

Washington University in St. Louis
Washington University Open Scholarship

Engineering and Applied Science Theses &
Dissertations

McKelvey School of Engineering

Summer 8-15-2018

The Example Guru: Suggesting Examples to Novice Programmers in an Artifact-Based Context

Michelle Ichinco

Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds



Part of the [Computer Sciences Commons](#)

Recommended Citation

Ichinco, Michelle, "The Example Guru: Suggesting Examples to Novice Programmers in an Artifact-Based Context" (2018).
Engineering and Applied Science Theses & Dissertations. 366.
https://openscholarship.wustl.edu/eng_etds/366

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering & Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:

Caitlin Kelleher, Chair

Roger Chamberlain

Sanmay Das

Bjoern Hartmann

Alvitta Ottley

The Example Guru: Suggesting Examples to Novice Programmers in an Artifact-Based
Context

by

Michelle Ichinco

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2018
St. Louis, Missouri

© 2018, Michelle Ichinco

Table of Contents

List of Figures	x
List of Tables	xiii
Acknowledgments	xv
Abstract	xvii
Chapter 1: Introduction	1
1.1 Resources for novice programmers	2
1.2 Examples.....	3
1.3 Approach.....	4
1.3.1 The Example Guru system overview	6
1.3.2 Looking Glass	8
1.4 Hypotheses	10
1.4.1 Hypothesis 1	10
1.4.2 Hypothesis 2.....	11
1.4.3 Hypothesis 3.....	11
1.5 Contributions	11
1.6 Summary.....	12
1.6.1 Related work	12
1.6.2 Hypothesis 1: Studies of novices using examples.....	12
1.6.3 Hypothesis 2: Suggesting example code to novices with the Example Guru.....	13
1.6.4 Hypothesis 3: Generating large-scale suggestion systems	13
1.6.5 Summary and future work	13
Chapter 2: Related Work	14
2.1 Examples and learning from a cognitive perspective	15

2.1.1	Analogical problem solving	15
2.1.2	Worked examples for learning.....	17
2.1.3	Takeaways.....	20
2.2	Comprehending and presenting example code in programming and software engineering	20
2.2.1	Code comprehension.....	21
2.2.2	Code example use	21
2.2.3	Quality of examples	22
2.2.4	Code summarization.....	23
2.2.5	Code visualization	23
2.2.6	Takeaways.....	24
2.3	Supporting access of relevant examples	24
2.3.1	Recommendations for API or command usage	24
2.3.2	Recommendations to improve code quality.....	26
2.3.3	Recommendations to fix errors	28
2.3.4	Providing examples as a way of recommending examples.....	30
2.3.5	Takeaways.....	32
Chapter 3: Understanding Novice Example Use		33
3.1	Introduction.....	33
3.2	Exploratory study	35
3.2.1	Materials	35
3.2.2	Study design.....	37
3.2.3	Participants.....	39
3.2.4	Analysis and results	40
3.3	Hurdles and strategies	45
3.3.1	Content distraction hurdle.....	45
3.3.2	Example comprehension hurdle.....	45
3.3.3	Programming environment hurdle	46
3.3.4	Code misconception hurdle	47
3.3.5	Code comprehension hurdle	47

3.3.6	Idea generation strategy	48
3.3.7	Code-example comparison strategy	48
3.3.8	Example emphasis strategy	49
3.4	Task behavior groups	49
3.4.1	Long conclusion group	50
3.4.2	Slow start group	51
3.4.3	No realization group	52
3.4.4	Quick group	53
3.5	Threats to validity	56
3.6	Discussion	56
3.6.1	Implications of slow start behavior	57
3.6.2	Implications of long conclusion behavior	57
3.6.3	Implications for the design of the Example Guru	58
3.7	Conclusion	61
Chapter 4: Towards Better Code Snippets: Exploring How Code Snippet Recall Differs with Programming Experience		62
4.1	Related work: novice and expert chunking in recall	64
4.2	Methods	65
4.2.1	Participants	65
4.2.2	Materials	66
4.2.3	Study setup	68
4.3	Analysis	70
4.3.1	Metrics	70
4.3.2	Comparing responses to correct code snippets	73
4.4	Results	73
4.4.1	Overall	74
4.4.2	Which elements do programmers initially recall?	74
4.4.3	What did programmers fill in after the first attempt?	79
4.4.4	Errors	81
4.5	Threats to validity	83

4.6	Lessons learned: recommendations for improving code examples	83
4.6.1	Selecting or creating effective code for examples	84
4.6.2	Purposely position elements within example	84
4.6.3	General emphasis and deemphasis	85
4.6.4	Example-specific emphasis and deemphasis	86
4.6.5	Emphasize important arguments	86
4.6.6	Emphasize tokens that stray from the norm	87
4.6.7	Deemphasize unimportant early code elements.....	87
4.7	Implications for the Design of the Example Guru.....	87
4.7.1	Selecting effective examples.....	88
4.7.2	Emphasizing elements.....	88
4.8	Conclusion	89
Chapter 5: Exploring Suggestion and Rule Design through Expert Content Creation		90
5.1	Methods.....	91
5.1.1	Materials	92
5.1.2	Study procedures	94
5.1.3	Participants.....	96
5.2	Analysis	96
5.2.1	Suggestions	97
5.2.2	Rule pseudocode.....	97
5.2.3	Rule implementation	98
5.3	Results.....	98
5.3.1	Suggestions	99
5.3.2	Rule pseudocode.....	102
5.3.3	Rule implementation	104
5.4	Threats to validity.....	108
5.5	Discussion.....	108
5.5.1	Quality in expert created content	108
5.5.2	Relevance of content for novices.....	109

5.6	Implications for the design of the Example Guru	110
5.6.1	Suggestions	111
5.6.2	Rules	111
5.7	Conclusion	111
Chapter 6: Designing and Evaluating the Example Guru for Suggesting API Methods.....		113
6.1	Introduction.....	113
6.2	The Example Guru design.....	115
6.2.1	System design methods	115
6.3	Evaluation	121
6.3.1	Documentation condition	121
6.3.2	Participants.....	122
6.3.3	Methods	123
6.3.4	Study procedures	125
6.3.5	Data collection and analysis.....	127
6.4	Results.....	129
6.4.1	Access and use of suggestions and documentation.....	130
6.4.2	Do participants' demographics affect how they used suggestions and documentation?	136
6.4.3	Do participants take advantage of API information features?	139
6.4.4	Threats to validity	140
6.5	Discussion.....	141
6.5.1	Learning APIs.....	142
6.5.2	Gender and the Example Guru	143
6.6	Conclusion	144
Chapter 7: Large-Scale Suggestions: Semi-Automatic Generation		145
7.1	Related work	147
7.2	Programming environment & suggestion system	148
7.2.1	Looking Glass programming environment	149
7.2.2	The Example Guru (final version)	149
7.3	Suggestion generation approach.....	151

7.3.1	Input repository	151
7.3.2	Initial setup.....	152
7.3.3	Example extraction	153
7.3.4	Example grouping.....	153
7.3.5	Human moderation	155
7.3.6	Generate rules.....	156
7.4	Comparison of semi-automatically generated to hand-authored suggestions ...	157
7.4.1	Comparison methods	158
7.4.2	Comparison results	158
7.5	User study: novices' interaction with semi- automatically generated suggestions vs. tutorials	161
7.5.1	Tutorial control condition	161
7.5.2	Study protocol	162
7.5.3	Participants.....	164
7.5.4	Data and analysis	165
7.5.5	Study results	167
7.6	Threats to validity.....	173
7.7	Discussion.....	173
7.7.1	How our approach generalizes.....	174
7.7.2	Potential of suggestions to increase use and learning	174
7.7.3	Effect of the code repository on suggestion generation	175
7.7.4	Personalization.....	176
7.8	Conclusion	176
Chapter 8: Summary and Future Work		178
8.1	Summary.....	178
8.2	Future work	180
8.2.1	How can we apply large-scale in-context suggested content to help people learn other topics?	181
8.2.2	How can we automatically generate context-relevant support at a large scale?	183

8.2.3	What types of user interaction can help users learn and improve suggestion relevance?	184
References		186
Appendix A: Understanding How Novices Use Examples Study Materials. [208]		
A.1	Computing history survey	[208]
A.2	Intro instructions	[209]
A.3	Task programs and solutions.....	[211]
A.4	Interview questions	[217]
Appendix B: Comparing Novices and Experts Study Materials		[218]
B.1	Instructions.....	[218]
B.2	Demographic survey.....	[219]
B.3	Tasks	[220]
B.4	Post-task survey.....	[225]
Appendix C: Exploring Types of Suggestions Study Materials		[226]
C.1	Skill trees	[226]
C.2	Programs and pre-made suggestions	[231]
C.3	Template	[236]
Appendix D: Example Guru for API Methods Study Materials.....		[237]
D.1	Instruction sheets	[237]
D.2	Templates.....	[239]
D.3	Suggestions, rules, and examples	[243]
D.4	Surveys	[253]
D.4.1	Demographic history Survey	[253]
D.4.2	Learning style survey.....	[253]
D.4.3	In-task survey questions	[254]
Appendix E: Semi-automatic Suggestion Generation Study Materials		[255]
E.1	Surveys	[255]
E.1.1	Demographic and computing history survey	[255]
E.1.2	Post-study survey	[255]
E.2	Training tasks.....	[256]

E.3	Templates.....	[257]
E.4	Tutorials	[259]
E.4.1	Turn the character's head.	[259]
E.4.2	Increase the walk pace of a character	[259]
E.4.3	Make the flash happen multiple times.	[259]
E.4.4	Make actions happen together	[259]
E.4.5	Make Alice wave three times	[260]
E.4.6	Make a character jump multiple times.....	[260]
E.4.7	Make the two objects move together	[260]
E.4.8	Make the set of actions happen multiple times	[260]
E.4.9	Make a character talk and walk at the same time.	[261]
E.4.10	Make an object change size and color at the same time.....	[261]
E.4.11	Make a character turn and turn back multiple times	[261]
E.4.12	Make the dolphins flip at the same time	[262]
E.4.13	Make the jump more realistic	[262]
E.5	Suggestions and examples	[262]
E.6	Transfer tasks.....	[267]

List of Figures

Figure 1.1:	(A) Looking Glass programming environment. (B) List of suggestions. (C) An opened/accessed suggestion. (D) The two code examples with the primary one selected. (E) The code for this suggestion, with the <i>do together parallel execution</i> block emphasized. (F) Preview execution of the code.	6
Figure 1.2:	Looking Glass	9
Figure 3.1:	The augmented Looking Glass with a task and an example. A) Looking Glass programming environment. B) Task code designed by the researcher. C) Example code and task instructions dialog box.....	36
Figure 3.2:	Study protocol	38
Figure 3.3:	Time before realization point vs. time after realization point, with correctness and behavior group annotated with color and shape.....	54
Figure 3.4:	Important labels and the average count for each of the behavior groups. The largest value is shown for each label.....	55
Figure 4.1:	An example of block and text versions for the same code snippet.....	67
Figure 5.1:	Skill group diagram.....	93
Figure 5.2:	Suggestion type category hierarchy.	100
Figure 5.3:	Suggestion novelty category hierarchy.....	101
Figure 5.4:	Example of a rule	102
Figure 5.5:	Iteration style category hierarchy.....	103
Figure 5.6:	Comparison style category hierarchy.	104

Figure 6.1:	The Example Guru implemented within Looking Glass. (A) List of all suggestions. (B) Code annotation button to open the most recently added suggestion. (C) Contrasting examples such as ‘walk fast’ and ‘walk slow’. (D) ‘Show me’ button that users can click to see the location of the suggested block.	117
Figure 6.2:	In-application API Documentation condition. (A) Users can access documentation using the ‘?’ button available beside APIs. (B) Examples with different values and the description. (C) The play button can be used to execute the code. (D) Button to expand or collapse the parameters information. (E) Users can navigate to other doc using these buttons.	123
Figure 6.3:	API information accessed and used grouped by frequency of API use by novice programmers.	131
Figure 7.1:	(A) Looking Glass programming environment. (B) List of suggestions. (C) An opened/accessed suggestion. (D) The two code examples with the primary one selected. (E) The code for this suggestion, with the <i>do together parallel execution</i> block emphasized. (F) Preview execution of the code.	148
Figure 7.2:	(A) An accessed tutorial. (B) List of tutorials, which always has the same set of 13 tutorials for everyone in the condition. (C) A short video that shows how to complete the step. (D) Written instructions. (E) Next button to go to the next step of the tutorial.	162
Figure 7.3:	The number of suggestions and tutorials participants accessed.	169
Figure A.1:	The intro task sheet.	[210]
Figure A.2:	Do together task.	[211]
Figure A.3:	For each loop task.	[212]
Figure A.4:	Function task.	[213]
Figure A.5:	API method task.	[214]
Figure A.6:	While loop task.	[215]
Figure A.7:	Repeat loop task.	[216]
Figure B.1:	For each loop block code snippet.	[221]
Figure B.2:	For each loop text code snippet.	[221]
Figure B.3:	Repeat While loop block code snippet.	[222]

Figure B.4: Repeat While loop text code snippet	[222]
Figure B.5: Simple Repeat loop block code snippet	[223]
Figure B.6: Simple Repeat loop text code snippet	[223]
Figure B.7: Conditional block code snippet	[224]
Figure B.8: Conditional text code snippet	[224]
Figure B.9: Difficulty Scale	[225]
Figure B.10: Mental Effort Scale	[225]
Figure C.1: Skill tree for ‘Jane’	[227]
Figure C.2: Skill tree for ‘Mike’	[228]
Figure C.3: Skill tree for ‘Molly’	[229]
Figure C.4: Skill tree for ‘Pete’	[230]
Figure C.5: One of the pre-made suggestions for ‘Jane’	[231]
Figure C.6: One of the pre-made suggestions for ‘Pete’	[232]
Figure C.7: One of the pre-made suggestions for ‘Mike’	[233]
Figure C.8: One of the pre-made suggestions for ‘Molly’, part 1	[234]
Figure C.9: One of the pre-made suggestions for ‘Molly’, part 2	[235]
Figure E.1: First Training Task	[256]
Figure E.2: Second Training Task	[257]
Figure E.3: Seaworld template	[257]
Figure E.4: Templates for open-ended programming	[258]
Figure E.5: Transfer task 1	[268]
Figure E.6: Transfer task 2	[269]
Figure E.7: Transfer task 3	[270]
Figure E.8: Transfer task 4	[271]

List of Tables

Table 3.1:	Labels	44
Table 4.1:	Participants' programming experience	66
Table 4.2:	Token Types	71
Table 4.3:	Average % of tokens recalled by everyday programmers and differences between groups for 1st and 3rd attempts.	72
Table 4.4:	Comparing blocks and text	76
Table 4.5:	% of errors for each token and total errors and attempts made by each programmer group	77
Table 4.6:	Correlation between token types and line number for first, second and third, and total. * $p < .05$, *** $p < .001$	79
Table 4.7:	Error Type	83
Table 5.1:	Rules, rule issues, and percentages of programs receiving suggestions for participant rules.....	107
Table 6.1:	Unsuccessful design attempts in formative testing	119
Table 6.2:	Rules, suggestions and examples	121
Table 6.3:	Time participants spent on the tasks	127
Table 6.4:	Categories of responses from suggestion participants about why they accessed and used, accessed and did not use or did not access suggestions.	134
Table 6.5:	Participant characteristics and information access and API usage. ...	136
Table 6.6:	Participants accessed the API information all of the different ways in both conditions	140
Table 7.1:	Objects and actions used for binning snippets.....	153

Table 7.2:	Human moderation criteria	156
Table 7.3:	Left: Hand-authored suggestions. Right: Semi-automatic suggestions and the numbers of suggestions received and accessed by children in our study.	160
Table 7.4:	Transfer task scores.....	172
Table 7.5:	Participants responded to Likert scales from 1-7. $\wedge p = .1$	172
Table E.1:	The 7 suggestions and rules for repeat and the suggestions and rules for do together with 10 or more examples in the cluster.	[267]

Acknowledgments

I am most grateful to my advisor, Caitlin Kelleher. From encouraging me to pursue a PhD during my summer REU through faculty job interviews, Caitlin taught me not only how to do research, but how to believe that I am a researcher. She knew when to push me to improve and when to tell me to take care of myself. I could not have asked for a better mentor for this journey. I hope to someday be as good of a mentor to my students as Caitlin has been for me.

Thank you to my committee members for their valuable feedback and support: Caitlin Kelleher, Roger Chamberlain, Sanmay Das, Bjoern Hartmann, Alvitta Ottley, and Robert Pless. This work was supported by NSF grants 1054587 and 1440996 and the Spencer T. and Ann W. Olin Fellowship.

A huge thank you to my lab mates and collaborators, especially: Kyle Harms, Wint Hnin, Dennis Cosgrove. To Kyle, for being a great friend, colleague, and best teacher of the tricks to running a successful user study at WashU. To Wint, for saving the day with secret snacks during long study sessions, for being an amazing editor, and for all of your hard work on our papers. To Dennis, for the generous code, editing, and teaching knowledge and for being an uplifting force throughout my PhD.

Thank you to my WashU Computer Science, Olin Fellowship, and HCI conference friends. I can't imagine this journey without these communities. Special thanks to Jordyn Maglalang for throwing things at the wall with me when times were tough, to Rebecca Gilson for all of the coffee shop work sessions, yoga, and hugs, and to Austin Henley for the conference and job-search camaraderie.

Thank you to Alana Lustenberger, Jessica Scolnic, Joanna Sebik, and Allie Wahrenberger for the many trips, chats, and love from afar!

Thank you to my family for believing in me, making numerous trips to St. Louis, and for making some of my most productive work sessions possible at the beach. My family has taught me to love learning, put me in a position where I was prepared and able to go to graduate school, and has been an amazing and endless source of encouragement.

Thank you to my partner, Randy. Randy has not only supported me through actions like last minute copy-editing, dinner breaks, and lemon bars, but also in his unwavering belief in me and his kind, yet firm encouragement to take care of myself over all else.

Michelle Ichinco

Washington University in Saint Louis

August 2018

ABSTRACT OF THE DISSERTATION

The Example Guru: Suggesting Examples to Novice Programmers in an Artifact-Based
Context

by

Michelle Ichinco

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2018

Professor Caitlin Kelleher, Chair

Programmers in artifact-based contexts could likely benefit from skills that they do not realize exist. We define artifact-based contexts as contexts where programmers have a goal project, like an application or game, which they must figure out how to accomplish and can change along the way. Artifact-based contexts do not have quantifiable goal states, like the solution to a puzzle or the resolution of a bug in task-based contexts. Currently, programmers in artifact-based contexts have to seek out information, but may be unaware of useful information or choose not to seek out new skills. This is especially problematic for young novice programmers in blocks programming environments. Blocks programming environments often lack even minimal in-context support, such as auto-complete or in-context documentation. Novices programming independently in these blocks-based programming environments often plateau in the programming skills and API methods they use. This work aims to encourage novices in artifact-based programming contexts to explore new API methods and skills. One way to support novices may be with examples, as examples are effective for learning and highly available. In order to better understand how to use examples for supporting novice programmers, I first ran two studies exploring novices' use and focus on example code. I used those results to design a system called the Example Guru. The

Example Guru suggests example snippets to novice programmers that contain previously unused API methods or code concepts. Finally, I present an approach for semi-automatically generating content for this type of suggestion system. This approach reduces the amount of expert effort required to create suggestions. This work contains three contributions: 1) a better understanding of difficulties novices have using example code, 2) a system that encourages exploration and use of new programming skills, and 3) an approach for generating content for a suggestion system with less expert effort.

