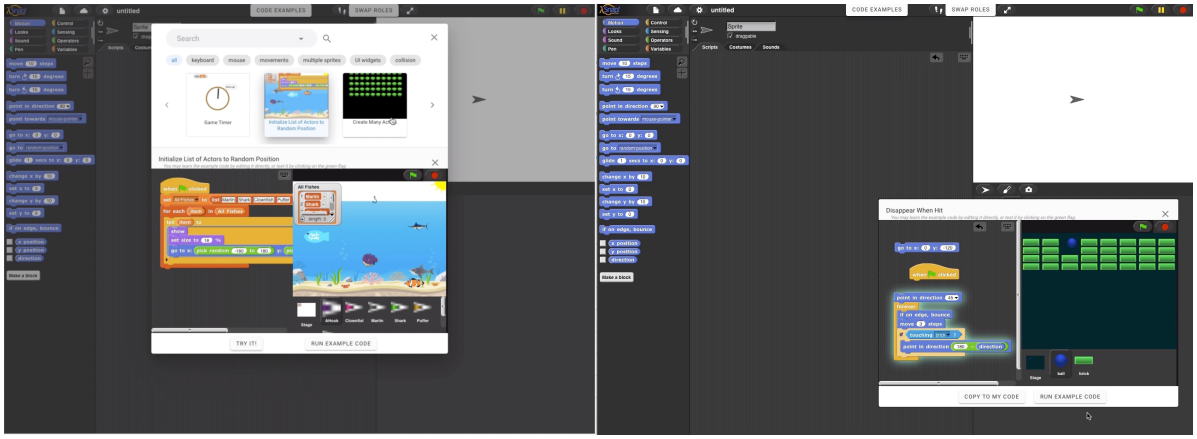


Figure 5.1: The EXAMPLE HELPER interface, which includes a selection-based gallery (left) and a playground view (right) for students to program while using the example as a reference.

that influenced students' example use, in an authentic, classroom study.

5.2 The EXAMPLE HELPER System

5.2.1 Interface Design

Figure 5.1 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 5.1-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, 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 **??**) [Wan21].

1) **Incentivise ideation.** A key activity exploratory programmers engage in is exploring and discovering new ideas in the middle of programming [Ker17b]. 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 5.1 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 [Har08; Ker17b; Ker17a]. Our prior work showed that students needed immediate, straightforward ways to experiment with the example, and need multiple modifications and test cycles to use examples effectively in their code, but was unable to do so efficiently in PROTOTYPE-EH [Wan21]. 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.

5.2.2 Example Content Design

While the Snap*!* website [Moe12] offers galleries of *complete projects* for students to browse, prior work has shown that novices [Ich17] and experienced programmers [Rob09] preferred using "snippet-sized" examples that teach an API usage pattern – how code can be organized to produce a certain behavior[Rob12; Tha21]. 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

Table 5.1: EXAMPLE HELPER design targets to address the search, decision, testing, and modification barriers students encounter when using code examples during open-ended programming.

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

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 [Wan20a]. 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.

5.3 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 from broader to more specific, 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 are the types of students who 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.

5.3.1 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. To ensure an authentic learning experience, we did not collect students' demographic information. The study happened during the second month of the students' programming course, and includes the following procedure:

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, we introduced students to the engineering design process [Hai18]. They were asked to design their products 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 on a project idea that they were both interested in. They could also choose to work independently on their project. This led to

36 student groups, among which, 10 were pairs and 26 were students who work independently[1]. After forming groups, students started with a week of planning in a digital planning system [Mil21], where they listed the features they wanted to complete in their project (e.g., "once the snake crash 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 can 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 [Gro18].

5.3.2 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 to the log data to generate patterns of interaction behaviors students engaged in when using examples [Gao20]. 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 conducted an initial coding of the note documents to generate a list of example interaction events that took place. After discussing and generating the initial lists of interaction events, 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 initial inter-rater agreement of 82.8%. They next discussed to resolve conflicts and refined the event lists and definitions, 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 sorted these events into high-level categories. This creates 3 high-level categories and 8

[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.

example-usage events. We present them in Section 5.4.1.

Pretest, planned & completed features. To investigate RQ2, we hypothesized that students' programming knowledge, or the complexity of their plans may affect 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 [Mil21]. 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 how many 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 mark those features as completed.

Example integration. To understand the outcome of using examples, for each example a group has requested, we also inspected the full log data to check whether the examples were integrated by the example. 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 into 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 variables and starter blocks. 3) structural modification, where students did bigger changes to the events, either deleting many blocks they did not need, or modifying them more to use them in their projects. We rated students' integrated examples according to the level of modifications, and present the result in Section 5.4.3.

5.4 Results & Discussion

5.4.1 RQ1: How did students use examples?

Our analysis revealed 3 high-level themes on students' example interaction behaviors: general example usage behavior, experimentation behavior, integration behavior, 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:

Experimentation behaviors.

Experimentation behavior 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 gallery window to test. This shows that many students benefited from the immediate test and experimentation features.

Integration behaviors.

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

Use cases

We illustrate below how students used the experimentation and integration strategies to understand and reuse an example code. We demonstrate how three different students used the "Move when the key is pressed" (*keymove*) example. *Keymove* demonstrates how to move actors with the key. 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 copied the example directly to their code by clicking on the "copy to my code" button. They then ran the example code 4 times, 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 had incomplete code about *keymove* 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 into their workspace.

Run-close-re-implement. Jo also 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.

Discussion

Our use case shows three different types of *opportunistic programming* strategies, summarized by prior work [Bra09; Ker17b]. In addition, *Copy-run-modify* also resembles the behavior of *debugging into existence* [Ros93; Ker17b], where students engaged in iterative test and modification to update an existing program. We found that EXAMPLE HELPER allowed students to interleave many experimentation behaviors with example copying strategies, with 85% and 57% students who ran and modified examples, respectively. This shows the potential for the EXAMPLE HELPER to enable active example integration strategies.

5.4.2 RQ2: Who used examples?

We found that only 22% (8/36) students integrated at least an 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 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 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 claim causal relationships, but only hypothesize that the students who make more ambitious plans integrated more examples, and also complete more complex projects at the end. 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 Different from prior work, which found that students with likely lower prior knowledge would request more code examples during closed-ended programming [Wan20c], our results on students' open-ended programming shows that students' prior knowledge is unrelated to whether they may benefit from using the examples. However, the complexity of students' plans – which shows how invested students are in their projects – does have a positive effect on how many examples students end up integrating 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., by detailed instructions or adaptive support during planning), which may lead to more example use, and potentially towards making better projects.

5.4.3 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. To better interpret results, we use our prior work [Wan21] 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 drawing qualitative 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 [Wan21], showing that providing students with autocomplete 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[3], 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, which EXAMPLE HELPER wasn't able to support. 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 5.4.1 shows that students actively interleave experimentation behaviors such as running the example, and modifying to test different functions on 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 shows that the editable example windows in EXAMPLE HELPER addressed students' testing barriers.

Outcomes. We also inspected students' integrated examples (Section 5.3.2) 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 5.3.2), 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.

[3]Among the 36 student groups, 17 have clicked on the "show example" button to open the gallery and use the EXAMPLE HELPER.

5.5 Limitations & Conclusion

Limitation: Students may still encounter search barriers. Our prior work showed that 47.6% searched queries were theme/asset-based, such as "people", "airplane" and "dining room" using PROTOTYPE-EH [Wan21]. 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 still encountered decision barriers. Our work found that about half of the student did not use the EXAMPLE HELPER system at all, showing that they may still encountered decision barriers. Prior work suggest that students avoid seeking help for many reasons, such as viewing requesting help as a threat to their independence and competence [But98], or forgetting about the choice to ask for help [Pri17]. The design choices we made for EXAMPLE HELPER assumes that students would open the interface at least once to use and benefit from its features, but none targeted those who would never use the system at all. In the future work, to encourage first-time usage, we may suggest 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.

5.6 Acknowledgements

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

CHAPTER

6

PINPOINT : A RECORD, REPLAY, & EXTRACT SYSTEM TO SUPPORT CODE COMPREHENSION AND REUSE

6.1 Introduction

Block-based programming environments, such as Scratch [Mal10] and Snap*!* [Har13], provide novice-friendly features such as block-based editors, and visual, interactive output. They engage programmers by allowing them to easily create artifacts that feel interesting and meaningful to them [Res09], such as games, apps, animations, or stories.

A common way for users to get started in Scratch or Snap*!* is by reusing and modifying another programmer's work as an example project [Mon12]. Reusing such example projects allows programmers to create artifacts that go beyond their own abilities, while maintaining a sense of ownership over their work [Roq16]. Such example-centric programming [Bra10] is commonly seen when learners *remixing* (i.e., copy and modify) other projects in Scratch's online community [Kha19], or when students learn new APIs by exploring

examples [Wan22].

However, program reuse requires not only knowledge of the programming language and APIs in the example [Wan21], but also skills such as code navigation and code comprehension [Gro10a]. Many Scratch or Snap*!* users are beginner programmers without strong prior programming experience [Fie14], and can easily encounter barriers when identifying, understanding, and integrating features from example programs [Wan21]. This suggests a need to help such users to navigate complex examples, identify and understand which code is responsible for specific features, and integrate them in their own program.

In this work, we present Pinpoint , a system that helps Snap*!* programmers to understand and reuse an existing program by isolating the code responsible for specific events during program execution. Specifically, a user can record an execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred. Pinpoint then identifies and displays the code responsible for creating that event, including the needed set-up code. Unlike prior systems to support example playback and understanding (e.g. [Bur13; Gro10b]), which only highlight a relevant line of code, Pinpoint presents users with complete, executable *code slices* [Xu05] that demonstrate specific functionality. We use dynamic program slicing [Agr90] to create such code slices for selected time intervals, and then further subdivide these according to different aspects of functionality (e.g., movement, cloning, etc.) by creating slice profiles [Ott93].

We evaluated Pinpoint in a user study with 17 programmers, with various levels of programming background. We found suggestive evidence that Pinpoint improved users' ability to integrate example code into their own projects. Follow-up interviews revealed specific ways that Pinpoint helped, including allowing users to relate specific example code to output, and helping to focus on relevant parts of the example code. Our primary contributions are 1) the Pinpoint system, including design goals derived from a formative study, 2) Suggestive evidence of Pinpoint 's impact on learners, and 3) a summary of students' perceptions of their reuse experience.

6.2 Related Work

While much prior work has been discussed in Chapter 2, we summarize the key findings from prior work, and discuss the gap from prior work to Pinpoint .

Pinpoint is based on Snap*!* , which is a block-based, graphical programming environ-

ment [Har13]. A core feature of Scratch and Snap*!* are their online communities, which are built on the culture of remixing [Das16], where users can click the "Remix" button to make a copy and modify to start their own version[Kha19], allowing them to creatively collaborate [Mon07] to share ideas and learn from one another [Roq16]. A large portion of projects in Scratch are remixed projects [Mon12], but as many projects to remix are relatively complex [Kha19], remixers might not always learn from reuse.

6.2.1 Code Reuse

Remixing is an example of code reuse, which refers to the process of identifying useful components of example code and integrating them into one's own program [Hol09]. Holmes et al. conducted four case studies on programmers' process of code reuse, and characterized the reuse process into two stages: 1) *locating and selecting* and 2) *integrating* [Hol09]. During the *locating and selecting* stage, programmers need to navigate through a complete example program to find relevant areas of interest [Hol09]. This process can be challenging for both experienced and novice programmers. For example, Ko et al. found that in this selection stage, software developers begin by searching for relevant information, but they often make use of limited and misrepresented cues in the program or the environment, causing failed searches [Ko06]. These results suggest that programmers need support that helps them make more accurate assumptions when relating functionality to a relevant code segment.

6.2.2 Supporting Code Comprehension & Reuse

Code comprehension refers to the process of programmers building a mental model of how a piece of code works [Von95; Gro10a]. Von Mayrhauser defined that a key cognitive process during code comprehension is generating a hypothesis of the causal effect from a code segment to its output [Von95]. Programmers of different levels may all form an incorrect hypothesis, but experts discard questionable hypotheses and form correct ones more quickly than novices [Von95].

Prior work has developed tools to support program comprehension for programming education and end users. For example, Python Tutor visualizes stack traces for students to see internal data representations of the program state [Guo13]. However, it is not designed for complex user inputs and graphical output of games and apps. Whyline in Alice[Coo00] helps users to ask why and why not questions for debugging their own code [Ko04a]. However, it can only answer object-specific questions such as "Why did Pacman resize .5?", but

not “object-relative” [Ko04a] questions such as “Why did Pacman resize after the Ghost moved”, which were frequently asked by Alice programmers [Ko04a].

Some prior work applied record/replay systems to help users understand or debug programs [Gro10b; Bur13]. Timelapse is a record/replay-based tool for debugging web applications, which points to the users the lines of code responsible for a point of interest during the recorded trace [Bur13]. Similarly, Gross et al. developed a record/replay tool to help users in Looking Glass to record and select the timeframe of interest during the playback. The system then highlights the code responsible for the timeframe [Gro10b]. However, both interfaces only highlight the lines of code responsible for the selected time frame in the output, but do not extract an *executable* code slice from the program. In contrast, Pinpoint directly addresses the learning barriers Snap*!* programmers encounter when reading and understanding code examples that are long and include multiple sprites, leveraging techniques including static [Xu05] and dynamic code slicing [Agr90]. Pinpoint allows users to select in the recorded replay a time interval, and inspect a decomposition of the original program, which only includes the part of the code responsible for the desired output, helping users learn their desired functionality in a targeted executable code example.

6.3 System Design Goals & Formative Study

Before introducing the Pinpoint system, we first present the design goals. To develop these design goals, we conducted a formative, think-aloud study with 6 students in our university’s introduction to engineering course (a required prerequisite for all CS courses) to investigate their code comprehension experience. The 6 students had various levels of programming experience. During the study, we asked the students to spend 2 minutes exploring the code of a mid-sized Snap*!* programming project (4 sprites, 57 code blocks), with the goal of being able to explain how that code achieves the output on the stage. The project was a simplified version of a space-invaders style game, explained in more detail in Section 6.4.1. Using thematic analysis [Bra12], two researchers transcribed the audio recordings of students’ think-aloud utterances and conducted open-coding on the transcripts, while referencing corresponding screen recordings for context. They next discussed and sorted the open codes into 3 high-level themes, which revealed patterns of students’ code comprehension experience when reading an unfamiliar, complex program.

Mapping from code to its runtime behavior

While reading code, students frequently make hypotheses about a piece of code's effect on the output, e.g.,"*this is the code for when the bullet touches the enemy, they disappear.*"[P1]. However, these hypotheses were frequently erroneous. Prior work has found that such false hypotheses could lead to errors and inefficiencies in code maintenance tasks [Ko04a]. Therefore **Design Goal 1** is to help students better map a code segment to its runtime behavior.

Bottom-up, linear reading for the whole program

Many students read code linearly from the top of the first sprite to the bottom of the last sprite (4/6[1]). Instead of starting from the output of the code and relating functionalities to code (top-down [Von95; Bro83]), many students' learning approach was primarily bottom-up [Von95; Gro10a], where they read code first, and then related the code to its output. As the students were trying to understand the whole program, our video data shows that only one student was able to completely read through the project within two minutes, perhaps due to the length of the program and the different task of being asked to explain the output. Therefore **Design Goal 2** is to help students to find and focus on the most important/relevant code for their goal.