Chapter 1

Introduction

Many novices who begin learning to program on their own in artifact-based contexts plateau in skills. *Artifact-based contexts* refer to situations where a programmer defines their own goals, without strict constraints, solutions, or pre-defined outputs. The artifacts learners typically create in these contexts usually involve more code than assignments or puzzles. While artifact-based contexts are often motivating, independent learners often fail to gain new skills or do so very slowly. Researchers have demonstrated this issue in blocks environments like App Inventor and Scratch [232, 234]. Plateauing in skills could lead to inappropriate work-arounds or inability to create complex and motivating projects. One reason novices do not seek out new skills may be their unawareness of the possibilities [112]. Because artifact-based contexts do not have specified solutions, systems cannot give hints or suggestions based on a task’s goal state or expected output. Instead, many children can only rely on existing online resources, since they often use artifact-based programming environments informally.

The goal of this thesis is to encourage novice programmers to explore and use new code in artifact-based contexts. To do this, we designed a system called the Example Guru.

The Example Guru suggests new code to novice programmers through example code in an artifact-based blocks programming environment for creating 3D animations. This work fits into a space that currently lacks significant research: in-context support and feedback for independent novice programmers in artifact-based blocks programming environments. To design a system to encourage new code exploration and use, we leverage the theoretical research on examples and learning, the availability of code examples, and the popularity of code example use in programming.

1.1 Resources for novice programmers

Many young novice programmers, especially in the US, who begin programming in artifact-based contexts lack the support of a traditional learning environment [97]. If they had access to an effective classroom context, instruction and feedback would likely foster new skills through a specified curriculum. Classrooms also typically involve projects where teachers or peers provide feedback. This personalized feedback on projects can point out misconceptions or missing skills that assignments with specified solutions may not catch. If programming environments could provide the type feedback typically provided in a classroom to novices as they work on motivating projects, novices might gain relevant and timely skills.

In order to learn these skills, independent novices in artifact-based contexts must seek out support that often requires them to leave their programming environment or current context. Most novice programming environments provide resources like documentation [214], tutorials [197, 218], or forums [113, 195]. They also often enable users to share programs so that novices can learn from each other. Many novice programming environments have online communities, such as Scratch, App Inventor, Looking Glass, Greenfoot, and Kodu [71, 114, 121, 196]. Novices can also seek out tasks to learn specific skills, like puzzles [6, 76] or online

courses [107]. All of these options require that the user seek out information outside of their programming context, which many novices will choose to do infrequently or not at all [83]. Despite whether novices have access to or seek out learning resources, many of these methods of learning inside and outside the classroom, including the one presented in this thesis, employ examples in some form.

1.2 Examples

Our approach uses example code for three reasons: the effectiveness of examples in learning, their large-scale availability, and their importance in the work-flow of programmers. Providing feedback in the form of examples may be a productive and scalable method for supporting novice programmers in artifact-based contexts. However, studies of less experienced programmers have indicated that reusing example code can be problematic [185]. This prompted our work to better understand novices' example use in order to employ examples in a support system.

Research has shown that providing examples to learners, especially in the form of worked examples, can be highly effective for learning [12, 35, 212]. Worked example theory and design are informed by *cognitive load theory*, which indicates that learning materials should limit the amount of cognitive load required [224]. Researchers have begun to explore how worked examples can help novices learn coding using subgoal labels and self-explanation [131, 149, 150]. Subgoal labels can reduce cognitive load when solving problems with examples as they segment the examples and describe the steps. Self-explanation is an effective strategy for learning from examples, as novices work to understand the example using context and reasoning [41]. The related work section provides an in-depth explanation of the theories and studies of worked examples in educational psychology research.

In addition to being effective learning resources, examples for programming are commonly available within online resources like code repositories, documentation, and forums. Code repositories, like GitHub [29], have code submitted by millions of users. Not all of the code within these repositories may be valuable as code examples, but many open-source projects likely have well-designed code that programmers use. Documentation or other static content, like written tutorials, often include code examples. Web forums for programmers, like Stack Overflow [203], contain many code snippets within questions and answers. These questions and answers often cover large portions of information, such as 87% of the Android API classes [163], and provide valuable example code snippets [155]. The availability of code examples makes them a common resource for current programmers and a potential resource for future systems that involve example code.

Programmers of varying skill level use available example code resources on an everyday basis. Experienced programmers often search for examples that they can find on the web or within programming environments [23, 84]. Non-expert programmers also attempt to utilize available example code to learn or fix errors. However, novices often have difficulties using examples without support [89, 183]. In order to be able to leverage the large quantity of available example code to support novices learning independently, this thesis first aims to better understand the difficulties in using examples before using the findings to design a system that suggests example code.

1.3 Approach

This dissertation aims to: 1) better understand the challenges novices have using examples in order to design support for examples, and 2) reduce novices' unawareness of relevant API methods and programming concepts by suggesting example code. To do this, we ran two

studies to understand how novices struggle when using example code. We then used these results, along with a study of experts making suggestions, to inform the design of a system that suggests API methods and programming concepts to novice programmers as they work on code projects. Finally, we designed an approach to generate content for this type of suggestion system with less human effort.

The system we designed to suggest example code to novices during artifact-based programming is called the Example Guru. The Example Guru contains a set of suggestions that introduce new skills. Each suggestion has a rule that checks a novice program for the opportunity to make the suggestion. The suggestions each have two example code snippets that demonstrate the skill and have associated descriptions.

Imagine a novice programmer named Joanna coding an animation with a bunny walking a far distance. The default speed for the bunny walking is very slow, which frustrates Joanna. She does not realize that the walk method has a parameter to change the speed called *walkPace*. The Example Guru analyzes her code and triggers a suggestion to change the speed of the walk animation. Joanna notices the suggestion and chooses to explore it by clicking on it. When she opens the suggestion, it has two examples: one that shows a character walking faster and another that shows a character walking slower. Joanna views both of the examples, which helps her figure out what speed she wants to use for the parameter. However, Joanna does not know where to find the *walkPace* parameter to change it, so she clicks the ‘show me how’ button. This button triggers the interface to provide instructions that demonstrate where to modify the *walkPace*. Now Joanna is able to change her program to make her bunny walk faster. She is happy that her animation looks better and has also learned about a feature of the API. As users gain skills, the suggestions introduce more complex API methods and programming skills, like joint movements to improve a simple turn, or parallel execution to make an object move diagonally.

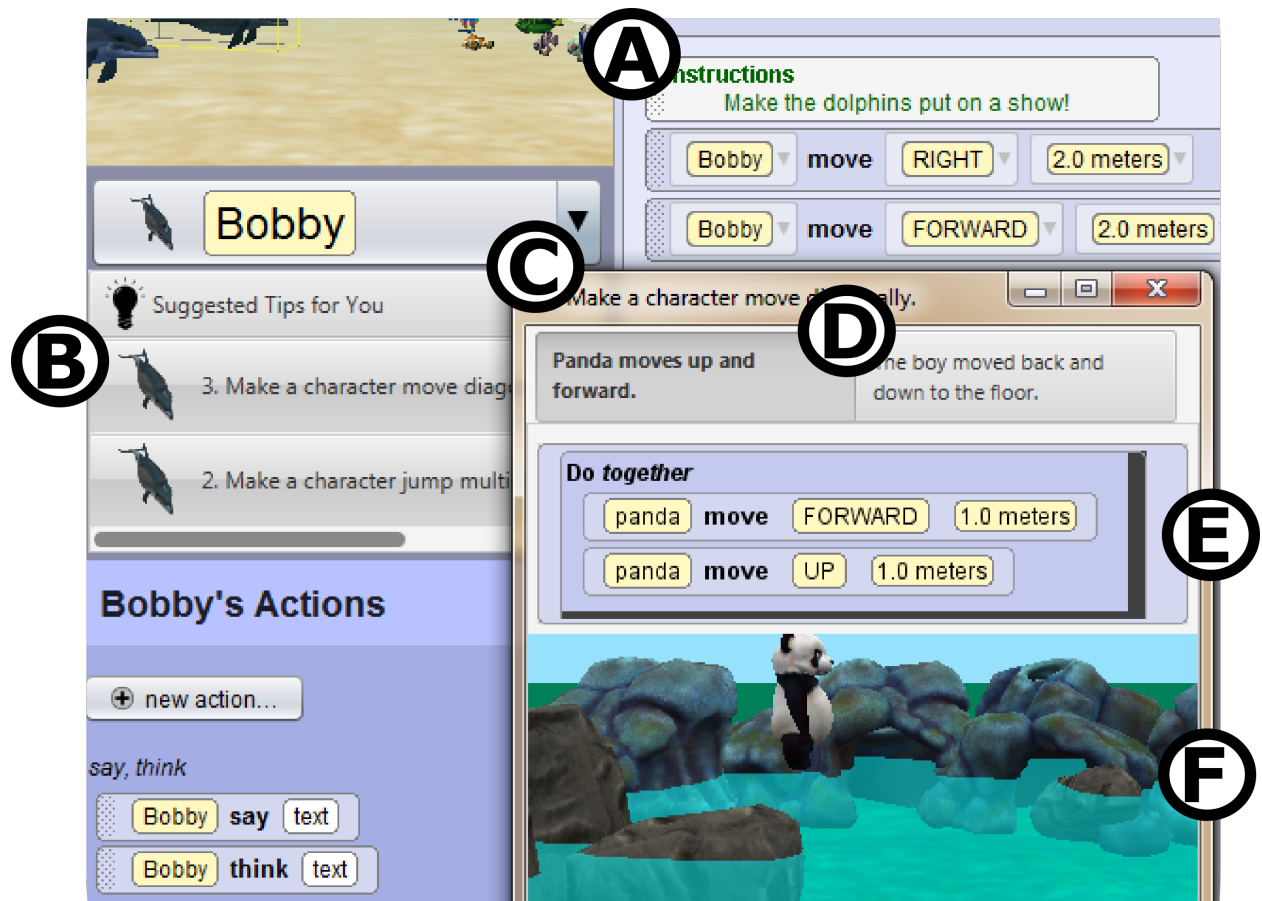


Figure 1.1: (A) Looking Glass programming environment. (B) List of suggestions. (C) An opened/accessed suggestion. (D) The two code examples with the primary one selected. (E) The code for this suggestion, with the *do together parallel execution* block emphasized. (F) Preview execution of the code.

1.3.1 The Example Guru system overview

The Example Guru implements an approach for suggesting new code concepts to programmers throughout the programming process. It has three main elements: rules, suggestions, and examples.

Rules

Rules statically analyze novices' programs for opportunities to make suggestions. Each suggestion has an associated rule, which triggers the suggestion when it finds a pre-defined combination of code. Rules statically analyze the abstract syntax tree of a program to find combinations of code that indicate opportunities to suggest API methods or programming skills.

Rules execute after a novice programmer executes their program. The point at which a programmer executes their program is likely when they want to check whether their code works correctly or when they have completed an idea. The rules execute at this point because this is likely to be a time when novices are most open to receiving suggestions about their programs.

Suggestions

The Example Guru makes suggestions to novice programmers as they program. Each suggestion encourages users to improve their artifact in a particular way using a code block that the user has not yet used. Users can choose to interact with suggestions, but are not required to, as the suggestions use a *negotiated* interruption strategy. Negotiated interruptions make information available, but allow the user to choose when and if they want to interact with the content [140]. This has been an effective interruption strategy for end-user debugging [179]. Novices can access the suggestions from a list of titles in a panel (see Figure 6.1-B) or from an annotation on the code block related to the suggestion. Hovering over the annotation shows the title of the suggestion. Once accessed, users can view two code examples and execute the examples. Hovering over the code triggers a tool tip to appear that explains how the code works and where users can find the new code block.

Code examples

Each suggestion shows two code examples. The user can execute each of the examples to see how they work by clicking on the scene (see Figure 6.1-D). The code examples demonstrate the code that enables novice programmers to implement the suggested idea. Each code example has a description of the output of the code, rather a description of the code itself. The idea is that the description should explain the effect of the example, rather than describe the code. The Example Guru's two code examples show contrasting information for API methods or similar information to reinforce programming concepts. Ideally, the two code examples will encourage the novice programmer to perform self-explanation [41]. By explaining to themselves how the two examples are similar and different, the novice programmer will likely better understand the concepts demonstrated.

1.3.2 Looking Glass

I implemented the Example Guru within a novice programming environment called Looking Glass. Looking Glass is a blocks-based novice programming environment designed for middle school children aged 10 to 15 to make 3D animations, as shown in Figure 1.2. Looking Glass is based on the Storytelling Alice programming environment [105] and is available online for free [121]. Looking Glass code is written in the Java programming language and uses the object-oriented paradigm. Blocks-based programming environments have been designed to reduce the complexity of the syntax for novice programmers and prevent novices from making syntactic errors. Looking Glass enables users to create animation programs by dragging and dropping blocks to animate objects rather than by typing code.

Each Looking Glass program, or 'world', has a 3D scene with a background, objects, and camera. Scenes can contain a variety of different objects like animals, people, furniture,



Figure 1.2: Looking Glass

plants, and other props. For example, Figure 1.2 shows a scene with a pig on an island and a helicopter. In general, Looking Glass programmers can choose a pre-created scene or can create their own. In the context of this work, we provided users with pre-created scenes in order to focus on programming rather than scene creation.

Children can create animations by dragging and dropping blocks of code that operate on the objects in the scenes or they can change the order of the actions. The code blocks allow users to make objects perform actions like move, turn, change size, and change appearance. Scene objects can also *speak* with speech bubbles to advance the story line. Figure 1.2 shows

code blocks that make the pig speak and then turn its shoulders to make it wave. Many code blocks require users to fill in argument values, like the direction to turn or the distance to move. They also commonly have optional arguments, like the duration or speed of an action. ‘Action ordering’ code blocks include basic programming concepts like simple parallel execution (‘Do together’), a simple loop with a numerical iterator (‘Repeat loop’), variables, conditional logic, and more complex loops. Once a programmer has added code blocks, they can click ‘Play’ to execute their program, which allows them to watch the animation they have created.

I chose to complete this work within Looking Glass because users primarily focus on creating artifacts, or animations. They can select their preferred scene and use code to create a story that they choose. This process is similar to many other artifact-based coding contexts where users do not typically receive feedback when working outside of a classroom, such as Scratch and App Inventor. Looking Glass is not as well known as other systems, making it easier to recruit participants unfamiliar with the system and API. Looking Glass also already had a built-in static code analysis system, making it more efficient to implement rules for the Example Guru.

1.4 Hypotheses

This dissertation has three main hypotheses surrounding suggestions and examples.

1.4.1 Hypothesis 1

Studies of novices using examples will indicate the challenges novices have using example code and will provide new directions for how to support example code use for novice programmers.

1.4.2 Hypothesis 2

Suggesting example code to novice programmers in artifact-based contexts will increase the number of new API methods and programming concepts novice programmers add to their programs compared to existing, static, forms of support.

1.4.3 Hypothesis 3

A support system for novice programmers can be created with less human effort than hand-authoring. The support system content will be equivalent to the hand-authored content.

1.5 Contributions

- My study of children using examples to solve tasks suggests that one main reason novices have trouble using examples is that they do not realize which element within the example is important [91].
- My study comparing novices and experts recalling example code suggests that novices focus on different elements than experts and cannot retain as much information [92].
- My study of adults making suggestions and authoring rules provides an understanding of how experienced programmers might make suggestions to novices and indicates what makes good suggestions and rules [94].
- I designed and built the Example Guru system, an example of how to implement the approach within a novice programming environment for animation creation which can be a model for systems in similar contexts [90]. This system could be easily translated to other similar novice programming environments.

- My studies show that suggesting content during artifact-based programming is more effective than static support for encouraging children to explore programming concepts. This approach likely applies broadly to artifact-based programming [90, 93].
- I designed an approach for semi-automatically generating suggestions and rules, which produced similar suggestions to a hand-authored set with less human effort [93].

1.6 Summary

The following chapters discuss related work, studies addressing my three hypotheses, and future work.

1.6.1 Related work

Chapter 2 provides an overview of related work and background in example use in programming and education. It also provides a summary of research on systems that suggest information to learn and use in artifact-based contexts. This work is related to the general idea of the Example Guru and also includes automatic and semi-automatic generation of support for programmers.

1.6.2 Hypothesis 1: Studies of novices using examples

Chapters 3 and 4 evaluate the hypothesis that studies of novices using examples will clarify the challenges of example use for novices. These chapters describe two studies exploring novice example use: 1) a study where children attempted to solve programming problems with example code available, and 2) a study comparing how novices and experts recalled code snippets.

1.6.3 Hypothesis 2: Suggesting example code to novices with the Example Guru

Chapter 5 describes an exploratory study of experts creating suggestions and rules that informs the types of suggestions and rules implemented in the Example Guru. **Chapter 6** evaluates the hypothesis that suggested examples will increase novices' use of new programming skills. It provides a detailed description of the design of the Example Guru, rationale for design decisions, and a study comparing suggestions to documentation. The study showed that novices chose to access suggestions about three times as often as documentation and used new API methods from suggestions more often than from documentation.

1.6.4 Hypothesis 3: Generating large-scale suggestion systems

Chapter 7 describes a method for semi-automatic suggestion generation. The evaluation of novice programmers using semi-automatically generated suggestions and comparison to hand-authored suggestions support the potential of this semi-automatic generation approach.

1.6.5 Summary and future work

Chapters 8 discusses the potential impact of this work and the possibilities for future work. The future work section discusses the following questions:

- How can we apply large-scale in-context suggested content to help people learn other topics?
- How can we automatically generate context-relevant support at scale?
- How can learners contribute to and engage with the support model?

Chapter 2

Related Work

Existing work related to this thesis spans from theories of learning to systems that support programmers. Educational psychologists have a long history of studying the cognitive processes behind learning, especially learning using examples. This body of work supports the benefits of incorporating examples into learning systems, provides a theoretical basis for thinking about learning with examples, and models ways to explore how learners use examples. As programming has become more popular, researchers have begun to focus on the cognitive processes programmers use when understanding code or examples, as these are critical and common activities for programmers. The existing research supports ways to help programmers comprehend large software projects, but does not specifically address comprehension of example code. Instead, much of the work surrounding examples for programmers focuses on making the example code more accessible. Many systems support programmers in accessing example code for more efficient software development, but none suggest examples to novices in artifact-based contexts with the goal of encouraging exploration of unused code. Research motivating and related to this thesis spans three major topics: 1) the cognitive processes of using and learning from examples, 2) the best ways of representing

examples for programmers, and 3) systems for making useful examples available to learners and programmers.

2.1 Examples and learning from a cognitive perspective

A significant body of research supports and studies the value of examples for problem solving and learning. This work has been important in the ideation and design of our studies of novices using examples and the Example Guru. In order to understand novice programmers' difficulties using examples, we draw on the theoretical groundwork of: 1) analogical problem solving, and 2) worked examples.

2.1.1 Analogical problem solving

In essence, analogical problem solving has the same properties as problem solving using an example. Cognitive psychologists define analogical problem solving as using one provided problem and solution (the base) to solve another problem (the target) [60, 62]. Although the base problem is not often called an example in the literature, it essentially functions in a similar way to an example. Novice programming with examples is most closely related to the research on analogical reasoning in mathematics [175]. In mathematics, a student might be asked to solve the problem $3x + 2 = 11$ using an example $2x - 4 = 6$. The student must first map the related elements of the problem and example. This problem and example pair has three sets of related elements: the $3x$ and $2x$, the $+2$ and -4 , and the $=11$ and $=5$. Understanding the relationships between these elements is critical to benefit from a provided example. If a learner can understand these mappings, it may help them to adapt the steps in the example to their problem. Prior work is divided on whether an understanding

of the mappings between base and target problems is necessary or sufficient in solving the task. Gentner describes the structure-mapping theory, arguing that the base problem and the target have equivalent sets of relations between problem elements [60]. The analogy is a mapping between the set of relations for the base and target problems. Gentner’s work suggests that the primary difficulty associated with solving a problem using an analogy comes from mapping the example and the target. If learners can correctly map the example and target, they can likely solve the problem correctly. In contrast, Novick and Holyoak’s research suggests that a learner must understand the mappings between the example and target problems, but that the mappings may not be sufficient to enable a learner to solve a task [158]. When learners need to adapt an example to fit their target problem, some learners may succeed at mapping but struggle to construct a full solution [158]. One study looking at analogical reasoning in novice programming supported Novick’s theories, finding a weak correlation between a mapping task and programming task success [89]. This indicates that mappings may be one element of the difficulties using examples.

Because understanding the mappings likely benefits learners to some extent, systems should try to support learners in finding and understanding the mappings. One way to do this may be to make the base and target problems have high surface similarity, which learners often find easier to map [61]. Surface similarity means that the problem and example elements that are correlated are also similar, such having the same location within the problem. As a learner becomes more familiar with a concept, reducing the surface similarity could then help them to gain a deeper understanding. Another related way to support learners in using examples may be integrating cognitive load theory in the design of worked examples.

2.1.2 Worked examples for learning

The body of research surrounding worked examples supports the integration of examples into a system for independent learning. We first provide an overview of cognitive load theory, which is used in the design of worked examples.

Cognitive load theory

Cognitive load theory is a theory for reducing the information processing load typically employed in educational research to support learning of cognitively complex information [211]. This type of content typically requires the understanding of many interacting elements, which need to be understood individually and also together in order to gain a deep understanding. However, the short-term memory, or working memory, that processes these components is a highly limited, but critical resource for learning [13, 146]. Learners must use their short term memory to process new information until they create schema, structures that organize knowledge and are stored in long-term memory. In order to create schema, learners must be able to accommodate the amount of cognitive load associated with the new information. In addition to the cognitive load of the content being learned, cognitive load can also come from the design of the instructional information, like if learners must spend extra effort connecting important elements or figuring out directions. When creating instructional information, it is critical to reduce the cognitive load imposed by the format of the presentation.

Cognitive load theory specifies three types of cognitive load that play into difficulty learning complex information and the design of worked examples: intrinsic cognitive load, germane cognitive load, and extraneous cognitive load. The intrinsic cognitive load of learning certain content is determined by the element interactivity of the components [211]. In many cases, the many interacting elements of complex information must be learned simultaneously, due

to the importance of how they interact. Modifying the instructional format cannot change the intrinsic cognitive load of content. The only way to reduce intrinsic cognitive load is to alter the type of information or problem presented. Germane and extraneous cognitive load are controlled by the instructional designer. When the instructional design supports schema formation, it imposes germane cognitive load. When it interferes and prevents schema formation, the instructional design imposes extraneous cognitive load. One common case of imposed extraneous load is when tasks require that the learner seek out information from a different location in the content [161]. This interrupts learners' focus on the information they need to actually learn. One instructional method, worked examples, has been shown to reduce the extraneous cognitive load imposed on learners.

Worked examples

Worked examples are problems with worked solutions designed based on cognitive load theory [212]. Worked examples can be highly effective for learning topics like mathematics and science [12, 212]. Learners often receive worked examples alongside problems to solve in order to aid them in solving their problem. A mathematics or physics solution naturally breaks down into steps, but the design of a worked example for programming is less clear. Their integration into programming education contexts is becoming more common, as researchers build systems to help educators create them more easily [122] and as systems provide interactive worked examples, like the Problem Solving Tutor [120]. Researchers integrating worked examples into programming contexts are also integrating cognitive science findings about worked examples, like the benefits of: self-explanation, providing multiple worked examples, providing subgoal labels, and faded worked examples.

Self-explanation: During effective self-explanation, learners generate explanations of learning materials and relate those explanations to the relevant generalized principles, which deepens

their understanding [41, 171, 174]. Self-explanation is an example of an instructional practice that increases germane cognitive load in order to further learning. Studies have found that self-explanation questions helped students in learning programming [165], especially if they included extra questions to help focus the self-explanation [222]. Recent work has applied self-explanation to programming and combined it with subgoals by having novice programmers author subgoal labels [130, 150]. Recent work has also shown that having novices write in comments has been effective in helping them to solve programming problems [221]. This research supports the idea of combining self-explanation with examples in systems for novice programming.

Multiple Examples: Providing multiple worked examples for learners to study can help them to grasp content [12, 63]. However, Catrambone and Holyoak showed that multiple examples only support learners in solving problems when the learners are instructed to use the similarities between the examples [37]. This prompts the learners to perform self-explanation.

Subgoal Labels: More recently, researchers have begun to focus on the design and effectiveness of worked examples for programming using subgoal labels. Research on worked examples overall indicates that subgoal labels can likely help learners better understand programming examples [36]. Adding subgoal labels to instructional information as a way of simulating a worked example has been effective for programming [128, 129, 149]. Researchers have also designed a way to crowdsource subgoal labels for videos, engaging learners in self-explanation [109].

Fading: Fading worked examples can also help learners. Faded worked examples are sequences of worked examples in which stages of the worked examples are removed in order to fade from a fully worked example to only a problem [172, 173, 193]. Fading reduces the extraneous cognitive load of worked examples by reducing the amount of information learners need to

figure out on their own until they are ready for it. Researchers have shown that faded worked examples can be effective for programming [70].

2.1.3 Takeaways

This body of work addressing how to best design instructional material and support problem solving has inspired both our work on novices' issues using example code, as well as the design of the Example Guru. While existing work provides some insight into ways to help people learn from examples, it does not address the problems novices may have when they do not understand example code. Another body of work, rooted primarily in software engineering, looks at how programmers comprehend code and ways to help programmers comprehend code more efficiently.

2.2 Comprehending and presenting example code in programming and software engineering

In order to be effective software engineers and for software to be robust, programmers need to be able to quickly comprehend code. In order to be able to support code comprehension, researchers have begun to look at how programmers comprehend and use code, as well as beneficial ways of summarizing and visualizing code examples to better support programmers. This research inspires both our studies of novices' difficulties using example code, as well as to the design of the Example Guru's example presentation. This section discusses related work in the areas of: code comprehension, code example use, the quality of examples, code summarization, and code visualization.

2.2.1 Code comprehension

Code comprehension is a core element of using code examples, as well as programming in general. Researchers have developed both theoretical and empirical bodies of work on code comprehension.

Review papers cover the extensive body of work on code comprehension and emphasize two main high-level theories: top-down and bottom-up [204, 223]. In top down theories, expert programmers use beacons, programming plans, and rules to make sense of code [25]. In bottom-up theories, programmers chunk related elements during comprehension based on high-level schema in long-term memory [199].

Empirical work on code comprehension has used a variety of methods. Researchers have evaluated code comprehension through eye tracking [32, 33], cerebral blood flow measurement [154], answering questions [14, 229], code modification [115], and debugging [219]. Research in computer science education has shown that code comprehension activities can effectively assess programming knowledge, as well as help students learn [202, 208]. While code comprehension at a high level likely affects the way people comprehend examples, the work often addresses code at a much larger scale than a code snippet. Researchers have also looked at how programmers try to use examples.

2.2.2 Code example use

Several studies seek to understand how programmers use example code naturally, though some focus on experienced programmers as opposed to novices. One such study explores experienced programmers reusing example code and, similar to this work, describes the programmers' behaviors during tasks using example code [185]. Rosson and Carroll find

that expert programmers ‘debugged into existence’ and only used examples as an initial source of information. It is important to note that this study took place before example code was widely available online. Another study looked at how programmers search the internet throughout programming tasks and found that programmers used online code examples for learning and reminding themselves of what they already know [23]. They also discovered that programmers started to use code they found before fully understanding it and made mistakes while adapting copied code. However, we do not know how these behaviors and problems apply to novices.

Few studies focus on non-expert programmers and those that do only briefly discuss code examples and reuse as a part of larger works. In analyzing the practices of informal web development, Rosson, Ballin and Nash found that programmers often use example code as a model when looking for general ideas of ways to design websites [183]. However, when they try to use the code, the programmers cannot effectively integrate it into their projects. One factor that may play into programmers’ abilities to effectively use example code is the quality of the existing examples.

2.2.3 Quality of examples

Researchers have looked at the effective qualities of code examples to establish how to design code examples for documentation. Studies have looked at how programmers rated examples and what they wanted in code examples. They found that programmers preferred concise code examples that highlight relevant code [213], are segmented and described [155], provide placeholders to indicate where to insert new code, and have understandable names and variables [31]. Another study found completeness and correctness to be important qualities [213]. Several of these studies list ‘readability’ and ‘understandability’ as important features, but do not clearly define what those qualities mean [31, 213]. Research has also suggested

providing natural language explanations to improve examples [38, 80]. However, this type of work has focused mainly on more experienced programmers who are often adults using text programming languages. Much of the work addressing how to help programmers comprehend and quickly make sense of code has been in the areas of code summarization and visualization.

2.2.4 Code summarization

Code summarization is the process of creating or generating brief descriptions or shorter code snippets, like summarizing English text. Code summary generators often use heuristics, information retrieval techniques, or machine learning to automatically create the summaries [74, 148, 235]. Researchers have explored what programmers think is important, using eye-tracking [182], and by asking programmers to manually create code summarizations [236]. They found that programmers spent more time looking at method signatures and often included structural components and ‘easy to miss’ code, but did not focus on control flow keywords, method invocations, or exception handlers. Many researchers motivate code summarization work with the inefficiencies of code maintenance. They do not address code summarization for novice programmers, which could explain more about which parts of code novices think are important.

2.2.5 Code visualization

Beyond code summarization, another way to make it easier for programmers to use code or code examples is to provide visualizations. Researchers have developed effective interactive visualizations for helping novice programmers understand animation programs [238]. Others have developed rich code visualization support for text programs, and found that code visualization can reduce time needed to answer questions about code, compared to basic

syntax highlighting [11]. However, work on code visualization has focused more on large-scale program understanding than code examples that demonstrate specific concepts.

2.2.6 Takeaways

Better understanding how programmers understand and use code, as well as using that information to better present code, is critical in supporting programmers in maintaining large software projects and learning new programming skills. The findings often align with cognitive load theory, in terms of code comprehension theory and properties of effective examples. The findings also support the need for a better understanding of novice programmers, who have particular difficulties integrating and understanding example code.

2.3 Supporting access of relevant examples

In addition to presenting examples in ways that novices can understand, novices need to receive example code relevant to their program at the correct time. Like the Example Guru, many systems try to help programmers gain access to relevant examples. Existing systems typically focus on supporting programmers in accomplishing programming tasks, rather than learning new skills. These systems try to make examples accessible in order to: 1) help programmers use correct API methods or commands, 2) fix code quality, or 3) fix errors.

2.3.1 Recommendations for API or command usage

APIs, as well as other complex software systems, commonly have many available methods and commands. Many features remain unknown by users, due to users being unaware of the possible capabilities or because of naming issues. Chapter 6 describes a study in which the Example Guru suggests API methods to encourage exploration of unknown methods. To

help users better use the full capabilities, researchers have developed support systems for APIs and complex software that provide recommendations to users. These systems typically base the recommendations on community data or individual usage.

Recommendations from community data

Some existing systems leverage overall community usage and sets of community-created artifacts to make suggestions for API methods and commands. The Example Guru also uses a repository of programs as a basis for designing suggestions, but focuses more on introducing previously unused API methods.

One way systems provide API and software support is by using community data to recommend commonly used commands. Some of these systems provide rankings of commands by basic counts of how often they are used, such as by: providing lists of API methods in a programming environment [86], providing lists of commands within software [135], or by emphasizing more commonly used API methods in documentation [206]. Other recommendation tools use collaborative filtering algorithms, which classify a user’s behavior within community usage data in order to recommend API methods [139] or software commands [119, 134].

Research has also leveraged communities of user examples, Q&A information, and code completion methods for making recommendations. Rather than making recommendations directly, these systems typically try to improve example code retrieval by recommending examples using the programmer’s context. They do this by: comparing users’ code against repositories [87], mining patterns of APIs often used together [239], matching related words to find code examples for similar types of functionality [15], or using input and output types [124, 215]. For more project-specific examples in open-source projects, recommendations have been based on the program history and types of tasks [47, 123]. Other than examples,

systems also use community resources to inform relevant Q&A recommendation [46], code completion [10, 26], and parameter completion [9, 237].

Recommending support based on individual usage

Rather than using community data, other systems provide recommendations for effective programming, APIs, or software commands based on either the user’s behavior or the artifact they are working on and pre-defined suggestion criteria.

Research has used current or past behavior to recommend support to users of complex software and APIs. AmbientHelp recommends information based on the commands a user is working with at any point in time [133]. Similarly, CoDis suggests unfamiliar commands based on command patterns and the time elapsed since the user’s last activity [241]. Another tool bases API recommendations on the user’s programming history [178].

Some systems only use the artifact a user is creating to recommend help for APIs, programming, and commands in complex software. Tools for recommending APIs consider the structure of code to recommend API methods, such as by looking for redundant code [104] or using identifiers from a class’s abstract syntax tree [82]. Documentation recommendations can also rely on artifacts, like by connecting method invocations to documentation [51], or by relating software interface elements to documentation [108]. Systems also recommend commands to users in sketching software based on the drawing artifacts that users create [58, 95, 152].

2.3.2 Recommendations to improve code quality

Like helping users learn complex APIs and software systems, existing systems support programmers in improving their code quality through suggestions. Improving code quality is most related to the Example Guru’s abstract programming concept suggestions in Chapter

7. In order for code projects to be maintainable, they need to follow best practices in code style, like keeping methods short and using abstraction appropriately. Three types of systems define and assess code quality more automatically: code smell detection methods, tools for code style checkers, and a subset of automatic grading systems.

Code smells

Code smells are patterns of code, such as long methods or duplicated code, which may indicate a problem in the maintainability of code and can be used to detect issues. These definitions and the implementation of the definitions closely align with the Example Guru’s rules. Fowler and Beck developed categorizations and definitions of “code smells” [57]. Based on these definitions, a body of research has investigated humans and metrics as detectors of code smells. Several studies explore human evaluations of code smells, finding low agreement for detection of complex code smells [126, 127, 192]. Mantyla, Vanhanen and Lassenius developed a taxonomy to enable better understanding of code smells for human detection [125]. They found that their taxonomy aligns with correlations software developers noted between code smells. DECOR and “detection strategies” [132, 147] enable humans to operationalize code smells using software metrics, while another set of systems use metrics to automatically detect code smells [57, 106, 116, 151]. Based on code smells, Cunha, Fernandes, Ribeiro and Saraiva identified smells in spreadsheet code and created a tool to find these smells [48].

Code style checkers

Similar to code smells, a number of tools allow programmers to check the quality and style of their code. PMD, Klokwork, SourceMonitor, QJ-Pro, and StyleCop are a few examples of these types of tools [45, 111, 166, 201, 205]. These tools provide standard metrics and

allow customization of metrics, but are aimed at professional programmers and do not have support for suggesting new programming concepts or skills.

Automatic grading

On a smaller scale than code smells and style checkers, a number of automatic grading systems measure the quality of student code using standards and structural properties. Some systems use standard metrics as code quality measures for student assignments, like the ISO/IEC 9126 standard [1, 24] or Berry and Meekings’ style metrics [20, 98]. Other automatic grading systems consider the structure of code. For example, one framework employs cyclomatic complexity [138, 217], while another study uses LOGISCOPE to find knots [143]. Cyclomatic complexity and knots find problematic code by looking at the paths through a program. These evaluation techniques focus on complex structural and style issues or aspects of code such as whitespace, all of which novices can often ignore while learning basic programming concepts.

Suggestions to help programmers learn APIs and improve their style of code most closely relate to the Example Guru’s suggestions for artifact-based programming. The set of systems that support programmers in fixing errors and incorrect solutions through suggestions use relevant strategies for providing information, but do not need to consider the programmer’s motivation.

2.3.3 Recommendations to fix errors

Researchers have made progress on suggested support to help programmers for: solving programming assignments and resolving bugs.

Automatic grading and intelligent tutoring systems

Automated grading, similar to rules in the Example Guru, check code correctness in programming assignments and can provide quick feedback to a large number of students. A variety of automated grading systems allow teachers to specify assignments and tests that evaluate the correctness of code output [7, 21, 55, 59, 88, 101, 187]. While these systems require assignments with defined answers, independent learners using novice programming systems often work on artifact-based programs that do not have right or wrong answers. Researchers also use automated evaluation of code to provide programmers with hints along the way [169]. However, these systems and tools focus on contexts where learners are working toward a specific solution, making it possible to generate hints based on the differences between the learner’s code and the correct solution. The Example Guru is designed to support programmers in unbounded contexts, where there is no specific solution.

Finding bugs

Methods for evaluating source code to locate bugs and errors are also similar to Example Guru rules, which often affect whether code executes correctly. PREfix and PREfast are systems that successfully determine the density of defects by analyzing code [153]. Static code analysis can also find issues in large-scale multi-threaded programs and detect security vulnerabilities [8, 103, 225]. However, novice programmers, focused on learning programming constructs, are mainly shielded from these complex bugs.

Studies have used crowd-sourcing to assist new programmers in understanding compilation errors and bugs. HelpMeOut, BlueFix and Crowd::Debug utilize example code from a database of users’ error fixes, in conjunction with expert explanations, to assist novice programmers in understanding and fixing bugs [5, 79, 226].

Software systems and programming environments suggest information to users based on errors. Tools for non-expert programmers recommend information to try to help users who have hit a barrier in completing a code task [100] or who have errors in their code [79]. Two specific scenarios where recommendations based on errors can be especially useful are: in pasting and adapting code examples [54], and in complex software systems where commands are easily mistaken for each other [117]. While these systems effectively suggest examples to help resolve errors, they do not necessarily introduce new skills and require that the user has hit a problem in order to know what to suggest.

2.3.4 Providing examples as a way of recommending examples

Another way of providing examples is to make it easier to access them through repositories or search. Educational systems and systems for programmers provide specific examples or support programmers searching for examples. While not specifically recommended, the methods used to select and support these examples are often similar to the ways context-sensitive suggestions are triggered.

Educational systems

Educational systems for programming often provide examples similar to the Example Guru, where code is available along with the ability to run the code. Researchers have worked on example selection [28], as well as presenting examples as learning material for learning programming [27, 156, 227]. One of these tools, the ‘Explainer’, provides support for learning from programming examples, by allowing programmers to view multiple forms of the example as well as programming plans [170]. With Explainer, participants were more consistent and direct in how they completed tasks. Yet, these studies focus primarily on the design of the

systems, as opposed to understanding how novice programmers use examples and what issues they have.

Supporting code search and example integration

End-user programming systems focus on enabling correct selection of examples and supporting the re-purposing of example code, but tell us little about what programmers are confused about as they try to use the examples. Some tools, like Blueprint and Fishtail, integrate example search into programming environments, improving programmers' abilities to search for examples without having to switch contexts [22, 188]. Other tools integrate with web browsers. Mica and Codetrails augment searches to improve the results for programmers looking for examples [68, 207]. In addition to improved example search, Snipmatch also supports integration into code, similar to Codelets and Webcrystal [38, 159, 230]. On the other hand, Looking Glass provides a way for novice programmers to select which part of a larger program they want to reuse [72]. These studies often compare programmers' success with and without the tools, but do not address the behaviors of programmers using examples. In this work, we seek to describe how novices utilize examples when programming.