Not running or modifying code

Many students either ran the code only once (2/6) or did not run the code at all (3/6), which may explain students' misconceptions about its output, perhaps due to the complexity of the project and not knowing where to start. Therefore **Design Goal 3** is to present users with relevant, executable code examples that are small and specific enough to run and modify.

6.4 The Pinpoint System

6.4.1 The Pinpoint Design

To illustrate the use of Pinpoint , assume a user wants to create a game with a character that shoots a bullet, based on a space invaders game (shown in Figure 6.1), which includes

[1] 4 among 6 students. Among the 2 other students, one started from another sprite, perhaps due to it having the least amount of code blocks; one student ran the code multiple times, but it was unclear how the student read the code, as they did not think-aloud to verbalize their thoughts although prompted

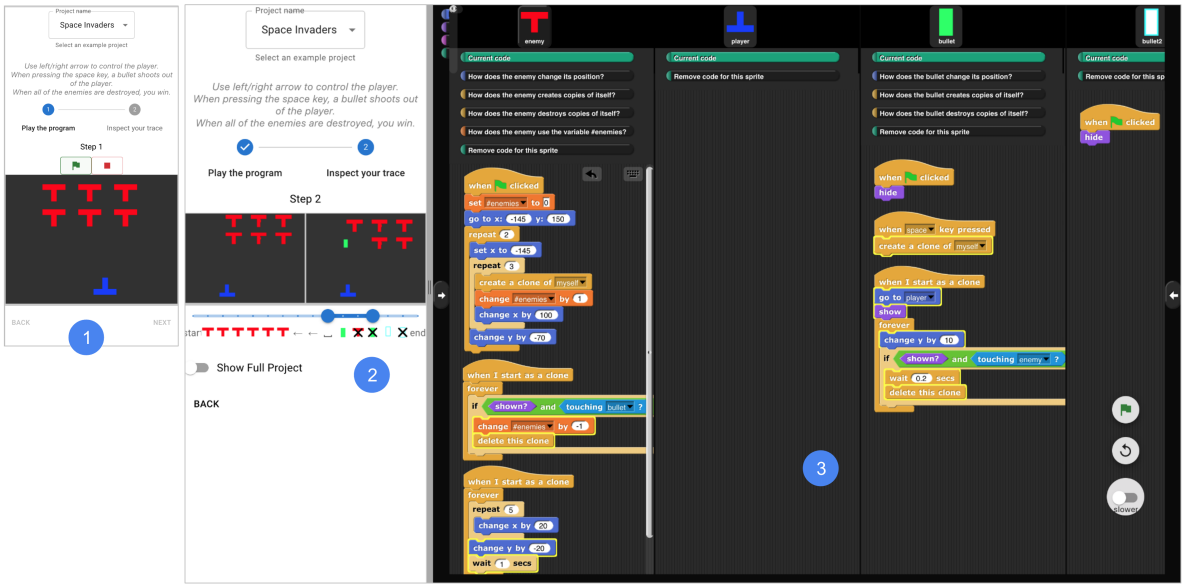


Figure 6.1: Pinpoint users can 1) record a program execution (including user input and graphical output), 2) replay a recording and select a time interval where an event has occurred, and 3) inspect an executable code slice relevant to the event, where the code executed inside the selected time interval is highlighted.

this desired feature. In Space Invaders, the player controls a ship (blue) and tries to destroy a group of enemy ships (red) by shooting bullets with the space key. Space Invaders also has other features that add code complexity: the enemy ships also move and fire randomly toward the player, and the player can dodge left and right. The player wins when all enemies are destroyed. The user wants to identify the relevant shooting code and incorporate it into their own program. Using Pinpoint , they can do the following steps, whose numbers correspond to those highlighted in Figure 6.1.

Step 1: Record an execution

Our first design goal is to help users visually map code to its runtime behavior. Pinpoint allows students to record their program execution by pressing the green flag to start recording and then pressing the stop button when they want the recording to stop. Users can view a replay of their program execution by clicking the “NEXT" button, or re-record by pressing the green flag again. While recording, Pinpoint creates an execution trace that records all user interactions (e.g., key presses), program states (e.g., variable values and sprite positions), and code executed. This is used to completely reproduce the program execution.

Step 2: Select an event

Our second design goal is to help users find and focus on the most important/relevant code for their goal. To do that, Pinpoint uses the slider bar for students to navigate through the recorded execution trace (Figures 6.1 - 2). The slider bar is automatically annotated with key events during the program execution, such as clone creation/deletion and user inputs (e.g., ←, which refers to when the left arrow key pressed). A student can select the start and end frame (each corresponding to the Snap*!* stage at the time index) to identify an event they want to explore. For example, an event of interest could be "when the space key is pressed, the bullet shoots". For this, a user could select the time interval demonstrated in Figure 6.1-2.

Step 3: Inspect the code

Our third goal is to present users with relevant, executable code examples that are small and specific enough to run and modify. To do that, as the user selects a time interval in the slider bar, Pinpoint automatically updates the relevant code slice to show only the code necessary to 1) set up the relevant event (e.g., moving the sprite to a starting location) and 2) carry out the event (cf. Section 6.4.2).

To help students understand the extracted code, Pinpoint includes the following 3 features: 1) **"How" questions.** One way to improve code comprehension is asking students to explicitly track changes to variables while reading code [Xie18]. However, as code slices may include multiple variables and implicit properties (e.g., a sprite's position, size, and appearance), it is difficult to track all changes. The user can therefore filter the code using the menu tabs above each sprite to select questions, such as "How does the enemy change its position?". This will show only the code relevant to movement and relevant control structures. This may also be useful if students are only interested in one aspect of an event (e.g., how the bullet was destroyed) but not others (e.g., how the bullet moves). We discuss the implementation of "how" questions in Section 6.4.2. 2) **Highlights for executing blocks**. Helping users quickly navigate to the key blocks for a code example has been shown to improve code comprehension [Ich15]. Therefore, Pinpoint highlights the executing blocks in yellow for a selected interval, while the code blocks that are not highlighted are required for the program to execute (e.g., setup code). 3) **The "show full project" button**. To allow users to compare the simplified code slice with the original program, students can toggle the "show full project" button to view the complete original program.

Pinpoint provides an augmented Snap*!* editor, with the following features: 1) It places

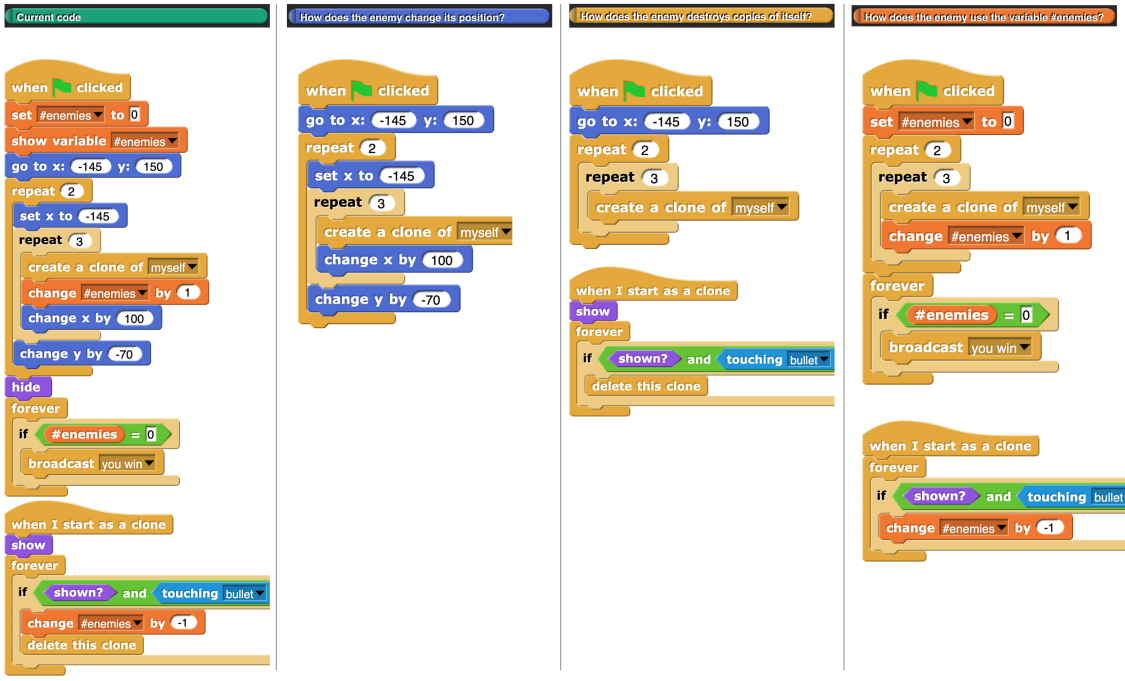


Figure 6.2: Users can also trace changes to individual variables by selecting "How" questions on different variables and attributes.

all sprites into different columns, as a selected event could impact several sprites (e.g., in "bullet shooting enemy"). 2) To allow users to both browse and execute/modify the code when needed, it uses toggle switches to open/close the code palette and Snap*!* stage. Users can add/delete/modify code blocks to tinker with any example code to verify what the example does (and if it matches their query) and explore changes that might alter its behavior.

6.4.2 Pinpoint Implementation

During Step 3 of the user experience (Section6.4.1), Pinpoint uses dynamic slicing [Agr90] to find the code relevant for the selected time interval. It then uses static slicing [Xu05] to automatically generate the "how" questions, which students can use to further isolate a code slice for the specific properties and variables of interest.

Using dynamic slicing to generate code corresponding to students' selected time interval

The program slice for a selected time interval (i.e., an executable subset of the program) is created using the program dependence graph (PDG) for the program being inspected. The PDG is a directed graph that consists of nodes which represent the program statements, and directed edges between these nodes which indicate control-flow, data-flow, or temporal (wait) dependencies. Pinpoint uses LitterBox [Fra21] to create an inter-procedural PDG which includes all scripts of the program, as well as their dependencies caused, for example, by broadcast or clone events. In general, program slicing consists of a backwards traversal of the PDG starting from a chosen slicing criterion (e.g., target statement). In Pinpoint , the code executed in the selected time interval serves as a slicing criterion, such that the dynamic slice represents the subgraph consisting of nodes that (1) are covered in the execution trace, and (2) are reachable in a backwards traversal from any nodes executed within the selected time interval. The result, for example shown in Figure 6.1, is a subset of the code which was either executed in the selected time or sets up the start state of the time interval.

Using static slicing [Xu05] to generate "How" questions

In order to generate "How" questions, Pinpoint further slices the program based on different sprite attributes (position, direction, size, visibility, sound_effect, volume, layer, appearance, and clone status). To achieve this, Pinpoint creates the *slice profile* [Ott93] for the dynamic slice of the current selection, i.e., it separates the slice into further sub-slices, each of which contains only code relevant to one attribute. For each attribute, the union slice [Kes21] is created as the union of the backwards slices from all statements that use (read) the attribute with the forward slices of the statements that define (write) the attributes within these backwards slices. The answer to a "How" question is given by the union slice for the attribute underlying the question, as for example shown in Figure 6.2.

6.5 Methods

To collect formative data on how students use Pinpoint , and understand the strengths and weaknesses of the Pinpoint system, we recruited a small group of students for a within-subject comparison study. We used the following research questions to direct our user study and analysis:

1. What is the impact of Pinpoint on students' ability to reuse example programs?
2. What are students' perceptions of their reuse experience?

6.5.1 Participants and Study Design

The goal of the study was to see how Pinpoint supported students in a standard reuse task: The students' goal is to create their own relatively simple target project, given a more complex example project implementing similar, but also additional, functionality as a reference.

Population

We recruited 17 participants from an introductory engineering course (5) and introductory computer science (12) course, including 11 males and 6 females; 10 Asian, 3 African American, 3 White and 1 Hispanic/Latino. Participants reported various levels of prior CS experience: 2 had no prior programming experience, 3 had taken only some tutorials, 11 reported taking or having taken 1-2 programming courses; 1 reported taking or having taken 3-4 programming courses. 10 students had not worked with Snap*!* or Scratch before, 6 "a few times", and 1 reported working with Snap*!* or Scratch "more than a few times".

Procedure

We used a within-subject design to understand how students use the Pinpoint compared to a standard Snap*!* interface, through a 90-minute, Zoom-based 1-on-1 study. Students first completed informed consent, a pre-survey, Snap*!* tutorial (to familiarize them with the interface), and an 8-minute Snap*!* programming pre-test, which measured students' prior knowledge in Snap*!* programming, including questions about loops, conditional statements, variables, concurrency, cloning, and message passing. They then completed two reuse tasks: Space Invaders and Catch the Dots. To explore examples, all students used Pinpoint in one task, and the standard Snap*!* interface in the other (details below). In both tasks, the students were first given a description of the target task, including a working version they could play (without seeing the code). They were then shown the corresponding example project, and asked to identify two key features from that project that would help them complete the target task. They were then given up to 20 minutes to explore the example project, and to complete the target task. We chose to limit time to 20 minutes to reduce ceiling effects (the target tasks were relatively simple) and to force students to explore the

example efficiently, since we hypothesized that given enough time students could reason through even complex examples unaided. In the first 5 seconds, students were encouraged to read the example program without directly starting to program, as learning an example prior to programming has been shown to be beneficial for students [Tra94]. After each reuse task, we conducted semi-structured interviews to ask students about their experience reusing the example program for their own code, the challenges they encountered when using the Pinpoint system, and in what way was it helpful or unhelpful.

To moderate possible effects of task difficulty, we randomized which assignment (Space Invaders or Catch the Dots) the students used Pinpoint on (i.e., a random half of students used Pinpoint first and half used it second; and independently, a random half used it on Space Invaders, and half used it on Catch the Dots). Before working on the Pinpoint task, students learned a short tutorial of the Pinpoint system using a third task with distinct features from both the Space Invaders and the Catch the Dots program. To reduce the effect of learning from the first task, we designed the requirements of the two tasks to use distinct code patterns, so completing one task would not give away the solution to the next. For example, even though the two projects both have functionalities for collision detection, the implementation was different (one used forever-if blocks, and the other one used repeat-until block).

6.5.2 Materials: Two Reuse Assignments

Each of the assignments included two parts: the learning part, where the students learn an example program; and the reusing part, where students use parts of the example program to build their target program. We created two sets of example-target project pairs: Space Invaders / Rain Game and Catch the Dots / Flower Collection Game. The two example tasks were chosen as they represent the type of programs users may prefer to reuse in Snap*!* : they include engaging elements such as user interactions and multiple-sprite interactions; they were representative of the size of programs users may remix on Scratch, and they were relatively high in code quality - which has been shown by prior work to enable more high-quality code reusing [Hil13]. To simplify the reuse tasks, students had access to a list of blocks that they may use. To allow two tasks to include distinct, differentiated features, the code for the overlapping feature in the two assignments (moving/rotating an actor with arrow key) was already provided.

Space Invaders – Rain Game

In one of the tasks, students first explore the Space Invader game from Section 6.4.1. The program includes 78 blocks across 5 sprites. They next complete a Rain Game with 7 distinct features, all having analogs in the space invader example: 1) when the space key is pressed, a water drop shows and go to the location of the cloud; 2) The water drop can move slowly downwards; 3) Can create 3 clones of trees; 4) Tree clones are created with different x coordinates; 5) when water hits a tree, water disappears; 6) when water hits a tree, tree grows; and 7) when water hits the lower edge, water disappears. Additionally, Space Invaders also includes many other features, such as using a variable to track number of enemy clones, that are not needed for the Rain Game, requiring the student to select relevant code.

Catch the Dots – Flower Collection Game

In the other task, students first explore another example program (Catch the Dots), which is similarly complex as the Space Invader Program (4 sprites, 85 blocks). Students next use the Catch the Dots program to complete a Follower Collection Game, which includes two sprites, a flower and a pot. The project includes the following 7 distinct features that have analogs in the Catch the Dots program: 1) When the green flag is clicked, a flower shows at random position and point towards the pot. 2) When the green flag is clicked, set score equals to 0; 3) The flower can move towards the pot; 4) When touching the pot body, the flower returns; 5) When touching the opening of the pot, the flower disappears and score increase by 1; 6) The flower can restart its movement after it disappears or returns to the edge; 7) When the score equals to 5, the pot says “you win”, and game stops. Similar to the Space Invaders program, The Catch the Dots game also includes many other features that are not needed for the Rain Game, requiring the student to select relevant code. Note that Flower Collection and Rain Game both include collision detection functionality, but this was implemented differently in the example programs.

6.6 Data Collection and Analysis

To understand in what way Pinpoint was helpful for students, and investigate students’ strategies when using Pinpoint , we conducted the following data collection and analyses based on our online tutorial sessions.

6.6.1 Pretest

To understand students' programming knowledge, we collected students' scores on the 8-minute pretest they completed prior to the two reusing tasks.

6.6.2 Task Performance

We collected data on the amount of time it took to complete the programming tasks and their features, based on the screen recordings and log data. As students were allowed for at least 20 minutes[2] to read and write code, but some students completed the task before the 20-minute time limit (7 for task 1, 7 for task 2, with 5 students finishing early on both tasks), we used the following two measures for task performance:

1) *Task completion*: we analyzed students' code at the 20-minute timestamp to identify completed programming features. Tasks were graded by one researcher, blind to students' condition, on a binary rubric, where each feature was either complete (correct functionality in the output) or incomplete/unattempted. Each task had a total of 7 features, detailed in Section 6.5.2.

2) *Time on task*: For the students' who completed the assignments early, we measured the amount of time spent to complete the task features. In addition, we collected students' example reading time, capped at 5 minutes, as all students were suggested to use the first 5 minutes of their programming time to read and learn the example project, and are required to start programming at the 5th minute, but they were also allowed to start earlier than the 5th minute, if they believed they had learned what was needed from the example program.

6.6.3 Qualitative Interview Analysis

We used thematic analysis [Bra12] to conduct qualitative analysis to our interview data. As P12 consented to the study but did not consent to be audio-recorded, our interview data includes 16 students. We began by performing open coding on the first 6 interview transcriptions. Two researchers independently read through the interview transcripts and conducted open coding. They next met to discuss and merge codes, and then arranged codes into high-level themes, by finding commonalities between codes. They discussed and wrote down definitions of these themes, and compiled a codebook, which includes each theme and their definitions. One researcher then coded the remaining 10 transcriptions based on the codebook, updating the codebook to include 3 themes.

[2]One student requested to stop after only 17.5 minutes on Task 2, when using the standard Snap*!* interface.

During the semi-structured interviews at the end of each student's programming session, we asked students whether they would prefer using Pinpoint or a standard Snap*!* interface for code comprehension. As part of students' response to this question (e.g., preferring Snap*!* , or Pinpoint , or mixed) is quantitative rather than qualitative, we report students' preference along with the other quantitative data in Section 6.7.1.

6.7 Results and Discussion

6.7.1 RQ1: What was the impact of Pinpoint on students' ability to extract and reuse code from an example?

Participants' interview responses show that 15 out of 16 preferring Pinpoint to Snap*!* 's interface for example comprehension (one had mixed feelings). This shows that students in general perceived Pinpoint as being helpful for code comprehension. We next discuss the impact of Pinpoint on students' task performance.

Students completed more features when using Pinpoint . We first conducted a within-subjects comparison of students' performance (i.e., # of features completed) on the task where they had Pinpoint versus when they did not. We found that students completed more of the 7 features in the task that uses Pinpoint ($M = 5.18$; $SD = 2.21$; Med=6) than when they did not ($M = 4.47$; $SD = 2.89$; Med=5), and this difference was not significant according to a Wilcoxon signed-rank test[3] ($p = 0.056$; Cohen's $d = 0.28$). This shows that overall, students were able to complete an average of 0.7 more features (10% more of the task) when using Pinpoint , compared to not using Pinpoint .

Students improved significantly more from Task 1 to Task 2 when using Pinpoint second.

One challenge with directly comparing students' performance on the two tasks (within-subjects) is that this does not control for time (Task 1 vs Task 2). Students generally improved their performance in terms of features completed from Task 1 ($M = 4.53$) to Task 2 ($M = 5.11$), having had additional practice in Snap*!* . However, if Pinpoint is helpful, we would expect students to improve *more* when they had Pinpoint on Task 2 (the **Late** group) than when they had it on Task 1 (the **Early** group). To investigate this, we calculated students' *improvement* from Task 1 to Task 2 based on the number of completed features. We found that 6 students (2 Early; 4 Late) performed perfectly on both Tasks (i.e., a ceiling effect),

[3]We use non-parametric statistical tests, as our dependent variables were not normally distributed; however, we also report means, SDs and effect sizes for completeness.

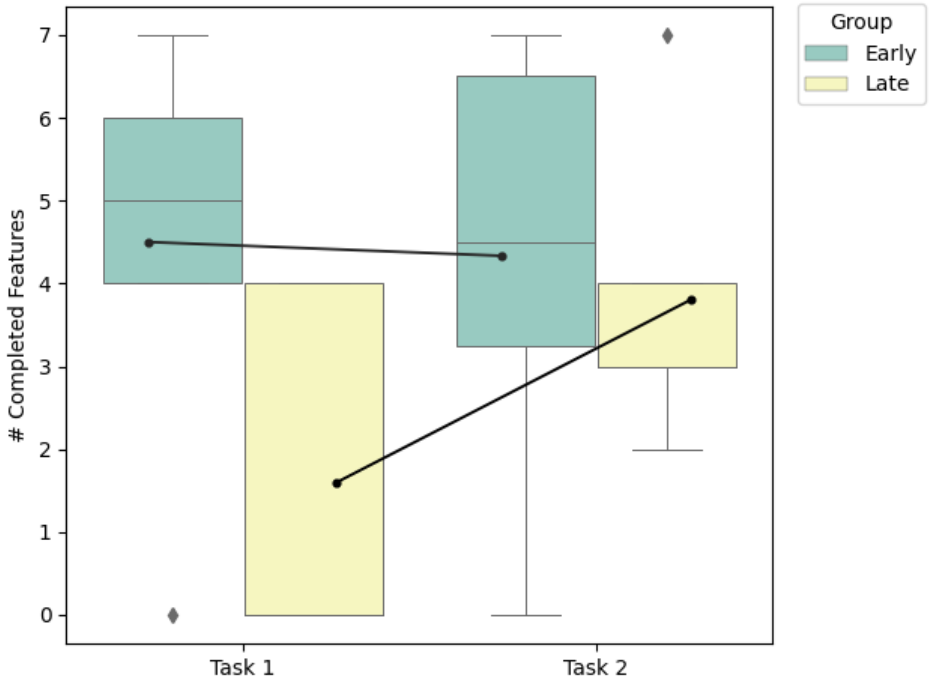


Figure 6.3: For the 11 students without perfect performance, The Late group (shown in yellow) showed significantly more improvement than the Early group (shown in green).

likely due to higher prior programming experience[4]. Since the 6 students who completed both tasks perfectly could *not* have improved, we analyze their data separately below.

For the remaining 11 students (6 Early; 5 Late), we compared their improvement from Task 1 to Task 2 (Figure 6.3). We found that on average the Early group showed far less improvement ($M = -0.17$; $SD = 1.30$; Med=0) than the Late group ($M = 2.20$; $SD = 1.30$; Med=3). A Mann-Whitney U-test shows the difference is significant ($W = 3.0$; $p = 0.031$) and the effect size (Cohen's $d = -1.92$) is quite large. Put another way, the Early group did just as well on Task 1 (with Pinpoint) as on their second task (after 20 minutes of practice, but without Pinpoint), while the Late group did much better on Task 2 (with both Pinpoint and practice), getting a median 3 out of 7 *more* features complete. It is worth noting that the Late group (after removing the 4 perfectly-performing students) performed worse overall (see Figure 6.3), which may have meant they had more room to improve. However, given the large differences between Early and Late groups, it seems safe to conclude that for students *who did need help* with the reuse task, Pinpoint had a large impact on their

[4]The 6 students who completed both tasks perfectly scored higher ($M = 7.83$, $SD = 1.33$) in the pretest than the 11 students who did not ($M = 6.36$, $SD = 1.75$).

performance.

For both of the above findings, it is also possible that which assignment the student did with Pinpoint impacted their performance somewhat; however, we found little difference between learners' performance on the Space Invaders assignment ($M = 4.59$; $SD = 2.90$; Med=6) and the Catch the Dots assignment ($M = 5.06$; $SD = 2.25$; Med=6).

When using Pinpoint , students used more time to read the example, and less time to complete the programming task. We also looked at how students spent their time when working on the reuse task. Considering all 17 students, we found that they spent on average around half a minute longer reading the example when using Pinpoint ($M = 254$ seconds; $SD = 62.0$; Med=289) than when using the standard Snap*!* interface ($M = 220$ seconds; $SD = 92.5$; Med=277), suggesting that Pinpoint may encourage students to spend more time reading and searching example code before beginning to code. However, recall that this reading time still counted towards students' 20 minute time limit. Given that students generally completed *more* features when using Pinpoint , this suggests that the extra time for reading paid off. Finally, for the 6 students who completed both tasks perfectly, the time that they spent improved from Task 1 to Task 2. We found that students completed Task 2 faster in both Early ($N = 2$; $M = 169.5$ seconds reduced; $SD = 29.0$; Med=169.5) and Late groups ($N = 4$, $M = 186.75$ seconds reduced; $SD = 199.3$; Med=147), but there was little difference between the groups. Therefore, Pinpoint may be most helpful for students with more difficulty in programming.

6.7.2 RQ2: What are students' perceptions of their reuse experience?

We discuss the key themes on participants perceptions and mindset during code reuse, based the qualitative interview analysis.

Certainty

Certainty refers to students' self-perceived confidence when making decisions or generate hypotheses during code reuse. Participants expressed relatively high levels of certainty when using Pinpoint to find a code segment. For example, participants discussed that the slider bar "*makes it a lot easier finding certain codes*"[P13], as alternatively in Snap*!* , "*you may have to search through ... each sprite, to ... find code*"[P13]. 8 students discussed that the "How" questions helped them make better searches: "*most of these questions are questions I was having ... when I would be writing code without [Pinpoint]*"[P9]. Participants also expressed relatively high levels of certainty when using Pinpoint to hypothesize the runtime

behavior of specific code segments: "*assigning an action to ... the code really helps solidify in your brain what you're supposed to do*"[P5].

Participants discussed uncertainty about understanding unfamiliar code blocks both when using and not using Pinpoint , e.g., "*it was harder to understand the repeat until [code block]*"[P13]. One participant also discussed uncertainty when trying to navigate through the slider bar to "*get it to the right time frame*"[P15].

Direction

Direction refers to the approach participants employ to navigate through the example program. Many discussed a depth-first, top-down approach when using Pinpoint , where instead of learning the entire program, they concentrate on the desired code segment: "*it basically changes the scope, like instead of having to read through the whole code, you can just zoom in on the specific piece you want and look at that.*"[P2] Three students specifically mentioned that the code highlights were helpful for further focusing their attention on the most relevant code (i.e. code that ran, as opposed to setup code): "*I could see for specific actions, which one I had to focus on*"[P10],in contrast to their experience without Pinpoint : "*it kind of points it out for you... which section of the code is working versus me just looking at the code and not being able to recognize which portions are being used.*"[P6].

For the task using the standard Snap*!* interface, similar to the formative study (Section 6.3) students in general described a breadth-first, bottom-up approach, where they "*I clicked on each [sprite] and see ... what happened*"[P4]. But some discussed negative feelings about their learning experience: "*I haven't ... learned anything ... in detail about this program*"[P4].

Annotation

Annotation refers specifically to the mental process defined by Letovsky in the Code Cognition Model [Let87], where programmers mentally *manage and structure* hypotheses about how multiple code segments interrelates, and how they map to the program's runtime behavior. Some students, while using standard Snap*!* , discussed challenges with the annotation process: "*I was trying to understand ... (the) timeline, (as) I know there's a moment when I have to add when clones start, and then there is a moment I have to put ... clone myself.*"[P8]. But many discussed being able to build the connection of different code segments more easily when using Pinpoint : "*[Pinpoint] showed me what was being done by section and so ... when I ... was progressing through [the slider bar]. [Pinpoint] added more [code*

blocks], I was like, oh, okay, so that's what this additional section does."[P6]. In addition, students discussed feeling easier to perceive code dependencies when using Pinpoint : "*I prefer ... looking at [Pinpoint] where the code's divided up. ... I think it's easier to look at how stuff are related that way, instead of just looking at it line by line.*"[P11], as for a specific event of interest, the full program include many irrelevant dependencies, which may distract students.

6.8 Conclusions and Future Work

In this paper, we present Pinpoint , a novel interface that helps Snap*!* programmers to understand and reuse an example program by isolating a specific code slice relevant to students' target event of interest. Our user study with 17 students provides suggestive evidence that Pinpoint improves students' ability to integrate code examples into their own projects. The students explained forming more confident hypothesis about a code segment's runtime behaviors, employing more focused, targeted example learning approach, and connecting different code segments more easily using Pinpoint . Overall, this suggests that Pinpoint achieved its design goals.

While these are promising results, Pinpoint currently has limitations that we will address in future work. For example, users encountered barriers when trying to understand unfamiliar code blocks both when using and not using Pinpoint . This shows that although Pinpoint allowed students to directly map a code segment to its functionality, it does not teach students the fundamental conceptual knowledge of a piece of code, which is an important sub-component of robust API knowledge [Tha20]. In the future, it may be helpful to point users to the documentation and tutorials inside Pinpoint , when they encounter unfamiliar blocks.

We may also further simplify the search experience: For example, one student noted in the interview that, as the reuse program becomes longer and more complex, the recorded trace can be long and difficult to navigate. Therefore, similar to how Pinpoint automatically generates the "How" questions to simplify search, in the future, we may intelligently generate higher-level questions regarding potential selections on the slider bar (e.g., auto-generating a "How to shoot the bullet" question), which users can click on and directly inspect a code slice, based on auto-selected time interval.

CHAPTER

7

INVESTIGATING THE IMPACT OF ON-DEMAND CODE EXAMPLES ON NOVICES' OPEN-ENDED PROGRAMMING EXPERIENCE

7.1 Introduction

Open-ended programming projects encourage students to build artifacts of their own design, and are widely used in many introductory programming courses (e.g., [Gar15; Gro18; Gon22]). However, novice programmers encounter many challenges when building open-ended projects, as they may be unsure what to make or what ideas are appropriate for the programming environment [Wan21], or lack knowledge of the programming concepts and APIs necessary for implementing an idea [Gro18].