Processes for selecting relevant code use varying levels and types of extra context. Some take short queries to find related code [40, 142, 186, 216]. Others utilize broader code structures [16, 85, 157] or add in more information, like the frequency of terms [167, 207], the programming language and framework [22], or timing [240]. Very closely related to our method, some systems determine the behavior of code using information retrieval techniques on the text of the code and text about code [190, 209].

2.3.5 Takeaways

An extensive set of research and systems aims to make example code more available to programmers of all levels and contexts. These systems support the effectiveness of recommending examples to programmers as they work for both concrete information like API methods, as well as more abstract information, like code smells and hints toward task solutions in intelligent tutoring systems. There are two key features of the Example Guru that set it apart from other systems: 1) the Example Guru suggests new code in order to reduce plateaus in coding skills, and 2) the Example Guru aims to motivate programmers by suggesting new code that also improves the programmers' artifacts.

Chapter 3

Understanding Novice Example Use

Note: Parts of this chapter were published in Visual Languages and Human-Centric Computing 2015 [91].

3.1 Introduction

Prior work found that novices struggle to make effective use of programming examples, but researchers have not yet delved into the specific reasons why this happens. A better understanding of these difficulties is critical to the design of the Example Guru and other systems that suggest examples to novices. This chapter addresses hypothesis 1: that studying novice use of examples will reveal challenges and inspire support. To explore this hypothesis, we ran a study in which novice programmers attempted to use examples to solve specified problems. We focus specifically on how middle school children use examples to inform support for young novice programmers.

This study was inspired by the design and analyses of previous research on the challenges and strategies of non-expert programmers. At a high level, researchers have studied the general behaviors exhibited during programming, such as debugging [110] and barriers in learning programming [112]. These studies typically record participants talking out loud to get a better understanding of participants' thought processes throughout the tasks. More specific to code reuse, research has also investigated the behaviors of non-expert programmers during mashup programming [34] and when attempting to locate functionality in unfamiliar code [73]. However, the body of research on novice and end-user programming behavior lacks focused explorations of example use.

To work towards this deeper understanding of novices' example use, we ran an exploratory study of young novice programmers using example code to solve programming tasks. The children worked in pairs editing code using examples. We recorded their conversations to capture their natural discussion about the tasks, what they were confused about, and what plans they had to solve the tasks. This study answers two questions: 1) what hurdles do novice programmers encounter, and 2) what strategies do novices use while attempting to use examples?

To find hurdles and strategies, we analyzed the transcripts of participant conversations using a grounded-theory-like approach. The transcripts show that in most tasks, participants have a 'realization point', in which they talk about now understanding what they need to do to solve the task. The existence or lack of a realization point reveals which participants had the most difficulties and indicates which parts of the task they struggled with the most. Based on this analysis, we present the strategies novice programmers used and the challenges they encountered. Finally, we suggest how systems can support novice programmers in overcoming hurdles.

3.2 Exploratory study

We ran an exploratory study to understand the hurdles encountered and strategies used by novice programmers working with examples.

3.2.1 Materials

To design the materials for this study, we had 9 pilot study participants, who completed tasks with examples. We augmented Looking Glass with the examples, so that they would be within the programming environment. We iterated on the design of the tasks and examples in order to make them challenging in order for participants to need to discuss the tasks to figure them out.

Looking Glass augmentation

In this study, we augmented Looking Glass with example code in an un-closable dialog box, as shown in Figure 3.1. This dialog box provides instructions for the task at the top. It also instructs the programmer to try to use the following example code to solve the task. Each task had a different example related to each specific task. Finally, a button below the code example allows users to execute the code. If a user clicks this button, they can watch the animation to see how the code works.

The example always showed a red outline around the important concept for emphasis, such as the ‘Do together’ code block in Figure 3.1. We chose to provide this emphasis because previous work indicated that this red highlighting could assist novice programmers in identifying the important part of example code [89]. We did not provide textual explanations because we wanted novices to focus on the example code and try to figure out how it worked to solve the

task. Furthermore, most examples that programmers find online are often not annotated. The emphasis merely gave direction without explanation and is something that a system could add automatically.

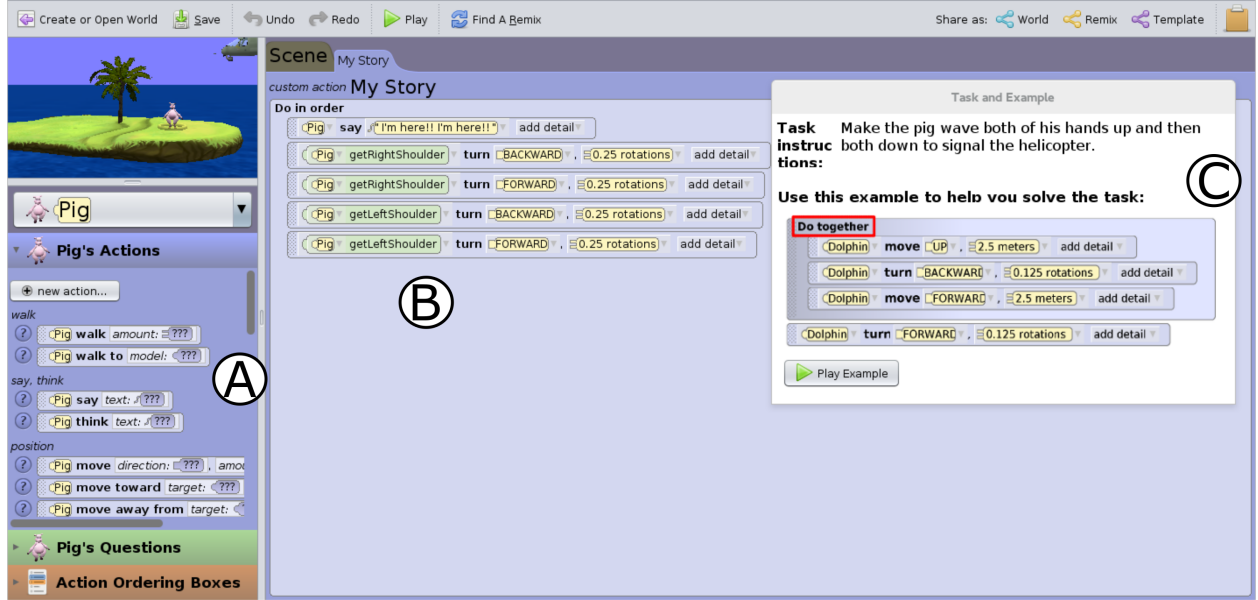


Figure 3.1: The augmented Looking Glass with a task and an example. A) Looking Glass programming environment. B) Task code designed by the researcher. C) Example code and task instructions dialog box.

Task programs

We created six task programs based on six concepts of varying difficulty. Each task required the pair of participants to correctly add a unique programming concept: simple parallel execution, a for loop, an unfamiliar API method, a function as a parameter, a while loop condition, or a for each loop iterator. The instructions for each task ask participants to add to or modify the given program to create a specific animation. The completion program was always a very simple program, only including basic programming statements similar to those participants had seen in the training task. The solution for each task required adding

the complex concept in the example code, such as simple parallel execution, as shown in Figure 3.1. The solution to the task in Figure 3.1 requires that the participants insert two parallel execution blocks: one to make both shoulders turn backward, and one to make both shoulders turn forward. The complete set of task programs and solutions is in section A.3

Examples

We created a code example for each program completion task to simulate a well-selected example found online. Each example contained the concept necessary to complete the associated task. The examples for each task are shown in section A.3. To prevent the tasks from being obvious, we used formative testing to ensure that the example did not directly map to the solution. For example, in Figure 3.1, participants needed to add two *Do together* blocks and rearrange the statements, while the example only shows one block. The example *Do together* also has one character doing multiple actions, while the solution requires the user to make two arms move. Formative testing showed that novices think about these ideas differently, which made the task more challenging.

3.2.2 Study design

Participants completed a demographic and computing history survey (see section A.1), a training task, and six program completion tasks, as shown in Figure 3.2. We allowed participants to ask questions at any point during the study. If users asked questions during the program completion tasks, we directed them to try to use the example. This study took a total of 90 minutes. If participants finished early, they were allowed to work on optional extra tasks which we did not analyze, or create their own program.

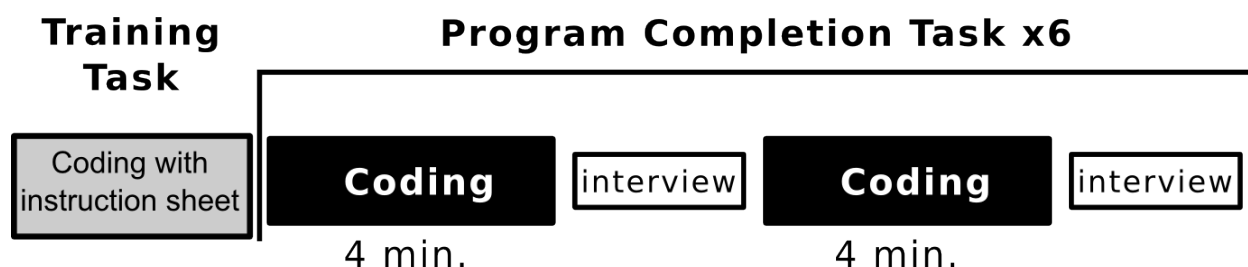


Figure 3.2: Study protocol

Training task

Pairs first completed a training task that was designed to familiarize them with the Looking Glass programming environment and the task format. They received an instruction sheet with directions and images that showed where to find essential elements in the interface (see section A.2). Pairs then started the program completion tasks.

Program completion tasks

Participants all worked on each of the six program completion tasks. Pairs saw the tasks in one of six orders that were balanced across participants to balance the effects of the tasks on each other.

In these tasks, participants worked on completing a program, given instructions and an example (Figure 3.1). For each task, participants had a total of eight minutes to work on the task, split into two four-minute halves. We chose the number of tasks and task times based on formative and pilot studies.

After the first 4 minutes of the task, a researcher asked the participants questions as part of a mid-task interview. In this interview, the researcher asked the participants questions about what they had tried so far, what they planned to do next, and why they had or had not used the example. The purpose of this interview was to: 1) prompt participants to discuss

their thought process, and 2) to encourage participants to use the example, if they had not yet used it. Because this study aimed to understand difficulties using examples, we wanted participants to try to use the example at the halfway point if they had ignored it previously.

At the end of the mid-task interview, pairs had another 4 minutes to complete the task. We encouraged participants to keep trying if they told us they completed the task but it was not correct. This likely increased success rates, but gave us more valuable information about their process trying to solve the tasks.

Once the task was complete or the eight minutes had passed, the researcher performed a final interview for that task. The goal of this interview was to ask questions to elicit any processes that the participants had not talked about yet and to gauge how well they understood the concepts in the example. This interview asked questions about: 1) how they figured it out, 2) what they would have done next if they had not figured it out, and 3) how the example worked.

3.2.3 Participants

We recruited 21 children aged 10-15 with minimal programming experience from the St. Louis Academy of Science mailing list. We screened participants to ensure that they had three or fewer hours of programming experience. Three children had programmed for more than three hours, so they participated in another concurrently running study instead. Our 18 participants had an average age of 11.4 ($SD = 1.4$). We had ten female participants and eight male participants.

For each session, we randomly assigned participants to pairs, such that in the end, we had nine pairs of participants. Participants worked in pairs because formative work showed that children were not actively ‘thinking out loud’ on their own, even when instructed to do so.

Working in pairs prompted most participants to have natural and continuous conversations about the tasks. Having the participants work in pairs did change the dynamics of the situation and likely improved overall performance. Both the difficulties and the strategies participants used help us to understand more about novice example use.

3.2.4 Analysis and results

We collected demographic and computing history survey data, logs from the programming environment during the sessions, audio logs, and task programs. In order to better understand how novices used examples, we analyzed the correctness of the programs and the statements novices made throughout the tasks.

Program correctness

We scored each task as either correct or incorrect based on the instruction criteria given to participants. In four cases, tasks did not fit one of the criteria, but they used the correct concept fully and correctly, so we also marked those as correct. For example, one criteria was to not add extra code blocks into a loop task to ensure that they used the loop instead of repeated code blocks. If the resulting code used the loop correctly but they had extra code statements added elsewhere, we still counted solutions as correct. Some participants added extra code at the end of programs for fun before notifying the researcher they had completed the task.

Out of 54 total tasks, participants correctly completed 37 tasks (69%) and failed to complete 17 tasks (31%). On average, it took participants 5.27 minutes ($SD = 2.24$ min.) to complete a task. This average includes those who spent the whole 8 minutes and did not finish the task. Task times ranged from 1.63 to 8 minutes. In one of the 54 tasks, the timer did not

stop the participants at the 8 minute mark, but from the logs we can determine what they accomplished within the 8 minutes and only analyzed that period of the task.

Transcription labeling

To analyze the audio recordings, we transcribed them, created two sets of labels to categorize the focus area and processes, and determined the ‘realization point’ for each task.

We transcribed a total of 7.6 hours of audio from the program completion tasks in order to analyze what participants talked about as they completed the tasks. We then broke the transcriptions up into segments in which participants focused on a single topic, such as a question and an answer. We then grouped 4% of the statements based on qualitative similarities and labeled those groups. We iterated on these labels four times to clarify the labels. To verify the quality of the labels, an independent researcher labeled 20% of the statements. The researchers achieved >80% agreement on the 20% of statements when labeling the transcripts independently. I labeled the remaining transcriptions.

We created two high-level sets of labels to categorize how participants spent their time during programming tasks with examples. We wanted to know 1) which part of the interface or task the participants were focusing on, the ‘focus area’, and 2) what they were doing or talking about within that context, the ‘process’. Table 3.1 shows these sets of labels, which we used to label the statements from the transcriptions.

We chose to categorize statements by their focus area in order to gain insight into the typical places novices had difficulties in these types of tasks. Participants’ statements roughly cover all of the elements of the task, as shown in the top section of Table 3.4.

Our process labels aim to explain the types of statements novices made while trying to solve the tasks with examples. Our labels describe participants as making three main types of statements: descriptions, ideas, and evaluations. In descriptions, the participants are either reading something off of the screen or summarizing something they are looking at. When participants make idea statements, they are describing something they think they should do or are planning to do. Finally, evaluation statements tell us whether the participants believe that their code is working. A small number of statements did not clearly fit into these categories or were off-topic.

Realization point

We define the ‘realization point’ as the point in the transcription when one of the participants first mentions the necessary concept in the example. For each task, we either find one statement as the realization point, or find no realization point. One possible limitation of the realization point is that participants may have thought about the concept before they said it out loud. The natural flow of conversation between most pairs of participants makes it likely that that participants talked about their realization right away.

Through our analysis of the audio transcriptions, we discovered that all but two tasks had a definitive point when the participants first noticed which part of the example to use. We believe this is a valuable feature of example use, and call the point when a participant first talks about the critical element of the example the ‘realization point’. We believe that identifying realization points and looking at behavior before and after the realization points is a new way of analyzing the behavior of programmers working with examples. The realization point separates the task into two parts: 1) the time before participants know what concept to use, and 2) the time the participants spend trying to figure out how to apply that information to the task code.

Across all tasks, pairs spent an average of 1.9 minutes ($SD = 1.5$ min.) before the realization point, ranging from 0.2 minutes to 7 minutes. For only 2 of the 54 tasks, participants never reached a realization point, so we exclude these task times from the averages for both before and after realization times. After the realization point, pairs spent 3.4 minutes on average ($SD = 1.9$ min.), with times ranging between 0.3 and 7.5 minutes. Notice that participants spent longer after the realization point than before (3.4 vs. 1.9 min.), which suggests that using the concepts from the example was more challenging than identifying them.

Table 3.1: Labels

Focus Area Labels	Description	% of statements
Instructions	Talking about or reference the task instructions.	15%
Programming Environment	Talking about a part of the programming environment without mention of the task or example code.	9%
Example Code	Reading or talking about the example code, specifically referring to objects or parameters used in the example.	27%
Example or Task Execution	Focusing on executing either the task program or the example code, as differentiated by task logs.	10%
Task Code	Reading or talking about the task code, specifically referring to objects or parameters used in the task.	26%
Off-topic	Not talking about the task.	5%
Unknown/Other		8%
Process Labels	Description	% of statements
Description	Reading, paraphrasing or explaining part of the focus area.	28%
Description-realization	Describing something when they make a realization or describing their realization. This is often signaled by an “Oh!”- like statement.	2%
Description-don’t understand	Describing something and making an explicit statement about not understanding how something works.	7%
Idea	Talking about an idea for something to complete the task. It may be abstract, concrete, or not even explicitly stated. Ideas can also be negative, such as telling their partner not to do a certain thing. (*This does not include actions like “play the example.”)	32%
Idea- realization	Talking about an idea about what to do next in which they seem to suddenly understand what needs to happen. This is often signaled by an “Oh!”-like statement.	2%
Idea- don’t know how	Talking about an idea about something to do next to solve the task, but they do not know how to carry out the idea.	3%
Evaluation-working	Declaring that their program is correct.	4%
Evaluation- possibly working	Declaring that that their program might be working.	1%
Evaluation- not working	Declaring that their program does not work.	6%
Unknown/other		15%

3.3 Hurdles and strategies

Because behavior before and after the realization point differs, we first describe two hurdles that occur before the realization point, followed by those after the realization point. Then, we describe three strategies. We call these ‘hurdles’ because many pairs overcame the challenges on their own.

3.3.1 Content distraction hurdle

Often, participants spent time at the beginning of a task exploring the task code and programming environment or generating ideas from those contexts. For instance, in a few tasks, participants wanted to move a UFO to the ground. Even though the instructions told them they could not use numerical values to accomplish this, a few pairs wanted to explore the different numerical values to see how they worked. In another task, a participant wanted to explore a parameter called *as seen by* after the pair talked about not having any ideas about how to complete the task. In this task, the participants needed to insert a function. Participants first wanted to explore what *as seen by* does: “Wait, can you, wait click *as seen by*, just out of curiosity, a little more. Just try one of those things: begin gently, begin gently and... do you know?” We can also see this hurdle through the transcription labels, where tasks have multiple *task code-idea* and *programming environment-idea* labels before participants looked at the example.

3.3.2 Example comprehension hurdle

In some cases, participants’ confusion about the example prevented them from using it or being able to generate ideas based on it. After having the researcher suggest that they use

the example during the mid-task interview, one pair had the following conversation: “Play example. I don’t get how that’s supposed to help us. Yeah, I have no idea.” In this case, the participants did not understand how the example was related to their task, so they did not even consider using it to prompt ideas. In other cases, participants did not understand what was happening in the example. One participant who described an example where a ghost moves toward a treasure chest until the two objects overlap. In this quote, the participant was reading part of the example code: “Ghost move toward treasure chest. Huh. That’s weird. Hmmm.” However, he did not read the next part of the code, which was the critical component. In these cases, the transcripts often have *example code* or *example execution* labels early in the task with a much later realization point.

3.3.3 Programming environment hurdle

After participants discovered which programming concept to use, they sometimes could not find it in the programming environment. For example, a pair of participants has this conversation about using the *repeat loop*: “then you do repeat two times. How? But it says that you can repeat. Where is the times thing? I don’t see that. Stop. Oh here, jump. We got that. I was just trying to find the...” At that point, the participants have been talking about the repeat loop for two minutes and it is time for the mid-task interview, so they tell the researcher about their problem finding the repeat: “so I was kind of confused because we can’t find the. [...] We can’t find how to do the repeat.”

In other cases, participants found what they wanted to use, but could not figure out how to select or move it to accomplish their goal. One such participant had a clear idea of what they wanted to do, but did not know how to accomplish it: “Well we take the collection and put it where the girl was so that it moves them all up at once. Okay, so how are we supposed to

do this?” These types of issues are commonly labeled *programming environment: don’t know how* and *programming environment: description-don’t understand*.

3.3.4 Code misconception hurdle

Sometimes participants had misconceptions about how their programs worked. In these cases, participants thought they knew what to do to complete the task, but that idea was actually incorrect. One participant incorrectly thought that changing the ordering of their code would make two things happen at the same time: “Maybe you put the right shoulder, maybe you switch those around. So put this one right there and that one right there. Why would we do that? Cause then it would go in sync.” However, their real problem was that they needed multiple parallel execution blocks. Sometimes, these misconceptions led participants to generate new ideas that helped them to succeed. However, misconceptions added to task time, as they required participants to debug the problem. In other cases, code misconception hurdles were followed by code comprehension hurdles, in which participants expected their code to do one thing, but it did another.

3.3.5 Code comprehension hurdle

Participants sometimes talked about not understanding how their code worked: “Why is he not on the ground,” “Let’s see how this works out. Why didn’t the rabbit move,” “What the heck happened with this jump,” and “What did we do? I thought he’d jump again.” In these questions, participants had an expectation of what would occur when they executed their code, but that expectation was not met. Responding to these questions lead to other hurdles, like context distraction, but also spurred strategies like idea generation and code-example comparison. Common labels for these types of problems are *task execution: description-don’t understand*.

3.3.6 Idea generation strategy

After the realization point, if participants did not have a plan for how to actually use the programming concept to solve the task, many still generated ideas based on the task code. We classify behaviors as part of this strategy if they are not based on the example code nor on a previous failed attempt. One participant asked their partner a slew of questions about what to do next “Do we have to put that up there or what? Do we move them in there or something? For it to work? Do we move this?” These questions refer to multiple different possible next actions, none of which the participant seems to base on any specific rationale. Another participant stated “Huh. I have no idea what you’re supposed to do, but I’ll try something.” While this process can be haphazard, the willingness to keep trying often resulted in success. The *task code: idea* and *execution* labels often accompany this strategy.

3.3.7 Code-example comparison strategy

Revisiting the example after the realization point while trying out ideas helped participants to complete the task. For example, a pair of participants were working on a task where they need to get a girl to walk a certain distance and then have a rabbit run away. Solving the task depended on them figuring out to use the expression ‘not overlapping’, but the not operator had to be added separately. They first get the ‘overlapping part’ and then return to the example and eventually figure out that they are missing the ‘not’: “Okay. Now, when I play it, she walks up, but the rabbit doesn’t run. Overlapping. Overlapping with... Play. It doesn’t do it. That’s weird. Not is true. But here it’s just is true ... That looks like the example. Yeah, but it’s got this whole red line around it, but it’s got this not thing.” After participants have worked with the task code for a little while, they are better able to identify meaningful differences between the example and task programs.

3.3.8 Example emphasis strategy

Some participants stated that the red outline helped them find the important part of the example, even though we did not provide any explanation of the outline (see Figure 3.1). When asked how they decided to use a certain concept, one participant stated, “we just saw the outline.” Another participant asked the researcher “where is the repeat? We saw it outlined.” We provided visual emphasis because we wanted participants to have a cue to help them move through the task, but we did not want to provide hints as to how the example actually worked.

3.4 Task behavior groups

Overall, this data contains a variety of task behavior profiles. Figure 3.3 shows a graph of the 54 tasks where the x-axis is the time before the realization point and the y-axis is the time after the realization point. We noticed that there are tasks that spent much more time than the average before and after the realization point, as well as tasks that were overall completed much more quickly than most. In this section, we wanted to explore what happened in these extreme cases. To do this, we selected 5 tasks (approximately 10% of the data) that performed best and worst before and after the realization point:

- Long conclusion: the 10 tasks where pairs spent the longest time after the realization point (5 correct, 5 incorrect)
- Slow start: the 10 tasks where pairs spent the longest time before the realization point (5 correct, 5 incorrect)
- Quick: the 5 tasks correctly completed the quickest

- No realization: the 2 tasks where participants never reached a realization point

For each of the groups, we describe their behaviors, hurdles, and strategies based on the transcriptions. Figure 3.4 shows a set of relevant transcription labels for this discussion and the average count of each label.

3.4.1 Long conclusion group

Since participants, on average, spent more of their task time after the realization point, we wanted to know what caused long conclusions, shown in the top grouping of Figure 3.3.

Correct long conclusion

Tasks in this group were slowed down by the number of ideas participants had, as well as participants' incorrect expectations of the code. Likely, participants successfully completed these tasks because they continued to generate ideas, and because they revisited the example. While participants in other groups spent time talking about not understanding why the task code executed a certain way, participants in this group revisited the example to try to figure out how their code and the example differed. Figure 3.1 also shows that this group had the most programming environment ideas, but not many statements where the participants talked about not understanding or not knowing how to find a code block. This likely means that they just needed to try a few ideas before finding what they needed. Behaviors after the realization point included two main hurdles: code misconception and programming environment, but participants used the code-example comparison strategy and the idea generation strategy.

Incorrect long conclusion

The tasks in the incorrect long conclusion group seem to have been the most slowed down by the programming environment (see Figure 3.1). This means that after the realization point, participants spent time trying to find code blocks or struggling with system mechanics. However, participants still used the idea generation and code-example comparison strategies, during which they thought of ideas from the task code and executed the code to see if the ideas worked. Unfortunately, participants in this group were the most confused about how their code worked, which likely meant that they generated many incorrect ideas. Overall, these tasks had similar hurdles and strategies to tasks completed correctly: programming environment and misconception hurdles and the idea generation strategy. These tasks, though, also suffered from the code comprehension hurdle.

3.4.2 Slow start group

In this section, we discuss both the correct and incorrect tasks during which participants spent the most time before the realization point (the middle group in Figure 3.3).

Correct slow start

Participants spent a long time before the realization point on these tasks primarily due to the distraction hurdle and because they did not always fully understand the task instructions. In these tasks, participants did not appear to look at the example before they created a plan based on the task code or programming environment. Accordingly, the first time participants have an example code focus label is not until near the realization point. The study context may have also contributed to the extended time before realization for some correct slow start tasks. In order to control what programs participants worked on for the study, we had to

provide participants with tasks and instructions, which not all participants may have been motivated by or understood immediately. Transcriptions for these tasks show that correct slow start tasks had on average one *instruction-description don't understand* label in their transcripts, which was the highest of all of the groups (see Figure 3.4).

Incorrect slow start

Interestingly, as shown in Figure 3.3, the incorrect slow start tasks have similar times before the realization point to the correct tasks. This means that both groups of tasks had similar amounts of time after the realization point to complete the task, so lack of time did not contribute to the incorrect end state. On incorrect slow start tasks, participants had the context distraction hurdle, but these tasks seem to have a different pattern than those completed correctly. Task transcriptions in this group contain the *example code* or *example execution* labels near the beginning of the task, but participants do not return to it again until the researcher reminds them during the mid-task interview, possibly caused by example comprehension hurdles.

3.4.3 No realization group

Participants working on tasks in the ‘no realization group’, shown on the bottom right of Figure 3.3, do not reach a realization likely because they do not discuss the example code even though they execute the example (see Figure 3.1). This likely means that they do not know how it would be useful. Consequently, the example comprehension hurdle will be especially important to resolve, as it can prevent participants from even realizing what concept to use. Unexpectedly, however, participants working on these tasks do not use the idea generation strategy, shown by the small number of *task code: idea* labels. Most likely,

participants during these tasks were overwhelmed, which is supported by the fact that both of these tasks were first in the series of six for the two pairs of participants.

3.4.4 Quick group

Participants who completed a task extremely quickly primarily described the instructions and task and generated ideas from the example early, rather than getting distracted. After the realization point, transcripts from these tasks have zero or one *programming environment* labels, which means that the participants did not have many conversations or questions about where to find code blocks. However, some participants in the quick group did use the idea generation strategy: “at first we tried putting them all in the do together box and then we tried putting two out and then one out and then put another box and put them in it.” Since these were often simple tasks, participants could guess about different configurations and still complete the tasks quickly.

Participants in this group also used the outline and code-example comparison strategies. They discover the correct concept to use almost immediately, mainly by finding it in the example. However, participants may have also noticed these concepts in previous tasks. In the quick group, on average, the tasks had 1.6 *example code* labels and one *example execution* label. Two of the five tasks in this group contained the code-example comparison strategy when the participants did not necessarily grasp the concept well enough to complete the task directly. In the other three tasks, participants did not need more information to correctly complete the tasks, or quickly generated several ideas, which happened to work.

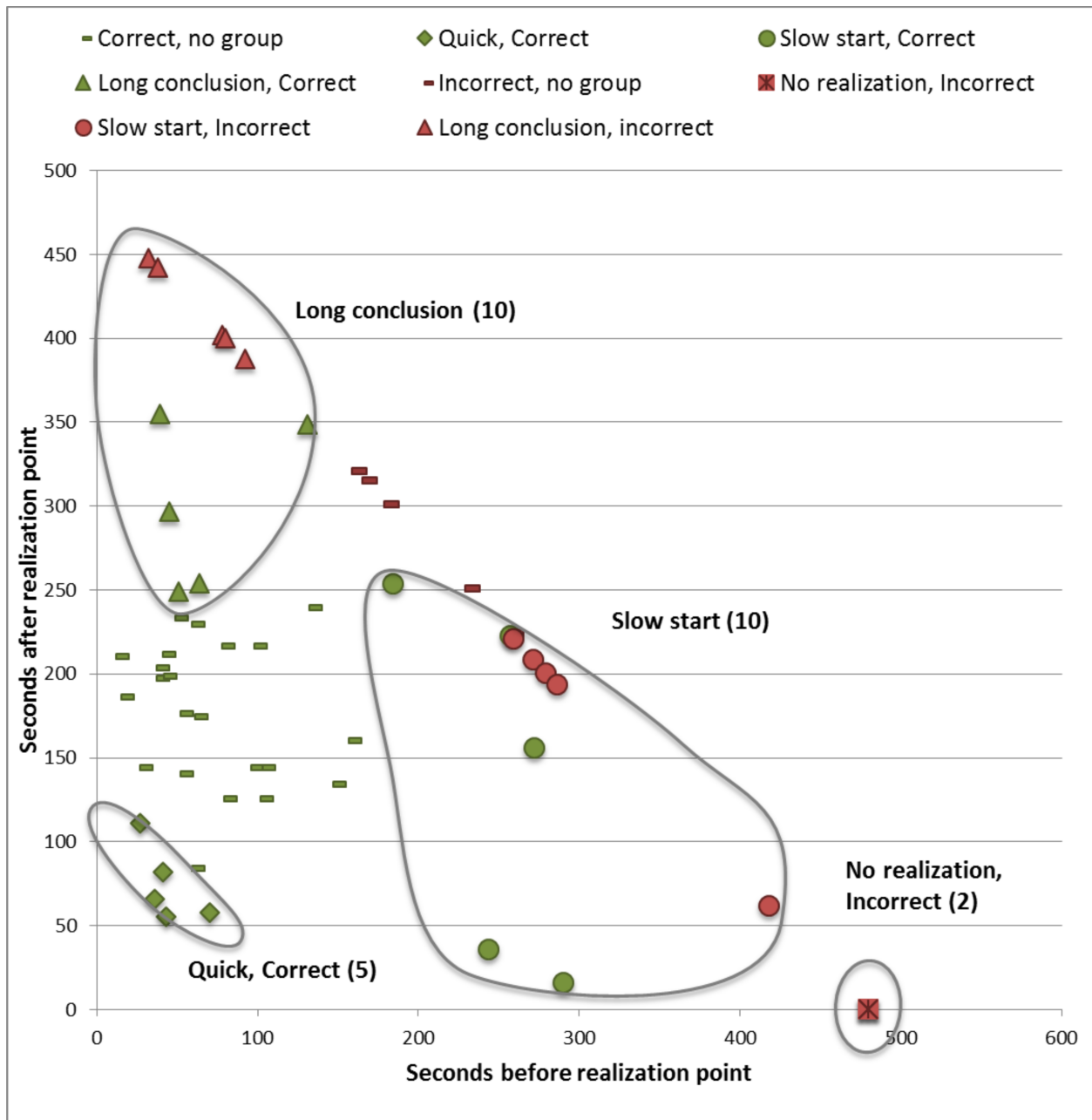


Figure 3.3: Time before realization point vs. time after realization point, with correctness and behavior group annotated with color and shape.

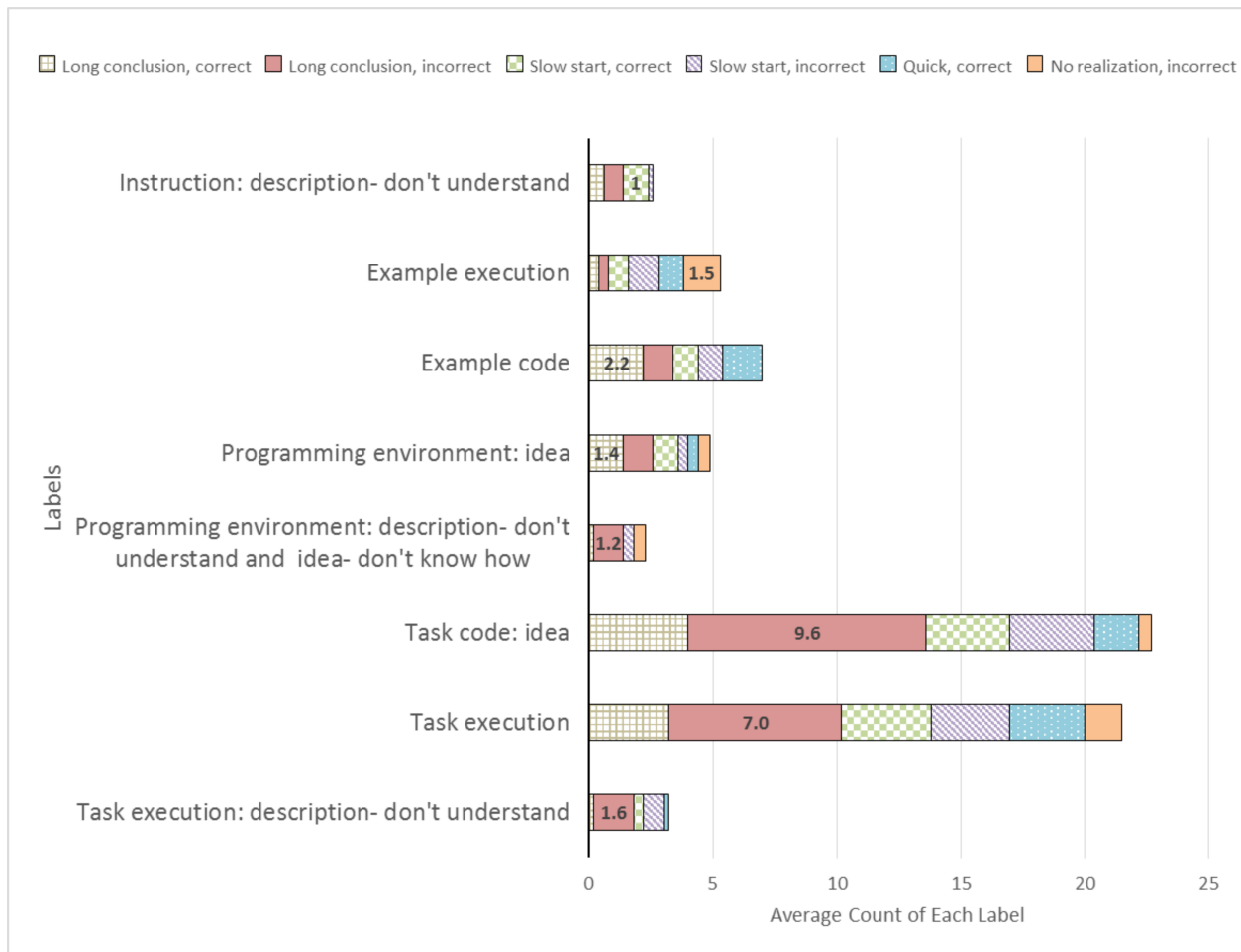


Figure 3.4: Important labels and the average count for each of the behavior groups. The largest value is shown for each label.

3.5 Threats to validity

Our threats to validity for this study primarily revolve around the number and types of participants. In this study, pairs of children worked on tasks using examples. Having pairs of children complete the tasks may have changed the participants' work-flow or concentration. We also had a relatively small sample size: nine pairs of children. We recruited participants through a science-focused local mailing list. The children who chose to attend our study may be more interested in programming than the typical child. Their parents also may be more invested in their education than the typical child.

3.6 Discussion

In this study, we explored how novice programmers use examples to complete programming tasks. Specifically, we looked at the case where a novice programmer is highly unfamiliar with their own code, as well as the example. The combination of many new concepts can create an overwhelming experience. Yet, this situation likely encompasses the experiences of many end-user and novice programmers when they begin and look to examples as a way to try to accomplish their goals.

A key result of this work is that the time spent before or after the realization point can indicate the types of problems participants likely experienced. In slow start tasks, participants' focus on the task and programming environment before addressing the example. In the long conclusion group, participants noticed the key to the example early, but still struggled to solve the task. These groupings suggest ways to design support for novice programmers using examples in general. We also used them in the design of the Example Guru.

3.6.1 Implications of slow start behavior

When participants had slow starts, it was often because of the example comprehension and context distraction hurdles.

Participants sometimes took a long time to reach the realization point because they were executing the example code more than reading the example code. The majority of the support provided for understanding examples accompanies the example code, but this might indicate that we should consider ways to augment the example execution. For example, this type of support could be more along the lines of a debugger than a textual annotation. Furthermore, some participants did not understand how the example related to their own code, which prevented them from trying harder to understand it.

Since participants were new to both the programming environment and task, spending time becoming familiar with those aspects of the task can be valuable. Thus, we do not always want to force novices past the context distraction hurdle. However, especially in educational contexts, we may want to nudge novice programmers to return to the example once they feel comfortable with the code and environment.

3.6.2 Implications of long conclusion behavior

Interestingly, many participants had quite a bit of trouble completing tasks even after the realization point. Our analysis of participant behavior starts to explain why participants still struggled after the realization point: programming environment, code misconception and comprehension hurdles.

The programming environment hurdle is specific to visual programming environments, where programmers may not be able to find a code block. However, this issue it is not necessarily

specific to the first 90 minutes of programming. Even if a novice programmer has become familiar with the programming environment, they still might not know where to find a code block that they have not used before. One way to improve examples to help novices would be to augment examples with assistance to find code blocks in the interface.

For the code misconception and code comprehension hurdles, we may be able to help novices by encouraging more revisiting of the example and by helping them to make a plan from the example. While some participants revisited the example while they were working on using the programming concept to complete the task, this was rare, yet helpful. Instead, many participants either used the idea generation strategy or ‘debugged into existence’, based on their misconceptions and code comprehension hurdles [184]. The participants who tried a few ideas and then returned to the example to see how their code was different seemed to be more effective in generating ideas that succeeded. However, the long conclusion pattern likely occurs because at the realization point, participants are not familiar enough with the task to generate a complete plan to solve the task. This means that just augmenting an example with a lot more information would probably cause novices to be even more overwhelmed when they first look at it. Instead, we would want to encourage participants to return to the example and provide support that they can request when they need it.

3.6.3 Implications for the design of the Example Guru

Ideally, the Example Guru’s design should help programmers overcome as many critical hurdles as possible. In application of these findings, we focused on the two hurdles and two strategies that directly relate to issues with example use: the example comprehension hurdle, the programming environment hurdle, the code-example comparison strategy, and the example emphasis strategy. We do not focus on the content distraction hurdle, code

misconception hurdle, code comprehension hurdle, or idea generation strategy because they were either related to these specific tasks or programming in general, rather than the example.