Many students use code examples to incorporate API usage patterns into their open-ended projects, for example, by searching for them on the web and "patching them together"

into their own code [Ker17b], or by "remixing" an existing project [Kha19]. Instructors also sometimes provide a set of examples for students when giving an open-ended project assignment [Lim22]. Prior work suggests examples may support students in open-ended projects by helping them understand what is possible and doable, and introducing new APIs they could use to get started [Bra09; Wan20c]. Some systems have been built to support example use during programming [Ich17]. However, few studies have been conducted to study the effect of having access to these examples during open-ended project-making. While examples have been studied on closed-ended problems with instructor-defined goals [Zhi18; Tra94], supporting open-ended programming represents a new challenge because they inherently include unique requirements for students, such as to express their ideas creatively [Hul15], and to learn new concepts and APIs during the process of project-making [Gol20]. Without experimental data, it is unclear whether having access to code examples helps students achieve these requirements.

In this work, I investigated three Research Questions: to what extent did having access to code examples affect students' RQ1) project complexity, RQ2) self-reported task load and creativity, and RQ3) learning outcomes. I ran a controlled study in a full-day coding workshop, with students recruited from local high schools. The examples were provided through an example support system called EXAMPLE HELPER 5, on which students can search, select, test, and use an example from a gallery of code examples. The experimental group had access to 37 on-demand examples in EXAMPLE HELPER, introducing new API usage patterns, while the control group had no access to novel examples[1]. During the workshop, students experienced an introduction to Snap*!* , and then complete a 2.5-hour open-ended programming activity, with pre and post-tests. The primary contribution of this work is a formal controlled study about the impact of code examples on novices' open-ended programming.

7.2 Related Work

7.2.1 Open-Ended Programming

In this work, I use the term "open-ended projects" to mean *projects where students partially or fully design the programming artifact they create, where they set their own goals that are meaningful to them [Gro18]. This is in contrast to more traditional, "closed-ended"

[1]To keep conditions comparable, the control group did have access to EXAMPLE HELPER, but it only provided 5 examples from an earlier tutorial. See Section 7.3 for details.

assignments, where students are given tasks with clear, defined goals, often accompanied with step-by-step instructions [Fra20]. The interest-driven coding experience of open-ended projects has been shown to provide a sense of ownership and self-confidence in various engineering, science and design classrooms [Mor13; Cue05; Bar12]. In computer science classrooms, according to a survey on 134 introductory programming instructors in the UK, many instructors found that the creative experience of building an app or game by themselves, makes computing more interesting to novice students [Bla13]. The BJC curriculum designers also highlight that through open-ended projects, students learn the important CS concepts and APIs "not by doing exercises on them, but by building things that need them", which sparks creativity and sense of self-confidence [Gol20].

However, multiple challenges arise when students freely choose what they want to build. Students may encounter difficulties exploring or generating ideas and may choose expedient or simple solutions, preventing them from committing to thoughtful work [Ker17b; Kok16]. In addition to the need for ideation, novices also encounter barriers, such as difficulties in applying knowledge of programming concepts into code implementation [Gro18] and difficulties understanding and reusing unfamiliar APIs [Kha19]. These challenges show the importance of balancing students' free-form discovery with some level of external support [Nos96].

To overcome these challenges, some instructors provide students with a set of example programs when building open-ended projects, as examples offer new ideas, and demonstrate how unfamiliar APIs and idioms can be used [Ker17b; Lim22]. However, little work has investigated the impact of having access to these examples on students' open-ended programming. Additionally, although an example system can help students to better explore and integrate new ideas, example systems for open-ended programming has been relatively scarce – as students freely choose and define their own goals, it is difficult to anticipate content to offer to the students [Wan21].

7.2.2 Code Examples

Code examples demonstrate a sample solution for a programming task or subtask. Most frequently, they are used in closed-ended programming tasks, but recent work has explored open-ended settings as well.

Using code examples on closed-ended programming tasks.

The "worked example effect" from Cognitive Load Theory [Swe06] explains how examples support learning: reading and understanding a step-by-step demonstration of a

problem (a "worked example" [Swe06]) prior to solving a related problem helps learners to complete the latter problem more efficiently. From a theoretical perspective, the worked examples reduces the amount of mental resource needed for searching for a problem-solving schema, and therefore retain more mental resources for processing and learning the material [Swe06]. Many prior studies have found that learning a worked example prior to solving a similar problem led to higher learning gains than solving both problems from scratch [Tra94; Sha13; Jen21]. A meta-analysis conducted by Crissman on 62 studies across various disciplines found that learning a worked example prior to problem-solving produced an overall moderate, positive effect on students' final learning outcome [Cri06]. These results also hold for programming worked examples. For example, Trafton and Reiser experimented with teaching undergraduate students Lisp programming by interleaved examples and solving an isomorphic problem, and found that compared to the group with only problem-solving pairs, the interleaved group achieved higher post-test scores [Tra94].

These studies show strong evidence of the effectiveness of using worked example examples [Swe06] with step-by-step instructions, presented before students solve a programming problem. However, many students – especially when working on a programming problem, also use what I call *on-demand examples*, such as those found on the internet [Gao20], in course resources (e.g., textbooks), or from an on-demand example system (discussed below). Unlike worked examples, students seek this help *during* problem-solving, and these examples may not have step-by-step instructions. However, less research has evaluated the effectiveness of these "on-demand" examples - and almost none in the context of open-ended programming. The Peer Code Helper [Zhi19] is an on-demand example system in Snap*!* . The system provides code examples as a sample solution of the same task, when students complete a programming assignment in Snap*!* . A study with 22 high school novice students showed that these examples help students solve the task quicker than the group without these examples. However, unlike the learning effect commonly seen with worked examples, the two groups' post-test performances were not different [Zhi19]. Ichinco and Kelleher augmented the Looking Glass system [Pow07] to include an example dialog, which presents code examples for a similar problem when students work on a closed-ended program completion task [Ich15]. In a formative study with 9 pairs of children aged 10-15 using the example system, Ichinco and Kelleher found that students encountered difficulties understanding the examples and connecting the examples to their own tasks. Some students also felt distracted by looking at the examples [Ich15]. This raises the question of whether the learning effect found from the structured learning of a worked example can transfer to the on-demand example learning scenarios, as students primarily

focus on solving their current problem, and examples from a different context introduce additional barriers.

Using code examples for open-ended programming.

Example use during open-ended programming is very different from that during closed-ended programming – unlike learning an example for solving a closed-ended programming problem, prior work has shown that example use during open-ended programming is usually "opportunistic" [Ker17b; Bra09], with the primary goal of saving time [Bla02; Ros93; Ros96; Lan89]. For example, Rosson and Carroll investigated Smalltalk programmers' example use, and summarized a typical reusing process as "getting something to work with" by directly copying an example to their own workspace, and then testing the example code and "debug into existence" [Ros93].

Prior work has noted a number of challenges programmers face when using examples during open-ended programming. Gao et al. analyzed the log data on 18 programmers' web use when completing a React programming task, and found that the programmers heavily relied on searching for and copying code examples from the web, but more than half of the copy-and-pasted examples were later deleted; with most of the kept examples modified to fit in programmers' own programs [Gao20]. 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 [Ama19; Kha19]. These challenges observed from remixed projects show that while on-demand examples are commonly used for programmers, it is still unclear whether using examples can lead to more advanced API use, or better project-making outcomes for novice programmers.

Only one study, conducted by Ichinco, Hnin, and Kelleher on the Example Guru system, has evaluated the effect of code example use during *open-ended* programming. The Example Guru uses pre-defined rules to check what examples a student may need, and then prompt an example to the student during open-ended programming. In a study with 78 children aged 10-15, Ichinco, Hnin, and Kelleher asked students to use 30 minutes to create two animations in Looking Glass, and found that students in the Example Guru group accessed more examples compared to another group that uses documentation, and applied more APIs from those examples into their own code [Ich17]. Building on this evaluation of the Example Guru system, in this work I measure the impact of example use in a longer (3 hours), ecologically valid learning setting (a day camp classroom), where students made a project from start to finish, which they had the freedom to design and plan themselves. Further, I not only investigate the extent of API use from code examples, I also investigate how access

to examples affects the complexity of students' projects, their post-task performance, and students' perceptions of task load and creativity.

7.2.3 Open-Ended Programming

I use the term "open-ended projects" to mean projects where students partially or fully design the programming artifact they create, where they set their own goals that are meaningful to them [Gro18]. This is in contrast to more traditional, "closed-ended" assignments, where students are given tasks with clear, defined goals, often accompanied by step-by-step instructions [Fra20]. The interest-driven coding experience of open-ended projects has been shown to provide a sense of ownership and self-confidence [Mor13; Cue05; Bar12]. The BJC curriculum designers highlight that through open-ended projects, students learn the important CS concepts and APIs "not by doing exercises on them, but by building things that need them", which sparks creativity and a sense of self-confidence [Gol20].

However, multiple challenges arise when students freely choose what they want to build. Students may encounter difficulties exploring or generating ideas, applying knowledge of programming concepts into code implementation [Gro18], or in understanding and reusing unfamiliar APIs [Kha19]. To overcome these challenges, some instructors provide students with a set of example programs when building open-ended projects, as examples offer new ideas, and demonstrate how unfamiliar APIs and idioms can be used [Ker17b; Lim22]. However, little work has investigated the impact of having access to these examples. Additionally, Example systems for open-ended programming have been relatively scarce – as students freely choose and define their own goals, it is difficult to anticipate content to offer to the students [Wan21].

7.2.4 Code Examples

Code examples demonstrate a sample solution for a programming task or subtask. Most frequently, they are used in closed-ended programming tasks, but recent work has explored open-ended settings as well.

Using code examples on closed-ended programming tasks.

The "worked example effect" from Cognitive Load Theory [Swe06] explains how examples support learning: When students learn a step-by-step demonstration of a problem (a "worked example" [Swe06]) prior to solving a related problem, they expend less mental resource to search for a problem-solving schema, and therefore retain more mental resource

for processing and learning the material [Swe06]. A meta-analysis conducted by Crissman on 62 studies across various disciplines found that learning a worked example prior to problem-solving produced an overall moderate, positive effect on students' final learning outcome [Cri06]. These results also hold for programming worked examples. For example, Trafton and Reiser experimented with teaching undergraduate students Lisp programming by interleaved examples and solving an isomorphic problem, and found that compared to the group with only problem-solving pairs, the interleaved group achieved higher post-test scores [Tra94], showing strong evidence of the effectiveness of using worked example examples [Swe06] with step-by-step instructions.

However, many students – especially when working on a programming problem, also use what I call **on-demand examples**, such as those found on the internet [Gao20], in course resources (e.g., textbooks), or from an on-demand example system (discussed below). Unlike worked examples, which presents an example solution of a similar problem *before* problem-solving, students seek on-demand examples *during* problem-solving, and these examples may not have step-by-step instructions. However, less research has evaluated the effectiveness of these "on-demand" examples – and almost none in the context of open-ended programming. The Peer Code Helper [Zhi19] is an on-demand example system in Snap*!* , which provides code examples as a sample solution of the same task, when students complete a programming assignment in Snap*!* . A study with 22 high school novice students showed that these examples decreased the time needed for task completion, but did not led to an improved post-test performance. Ichinco and Kelleher augmented the Looking Glass system [Pow07] to include an example dialog, which presents code examples for a similar problem when students work on a closed-ended program completion task [Ich15]. A formative study with 9 pairs of children aged 10-15 found that students encountered difficulties understanding the examples and connecting them to their own tasks, and felt distracted by looking at the examples [Ich15]. This raises the question of whether the learning effect found from the structured learning of a worked example can transfer to the on-demand example learning scenarios, as students primarily focus on solving their current problem, and examples from a different context introduce additional barriers.

Using code examples for open-ended programming.

Prior work analyzing the phenomena of example use during open-ended programming has been focused on how examples were found and reused from the web (e.g., [Gao20; Kha19; Bra09]. Specifically, prior work has noted a number of challenges programmers face when using examples during open-ended programming. For example, an analysis by Gao et al. on 18 programmers' web use during React programming found that programmers

heavily relied on searching for and copying code examples from the web, but more than half of the examples were later deleted; with most of the kept examples modified to fit in programmers' own programs [Gao20]. 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 [Ama19; Kha19]. These challenges show that while on-demand examples are commonly used for programmers, it is still unclear whether using examples can lead to more advanced API use, or better project-making outcomes for novice programmers.

I found only one controlled study, conducted by Ichinco, Hnin, and Kelleher on the Example Guru system [Ich17], to have evaluated the impact of having access to code example use during *open-ended* programming. The Example Guru uses pre-defined rules to find and prompt an example to the student during open-ended programming. In a study with 78 children aged 10-15, Ichinco, Hnin, and Kelleher asked students to use 30 minutes to create two animations in Looking Glass, and found that students in the Example Guru group accessed more examples compared to another group that uses documentation, and applied more APIs from those examples into their own code [Ich17]. Going beyond this work, I measure the impact of example use in a longer ($\tilde{3}$ hours), ecologically valid learning setting (a coding workshop classroom), where students made a project from start to finish, which they had the freedom to design and plan themselves. Further, I not only investigate the extent of API use from code examples, I also investigate how access to examples affects the complexity of students' projects, their post-task performance, and students' perceptions of task load and creativity.

7.3 Methods

7.3.1 The EXAMPLE HELPER System

To investigate the impact of students' example use during open-ended programming, I aim to use an example system that can provide access to multiple different examples. This system can can allow students to browse, search, test, and copy examples, which is consistent with how students search for and reuse examples on the web [Gao20; Bra09]. Therefore, I chose the EXAMPLE HELPER system [Wan22] from my prior work (Chapter 5). The EXAMPLE HELPER is an example gallery developed as an extension of Snap*!* [Moe12], a block-based novice programming environment. A student can browse and search for

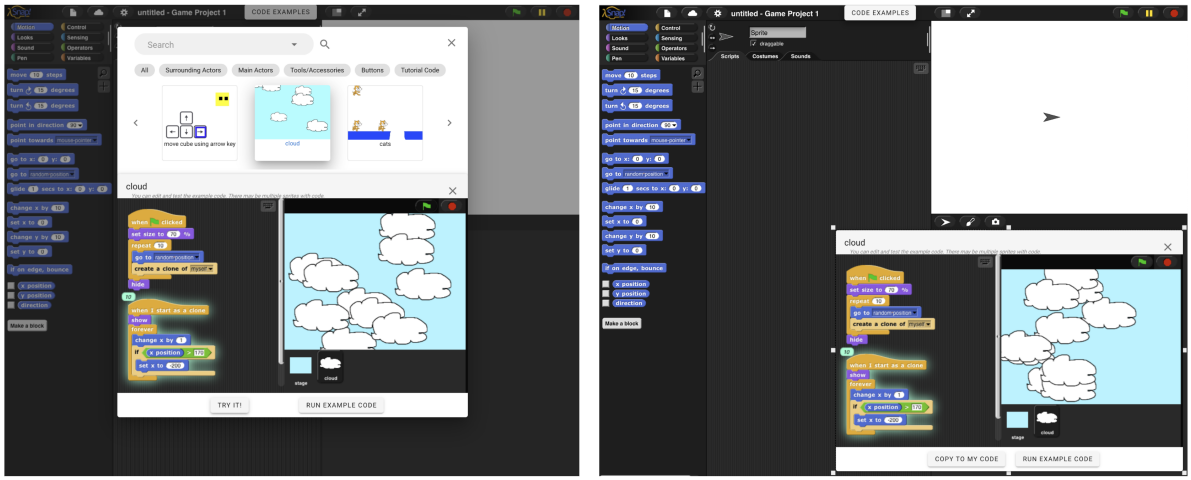


Figure 7.1: The EXAMPLE HELPER interface [Wan22]. Students can browse and search for examples in a gallery (left) interface, and then test and modify them in a sandbox (right), where they may also click to copy the example into their own workspace.

examples from an example gallery, and to select, modify, test and copy examples to their own code. The interface of EXAMPLE HELPER is described in Chapter 5.