Example comprehension hurdle

Because the example comprehension hurdle can prevent novices from reaching the realization point or paying attention to the example, we wanted the design of the Example Guru to address it. The Example Guru has three features that attempt to help novices better comprehend the examples: 1) the titles of the suggestions and examples describe the code *output* rather than the code, 2) two examples demonstrate the concept or contrasting code, and 3) rules that match suggestions to novice code based on the types of objects (in the semi-automatic version). The examples have titles describing the code output because this explains to the programmer what the code will do if they add it to their program. This can help them to evaluate whether a suggestion is useful to them and seems more accessible than a description of how the code works abstractly. To demonstrate the meaning of the abstract concepts, each suggestion has two examples. The examples show different ways of using the code or contrasting examples. These two examples encourage novices to explain to themselves how they are similar or different. Finally, in the final version of the Example Guru, the suggestions match examples to novice code that uses similar types of objects. By connecting the suggestions to the code's objects, the suggestions contain examples with much more surface similarity to the novice code. This should reduce some of the cognitive load associated with new content.

Programming environment hurdle

After novices reach the realization point, they can still get stuck attempting to solve the problem if they cannot find the critical code blocks in the programming environment. This

could happen frequently for examples in blocks programming environments, as examples are typically images rather than editable or copyable code. If a novice cannot find it, they sometimes give up and try other incorrect strategies. To resolve this issue in the design of the Example Guru, we had two different support methods: a button called *show me how* that provides an overlay showing where to find the code block, or a tool tip revealed by hovering over the code. The tool tip provides instructions on where to find the code block. Ideally, the tool tip will encourage novices to learn where the blocks are, rather than relying on a button that will give them the answer.

Code-example comparison strategy

Because novices benefited so much from comparing their code to the example, we wanted the Example Guru to encourage this strategy. For this reason, suggestions appear in two locations: as annotations to the code and in a list. The annotations provide a direct connection between the novice's code and the relevant suggestion, hopefully helping novices to make that link. Only the most recent suggestion appears as an annotation to reduce overwhelming the interface. Suggestions are always available in the list, which enables novices to return to them easily for further comparison later.

Example emphasis strategy

In this study, examples had red outlines around the critical code elements. Since novices found these helpful in solving the tasks, we wanted the Example Guru to also support emphasizing code. For API method suggestions, the Example Guru emphasizes the code by having the example only have one code block. API methods in Looking Glass do not typically require multiple lines of code to demonstrate, so having only one line of code ensures that extra irrelevant code does not distract novices. For complex and abstract code examples, the

critical code block is emphasized with a 3D shadow. This shadow reduces any confusion about whether the red outline will actually appear on the code itself.

3.7 Conclusion

This study begins to show the challenges novices have using examples and provide directions for supporting novices in using examples generally and specifically for the Example Guru. One major remaining issue is that when novice programmers look at an example, they do not realize which elements relate to their task. This prevents them from spending more time trying to understand the example. Without an understanding of why the example could help them, novices spend time trying to figure out how the code works on their own, often with little success. While emphasis and code comprehension support can begin to help programmers with this issue, this study raises a question about what misconceptions participants had in finding the important elements of examples. This question prompted the next chapter, which explores the differences in how novices and experts study and recall code snippets.

Chapter 4

Towards Better Code Snippets: Exploring How Code Snippet Recall Differs with Programming Experience

Note: Portions of this chapter were published in Visual Languages and Human-Centric Computing 2017 [92].

Our exploratory study of novices using examples indicated that not realizing the important part of an example can prevent novices from being able to use an example to solve a task. This chapter explores the ways novices and experts pay attention to aspects of code by exploring how they recall code snippets. Like the previous chapter, this study supports hypothesis 1, looking at how novices interact with examples in order to better support novices in using examples. Systems should especially support novices in identifying important elements because otherwise, novices will likely not return to the example for help or spend time trying to better understand it. Current systems instead commonly focus on supporting

access and integration of examples, rather than supporting example perception [22, 38, 159, 188]. A better understanding of how novices focus on examples could enable us to engineer the presentation of examples such that novices notice critical elements of examples more often and more quickly.

Novices may have trouble identifying important elements of code snippets because their lack of knowledge forces them to process each element of the code individually [67]. This likely makes it difficult to determine the weight of each element. In contrast, experts can automatically *chunk* multiple elements and remember them as one unit. This enables experts to process and recall more elements using the same working memory limitations as novices. Experts can chunk because they have schema, which are long-term memory knowledge structures that help them to organize new information [39, 50, 67]. Schema likely also help experts identify essential elements of content because they can quickly identify structures that align with their knowledge.

Research has confirmed that expert programmers can remember and organize code better than novices [2, 141, 176]. However, work has not addressed how the specific types and order of elements that novices and experts recall can inform code example design. Furthermore, the majority of these studies occurred before blocks programming languages became popular and thus do not consider how syntax affects recall. Research has compared how students perceive text and blocks-based programming languages [228]. We are unaware of work comparing what novices and experts focus on when recalling blocks compared to text. Understanding the differences in how experts and novices recall code snippets could provide insight into how to help novices focus their attention more effectively on critical elements of code snippets, in both text and block contexts.

We ran an exploratory study comparing how everyday, occasional, and non-programmers recall snippets of code. To explore differences between text and block code, participants studied and recalled two text and two block snippets. To investigate the order of element recall, participants had three attempts to study and recall each snippet. This work seeks to answer two questions:

- RQ1: Which code snippet elements do different levels of programmers initially recall?
- RQ2: What do different levels of programmers fill in after the first attempt?

This chapter has two main contributions: 1) the key similarities and differences in recall between everyday, occasional and non-programmers, and 2) recommendations for beginning to improve code snippet presentations. Everyday programmers focused primarily on structure and meaning, especially in the first attempt, and struggled the most with domain-specific code elements. Those with less experience had the most success recalling concrete and natural-language tokens close to the beginning of the code snippets.

4.1 Related work: novice and expert chunking in recall

This study has been inspired by past work in novice-expert recall and code comprehension.

Research has explored chunking through comparisons of novice and expert recall in a variety of fields. Early work on chunking began comparing novice and expert recall in chess. One famous study found that expert chess players could recall many more chess pieces than novices within a valid game configuration, supporting the theory that the experts can chunk common chess configurations using schema [39, 50]. Accordingly, experts and novices did not differ in their abilities to recall random configurations of chess pieces. Researchers have also

investigated the differences between novices and experts in other fields, such as categorizing physics problems [42], and programming [43, 64, 118, 191, 200].

Studies looking at schema and chunking in programming have often involved novice and expert programmers recalling code. Some of these studies essentially replicate the chess study for programming, finding that experts can recall more correctly structured code than random code [2, 141, 176, 198, 200]. Studies have also looked at how recall correlated with other skills, like comprehension and debugging [198, 220]. In addition to snippets and programs, researchers have explored how different levels of programmers attempted to recall reserved words. They found that novices recalled based on common memorization mnemonics, like natural language and story mnemonics. Novices also focused more on objects, while intermediates and experts used their programming knowledge, focusing on functionality [49, 141]. While prior work considered the differences between novices and experts in terms of knowledge and processing, our study design and analysis enable us to recommend improvements for example code.

4.2 Methods

We ran an exploratory study to answer two primary questions: 1) which code snippet elements do different levels of programmers initially recall, and 2) what do different levels of programmers fill in after the first attempt? We also wanted to know if participants recalled text and block code differently and whether having a problem to solve along with the code snippets would affect what participants recalled.

4.2.1 Participants

We recruited participants through Amazon’s Mechanical Turk (MTurk), an online crowdsourcing platform [4]. We sought a diverse population of participants with a variety of

Table 4.1: Participants’ programming experience

Group	How often	HS	Col- lege	On- line	Infor- mally	Top 5 Languages
Everyday	Everyday	11%	22%	6%	61%	c:13, java:10, javascript:11, python:8, sql:8,
Occasional	Sometimes and Past	14%	10%	0%	76%	c:11, java:10, javascript:7, python:4, sql:4, html:2

programming experience levels. To increase the chances of receiving reliable results, we required that participants live in English-speaking countries and have completed 100 tasks. After receiving three incorrectly completed tasks, which we excluded, we increased the requirement to 1000 tasks.

We recruited three populations based on self-reporting: 1) 21 participants with no programming experience (non-programmers), 2) 21 participants who program once in a while or used to program in the past (occasional programmers), and 3) 18 participants who program on an everyday basis (everyday programmers). The majority of both everyday and occasional programmers learned programming informally (61% and 76% respectively), while others learned online, in college or in high school. We had 24 female, 35 male, and 1 unspecified gender participants. Participants ranged in age from 22 to 50 ($M = 33.3$, $SD = 7.3$).

4.2.2 Materials

We wanted to explore what participants would recall from block and text code snippets across multiple attempts. We iteratively created four snippets of code through pilot testing. Figure 4.1 shows block and text versions of the *for each* loop code snippet (see section B.3 for all tasks). In the design of the code snippets, we had 3 priorities: 1) make the snippets challenging for all programmers to recall, 2) have distinct control flow programming constructs to reduce, and 3) make the blocks and text as comparable as possible.

In order to understand the order of participants' focus, the snippets needed to be complex enough that even experts would require multiple attempts to recall the code. In total, each snippet had between 8 and 11 lines of code. Each code snippet contained one control flow construct, like a loop, surrounded by API method calls. We designed the snippets to limit learning effects for specific constructs that novices may not initially know. Each code snippet included one of four distinct programming control flow constructs: 1) a *while* loop, 2) an *if-else* block, 3) a *for* loop iterating three times, and 4) a *for* loop iterating through a list of objects.

Because we wanted to compare recall for blocks and text, we created each of the examples in Java and in blocks. To reduce the differences between the text and blocks other than the presentation, we created code snippets in Looking Glass, which is written in Java.

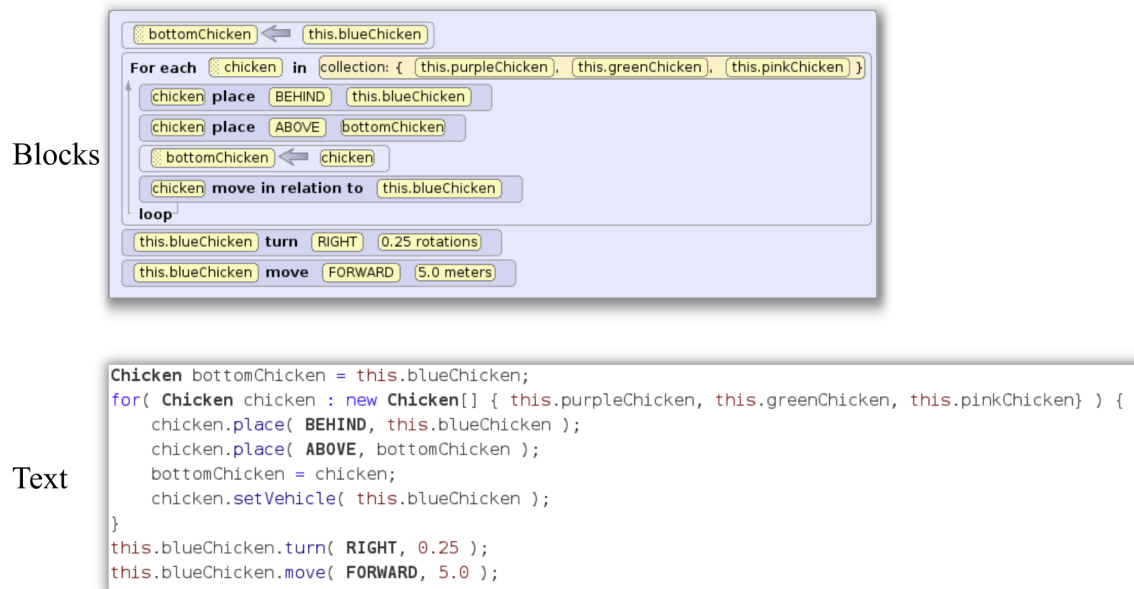


Figure 4.1: An example of block and text versions for the same code snippet.

4.2.3 Study setup

Our study had: 1) an introduction phase, and 2) a study and recall phase, in which participants had three chances to memorize and recall four code snippets.

Introduction and surveys

The first part of the study included an introduction, sample tasks, and a demographic survey that asked participants about their age, gender, and programming experience. To introduce participants to the mechanics of the study, participants first stepped through instructions and completed two sample recall tasks: one for text and one for blocks (see section B.1).

In order to determine participants' programming experience level, participants filled out a survey that asked how often they program, how they learned to program, and which programming languages they know (see section B.2). We used how often participants program as our measure of expertise, assuming that those who program every day will have more developed schema for programming. We grouped participants who program once in a while and in the past because prior work has suggested that past programmers often forget many details [94]. General programming experience does not indicate how familiar participants were with specific programming constructs. After each task, we asked participants to report their familiarity with the construct in the code snippet.

Study and recall

Participants completed four study and recall tasks. In each of these tasks, participants saw a snippet of code (like one of the two shown in Figure 4.1) and then attempted to recall it. Participants had three chances to memorize and recall each code snippet. Participants had

90 seconds in their first attempt and 30 seconds in the second and third attempts to study each code snippet. We did not want to limit participants' recall by their typing speed, so participants did not have time limits for recalling the snippets. We based the time limits on pilot testing in order to make the tasks difficult so that participants would not be able to recall the entire code snippet.

In order to explore the differences in recall for text and blocks code, two of the consecutive tasks showed code in Java, while the other two tasks showed code in blocks. We randomized and balanced whether participants saw the two blocks tasks first or the two Java tasks first. During the block tasks, participants' typed text appeared on a canvas that they could draw blocks on to represent the structure of the code. We designed this canvas for block recall through pilot testing. After each pair of block and text code snippets, participants answered questions about their cognitive load for those tasks, using the validated difficulty and mental effort scales [160].

Participants also received problems along with two tasks and answered growth mindset survey questions (see section B.4). Due to concerns of validity, we do not analyze these factors. We hypothesized that having problems to solve along with code snippets would focus experienced participants on recalling the constructs in the problems. To explore this, participants saw a related problem for two tasks. Unfortunately, participants' feedback indicated that the problems added too much work to the tasks, so we believe they may not have consistently paid attention to the problems. Based on participants' feedback during pilot testing, we also hypothesized that novices may have different perceptions of their ability to gain programming skills for block and text snippets [194]. To measure this, participants answered the growth mindset survey after each pair of blocks or text tasks. The survey was unreliable in this case (Cronbach's $\alpha = 0.5$) due to one question that inversely correlated with related questions. Because these elements of the study were unreliable, we do not report these results.

4.3 Analysis

In order to interpret the data, we needed to compare the participants' responses with the ground truth code snippets. This section first describes how we matched response lines to code snippet lines and found the differences. It then discusses the metrics used to analyze the data.

4.3.1 Metrics

In analyzing participants' responses, we wanted to explore patterns in the differences between what programmer groups recalled. To do this, we initially created visualizations of the code snippets, with size representing differences between the groups and color representing attempt number. This process revealed three patterns that we wanted to test quantitatively: 1) the first attempt seemed distinct from the second and third attempts, 2) some token types had much larger differences than others, 3) there were trends based on the line number. The incorrectly recalled tokens also seemed to provide insight into participants' thought processes. To explore these patterns, we analyzed differences in: 1) attempt number, 2) correct and incorrect token types, 3) position of tokens recalled, and 4) survey data.

Attempt number

We allowed participants three chances to recall each of four snippets. Participants recalled very few tokens in the second and third attempts individually, so we combine the last two attempts to answer what participants filled in after the first attempt. We discuss what occurred in the second and third attempts together,

Table 4.2: Token Types

Group	Token Type	Example of token type
con- trol flow	constructs	if, for, while
	keywords	new, final
	variable types & identifiers	Integer, index
	conditionals	isCollidingWith, true
	operators	!, not, ++
objects	scope	this
	subject	blueChicken, purpleChicken
	accessors	getRightHip, getLeftHip
separators	separators	;, (,) {, }
actions	action identifiers	move, turn, say
arguments	numerical literals	5, 0
	string literals	`Uh oh`
	enum literals	ABOVE, FORWARD
	function literals	getDistanceTo
	unit literals (only in blocks)	meters, rotations

Correct and incorrect token types

To explore whether participants focused on and recalled different elements of the code snippets, we analyzed the token types shown in Table 4.2 for the attempts, blocks and text, and errors. We based the token types on compiler types, but split and grouped them based on the meanings of the tokens and their structure within the code snippets for this study.

Since the tokens and amounts of tokens differed between blocks and text snippets, we compared how participant groups recalled the token types across blocks and text. Due to issues participants had drawing the block structures on the canvas, we only analyze the text participants typed. To explore the amounts of errors participants made for each token type, we manually labeled the token types for each error. We labeled errors based on why the tokens were incorrect, as shown in Table 4.7.

Table 4.3: Average % of tokens recalled by everyday programmers and differences between groups for 1st and 3rd attempts.

	Token Type	Every-day 1st at-tempt	%Every1 - %Occas.1			Every 2/3 at-tempt		Every 3rd at-tempt		%Every3 - %Occas.3		
			%Every1 - %Occas.1	%Every1 - %Non1	%Occas.1 - %Non1					%Every3 - %Occas.3	%Every3 - %Non3	%Occas.3 - %Non3
	All	64%	12%	26%	14%	15%	79%	8%	24%	16%		
	String	74%	16%	25%	9%	15%	89%	12%	13%	1%		
Structural	Key-words	71%	12%	30%	12%	15%	89%	9%	12%	4%		
	Construct	61%	5%	34%	29%	27%	88%	11%	37%	26%		
	Scope	61%	16%	24%	9%	21%	82%	14%	22%	7%		
	Object	52%	15%	23%	8%	24%	78%	18%	26%	8%		
	Separators	52%	21%	28%	7%	25%	77%	22%	31%	9%		
	Actions	46%	13%	22%	9%	29%	75%	15%	27%	12%		
	Variables	50%	15%	23%	8%	23%	73%	25%	30%	15%		
Meaning	Operators	56%	30%	35%	5%	20%	76%	28%	33%	5%		
	Conditional	36%	19%	23%	4%	26%	62%	14%	19%	6%		
	Enum	39%	18%	15%	-4%	27%	66%	11%	18%	8%		
	Numeric	34%	21%	21%	0%	31%	65%	24%	30%	6%		
	Unit	21%	16%	16%	0%	24%	46%	17%	27%	13%		
API	Accessors	24%	2%	5%	3%	42%	66%	13%	24%	11%		
	Functions	13%	8%	9%	1%	40%	53%	31%	37%	6%		

Position

Based on exploration of the data, we noticed that participants recalled more tokens at the beginning of the code snippets. We wanted to quantify this trend and compare it across groups. We analyzed the correlation between the line number and the percentages of tokens recalled in that line, using Spearman’s R. If participants focused on the beginning of an example, they have a negative correlation. If they focused on the end of the example, they have a positive correlation.

4.3.2 Comparing responses to correct code snippets

In order to compare what programmers recalled, we needed to know which lines of participants' responses matched up correctly with the code snippets in the tasks. We tried a variety of text and code comparison techniques (e.g. Google's 'diff-match-patch' [53]), but they did not work due to many missing, swapped, and out of order tokens. We designed an algorithm that matched lines based on the alpha-numeric tokens in the lines and only minimally factored in missing tokens. This worked better than metrics based on edit distance because those metrics overestimate the error for incomplete lines, while ours does not. If less than 50% of the response line did not match a code snippet line, we did not match or analyze it. We also hand-coded erroneous tokens and found that a small number were due to mismatched lines. Of the 4495 total lines of code recalled, our algorithm did not match 3% of lines and incorrectly matched 1% of lines, which we did not analyze. For the correctly matched lines, we used Python's DiffLib library [52] to compare the response lines to the code snippet lines, resulting in the identification of correct and incorrect tokens for each line.

4.4 Results

Programmers focused on recalling different token types and positions initially and after the first attempt. The elements of code snippets that programmers recalled first indicate what they thought was the easiest to remember and what they deemed most important, likely based on their schema. The elements programmers filled in later indicate what they thought was secondarily important and/or harder to recall. This section reports: A) the overall differences in recall, difficulty, and mental effort, B) participants' recall in the first attempt, and C) participants' recall after the first attempt.

4.4.1 Overall

As expected, everyday programmers recalled larger overall percentages of the code snippets and found the tasks easier. Everyday programmers recalled higher percentages of overall tokens than occasional and non-programmers, and occasional programmers also recalled more tokens than non-programmers (see Table 4.3). Participants recalled the majority of these tokens in the first attempt. We report the percent of token types added by everyday programmers in the second and third attempts alone, as well as in the end, in Table 4.3, to provide a flavor for the second and third attempts.

Everyday programmers found the tasks significantly less difficult ($p < .05$) and everyday programmers required less effort than programmers with less experience ($p < .05$). Occasional programmers also found the tasks significantly less difficult than non-programmers ($p < .05$), but they did not need significantly less mental effort than non-programmers. Occasional programmers may have also slightly improved over the four tasks, as their recall percentages had a marginally significant correlation with the task number ($r = 0.21, p = 0.06$).

4.4.2 Which elements do programmers initially recall?

Programming experience aided in early recall of: 1) tokens that make up the structure, and 2) tokens that fill in the meaning. All programmers initially recalled: 3) natural language tokens and 4) tokens near the beginning of the code snippet.

Structural tokens

Everyday and occasional programmers primarily focused on core structural tokens in the first recall attempt (see Table 4.3-Structural). These tokens set up the overarching control

flow, objects, and the actions objects complete, but do not include the specifics of the control flow or actions. Core structural tokens include: construct tokens (e.g., `for each in`), related keywords (e.g., `collection`), separators (e.g., `{, }, ;`), objects (e.g., `this.bluechicken`) and actions (e.g., `turn`). For the most part, both everyday and occasional programmers recalled the highest percentages of core tokens. Occasional programmers also recalled higher percentages of them than non-programmers. This suggests that both everyday and occasional programmers likely use schema to chunk critical structural components, but that occasional programmers do so somewhat less successfully. Specifically, occasional programmers may need more assistance with separator tokens. Non-programmers do not have knowledge to support recall of structural tokens and fall furthest behind in recalling construct tokens.

Occasional programmers differed from everyday programmers in correctly recalling separator tokens more than any other core structural token. Occasional programmers had almost twice as many separator errors as everyday and non-programmers (Table 4.5). The large number of errors resulted from issues such as recalling extra separators for block snippets and swapping separators such as `{` and `(`. Having fewer separators may help less experienced programmers remember them. Both occasional and non-programmers recalled higher percentages of correct separator tokens for block snippets than text snippets (see Table 4.4). Separators can be critical for indicating structure and scope, so it may be especially important to consider the role of separators in code snippets.

Occasional programmers recalled only 5% fewer construct tokens, such as `for` and `while`, than everyday programmers, while non-programmers recalled 34% fewer construct tokens than everyday programmers (Table 4.3). Occasional programmers were often familiar with these tokens, reporting prior familiarity with the constructs in 56% of tasks. Knowledge of constructs likely helped the most when the construct names aligned with familiar languages, such as the text versions: `for(int i=0...)`. Both everyday and occasional programmers

Table 4.4: Comparing blocks and text

Token	Everyday		Occasional		Non	
	Text	Blocks	Text	Blocks	Text	Blocks
Separators	ns	ns	42%	63%	35%	52%
Construct	93%	82%	85%	67%	ns	ns
Conditional	ns	ns	30%	53%, $p = .1$	17%	54%
† Unit tokens were only present in block snippets						
† All other tokens were non-significant $p > .1$						

recalled more correct construct tokens for text than block code (see Table 4.4). For block code, some constructs have been modified to clarify meaning, such as **repeat 3 times**. Non-programmers seemed to find constructs especially difficult to focus on or recall for both types of code snippets. While many informal occasional programmers likely have some exposure to constructs, non-programmers who do not know how the constructs work will not realize their importance to the code snippets.

Meaning detail tokens

Everyday programmers recalled many of the specific details that completed the construct and action statements in their first recall attempt. However, occasional programmers often recalled similar percentages as non-programmers (see Table 4.3-Meaning). The variables, operators, and conditionals specify details, such as the iteration in a loop like `Integer index = 0; index<3; index++` or the condition for a while loop `! this.woodenBoat.isCollidingWith(this.iceberg)`. The enums, numbers, and units specify the details of actions, such as in `move(FORWARD, 5.0 meters)`, in which `FORWARD` is the enum. These tokens require a deeper understanding of the code snippet functionality in order to be memorable. Many of these tokens are similar, making them somewhat difficult to recall correctly through direct memorization. Everyday programmers can likely recall these more easily using schema,

enabling them to reconstruct the snippet based on the meaning, rather than recall the tokens individually.

Errors and syntax provide more information about occasional and non-programmer recall patterns for these tokens. Occasional and non-programmers had high error rates on token types in this group, such as operators and units (Table 4.5). They often recalled meaning tokens in the wrong line or order, likely caused by memorization of specific tokens rather than understanding of the overall animation. Non-programmers also recalled more conditional tokens for block snippets than for text, possibly due to the natural language conditionals in block snippets. For example, text snippets use `isCollidingWith`, while the block snippet uses `is near`. This aligns with the recall rates for string tokens, as discussed next.

Table 4.5: % of errors for each token and total errors and attempts made by each programmer group

		% Erroneous attempts, Total attempts					
Group	Token	Everyday		Occasional		Non	
Structural	String	4%,	354	5%,	359	4 %,	273
	Keyword	4%,	130	7%,	134	3 %,	110
	Construct	6%,	365	5%,	391	10 %,	249
	Scope	2%,	1383	1%,	1352	0 %,	1087
	Object	14%,	1597	24%,	1687	22 %,	1282
	Separators	4%,	5726	7%,	4554	4 %,	3259
	Action	9%,	968	16%,	975	18 %,	735
Meaning	Variable	9%,	509	7%,	440	9 %,	327
	Operator	15%,	453	23%,	295	13 %,	216
	Conditional	6%,	489	7%,	441	9 %,	329
	Enum	16%,	544	16%,	516	12 %,	402
	Number	7%,	1155	7%,	805	7 %,	631
	Unit	31%,	180	26%,	160	50 %,	114
API	Accessor	20%,	327	16%,	343	15 %,	270
	Function	40%,	110	55%,	77	59 %,	46

Natural language tokens

All of the programmer groups recalled surprisingly high percentages of string tokens. In this context, string tokens are the natural language arguments for **say** and **think** actions, like a penguin that says ``uh oh'' before crashing. We might expect occasional and non-programmers to recall these with ease, due to their simplicity, natural language, and uniqueness amongst the other code tokens. However, we did not expect everyday programmers to focus on string tokens, as strings are often even less critical to the overall animation than meaning details like **FORWARD 5.0 meters**. Due to the storytelling nature of these code snippets, string tokens provide contextual information related to the overall animation, possibly making them easier to recall for everyday programmers as well. The position of string tokens may have also been a factor.

Beginning of the code snippet

In the first attempt, participants recalled tokens closer to the beginning of the code snippet more than lines further down, with a few exceptions. All participants had significant negative correlations between the line number and the percentage of token types recalled (see 1st attempt columns in Table 4.6). This indicates a relationship between the line number and recall. Everyday programmers often had weak correlations ($< .3$), while occasional and non-programmers had more moderate correlations ($.3 < r < .5$). Thus, occasional and non-programmers had stronger relationships between the position of tokens and recall.

Some token types had no correlations, either due to occurring only at the beginning, like keywords, or low recall rates. Only occasional programmers had more numbers and accessors, such as **getRightHip**, in the first attempt. This may indicate that occasional programmers focused on less important tokens in the first attempt due to their location.

4.4.3 What did programmers fill in after the first attempt?

Programmers filled in two main token groups after the first recall attempt: 1) obscure API methods, and 2) objects, and 3) did so in a variety of positions.

Obscure API methods

Everyday programmers recalled 42% of accessors, such as `getRightHip` and 40% of functions, such as `getDistanceTo` after the first attempt. Everyday programmers recalled more function tokens than occasional and non-programmers in the second and third attempts. By the end of the third attempt, everyday programmers recalled more accessors and functions than other groups (see Table 4.3-API). This is an increase from the first attempt, in which everyday programmers only recalled 24% of accessors and 13% of functions. Everyday programmers

Table 4.6: Correlation between token types and line number for first, second and third, and total. * $p < .05$, *** $p < .001$

Token	Everyday			Occasional			Non-programmer		
	1st	2+3	End	1st	2+3	End	1st	2+3	End
All	-.26***	.12*	ns	-.37***	ns	-.20***	-.37***	-.11	-.32***
Strings	-.55***	.41***	-.25*	-.67***	ns	-.55*	-.57***	ns	-.61***
Constants	-.19*	ns	ns	-.42***	ns	-.32***	-.32	ns	-.26***
Separators	-.25***	.15***	ns	-.33***	ns	-.19***	-.32***	-.10*	-.30***
Actions	-.21***	.21***	ns	-.28***	ns	-.23***	-.32***	ns	-.27***
Scope	-.25***	.18***	ns	-.27*	.09*	-.19***	-.29***	ns	-.29***
Subjects	-.24***	.14*	-.11*	-.25***	.09***	-.17***	-.25***	ns	-.23***
Variables	-.32***	.21*	ns	-.42***	ns	-.25***	-.30***	-.19*	-.38***
Operators	-.26*	.19*	ns	-.33***	ns	-.30***	-.29***	ns	-.32***
Condition-als	-.37*	ns	ns	-.48*	-.51***	-.68***	-.44*	-.36*	-.57***
Numbers	ns	ns	ns	-.13	.10*	ns	ns	ns	ns
Unit	-.23*	ns	-.32***	ns	-.18*	-.26*	ns	-.31***	-.30***
Accessors	ns	ns	ns	-.19*	ns	ns	ns	ns	ns

†Keywords, Enum, and Function had no significant correlations.

likely left functions and accessors for the later attempts because these tokens often contribute less to the core structure and meaning of the code than other tokens. Based on high error rates, these token types may have been especially challenging to remember (see Table 4.5-API). Everyday programmers likely have schema for how these tokens work, but without API familiarity, trying to recall API methods with at least three words was likely difficult.

Object tokens

Occasional programmers particularly focused on object tokens, such as `bluechicken`, attempting to recall nearly 100 more object tokens than everyday programmers, as shown in Table 4.5. Occasional programmers also had positive correlations between recall and line number for scope and object in the second and third attempt (see Table 4.6). This implies that occasional programmers began to focus on these tokens throughout the example more after the first attempt. However, occasional programmers also had a high error rate for subjects, with 24% of attempts incorrect.

Some possible explanations for the attention to objects are: 1) the storytelling nature of the code snippets, 2) the natural language and concreteness of objects, and 3) the repetition of object token names on multiple lines. We have anecdotally noticed in studies of novices coding animations that novices often focus on the objects. This is likely in part due to the focus in this type of code on the story, which revolves around the objects. The objects complete the visible actions and are easily associable with the real world. These tokens also often reoccurred through the code, making their names easier to focus on and remember. However, without a strong schema for the meaning of the code, easily recalled tokens can often end up in the wrong location, causing errors.

Focus and lack of focus on beginning of snippet

In the second and third attempts, everyday programmers often had positive correlations between the line number and the percentage of tokens recalled, indicating that they filled in the end of the snippet. By the end of the third attempt, everyday programmers no longer had correlations between the line numbers and percents recalled for most tokens. Occasional and non-programmers often maintained the significant moderate negative correlations, as shown in Table 4.6.

Occasional, and especially non-programmers, sometimes continued to focus their attention at the beginning of the snippet even after the first attempt. The lack of difference for keyword tokens after the third attempt demonstrates this continued focus on the first few lines of code. Keyword tokens encompass abstract programming terms such as `constant`, `collection`, and `final`, which we might expect to have similar recall patterns to constructs. However, keyword tokens were only located in the first few lines of code. Occasional and non-programmers' focus on the beginning of the snippets enabled them to recall similar percentages of keywords as everyday programmers. This demonstrates how strong of an impact the position of tokens can have on less experienced programmers' focus and recall.

4.4.4 Errors

For all of the rest of the prior results, we focused on only the tokens in participants' responses that were exactly correct when compared to the code snippets provided. However, the frequency, token type, and type of errors can also tell us about the differences in what programmers are noticing and focusing on in examples. Table 4.5 shows the token types, the percentage of the total errors that the token types account for, and the percentages of each token type error that were committed by the different levels of programming experience.

For the most part, everyday programmer errors were primarily for argument token types, but with also some for variables, accessors and scope. However, for the most part, everyday programmers had errors with the types of tokens they recalled last, like accessors and functions, as well as other token types that occur at the ends of lines of code and that are more specific to the animation code snippets in this study. Because everyday programmers mostly have solid schema for things like order, spacing and separators, their main errors were in misplacing tokens, having extra tokens and having similar but not quite right tokens, as shown in Table 4.7.

The places where occasional programmers had the most errors, however, were primarily for objects and separators, but also had many errors relating to the core structure of the code, including many control flow token types and actions. Interestingly, if the erroneous objects were added to the correct numbers of object tokens recalled, occasional programmers actually attempted to recall more operator tokens than everyday or non-programmers. This is the only token type where occasional programmers at least attempted to recall more tokens than everyday programmers. In terms of their error types, they had high rates of extra and missing separators, which is likely connected to the combination of blocks and text code snippets that had differing types of separators. They also likely have only weak schema for these, so they make assumptions about what they should be, but don't know it as well as everyday programmers. Occasional programmers as had many similar/partial tokens, extra, and misplaced.

Interestingly, the only token type where non-programmers had the highest percentage of tokens was for construct control flow tokens, like 'repeat' and 'while'. In terms of error type, non-programmers had high percentages of extra and missing spaces, as well as spelling errors. The extra and missing space issues may be related to the mix of block and text programming

Table 4.7: Error Type

Group	Type	Everyday	Occasional	Non
Semantic:	misplaced	32%	19.9%	21.1%
	order	4.9%	5.4%	4.6%
	wrong	6.0%	7.7%	6.4%
	extra	17%	16.1%	9.3%
Syntactic:	extra/missing space	5%	6.2%	16.6%
	extra/missing separators	2.2%	11.1%	6.6%
	spelling	12.5%	10.9%	22.7%
	similar & partial	18.4%	20%	9.3%
	placeholder	.6%	1.5%	0%

languages and their different spacing, while the high rate of misspelling may be because non-programmers do not realize how critical spelling is in coding.

4.5 Threats to validity

Since we wanted to derive possible directions for further exploration and design, we chose to risk a possible higher false positive rate, rather than a higher false negative rate in our statistical analysis. While we did not correct for multiple comparisons, the small number of comparisons and the fact that we chose the comparisons in advance does reduce our risk of a high false positive rate.

4.6 Lessons learned: recommendations for improving code examples

This study explored how different experience levels affected how programmers studied and recalled code. Based on the results, presentation of snippets should consider three aspects:

1) the code in the example itself, 2) general features to emphasize and deemphasize, and 3) specific features of the example to emphasize or deemphasize.

4.6.1 Selecting or creating effective code for examples

When creating or selecting example code, our results suggest that three strategies might help programmers to focus on important elements: 1) Purposely position elements within the example, 2) minimize similar identifier names, and 3) use natural language appropriately.

4.6.2 Purposely position elements within example

All levels of programmers often focused, at least to some extent, on the beginning of the example more than the rest. There were often significant negative correlations between the percent of tokens recalled for a line and the line number. While many of the correlations disappeared for everyday programmers by the end of the third attempt, this still seems to indicate that regardless of programming experience, having critical elements of code snippets at the beginning is important. This will likely have an even bigger effect on occasional and non-programmers, who maintain moderate correlations throughout. There are a few possible ways to possibly address this: 1) place critical elements of code near the beginning of the example or emphasize them if they are elsewhere, 2) deemphasize less important, but necessary lines of code near the beginning of the example to focus the programmer's attention at the critical element, and 3) keep code example snippets short.

Minimize similar identifier names

For the token types that had repeated identifiers and similar identifiers (objects and actions), such as `bluechicken` and `greenchicken`, participants often recalled them in the wrong

locations. While these errors likely indicate that the programmers do not fully understand the code, similar identifier names add extra difficulty. When specifically designing code examples in educational resources or documentation, reducing similarities between identifiers will likely make the snippet easier to parse. For code snippets in the wild, systems can likely automatically rename tokens more uniquely. For example, if two identifiers contain common substrings, they could be renamed to reduce overlap.

Use natural language appropriately

Novices recalled natural language tokens similarly to more experienced programmers. Natural language within code snippets can be very powerful, as novices will be able to use it most effectively for understanding. However, the attention natural language draws can also be detrimental, causing programmers to focus on unimportant elements.

4.6.3 General emphasis and deemphasis

The results indicate that we may be able to apply some emphasis and deemphasis strategies generally to help programmers focus their attention: emphasizing structure and meaning, and deemphasizing non-critical syntax.

Emphasize structure and meaning

While everyday and occasional programmers recalled more structural tokens than non-programmers, non-programmers recalled especially low percentages of construct tokens, which are especially critical to the structure. Occasional programmers had similar difficulty to non-programmers recalling tokens that fill in the meaning of the snippet, likely due to lack of schema. Helping less experienced programmers discover, focus on, and understand these

critical elements of code snippets will be essential in enabling non-experts to effectively use example code.

Deemphasize non-critical syntax

Separators were a problematic token, especially for less experienced programmers. Programmers often had trouble remembering the correct separators or their correct location. However, occasional and non-programmers recalled higher percentages of separators for block snippets. Reducing the focus on separators unrelated to the critical elements of the example might help everyday and non-programmers notice and remember the important separators.

4.6.4 Example-specific emphasis and deemphasis

Depending on the specific example, example code may benefit from specific decisions to emphasize certain important elements or tokens that stray from the norm, and deemphasize unimportant early elements that may be distracting.

4.6.5 Emphasize important arguments

Everyday programmers generally paid less attention to tokens like argument values and accessors in the first recall attempt, as these were not part of the core structure. The control flow structure is typically critical in code snippets, but cases will exist where the code snippet actually aims to highlight one of these token types that does not fit into a typical schema. In these cases, everyday programmers will likely pay less attention to those components, so the code snippet would need to draw extra attention to them.

4.6.6 Emphasize tokens that stray from the norm

Programmers with experience had more difficulty recalling block construct tokens, like `repeat 3 times` that replace typical loop syntax. The programmers likely still understood the meaning of those tokens due to their schema. Yet, the slight difference from familiar programming languages made the exact syntax challenging. In code snippets where the programming language or API differs from typical practices, snippets should help the programmer to notice and remember the difference.

4.6.7 Deemphasize unimportant early code elements

Occasional and non-programmers often recalled unimportant argument values in the first few lines of code, rather than paying attention to structural elements of the rest of the example. Some critical elements of code will have a long list of arguments or tokens that are not all as important as the method name or construct name. Clearly showing which elements of the first few lines of code are important and not important will likely reduce the amount of information novice programmers need to try to comprehend at first glance.

4.7 Implications for the Design of the Example Guru

The lessons learned make generalized recommendations for systems using examples. As an example of how to apply these, we used these recommendations to inform the design of the final version of the Example Guru in two main ways: 1) defining the example selection process, 2) emphasizing code with multiple examples and 3D shadows.