Figure 7.1 shows the interface of the EXAMPLE HELPER system. When programming in Snap*!* , a student can click on the "show example" button at the top-center of the Snap*!* window to open a gallery of code examples (Figure 7.1-left). They can use the search box, or select different tags to find an example they need. They may also simply click through the left and right arrows to browse through different code examples, represented as gif animations. To inspect the code, they may click on the gif to see a preview window, shown in the lower half of Figure 7.1-left). Just like a standard Snap*!* window, the student may view code from different sprites (i.e., actors), edit the code, and run it to see its output on the right of the preview window. If they click on the "try it" button on the lower right of the preview window, a new sandbox will show up (Figure 7.1-right), where students can continue to work with the example, or click on the "copy to my code" button to copy the example code to their own code. A qualitative analysis on the log data of 46 undergraduate students using EXAMPLE HELPER has found that the system affords a variety of exploration and experimentation behaviors, showing that the system is suitable to be used to support open-ended programming.

Example Content Design: I followed suggestions presented in Chapter 5 to construct examples that are self-contained, small in size, and demonstrate programming patterns that are commonly seen in novices' open-ended programming projects [Wan20a]. They are

also designed to align with the learning goals of the coding workshop (explained below). I designed a total of 37 examples, among them, 5 were also used in tutorial sessions to introduce students to Snap*!* . The examples I included in my study had an average of 21.9 blocks (min = 5, max = 58, std = 13.13), which is consistent with the number of blocks typically needed to complete a specific feature in Snap*!* , summarized by [Wan20a].

7.3.2 Participants & Learning Context

To investigate the impact of code examples, I aim to conduct a controlled study in an authentic learning context where students have enough time to learn about Snap*!* and build a complex open-ended project. Therefore, I conducted a full-day coding workshop, which was repeated on 3 consecutive Saturdays with different participants. During each day, students learn and practice Snap*!* programming in the morning, built an open-ended project in the afternoon.

Participants: I recruited 46 participants from multiple high schools from a state in the Southeastern region of the United States, including 14 Females and 31 Males, and 1 who preferred not to disclose. The 46 students were split over each of the 3 consecutive Saturdays. There were 18 White, 13 African American, 10 Asian, 3 Hispanic or Latino, 1 Caribbean Islander, and 1 American Indian or Native Alaskan students. Participants reported various levels of prior CS experience: 8 reported having no prior programming experience, 22 reported having completed a few programming tutorials before, 8 reported having taken one programming course, and 8 reported having taken at least 2 programming courses. I recruited participants from diverse prior experiences, as open-ended programming projects are common at all levels, and this prior experience was balanced across conditions.

Learning Goals: To prepare students for building complex open-ended programming projects and to allow all students with different levels of background to learn the basics of Snap*!* and programming, I designed the workshop to cover a variety of CS concepts and Snap*!* API use-cases. Specifically, the Snap*!* environment features building visual, interactive programs (such as games and apps) and commonly include multiple sprites[2] that can interact with each other over time. Therefore, apart from general CS concepts such as loops and conditionals, my learning objectives also include more advanced Snap*!* concepts such as concurrency, which can be implemented in Snap*!* by using multiple scripts[3]; and Snap*!* -specific APIs such as cloning, which can create multiple copies of a sprite.

[2]Sprites in Snap*!* are similar to objects in object-oriented programming environments.
[3]Scripts refers to a set of connected code blocks in Snap*!* .

The curriculum and assessments for the coding workshop were developed around the following 3 learning goals, divided into 12 subgoals: 1) CS Concepts: understand and apply knowledge of basic CS concepts, including loops, conditionals, variables, and concurrency. 2) Understand and apply the various concepts and usage of Snap*!* 's cloning APIs, including: clones are copies of their parent sprite; clones inherit properties of the parent sprite when they are created; scripts under the header block "when I start as a clone" can run concurrently for each individual clones after they are created; a clone can be deleted using "delete this clone". 3) Understand and use other important Snap*!* APIs correctly, including message passing, randomness, appearance (e.g., graphic effects), touching and stopping the Snap*!* program. I applied these learning goals to design the example content, introductory materials, and the pre and post-tests, discussed below.

Conditions: Participants were divided across three days, and during each day, a random half of the students were assigned to the control condition, while the other half were assigned to the example condition. To understand students' example use in an authentic learning setting, conditions were assigned non-intrusively: all students had access to the EXAMPLE HELPER system throughout the day. In the morning, all students were introduced to Snap*!* programming and the EXAMPLE HELPER system; in the afternoon, students design and build an open-ended project in Snap*!* . All students had access to 5 "tutorial examples" throughout the day, which were used for introducing students to Snap*!* and the EXAMPLE HELPER system. However, the control group only had access to these 5 tutorial examples, while the example group (i.e., EXAMPLE HELPER group) an additional set of 32 examples, with various different contexts [4].

7.3.3 Procedure

I followed the same procedure for each of the three Saturdays. I describe the timeline of one day's coding workshop below:

Morning: Tutorial and pre-test. Students arrived at the classroom at 9:30 am. After completing the pre-survey, they learned a 2-hour Snap*!* tutorial. The goal is to help students learn the 12 sub-learning goals, and become familiar with the EXAMPLE HELPER interface. Students completed a scaffolded closed-ended programming exercise, while making use of 5 tutorial examples from EXAMPLE HELPER. Afterward, students completed a 15-minute

[4]For example, Figure 7.1 shows an interface for an example group student – all tags had relevant code examples underneath the search bar. However, the control group only had code examples underneath the final tag "tutorial code".

pre-test, which includes 12 questions, each corresponding to one of the 12 sub-learning goals. Afterward, they do a 30-minute lunch break.

Afternoon: Programming, post-survey, post-test, and post-assignment. Students planned and programmed an open-ended project in the afternoon. They first learned the project requirements: they should build a game under the theme "Animals and Nature", and follow requirements based on 2 dimensions: A) Game Experience dimension: they should build a complete game, that has 1) starting and 2) ending mechanism; 3) actor and 4) user interactions; 5) obstacles; 6) actors can gain score or lose life; and 7) artistic details (e.g., scrolling backdrop or graphical effects). B) Applying Programming Knowledge dimension – they should apply the programming knowledge and APIs they have learned in the morning (i.e., the 3 key learning goals) to their final projects. After learning the requirements, students did a 20-minute planning session, where they describe the key mechanics they wanted to include in their projects. After planning, students did open-ended programming in Snap*!* for 2 hours and 30 minutes. As discussed in Section 7.3, the control group had access to 5 tutorial examples, while the example group had access to all 37 examples. After programming, students were asked to use 30 minutes to complete a post-survey, a post-test, and a post-assignment. The post-survey includes questions on task load and creativity; the post-test includes 12 isomorphic questions as the pre-test; and the post-assignment (called "Catch The Egg"), is a close-ended task that combines knowledge from the 12 sub-learning goals, but is different from any of the code examples. The assignment asked students to build a game that has clones of eggs falling, and a cart at the bottom to catch the egg and increase their scores. The game should stop after a certain amount of time, and should print the user's final score.

7.3.4 Measures

To collect data based on the RQs from Section 7.1 – I discuss the measures I collected to evaluate students' project complexity (RQ1), their self-perceived Task Load and Creativity (RQ2), and outcomes on learning (RQ3).

Pre-test. I designed the pre-test questions based on the 12 sub-learning goals under the 3 learning goals. Each question corresponds to one learning goal. In specific, the 4 CS concept questions were adapted from sample questions from the CS Concept Inventory [Rac20]. Across both conditions, their pre-test score had a mean score of 7.11 (std = 2.48, min = 2, max = 11), meaning that an average student completed around 60% of the questions correctly. The pre-test scores had no significant difference between groups ($p = 0.57$) - The

control group had 21 students, with mean = 7.0 (std = 2.035), and the experiment group had 25 students, with mean = 7.2 (std = 2.75).

Example Usage Statistics. The Example Helper system tracks log data on all actions students perform with examples, such as opening an example, or copying an example to their own code. Based on the log data, I collected the timestamps and the content of the examples the students have requested. I next compared the examples students have requested and their final submitted project, to inspect how examples were integrated into students' projects. 44 among the 46 students have submitted their projects (20 in control, 24 in experimental)[5]. For consistency of comparison, my analysis of example usage statistics focus on the 44 students.

Project complexity score (RQ1). To evaluate the complexity of the 44 submitted students' projects, I constructed a grading rubric with three dimensions: 1) Game Experience, which calculates how many of the 7 requirements (Section 7.3) a student's project achieved, to measure whether a student's game is complete and interesting. 2) Applying Programming Knowledge, which calculates how many of the 12 sub-learning goals were applied in a student's project. For example, when a student's project has at least one loop statement, I grade the "the loops concept" sub-goal as complete. 3) Number of Game Mechanics, which calculates the number of game mechanics students implemented successfully in their projects. To define "mechanics", I followed the approach by Cateté, Lytle, and Barnes on developing scientific rubrics [Cat18]. Two researchers independently inspected 10 projects to generate a list of mechanics, each of which describes a distinct behavior of the game (e.g., an actor can move with an arrow key), that can be completed using a combination of programming concepts and Snap*!* APIs. They next discussed, merged, and verified the mechanics, generating a list of 26 mechanics. Next, the two researchers used the mechanic list to each grade 17 student's projects, and they adjusted the mechanic list to include extra mechanics, generating a final list of 37 mechanics. Based on the above grading rubric, an average student achieved 8.6/12 programming knowledge scores (std = 3.30), 4.77/6 game requirements (std = 2.20), and 6.57/37 game mechanics (std = 8.65). For each student, the overall project complexity score is calculated as the sum of the three scores (Game Experience, Applying Programming Knowledge, and Number of Game Mechanics), each score scaled to the range 0–1.

 # Blocks (RQ1). For each student's submission, I calculated the number of blocks to indicate the size of the project, and the number of distinct blocks in students' projects to

[5]Two students chose to not submit their projects. As a voluntary coding workshop, I did not oblige students to submit their final projects if they chose not to.

indicate the variety of APIs a student applied in their project.

Use rate difference (RQ1). To compare the preferences of block and mechanic usage between the control and example group, for each block, I define the "use rate" as the proportion of students who used this block in their project. For example, 66.7%(16/24) example group students and 20% (4/20) control group students used the block "delete this clone" at least once in their projects, so the block's use rate is 66.7% in the example group, and 20% in the control group. To compare the preference of block usage, I sorted the blocks based on the **"use rate difference"**, which indicates the use rate of a certain block in the example group, minus the use rate of the block in the control group. Therefore, a positive/negative use rate difference indicates that the block was more/less popular among the example group than the control group. Similarly, the use rate difference was also calculated for each of the Game Mechanics collected from the project complexity measurements.

Self-Perceived Task Load & Creativity (RQ2). I measured students' perception of the task load based on students' 5 Likert-scale responses to the 6 NASA-TLX questions [Har88]: 1) (Mental Demand) How mentally demanding was the task? 2) (Physical Demand) How physically demanding was the task? 3) (Temporal Demand) How hurried or rushed was the pace of the task? 4) (Performance) How successful were you in accomplishing what you were asked to do? 5) (Effort) How hard did you have to work to accomplish your level of performance? 6) (Frustration) How insecure, discouraged, irritated, stressed, and annoyed were you? I also calculated the total score of the task load by summing up students' ratings from all 6 questions. All 46 students completed the task load surveys, with an average score of 18.02 (std = 3.03). I measured students' self-perceived creativity based on students' 5 Likert-scale responses to 5 CSI questions [Che14]: 1) (Result-Worth-Effort) What I was able to produce was worth the effort I had to exert to produce it. 2) (Expressiveness) I was able to be very expressive and creative while implementing my ideas. 3) (Exploration) It was easy for me to explore many different ideas, options, designs, or outcomes during programming. 4) (Immersion) My attention was fully tuned to the programming activity, and I forgot about the system/tool that I was using. 5) (Enjoyment) I was very engaged in the programming activity - I enjoyed this activity and would do it again [6]. I calculated the total score of students' self-perceived creativity by summing up students' ratings for all 5 questions. All 46 students completed the CSI surveys, with an average score of 17.52 (std = 3.95).

[6]The last "collaboration" question was excluded, as students worked independently

Learning (RQ3). I measure learning based on two different aspects: 1) the Normalized Learning Gain (NLG) [Mar07]; and 2) the post-assignment performance. I designed the post-tests as 12 isomorphic questions from the pre-test, and calculated NLG as (post − pre)/(12 − pre), according to [Mar07]. Students' post-test scores had an average lower than the pre-test scores (mean = 6.67, std = 2.86, min = 0, max = 12), lower than the pretest, which has an average of 7.11 (std = 2.48, min = 2, max = 11). This is likely due to the fatigue of the full-day coding workshop – 47.8% (22/46) students' scores dropped during the post-test, with 6 students dropping at least 3 points. In addition, instructors anecdotally reported off-task behaviors during the post-test. A larger portion of score drop may have been caused by students randomly guessing their answers: Among the students whose score dropped, 32.8% (7/22) had post-test scores lower than or equal to a score resulting from guessing (post-test score <= 3), while among these same students, there were less than 10% (2/22) who had scores lower than or equal to 3 in pre-tests. This suggests that students did not fail to learn during the activity, but rather that in this voluntary camp setting, many chose not to meaningfully engage with the post-test. For completeness, I still present the measure; however, it is likely not reflective of differences in learning between conditions. I discuss the implication of this result in Section 7.5. Consequently, the NLG had a mean of -0.18 (std = 0.70, min = -2.5, max = 1). In addition to the NLG, students also worked on the "Catch The Eggs" post-assignment, where neither group had access to examples. I also calculated the number of objectives completed in the catch-the-egg assignment. Across both conditions, the mean is 3.84, meaning that students completed 3.84 / 8 objectives on average. Note that students complete the post-survey, post-test, and post-assignment in 30 minutes, so the lack of time may have caused an average student to have completed less than half of all objectives.

7.4 Results

To compare the control and example groups' data on 1) project complexity, 2) self-reported task load and creativity, and 3) learning outcomes, I first use Shapiro-Wilk test to test sample normality; then I compare the corresponding measures between groups using the Mann-Whitney U test for non-normal samples, and Student's t-test for normal samples[7] [8].

[7]The effect size based on the Mann-Whitney-U test is reported as r value, while the effect size based on the Student's t-test is reported as Cohen's d value.

[8]While my analysis includes multiple comparisons (e.g., project complexity score, task load and Learning), I follow suggestions from [Per98] to report original p-values and discuss its implications, rather than adjusting

Table 7.1: Statistics for the control (ctrl.) and example (exp.) groups, with the p-value and effect sizes. * means the data follows the normal distribution.

	1. project complexity					
	game	knwl.	mech.	total sc.	#blks*	dist. #blks*
n(ctrl./exp.)	20/24	20/24	20/24	20/24	20/24	20/24
mean(ctrl./exp.)	0.65/0.71	0.65/0.72	0.48/0.53	1.78/1.96	87.3/101.04	23.2/28.42
std(ctrl./exp.)	0.31/0.31	0.3/0.29	0.27/0.28	0.84/0.83	47.15/50.17	7.86/7.84
p-value	0.54	0.38	0.55	0.48	0.37	0.04
effect size	0.09	0.13	0.09	0.11	0.28	0.65

I present my findings below.

Descriptive statistics. I first collected data on example requests and integrations. I found that 1) most students from the example group has requested (83.3%, 20/24) and integrated (79.8%, 17/24) at least one example. An average example group student has opened 2.76 examples (min = 0, max = 13, std = 3.07); and has integrated 1.42 examples into their final code (min = 0, max = 5, std = 1.35). 2) Much fewer students from the control group (40%, 8/20) have requested and integrated at least one example (25%, 5/20), and they requested and integrated fewer examples too: An average control condition student has opened 0.9 examples (min = 0, max = 4, std = 1.22); and has integrated 0.3 examples into their final code (min = 0, max = 2, std = 0.55). Likely due to the more variety of examples, the example group had a significantly higher number of example requests ($p = 0.01$, $d = 0.82$) and integrations ($p = 0.001$, $d = 1.05$). This suggests that any differences in the outcomes of these groups, reported later, were likely due to their use of examples.

When integrating examples, students typically build features on top of it by removing parts of the code and including their own implementations – Among the integrated examples, students on average removed 34.4% (min = 0, max = 77.8%, std = 0.24) of the example code, and added 60.5% (min = 0, max = 828.6%[9], std = 1.36) of their own code on top of it.

Findings for RQ1: Having access to examples allowed students to build projects with significantly more variety of APIs, and with a marginally higher project complexity score (non-significant).

Table 1 shows that compared to the control group, the example group built projects with higher overall size (# blks), but non-significantly; but their distinct number of blocks (distinct #blks) was significantly higher ($p = 0.04$), with a large effect size ($d = 0.65$), showing

alpha values.

[9]The maximum amount of added code is 828.6%, as some students build more complex code than the example.

that the example group used significantly more variety of APIs. However, this significant difference in block variety did not lead to a significant difference in project complexity: while the example group had a consistently higher average on the game experience (game), applying knowledge (knwl.), and the number of mechanics (mech.) scores, the differences were not significant, with small effect sizes.

I investigated the control and example groups' "use rate difference" (Section 7.3.4) in their choices of Snap*!* APIs, and found that **the additional types of blocks used by the example group primarily came from the code examples**: I found that the top 3 blocks that have the largest use rate difference between example and control groups were: "delete this clone" (66.7% v.s. 20%[10]), "pick random _ to _" (50% v.s. 10%), and the operator "_ < _"(54.17% v.s. 15%). These blocks are often demonstrated by the examples, appearing in 35.14%, 27.03%, and 24.3% of all 37 examples, respectively, with none appearing in the 5 tutorial examples. But when I removed the code blocks that students copied from examples, the two groups' projects no longer had any significant difference in the variety of code blocks, with the control and example group having an average of 22.3 and 19.25 distinct code blocks, respectively.

Lastly, I analyzed each mechanic's use rate difference between example and control group, and found that **the control group may have spent more effort in building the basic mechanics they learned from the morning tutorials**: The mechanics with the highest positive use rate difference (i.e., more popular in the example group) frequently came from integrating one or more examples. For example, the "end game on collision" mechanic had the highest positive use rate difference (62.5% v.s. 35%), and can be integrated from examples, as 12 examples include collision (i.e., a sprite touching another), and another (non-overlapping) 3 examples include ways to stop the game. However, the mechanics with the lowest negative use rate difference (i.e., more popular in the control group) commonly came from the morning tutorials. For example, "having a sprite that can move with key press" is a mechanic demonstrated in the morning tutorial, and has the lowest negative use rate (50% v.s. 70%). Other mechanics used more commonly in the control group also had similar characteristics: the mechanics with the second and third lowest use rate difference are "rotate with mouse" (0% v.s. 15%) and "conversation between sprites" (8.3% v.s. 15%). These mechanics are similar to what they have learned from the morning tutorial, where students may have had more time to learn and practice these mechanics before the afternoon project-making session.

[10]Meaning the block "delete this clone" was used by 66.7% (16/24) example group students and 20% (4/20) control group students.

Table 7.2: Statistics for the control (ctrl.) and example (exp.) groups, with the p-value and effect sizes. * means the data follows the normal distribution.

	2. task load & creativity		3. Learning	
	total TLX sc.*	total CSI sc.*	NLG	post-assign. sc.
n(ctrl./exp.)	21/25	21/25	21/25	21/24
mean(ctrl./exp.)	17.43/18.52	16.71/18.2	-0.21/-0.17	3.05/4.71
std(ctrl./exp.)	2.97/2.93	3.81/3.86	0.58/0.77	2.92/3.06
p-value	0.23	0.21	0.43	0.07
effect size	0.36	0.38	0.12	0.26

Findings for RQ2: Having access to examples helped students experience a higher sense of creativity (non-significant), but also a higher task load (non-significant).

The example group reported a total creativity score ("total CSI sc.") higher than the control group (Table 7.2) – this difference is not significant, although having a medium effect size ($d = 0.38$). Specifically, the example group's creativity rating is consistently higher in *all 5 subscales*. On the other hand, the example group also reported higher task load ("total TLX sc."), but the difference was non-significant, with a medium effect size ($d = 0.36$). This rating is consistent over 4/6 TLX questions: Physical and Temporal Demand, Performance, and Effort. This shows suggestive evidence that, examples may have helped students feel more creative, but may also have increased students' self-reported task load, causing them to feel that they needed to spend more time and effort in building the projects.

Findings for RQ3: Having access to examples may have allowed students to perform better on a later, closed-ended programming task *without* examples.

I found that the example group performed more than 50% better than the control group on their post-assignment, shown by the "post-assign. sc." in Table 7.2. This difference was not significant ($p = 0.07$), but had a medium effect size ($r = 0.26$). This suggests that the example group may have learned more API usage patterns from the code examples, and applied them to a different post-assignment without examples, but further evaluations are needed to confirm this finding.

On the other hand, both groups produced negative normalized learning gains (NLG) (Table 7.2). This is likely due to the fatigue of the full-day coding workshop (discussed in Section 7.3.4). As many students did not engage meaningfully with the post-test, the NLG is likely not reflective of differences in learning between conditions. Additionally, the score drop in the example group was smaller than that in the control group, although the difference was not significant ($p = 0.43$), with a small effect size. One explanation is that the example group may have felt happier (e.g., shown by their higher creativity ratings)

after the programming experience, and therefore are more willing to spend more effort in the post-test.

7.5 Discussion & Conclusion

Discussion 1: More time on example learning can lead to less time on practicing learned knowledge. Findings for RQ1 discussed that the example group may have built more creative mechanics demonstrated from examples, and the control group may have spent more effort in successfully implementing more basic mechanics introduced from the morning tutorials. These differences in students' choices of features show that examples may encourage more exploratory programming behaviors, but may also cause students to spend less time practicing some basic programming knowledge they have previously learned. This may help students who want to explore and build a creative project, but may also harm students who need more practice in basic programming knowledge. This shows the importance of personalizing on-demand examples, so that students who need more practice may be prompted with examples that reinforce the knowledge and APIs they have learned, rather than introducing new knowledge.

Discussion 2: Students may potentially learn knowledge and APIs by modifying and integrating examples. Prior work shows that programmers commonly use on-demand examples "opportunistically", where they rely heavily on "getting something to work with", rather than learning the examples [Bra09]. Lab studies found that by simply copying examples, students can complete programming tasks faster, but don't do better in post-tests [Zhi18]. However, unlike findings in [Zhi18], my analysis shows mixed results, as although both groups did not do better in post-tests, the example group's post-assignment performance is better ($p = 0.07$). This is likely because students had to modify a large portion of an example to integrate it into their own code (Section 7.4). During this process, instead of passively copying, they can actively modify and experiment with the example, which may potentially lead to more knowledge assimilation [Chi14]. My curricula likely encouraged modification by providing examples that are small, self-contained, and commonly used in other student-designed games. This allowed students to find enough commonalities between the examples and their own ideas, to decide to reuse them; but also enough differences so that they could build their own mechanics on top of the examples.

Limitation 1: Generalizability. One limitation of the work is that I deployed only one example system (EXAMPLE HELPER). While the findings may not generalize to systems that

are drastically different to EXAMPLE HELPER, my findings may apply to future systems that are built similar to EXAMPLE HELPER, where students may browse, select and copy code examples; or to instructors, who may also provide a set of code examples to students when they work on open-ended programming projects (e.g., [Lim22]).

Limitation 2: Full-day coding workshop and worse post-test performance. Another limitation that impacted my results is the study duration of my workshops. Given that many students were unfamiliar with Snap*!* , the full-day coding workshop had a steep learning curve and lasted long hours. This had an impact on students' fatigue and engagement levels, as shown by instructor-reported off-task behaviors and students' negative NLGs. The NLGs, as a result, did not provide meaningful information on whether access to examples led to better code-tracing problem performances. Future work should investigate ways to reduce students' fatigue, for example, by breaking down the coding workshop into multiple days.

Conclusion In this work, I investigated the impact of having access to examples on students' open-ended programming experience. I found that having access to code examples allowed students to create projects with more variety of APIs, feel higher creativity, but also experience a higher task load. I also found suggestive evidence of students being able to apply the knowledge to a better code-writing program. My results show the potential for code examples to both increase creativity and support learning and the importance of designing personalized code examples that can both increase creativity and reduce task load.

CHAPTER

8

CONCLUSION

8.1 Research Questions

In this section, I discuss how this work has addressed the following research questions:

- RQ1: *What are novices' motivations, strategies, and barriers when using code examples during open-ended programming?*.
- RQ2: *How can we design code examples to address students' decision, search, mapping, understanding, and testing barriers?*
- RQ3: *What is the impact of having access to code examples on students' open-ended programming experience?*

8.1.1 Research Question 1

RQ1 asks, *What are novices' motivations, strategies, and barriers when using code examples during open-ended programming?* This RQ is addressed in Chapters 3 and 4. In Chapter 3, I conducted a study with 12 pairs of high school students working on open-ended game

design projects, using a prototype version of the Example Helper system, which allows students to browse examples based on their output, and to view and copy the example code. I analyzed interviews, screen recordings, and log data, and identified 5 motivations and 4 strategies that arise when students use examples. Specifically, I found that students request code examples primarily to explore ideas; to understand how to start a step; to debug incorrect code; to confirm their own code; or to avoid re-implementing the same code. I also found 4 different strategies students employ when requesting examples: by integrating one block/feature at a time; by comparing their code with the example code to identify the key differences; by using tinkering to understand an example code; or by implementing a feature after closing the example. I found that some example requests (13.8%) also exhibit a lack of strategy, which is indicated by students copying and replacing the example code with their own code blindly, and using shallow debugging methods, such as making arbitrary changes. I found that students almost always used a strategy of some form, which often led to success and adaptation; and that different strategies have different use cases and affordances. However, a lack of strategy can lead to failures of integration.

In Chapter 4, I explored novices' learning barriers when interacting with code examples during open-ended programming by deploying the same prototype version of the Example Helper system, with 44 novice students in an introductory programming classroom. During the study, students work on an open-ended project in Snap*!* , and I collected log data and interview data afterwards to understand students' experiences. I found three high-level barriers that novices encountered when using examples: decision, search, and integration barriers – students may not recognize the need to use examples while getting stuck (Decision Barrier), they may not know how to explain an example they want (Search Barrier), or not know how to integrate example code into their own code (Integration Barrier). Specifically, they may fail to integrate an example code into their own code when they do not know how to use an unfamiliar code block (Understanding Barrier), when they do not know how to map a property of the example code to their own code (Mapping Barrier), or when they encounter difficulties to test or modify a piece of example code (Modification Barrier).

Understanding the need, strategies, and challenges students encounter when using examples informed the second part of my thesis, where I build tools to support such needs, strategies, and challenges. Combining the analysis of students' needs and challenges during example use, I find the following insights for building tools to support novices during open-ended programming:

1. Students encounter many challenges to find an example that's relevant to their project. Chapters 3 and 4 found that students may not know *when* to look for an example;

and that they may not know how to explain a certain feature they wanted (Search barrier). Even when they find an example, they fail to recognize how the example relates to their own project (Mapping Barrier). This shows that we need to provide novel ways for students to look for examples they need, and to improve the designs of example support systems, so that students understand when and how to request an example, and how it can be applied to their own code.

2. Students may understand and integrate examples more successfully into their own code when they have experimented with or tested the example. As the prototype of the Example Helper system did not allow students to directly test and modify the example code, Chapter 4 explained that students needed such features in an example system to allow them better understand an example. On the other hand, findings from Chapter 3 show that the tinkering strategy, where students modify and test a specific part of an example they were unfamiliar with, has led to a higher example integration rate. Therefore, example support systems should allow students to experiment with the system (e.g., in a sandbox), so that students would understand how the code in the example relates to its output.

8.1.2 Research Question 2

RQ2 asks, *How can we design code examples to address students' decision, search, mapping, understanding, and testing barriers?* This RQ is addressed in Chapters 5 and 6, where I discuss the design and deployment of two complementary example support systems to address the above learning barriers.

Chapter 5 explores how to design code examples to support novices' effective example use by presenting our experience of building and deploying a remodeled Example Helper system (later referred to as Example Helper). It not only includes the gallery-based example searching feature from its prototype, but also a sandbox to test and experiment with the example. This remodeled Example Helper is built to specifically address three barriers identified from Chapter 4: 1) it addressed the search barrier by providing immediate search results and autocompleted suggestions; 2) it addressed the decision barrier by allowing previewing and testing the example in the browsing interface; and 3) it addressed the testing barrier by allowing students to run, modify, and view immediate output inside the example window. I 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. The results show

the autocomplete searches, as well as the accessible, editable preview and the sandbox feature, led to relatively low incidents of search, decision, and testing barriers.

Chapter 6 presents Pinpoint, a system that helps Snap*!* programmers understand and reuse an existing program by isolating the code responsible for specific events during program execution. In Pinpoint , a user can record an execution of the program (including user inputs and graphical output), replay the output, and select a specific time interval where the event of interest occurred, to view code that is relevant to this event. The Pinpoint system was designed based on three design goals: 1) To help students better map a code segment to its runtime behavior; 2) to help students find and focus on the most important/relevant code for their goal; and 3) to present users with relevant, executable code examples that are small and specific enough to run and modify. I conducted a small-scale user study to compare users' program comprehension experience with and without Pinpoint, and found suggestive evidence that Pinpoint helps users understand and reuse a complex program more efficiently. Specifically, students explained that when using Pinpoint they were able to generate more confident hypotheses about a code segment's runtime behaviors, and employ a more focused, targeted example learning approach (as they can specifically choose a subset of an example to inspect based on its output). And students explained that connecting different code segments was easier when using Pinpoint .

The remodeled version of Example Helper and Pinpoint are built to address the learning barriers identified from Chapter 4 from different perspectives. While the Example Helper aimed to address the decision, search, and testing barrier by using auto-complete suggestions and testing sandbox, the Pinpoint re-designed the example search experience as a student-centered, output-based example search. By allowing students to record an execution trace, replay the output, and select a specific time interval to search for a specific code slice, Pinpoint addressed students' decision and search barrier by allowing students to create and use an example based on an output they want. It also addressed students' understanding barrier by allowing students to interrogate the example in relation to its output, investigate and connect different code slices, and use highlights to help students focus on the code that caused the visual effect they are interested in.

8.1.3 Research Question 3

RQ3 asks, *What is the impact of having access to code examples on students' open-ended programming experience?* This RQ is addressed in Chapter 7, where I discussed the evaluation of the Example Helper system on students' open-ended programming experience. In this

chapter, I evaluated the impact of having access to code examples on open-ended programming, through a study with 46 local high school students in a full-day coding workshop. I conducted a controlled study, where half of the students had full access to 37 code examples using EXAMPLE HELPER, and the other half had 5 standard, tutorial examples. I found that students who had access to all 37 code examples used a significantly larger variety of code APIs, perceived their programming experience as relatively more creative, but also experienced a relatively higher task load. I also found weak evidence of a better post-assignment performance from the EXAMPLE HELPER group, showing that some students were able to learn and apply the knowledge they learned from examples to a new programming task. The results show that access to code examples during open-ended programming helped students become more creative, build projects with a larger variety of APIs, and learn new knowledge for future tasks.

8.2 Design Principles & Future Work

I summarize the following design principles based on the iterative experience of designing systems to support example use during open-ended programming:

Increase students' awareness and trust towards code examples by including students in the process of example co-creation. The evaluation of the Pinpoint system provides evidence that by involving student in the process of example co-creation and allowing them to inspect specific code based on its output, Pinpoint was effective in helping users understand a complex code example, integrate it into their own code, and produce more complex programming project, compared to a standard example code interface.

For Pinpoint , the example co-creation process was generated by allowing students to record, replay, and select an output they are interested in, and Pinpoint produces the relevant code slice. This process has been shown to allow students to feel more confident when making decisions about whether to use the example, or generating a hypothesis about what the example does. On the other hand, such example co-creation process can also take many other forms. For example, students can use any pictures they found in an open-source repository (e.g., the online Scratch community [Fie14]) to indicate a specific project theme they are interested in, and an example generation system can find simple examples that are only related to the specific theme, and help students to complete the project using examples.

Promote active code integration by encouraging prototyping and experimentation

As a result, students felt more willing to use the remodeled Example Helper system, and integrated more examples into their own code. Chapter 5 explained that by building a sandbox feature embedded in the Snap*!* , the remodeled Example Helper addressed the decision and testing barriers by supporting students' natural exploratory programming behaviors [Ker17b] (i.e., testing and tinkering), which are also effective strategies to promote example understanding and integrations, as discussed in Chapter 5.

Connect example use closely with students' programming experience. Chapter 7 shows that although examples allow students to build projects with a variety of APIs, and to perceive their project-making experience as more creative, examples also introduce extra task load to the students, causing them to feel their project is harder and needed more effort. Similarly, the evaluation of the Pinpoint system from Chapter 6 also shows that students need more time to use the interface to learn the example, although the time is paid off by them needing less time to complete the project. These results show that while examples support students' project-making, they may also lead to more time and effort for students to learn the examples.

To support students with examples that are more connected to students' programming experience, future work can generate examples based on the specific themes that students work on, and prioritize examples that are connected more closely to students' current programming progress. For example, by building an auto-detector, so that when students are developing code for aesthetic details, such as moving background, mouse-clicking, and actor appearance, prioritize examples that are specifically about such effects; and when students are working on actor mechanics such as jumping, prompt examples that are related to students' current goals.

8.3 Contributions

The contribution of this thesis includes:

- A systematic analysis of the goals, strategies, and barriers novices experience or encounter when using code examples during open-ended programming. This analysis generates insights on what novices need from using examples, how they use them, and the challenges they encounter. Consequently, this analysis creates insights into ways to build systems to support students' example use.

- The design and deployment of two interconnected systems, which support, respectively, testing-centered example integration (Example Helper); and just-in-time exam-

ple extraction (Pinpoint), each addressing one or more barriers students encounter during example reuse. Specifically, the evaluation of the Example Helper system shows evidence that this remodeled interface generates more example testing and integration behaviors compared to the prototype Example Helper; and the evaluation of the Pinpoint system shows that Pinpoint helps students to learn a complex code example better than a simple programming interface, allowing them to produce more complex programs.

- The empirical evaluations of the impact of code examples on students' open-ended programming experience. This is the first work to measure the impact of example use on open-ended programming, in an authentic learning setting (a full-day coding workshop), where students planned and programmed an open-ended project in 3 hours. This is also the first work to comprehensively investigate how access to examples affects the complexity of students' projects, their post-task performance, and students' perceptions of task load and creativity. The results show that access to code examples during open-ended programming helped students become more creative, build projects with a larger variety of APIs, and learn new knowledge for future tasks.

BIBLIOGRAPHY

[Agr90] Hiralal Agrawal and Joseph R Horgan. “Dynamic program slicing”. In: *ACM SIGPlan Notices* 25.6 (1990), pp. 246–256.

[Ald18] Shaban Aldabbus. “Project-based learning: Implementation & challenges”. In: *International Journal of Education, Learning and Development* 6.3 (2018), pp. 71–79.

[Ale06] Vincent Aleven, Bruce M McLaren, and Kenneth R Koedinger. “Toward computer-based tutoring of help-seeking skills”. In: *Help seeking in academic settings: Goals, groups, and contexts* (2006), pp. 259–296.

[Ale16] Vincent Aleven et al. “Help helps, but only so much: Research on help seeking with intelligent tutoring systems”. In: *International Journal of Artificial Intelligence in Education* 26.1 (2016), pp. 205–223.

[Ama19] 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)*. IEEE. 2019, pp. 1–8.

[Atk03] 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.” In: *Journal of educational psychology* 95.4 (2003), p. 774.

[Bai20] Gina R Bai, Joshua Kayani, and Kathryn T Stolee. “How Graduate Computing Students Search When Using an Unfamiliar Programming Language”. In: *International Conference on Program Comprehension (ICPC)*. 2020.

[Bar12] Moshe Barak. “From “doing” to “doing with learning”: Reflection on an effort to promote self-regulated learning in technological projects in high school”. In: *European Journal of Engineering Education* 37.1 (2012), pp. 105–116.

[Bec06] Laura Beckwith et al. “Tinkering and gender in end-user programmers’ debugging”. In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 2006, pp. 231–240.

[Ber16] Ilias Bergström and Alan F Blackwell. “The practices of programming”. In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2016, pp. 190–198.

[Bla02] 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*. 2002, pp. 2–10.

[Bla13] Jonathan Black et al. “Making computing interesting to school students: teachers’ perspectives”. In: *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 2013, pp. 255–260.

[Blu91] Phyllis C Blumenfeld et al. “Motivating project-based learning: Sustaining the doing, supporting the learning”. In: *Educational psychologist* 26.3-4 (1991), pp. 369–398.

[Bra09] Joel Brandt et al. “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 1589–1598.

[Bra10] Joel Brandt et al. “Example-centric programming: integrating web search into the development environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 513–522.

[Bra12] Virginia Braun and Victoria Clarke. “Thematic analysis.” In: (2012).

[Bro17] Brian Broll et al. “A visual programming environment for learning distributed programming”. In: *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 2017, pp. 81–86.

[Bro83] Ruven Brooks. “Towards a theory of the comprehension of computer programs”. In: *International journal of man-machine studies* 18.6 (1983), pp. 543–554.

[Bru01] Peter Brusilovsky. “WebEx: Learning from Examples in a Programming Course.” In: *WebNet*. Vol. 1. 2001, pp. 124–129.

[Bur13] Brian Burg et al. “Interactive record/replay for web application debugging”. In: *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 2013, pp. 473–484.

[But98] Ruth Butler. “Determinants of help seeking: Relations between perceived reasons for classroom help-avoidance and help-seeking behaviors in an experimental context.” In: *Journal of Educational Psychology* 90.4 (1998), p. 630.

[Cat18] Veronica Cateté, Nicholas Lytle, and Tiffany Barnes. “Creation and validation of low-stakes rubrics for k-12 computer science”. In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 2018, pp. 63–68.

[Che14] Erin Cherry and Celine Latulipe. “Quantifying the creativity support of digital tools through the creativity support index”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 21.4 (2014), pp. 1–25.

[Chi14] Michelene TH Chi and Ruth Wylie. “The ICAP framework: Linking cognitive engagement to active learning outcomes”. In: *Educational psychologist* 49.4 (2014), pp. 219–243.

[Cla11] Ruth C Clark, Frank Nguyen, and John Sweller. *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. John Wiley & Sons, 2011.

[Cla16] Ruth C Clark and Richard E Mayer. *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. John Wiley & Sons, 2016.

[Coe17] Jarno Coenen, Sebastian Gross, and Niels Pinkwart. “Comparison of Feedback Strategies for Supporting Programming Learning in Integrated Development Environments (IDEs)”. In: *International Conference on Computer Science, Applied Mathematics and Applications*. Springer. 2017, pp. 72–83.

[Col88] Allan Collins, John Seely Brown, and Susan E Newman. “Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics”. In: *Thinking: The Journal of Philosophy for Children* 8.1 (1988), pp. 2–10.

[Coo00] Stephen Cooper, Wanda Dann, and Randy Pausch. “Alice: a 3-D tool for introductory programming concepts”. In: *Journal of Computing Sciences in Colleges*. Vol. 15. 5. Consortium for Computing Sciences in Colleges. 2000, pp. 107–116.

[Cri06] Jennifer K Crissman. *The design and utilization of effective worked examples: A meta-analysis*. The University of Nebraska-Lincoln, 2006.

[Cro55] Lee J Cronbach and Paul E Meehl. “Construct validity in psychological tests.” In: *Psychological bulletin* 52.4 (1955), p. 281.

[Cue05] Peggy Cuevas et al. “Improving science inquiry with elementary students of diverse backgrounds”. In: *Journal of Research in Science Teaching: the Official Journal of the National Association for Research in Science Teaching* 42.3 (2005), pp. 337–357.

[Das16] Sayamindu Dasgupta et al. “Remixing as a pathway to computational thinking”. In: *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. 2016, pp. 1438–1449.

[Del12] Nicola Dell et al. “" Yours is better!" participant response bias in HCI”. In: *Proceedings of the sigchi conference on human factors in computing systems*. 2012, pp. 1321–1330.

[Don19] Yihuan Dong et al. “Defining tinkering behavior in open-ended block-based programming assignments”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 1204–1210.

[Dor13] Brian Dorn, Adam Stankiewicz, and Chris Roggi. “Lost while searching: Difficulties in information seeking among end-user programmers”. In: *Proceedings of the American Society for Information Science and Technology* 50.1 (2013), pp. 1–10.

[Fie14] Deborah A Fields, Michael Giang, and Yasmin Kafai. “Programming in the wild: trends in youth computational participation in the online scratch community”. In: *Proceedings of the 9th workshop in primary and secondary computing education*. 2014, pp. 2–11.

[Fra20] Diana Franklin et al. “An Analysis of Use-Modify-Create Pedagogical Approach’s Success in Balancing Structure and Student Agency”. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, pp. 14–24.

[Fra21] Gordon Fraser et al. “Litterbox: A linter for scratch programs”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE. 2021, pp. 183–188.

[Gao20] Gao Gao et al. “Exploring Programmers’ API Learning Processes: Collecting Web Resources as External Memory”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2020, Dunedin, New Zealand, August 10-14, 2020*. Ed. by Michael Homer et al. IEEE, 2020, pp. 1–10. DOI: 10.1109/

VL/HCC50065.2020.9127274. URL: https://doi.org/10.1109/VL/HCC50065.2020.9127274.

[Gar15] Dan Garcia, Brian Harvey, and Tiffany Barnes. "The beauty and joy of computing". In: *ACM Inroads* 6.4 (2015), pp. 71–79.

[Gen03] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. "Learning and transfer: A general role for analogical encoding." In: *Journal of Educational Psychology* 95.2 (2003), p. 393.

[Ger04] Peter Gerjets, Katharina Scheiter, and Richard Catrambone. "Designing instructional examples to reduce intrinsic cognitive load: Molar versus modular presentation of solution procedures". In: *Instructional Science* 32.1-2 (2004), pp. 33–58.

[Gic83] Mary L Gick and Keith J Holyoak. "Schema induction and analogical transfer". In: *Cognitive psychology* 15.1 (1983), pp. 1–38.

[Gol20] Paul Goldenberg et al. "Design Principles behind Beauty and Joy of Computing". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 2020, pp. 220–226.

[Gon22] David Gonzalez-Maldonado et al. "Investigating the Use of Planning Sheets in Young Learners' Open-Ended Scratch Projects". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 2022, pp. 247–263.

[Gre18] Jeffrey A Greene et al. "Capturing and modeling self-regulated learning using think-aloud protocols." In: *Handbook of Self-Regulation of Learning and Performance* (2018), pp. 323–337. DOI: 10.4324/9781315697048-21.

[Gro10a] Paul Gross and Caitlin Kelleher. "Non-programmers identifying functionality in unfamiliar code: strategies and barriers". In: *Journal of Visual Languages & Computing* 21.5 (2010), pp. 263–276.

[Gro10b] Paul A Gross et al. "A code reuse interface for non-programmer middle school students". In: *Proceedings of the 15th international conference on Intelligent user interfaces*. 2010, pp. 219–228.

[Gro14] Sebastian Gross et al. "Example-based feedback provision using structured solution spaces". In: *International Journal of Learning Technology 10* 9.3 (2014), pp. 248–280.

[Gro15] Sebastian Gross and Niels Pinkwart. “Towards an integrative learning environment for java programming”. In: *2015 IEEE 15th International Conference on Advanced Learning Technologies*. IEEE. 2015, pp. 24–28.

[Gro18] 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. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 999–1004. ISBN: 9781450351034. DOI: 10.1145/3159450.3159522. URL: https://doi.org/10.1145/3159450.3159522.

[Gui04] Marilys Guillemin and Lynn Gillam. “Ethics, reflexivity, and “ethically important moments” in research”. In: *Qualitative inquiry* 10.2 (2004), pp. 261–280.

[Guo13] Philip J Guo. “Online python tutor: embeddable web-based program visualization for cs education”. In: *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, pp. 579–584.

[Guz03] Mark Guzdial and Elliot Soloway. “Computer science is more important than calculus: The challenge of living up to our potential”. In: *ACM SIGCSE Bulletin* 35.2 (2003), pp. 5–8.

[Guz05] Mark Guzdial and Andrea Forte. “Design process for a non-majors computing course”. In: *ACM SIGCSE Bulletin* 37.1 (2005), pp. 361–365.

[Guz15] Mark Guzdial. “Learner-centered design of computing education: Research on computing for everyone”. In: *Synthesis Lectures on Human-Centered Informatics* 8.6 (2015), pp. 1–165.

[Hai18] Yousef Haik, Sangarappillai Sivaloganathan, and Tamer Shahin. *Engineering design process*. Nelson Education, 2018.

[Ham12] Lorna Hamilton and Connie Corbett-Whittier. *Using case study in education research*. Sage, 2012.

[Har08] Björn Hartmann et al. “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*. 2008, pp. 91–100.

[Har13] Brian Harvey et al. “Snap!(build your own blocks)”. In: *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, pp. 759–759.

[Har88] Sandra G Hart and Lowell E Staveland. “Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research”. In: *Advances in psychology*. Vol. 52. Elsevier, 1988, pp. 139–183.

[Hil13] Benjamin Mako Hill and Andrés Monroy-Hernández. “The cost of collaboration for code and art: Evidence from a remixing community”. In: *Proceedings of the 2013 conference on Computer supported cooperative work*. 2013, pp. 1035–1046.

[Hol09] Reid Holmes et al. “The end-to-end use of source code examples: An exploratory study”. In: *2009 IEEE International Conference on Software Maintenance*. IEEE. 2009, pp. 555–558.

[Hul15] Carol Hulls et al. “The use of an open-ended project to improve the student experience in first year programming”. In: *Proceedings of the Canadian Engineering Education Association (CEEA)* (2015).

[Hun04] 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*. Vol. 4. 1. 2004, p. 1722.

[Ich15] Michelle Ichinco and Caitlin Kelleher. “Exploring novice programmer example use”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2015, pp. 63–71.

[Ich17] 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*. 2017, pp. 1105–1117.

[Ich19] Michelle Ichinco and Caitlin Kelleher. “Open-Ended Novice Programming Behaviors and their Implications for Supporting Learning”. In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2019, pp. 45–53.

[Jen21] Jay Jennings and Kasia Muldner. “When does scaffolding provide too much assistance? a code-tracing tutor investigation”. In: *International Journal of Artificial Intelligence in Education* 31.4 (2021), pp. 784–819.

[Kar13] Stuart A Karabenick and Richard S Newman. *Help seeking in academic settings: Goals, groups, and contexts*. Routledge, 2013.

[Ker17a] Mary Beth Kery, Amber Horvath, and Brad A Myers. “Variolite: Supporting Exploratory Programming by Data Scientists.” In: *CHI*. Vol. 10. 2017, pp. 3025453–3025626.

[Ker17b] Mary Beth Kery and Brad A Myers. “Exploring exploratory programming”. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2017, pp. 25–29.

[Kes21] Max Kesselbacher and Andreas Bollin. “Towards the Use of Slice-based Cohesion Metrics with Learning Analytics to Assess Programming Skills”. In: *2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG)*. IEEE. 2021, pp. 6–10.

[Kha19] 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)*. IEEE. 2019, pp. 169–173.

[Kir06] Paul A Kirschner, John Sweller, and Richard E Clark. “Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching”. In: *Educational psychologist* 41.2 (2006), pp. 75–86.

[Ko04a] Amy J Ko and Brad A Myers. “Designing the whyline: a debugging interface for asking questions about program behavior”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2004, pp. 151–158.

[Ko04b] 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*. IEEE. 2004, pp. 199–206.

[Ko06] Andrew J Ko et al. “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks”. In: *IEEE Transactions on software engineering* 32.12 (2006), pp. 971–987.

[Ko11] Amy J. Ko et al. “The state of the art in end-user software engineering”. In: *ACM Computing Surveys (CSUR)* 43.3 (2011), pp. 1–44.

[Koe12] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. “The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning”. In: *Cognitive science* 36.5 (2012), pp. 757–798.

[Kok16] Dimitra Kokotsaki, Victoria Menzies, and Andy Wiggins. “Project-based learning: A review of the literature”. In: *Improving schools* 19.3 (2016), pp. 267–277.

[Lah05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A study of the difficulties of novice programmers”. In: *Acm sigcse bulletin* 37.3 (2005), pp. 14–18.

[Lan89] B. M. Lange and T. G. Moher. “Some Strategies of Reuse in an Object-Oriented Programming Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 69–73. ISBN: 0897913019. DOI: 10.1145/67449.67465. URL: https://doi.org/10.1145/67449.67465.

[Lee11] Irene Lee et al. “Computational thinking for youth in practice”. In: *Acm Inroads* 2.1 (2011), pp. 32–37.

[Let87] Stanley Letovsky. “Cognitive processes in program comprehension”. In: *Journal of Systems and software* 7.4 (1987), pp. 325–339.

[Lew11] Colleen M. Lewis. “Is pair programming more effective than other forms of collaboration for young students?” In: *Computer Science Education* 21 (2011), pp. 105–134.

[Lew15] 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*. 2015, pp. 41–50.

[Lim22] Ally Limke et al. “Case Studies on the use of Storyboarding by Novice Programmers”. In: *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. 2022, pp. 318–324.

[Lis04] Raymond Lister et al. “A multi-national study of reading and tracing skills in novice programmers”. In: *ACM SIGCSE Bulletin* 36.4 (2004), pp. 119–150.

[Lyt19] Nicholas Lytle et al. “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*. ACM. 2019, pp. 395–401.

[Lyt20] Nicholas Lytle et al. “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. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 832–838. ISBN: 9781450367936. DOI: 10.1145/3328778.3366943. URL: https://doi-org.prox.lib.ncsu.edu/10.1145/3328778.3366943.

[Mal10] John Maloney et al. “The scratch programming language and environment”. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010), pp. 1–15.

[Mar02] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: Women in computing*. MIT press, 2002.

[Mar07] Jeffrey D Marx and Karen Cummings. “Normalized change”. In: *American Journal of Physics* 75.1 (2007), pp. 87–91.

[Mar20] Samiha Marwan, Anay Dombe, and Thomas W Price. “Unproductive Help-seeking in Programming: What it is and How to Address it”. In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 2020, pp. 54–60.

[Mar94] Ronald W Marx et al. “Enacting project-based science: Experiences of four middle grade teachers”. In: *The Elementary School Journal* 94.5 (1994), pp. 517–538.

[McG18] Steven McGee et al. “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. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 50–55. ISBN: 9781450351034. DOI: 10.1145/3159450.3159529. URL: https://doi.org/10.1145/3159450.3159529.

[Mee11] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. “Habits of programming in scratch”. In: *ITiCSE’11 - Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science* (2011), pp. 168–172. DOI: 10.1145/1999747.1999796.

[Mil21] Alexandra Milliken et al. “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. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 232–238. ISBN: 9781450380621. DOI:

10.1145/3408877.3432552. URL: https://doi.org/10.1145/3408877.3432552.

[Moe12] J Moenig and B Harvey. "BYOB Build your own blocks (a/k/a SNAP!)" In: *URL: http://byob. berkeley. edu/, accessed Aug* (2012).

[Mon07] Andrés Monroy-Hernández. "ScratchR: sharing user-generated programmable media". In: *Proceedings of the 6th international conference on Interaction design and children*. 2007, pp. 167–168.

[Mon12] Andrés Monroy-Hernández. "Designing for remixing: Supporting an online community of amateur creators". PhD thesis. Massachusetts Institute of Technology, 2012.

[Mor11] Ralph Morelli et al. "Can android app inventor bring computational thinking to k-12". In: *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)*. 2011, pp. 1–6.

[Mor13] Teresa M Morales, EunJin Bang, and Thomas Andre. "A one-year case study: Understanding the rich potential of project-based learning in a virtual reality class for high school students". In: *Journal of Science Education and Technology* 22.5 (2013), pp. 791–806.

[Mor15] 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*. ACM. 2015, pp. 21–29.

[Nos96] Richard Noss and Celia Hoyles. *Windows on mathematical meanings: Learning cultures and computers*. Vol. 17. Springer Science & Business Media, 1996.

[Ott93] Linda M Ott and Jeffrey J Thuss. "Slice based metrics for estimating cohesion". In: *[1993] Proceedings First International Software Metrics Symposium*. IEEE. 1993, pp. 71–81.

[Pap80] Seymour Papert. "Mindstorms: Computers, children, and powerful ideas". In: *NY: Basic Books* (1980), p. 255.

[Par06] 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*. Australian Computer Society, Inc. 2006, pp. 157–163.

[Par11] 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*. 2011, pp. 25–30.

[Pat13] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. “Comparing and contrasting different algorithms leads to increased student learning”. In: *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM. 2013, pp. 145–152.

[Pau91] Delroy L Paulhus. “Measurement and control of response bias.” In: (1991).

[Pep07] Kylie A Peppler and Yasmin B Kafai. “From SuperGoo to Scratch: Exploring creative digital media production in informal learning”. In: *Learning, media and technology* 32.2 (2007), pp. 149–166.

[Per98] Thomas V Perneger. “What’s wrong with Bonferroni adjustments”. In: *Bmj* 316.7139 (1998), pp. 1236–1238.

[Pir94] Peter Pirolli and Mimi Recker. “Learning Strategies and Transfer in the Domain of Programming”. In: *ITLS Faculty Publications* 12 (Sept. 1994). DOI: 10.1207/s1532690xci1203_2.

[Pow07] Kris Powers, Stacey Ecott, and Leanne M Hirshfield. “Through the looking glass: teaching CS0 with Alice”. In: *Proceedings of the 38th SIGCSE technical symposium on Computer science education*. 2007, pp. 213–217.

[Pri17] Thomas W Price et al. “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*. ACM. 2017, pp. 127–135.

[Rac20] Arif Rachmatullah et al. “Development and validation of the middle grades computer science concept inventory (MG-CSCI) assessment”. In: *EURASIA Journal of Mathematics, Science and Technology Education* 16.5 (2020), em1841.

[Res09] Mitchel Resnick et al. “Scratch: programming for all”. In: *Communications of the ACM* 52.11 (2009), pp. 60–67.

[Rit07] Bethany Rittle-Johnson and Jon R Star. “Does comparing solution methods facilitate conceptual and procedural knowledge? An experimental study on learning to solve equations.” In: *Journal of Educational Psychology* 99.3 (2007), p. 561.

[Rob09] Martin P Robillard. “What makes APIs hard to learn? Answers from developers”. In: *IEEE software* 26.6 (2009), pp. 27–34.

[Rob12] Martin P Robillard et al. “Automated API property inference techniques”. In: *IEEE Transactions on Software Engineering* 39.5 (2012), pp. 613–637.

[Rob17] Gregorio Robles et al. “Software clones in scratch projects: On the presence of copy-and-paste in computational thinking learning”. In: *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE. 2017, pp. 1–7.

[Roq16] Ricarose Roque, Natalie Rusk, and Mitchel Resnick. “Supporting diverse and creative collaboration in the Scratch online community”. In: *Mass collaboration and education*. Springer, 2016, pp. 241–256.

[Ros93] Mary Beth Rosson and John M Carroll. “Active programming strategies in reuse”. In: *European Conference on Object-Oriented Programming*. Springer. 1993, pp. 4–20.

[Ros96] Mary Beth Rosson and John M Carroll. “The reuse of uses in Smalltalk programming”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 3.3 (1996), pp. 219–253.

[Sch71] Wilbur Schramm. “Notes on Case Studies of Instructional Media Projects.” In: (1971).

[Sha13] Amir Shareghi Najar and Antonija Mitrovic. “Examples and tutored problems: How can self-explanation make a difference to learning?” In: *International Conference on Artificial Intelligence in Education*. Springer. 2013, pp. 339–348.

[Shi08] Benjamin Shih, Kenneth Koedinger, and Richard Scheines. “A Response-Time Model for Bottom-Out Hints as Worked Examples”. In: Jan. 2008, pp. 117–126. DOI: 10.1201/b10274-17.

[Sor07] Juha Sorva. “Notional machines and introductory programming education”. In: *Trans. Comput. Educ* 13.2 (2007), pp. 1–31.

[Swe06] John Sweller. “The worked example effect and human cognition.” In: *Learning and instruction* 16.2 (2006), pp. 165–169.

[Swe88] John Sweller. “Cognitive load during problem solving: Effects on learning”. In: *Cognitive science* 12.2 (1988), pp. 257–285.

[Tha20] Kyle Matthew Thayer. “Practical Knowledge Barriers in Professional Programming”. PhD thesis. 2020.

[Tha21] Kyle Thayer, Sarah E Chasins, and Amy J Ko. “A theory of robust API knowledge”. In: *ACM Transactions on Computing Education (TOCE)* 21.1 (2021), pp. 1–32.

[Tho20] W. Price Thomas et al. “Engaging Students with Instructor Solutions in Online Programming Homework”. In: *To be published in the 2020 Association for Computing Machinery's Special Interest Group on Computer Human Interaction (ACM SIGCHI '20)*. 2020.

[Tra94] John Gregory Trafton and Brian J Reiser. “The contributions of studying examples and solving problems to skill acquisition”. PhD thesis. Citeseer, 1994.

[Tsa21] Jennifer Tsan et al. “Collaborative dialogue and types of conflict: An analysis of pair programming interactions between upper elementary students”. In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 2021, pp. 1184–1190.

[Urb00] Mark Urban-Lurain and Donald J Weinshank. “Is there a role for programming in non-major computer science courses?” In: *30th Annual Frontiers in Education Conference. Building on A Century of Progress in Engineering Education. Conference Proceedings (IEEE Cat. No. 00CH37135)*. Vol. 1. IEEE. 2000, T2B–7.

[Von95] Anneliese Von Mayrhauser and A Marie Vans. “Program comprehension during software maintenance and evolution”. In: *Computer* 28.8 (1995), pp. 44–55.

[Wan20a] Wengran Wang et al. “Comparing Feature Engineering Approaches to Predict Complex Programming Behaviors”. In: *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20* (2020).

[Wan20b] Wengran Wang et al. “Crescendo: Engaging Students to Self-Paced Programming Practices”. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 859–865. ISBN: 9781450367936. DOI: 10.1145/3328778.3366919. URL: https://doi.org/10.1145/3328778.3366919.

[Wan20c] Wengran Wang et al. “The Step Tutor: Supporting Students through Step-by-Step Example-Based Feedback”. In: *ITiCSE'20 - Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, To be published* (2020), pp. 391–397.

[Wan21] Wengran Wang et al. “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. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 394–400. ISBN: 9781450382144. DOI: 10.1145/3430665.3456370. URL: https://doi.org/10.1145/3430665.3456370.

[Wan22] Wengran Wang et al. “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*. 2022, pp. 619–625.

[Win11] Philip H Winne. “A cognitive and metacognitive analysis of self-regulated learning”. In: *Handbook of self-regulation of learning and performance* (2011), pp. 15–32.

[Wri07] Terry Wrigley. “Projects, stories and challenges: more open architectures for school learning”. In: *Storyline past, present and future* (2007), pp. 166–181.

[Xie18] Benjamin Xie, Greg L Nelson, and Amy J Ko. “An explicit strategy to scaffold novice program tracing”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM. 2018, pp. 344–349.

[Xu05] Baowen Xu et al. “A brief survey of program slicing”. In: *ACM SIGSOFT Software Engineering Notes* 30.2 (2005), pp. 1–36.

[Yin17] K Yin Robert. *Case study research and applications: design and methods*. 2017.

[Zhi18] Rui Zhi, Nicholas Lytle, and Thomas W Price. “Exploring Instructional Support Design in an Educational Game for K-12 Computing Education”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM. 2018, pp. 747–752.

[Zhi19] Rui Zhi et al. “Exploring the Impact of Worked Examples in a Novice Programming Environment”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM. 2019, pp. 98–104.