4.7.1 Selecting effective examples

The results of this study suggest heuristics for selecting examples, either by hand or automatically. Our heuristics for the Example Guru, based on this study, include optimizing the: position of the important element, size of the example, number of unimportant elements, and names of objects. In the Example Guru, we only selected code examples where the critical code element was the first code block and the only abstract concept. This makes the important part of the example first, which was one of the most important predictors of recall in this study. We only included examples with one of these abstract concepts, to prevent examples from becoming long and overwhelming. We also discouraged selection of examples that had many unimportant parameters. Early novices often use fewer of these extra parameters, so having them in the examples often draws attention away from the critical element. Finally, objects in the examples should be as clearly labeled as possible. This makes it easier to identify the objects when watching the animation, reducing extra cognitive load to figure out how the code relates to the animation. These four heuristics resulted in examples that were, for the most part, simple and clear examples of concepts.

4.7.2 Emphasizing elements

Programmers' focus on early and unimportant elements indicates the need for more support to help programmers find the critical element in an example. The Example Guru uses two methods to emphasize the critical element: showing multiple examples with the same concept and 3D shadows.

For programming concepts, the Example Guru suggestions each have two examples. Both of the examples show the programming concept being suggested. We aimed to show two examples where the code accomplishes similar goals using the suggested programming concept,

but in two different contexts. This ideally will help novices to notice the use of the same programming concept.

The Example Guru also places 3D shadows around the critical code element for multi-block examples to make it appear closer to the user. Due to the visual nature of Looking Glass code, emphasizing code blocks by changing the appearance of the code block, like with color, could confuse novice programmers or users unfamiliar with Looking Glass. The 3D shadow provides emphasis and prevents the code block from blending in with the background without changing the appearance of the code block itself.

4.8 Conclusion

This study explored what everyday, occasional, and non-programmers focused on and recalled for code snippets. Experienced programmers initially focused on structure and meaning, filling in other details later. Less experienced programmers lacked attention to key structural components, focused on natural language and object tokens, and skewed their recall toward the beginning of the example. Programmers ranging from experts to complete novices rely on code snippets to learn new programming skills and to attempt to accomplish programming tasks outside of formal education contexts. The findings from this study led to recommendations for selection and presentation of examples generally as well as specifically for the Example Guru. The recommendations from this and the previous chapter likely will help to make examples more effective in a suggestion system like the Example Guru. One question remains for the design of the Example Guru: what should the suggestions and rules look like?

Chapter 5

Exploring Suggestion and Rule Design through Expert Content Creation

Note: Portions of this chapter were published in Visual Languages and Human-Centric Computing 2013 [94].

The goal of this chapter was to explore how expert programmers would create suggestions and rules in order to guide the types of suggestions and rules the Example Guru would include. Many systems provide feedback to programmers working on specific tasks. We are not aware of existing systems that suggest new skills during artifact-based programming. In order to design these types of suggestions and determine when they should be triggered, we wanted to explore the types of suggestions and rules experts would create.

We had experienced programmers create the suggestions and rules because experienced programmers commonly provide feedback to programmers. They provide feedback individually and also create most of the resources available for learning new skills and technologies. In face-to-face interactions, teachers or mentors can give feedback to novices and experts peer

review each others' code. Experts or professionals also typically hand-author resources online like documentation and tutorials. Experienced programmers also on occasion formalize when to make certain types of suggestions to other programmers. For example, opportunities exist for programmers to author rules in other contexts, such as for style checking [166]. We believed these experiences would help us to explore the types of suggestions and rules a system should provide.

We ran an exploratory study of expert suggestion and rule authoring. We asked participants to suggest an improvement for a novice program and author a rule to identify more general patterns in code that indicate that their suggestion would be appropriate. This paper addresses the following questions:

- Do participants make suggestions that have the potential to teach a novice how to improve their program?
- What types of suggestions do experts make?
- What does the rule pseudocode tell us about ways to author rules?
- How well do rules generalize whether a program should receive the suggestion?

5.1 Methods

We designed a study to explore the suggestions expert programmers would make to novice programmers, how expert programmers would rule pseudocode, and whether the rules would find appropriate target programs.

5.1.1 Materials

We created example novice programs, pre-made suggestions, skill set diagrams, and the rule-authoring template used in this study to simulate the experience of an expert in a crowdsourced suggestion creation system.

Example novice programs

To gather a set of novice programs for experts to work with, we created “example novice programs.” To make these programs, we recreated Alice programs [3] and Looking Glass programs from a 2010 study in the current version of Looking Glass. We recreated the novice programs due to a lack of student programs compatible with the current system. The example novice programs also remove information that might identify the original authors. The example programs maintain the same structure and content as student programs, with changes mainly in the characters and props.

We selected the programs to represent a variety of skill levels and ensured that we could point to at least one potential suggestion for each program. Alice is a sister project of Looking Glass that has college-aged users with more advanced skills. In the 2010 Looking Glass study, participants had never programmed before.

Skill group diagrams

To help participants make skill-appropriate suggestions, we provided them with a skill group diagram like Figure 5.1. We based our hierarchy on groupings used by the Computer Science Teachers Association [56]. The diagrams indicated which skills the novice programmer likely

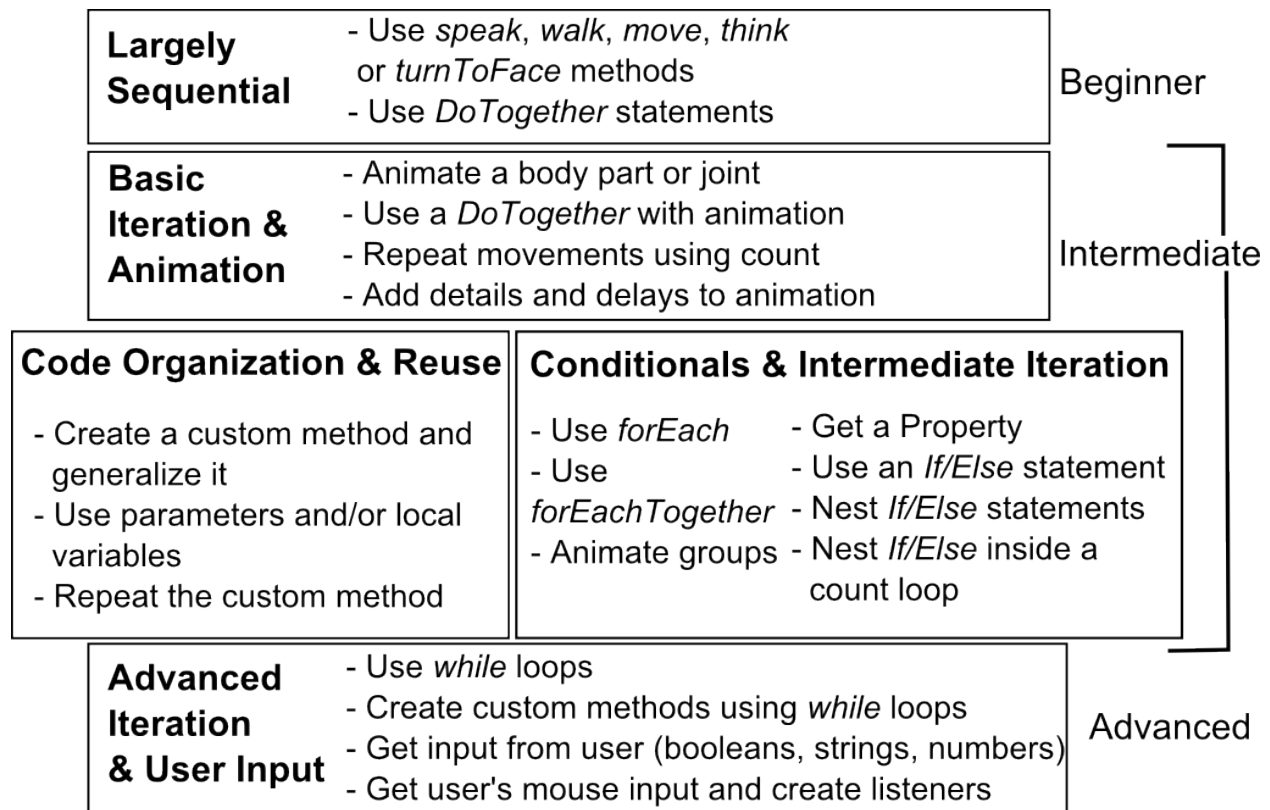


Figure 5.1: Skill group diagram

already knows, which might be appropriate to present next, and which might be too advanced. All of the diagrams are shown in section C.1.

Pre-made suggestions

In order to allow more time for experts to author rules, we created a pre-made suggestion for each novice program. Suggestions were one of two types: code-based or animation-based. An example pre-made suggestion for an animation change is “A more complex animation with body movements was added where Tami previously danced. She now moves her left leg and arms to be in a dancing position before she turns, instead of just spinning in a standing pose.” An example pre-made suggestion for a code change is “A list of characters all doing the same action was replaced with a *ForEach* loop with the array of characters.” To show a

participant a pre-made suggestion, we provided them with a sheet of paper that contained screenshots of the original program and the pre-made suggestion. The English descriptions of the suggestions were printed above the screenshots.

Rule-authoring template

We chose to use a Word document for pseudocode rule authoring to minimize the influence of the IDE on the participants' coding style choices. The document contained instructions asking participants to write the rule in a sentence and then in pseudocode (see section C.3). Each rule needs to return 'True' if the rule has found a target program and otherwise returns 'False.'

5.1.2 Study procedures

The study had three parts: an introduction to Looking Glass, one suggestion creation task, and four rule authoring tasks.

Looking Glass introduction

To enable participants to make a suggestion by editing a program, we introduced participants to the components of the Looking Glass IDE. First, we showed and executed a completed program created by a researcher as an example. We then asked participants to create a simple program to familiarize themselves with Looking Glass.

Suggestion creation

To investigate the types of suggestions domain experts would make for novice programs, we asked participants to create a suggestion for a novice program. To help experts make

suggestions at an appropriate level, we showed each participant an example program and a skill group diagram (Figure 5.1).

We then asked participants to make a suggestion by editing the program. We randomly assigned each participant a novice program to make a suggestion for. We instructed participants to edit the program to make an improvement, but left the type of improvement to their discretion.

We had each participant create only one suggestion due to the time consuming nature of thinking of and creating a code suggestion. Furthermore, the commonality of peer review suggests that expert programmers often give feedback, so this study emphasizes exploring the ability of experts to author rules.

Rule authoring

The rule authoring task explored whether experts could code scripts to check for when a novice program should generally receive a certain suggestion. We had participants author the rules in pseudocode to focus on the ideas of the suggestions, rather than the correctness of the code. We provided participants with a rule-authoring template, which is a Word document with instructions for the task and space for typing the rule. We asked participants to describe the rule in English by explaining what pattern of code should trigger the suggestion. We then asked participants to write the rule as a program in pseudocode that, when run on any novice program, would determine whether the novice program would benefit from the suggestion they created. Similar to the natural programming approach [162], we asked participants to use their own vocabulary. We did not provide any information about terms or structure that they should use.

We asked participants to author four rules: one rule for the suggestion they created and three rules based on pre-made suggestions designed by the experimenters. For the three pre-made suggestions, we showed the participant a novice program beside a pre-made suggestion and asked them to author a rule that finds target programs for the provided suggestion. Participants received the three rule authoring tasks for pre-made suggestions in a randomized order. We randomly assigned each participant an example program for each task such that each participant saw one example program from each of four skill groups in Figure 5.1. Due to a lack of student programs in the advanced skill group, we did not include advanced example programs in this study.

5.1.3 Participants

We had 21 participants, five of whom were female, ranging in age from 19 to 68. We recruited participants through the Academy of Science of St. Louis mailing list. Participants' programming backgrounds range from being self-taught to having a Ph.D in Computer Science. Most participants listed their occupation as software developer, software engineer or programmer. Fifteen participants described themselves as an expert in at least one programming language and all had experience programming.

5.2 Analysis

We analyzed the suggestions and rules in order to answer three questions: 1) what types of suggestions do programmers make, 2) what does the pseudocode tell us about designing support for rule authoring, and 3) what do the target programs tell us about the rules?

5.2.1 Suggestions

To explore the types of suggestions participants made, we used a grounded theory approach [65], which is an iterative process of labeling possibly important features of the data to develop categories and theories. We labeled suggestions based on the suggestion as a whole. This process resulted in hierarchies of categories for *Suggestion Type* and *Suggestion Novelty*, based on common labels and relationships between labels. The Suggestion Types group suggestions by the concept or idea the improvement presents. The Suggestion Novelty categories consider whether the suggestion utilizes new concepts. For categorization purposes, new concepts are programming constructs, structures, or methods that the example program does not contain.

This process resulted in hierarchies of categories for suggestion type and novelty, based on common labels and relationships between labels. After developing the categories using grounded theory, two researchers then individually selected categories for the 21 suggestions participants created. The categorization process had inter-rater reliabilities of 95% for Suggestion Types and 86% for Suggestion Novelty categories.

5.2.2 Rule pseudocode

We analyzed the rule pseudocode to understand whether experts could author rules and what support might help them to do so. If crowdsourcing is a viable way to produce rules at a large scale, researchers would likely need to develop support for making the rule-authoring process easier. The ways that experts author these rules tells us what kinds of support might help them.

The 21 participants wrote 72 rules, averaging 3.5 of 4 completed tasks, due to time. We analyzed pseudocode with a similar grounded theory approach as used for the suggestions.

For pseudocode, we labeled each line of code and categorized them individually. The rules contained 287 total lines, not counting lines that were braces or the required return statements. On average, rules contain four lines of code, with a standard deviation of 1.8. Consistent with the method for categorizing suggestions, two researchers independently categorized pseudocode lines, with a 94% inter-rater agreement rate. We will not discuss the 6% of disputed lines, as they fit into multiple categories or were ambiguous. The remaining 94% of lines fall into three overarching categories: iteration, comparison, and other.

5.2.3 Rule implementation

To evaluate whether rules are likely to work in practice, we implemented each of the rules participants authored for their own suggestion. We chose to implement these rules because they are based on suggestions participants made, rather than pre-made suggestions. Two pairs had identical implementations, so we tested the remaining 19 rules.

We tested each rule on 165 programs uploaded to the Looking Glass Community and a set of novice programs. We will refer to the Looking Glass Community programs as the ‘mixed group,’ since Looking Glass Lab members, ranging from college sophomores to a professor, contributed 92 of these programs. We also tested the rules on 55 programs created by middle school children, the ‘novice group,’ in an unrelated 2013 study. The novice group did not receive instruction on how to program.

5.3 Results

Our results answer our three questions: 1) what types of suggestions do programmers make, 2) what does the pseudocode tell us about designing support for rule authoring, and 3) what do the target programs tell us about the rules?

5.3.1 Suggestions

Our analysis of suggestions led to two high-level results about the types and novelty of expert-made suggestions.

Suggestion type

The data labeling and categorization process resulted in two major types of suggestions: code changes and animation changes, as shown in Figure 5.2. In a code change, the participant modified the style of the program’s code, like adding a variable when a value is used multiple times for the same purpose. In animation changes, participants modified the animation output of the program, like creating a new method *fallDown* where the character flails and falls realistically to replace a method that makes a character turn backward without bending any joints. Of 21 participants, 16 improved the code, 4 modified the animation, and 1 changed both the code and the animation. At the most detailed level, our analysis resulted in eight suggestion types with no more than 20% of suggestions in any one category.

Two common code change types are creating a new method and restructuring repeated code, as shown in Figure 5.2. One new method suggestion was for a program where a man pushes another man into the ocean. The suggestion extracted the animation to a custom method, which made the action more easily reusable. In another example program, a set of kids each turned to face the camera sequentially. The suggestion, which restructured repeated code, replaced the list of repeated statements with a *ForEach* loop, improving the code style and introducing or reinforcing *ForEach* loops. These suggestions are likely similar to ones experts would make in code review, as they make code more maintainable and easier to understand. However, in the context of children learning to program through animations, these suggestion seem unlikely to motivate novices.

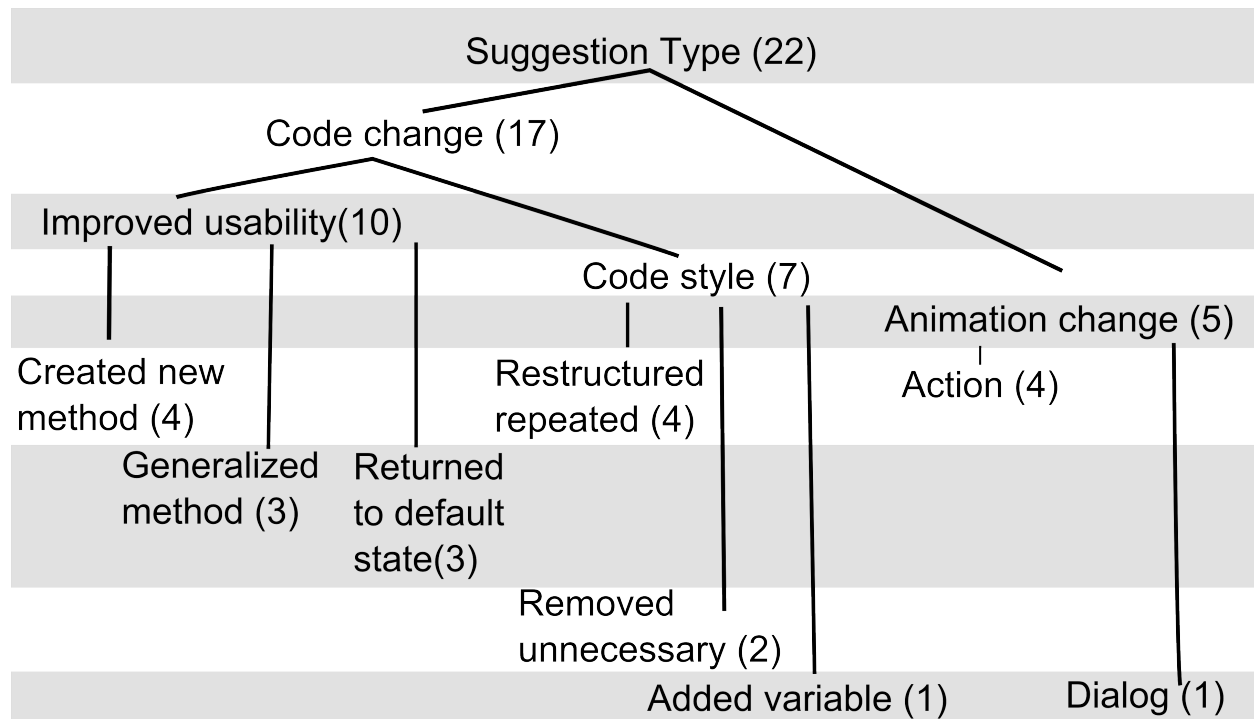


Figure 5.2: Suggestion type category hierarchy.

Six participants improved the usability of code by generalizing a method or returning the animation to a default state. Participants generalized a method by adding parameters or making a method accessible to a class of objects, rather than a single character. Returning code to a default state involved bringing a character to a position where the animation could continue or was more easily reusable. For instance, one example program had a dancer jump into the air and the suggestion returned the dancer to the ground.

Although we encouraged both code and animation changes, most participants made code changes likely for several reasons: lack of familiarity with Looking Glass, fear of changing a child’s creation, and difficulty generalizing animation changes. Since participants only had a short introduction to Looking Glass, changing programming constructs or restructuring code was easier than creating more complex animations. Several participants commented that they did not want to change the animation because they were unsure of the original intentions.

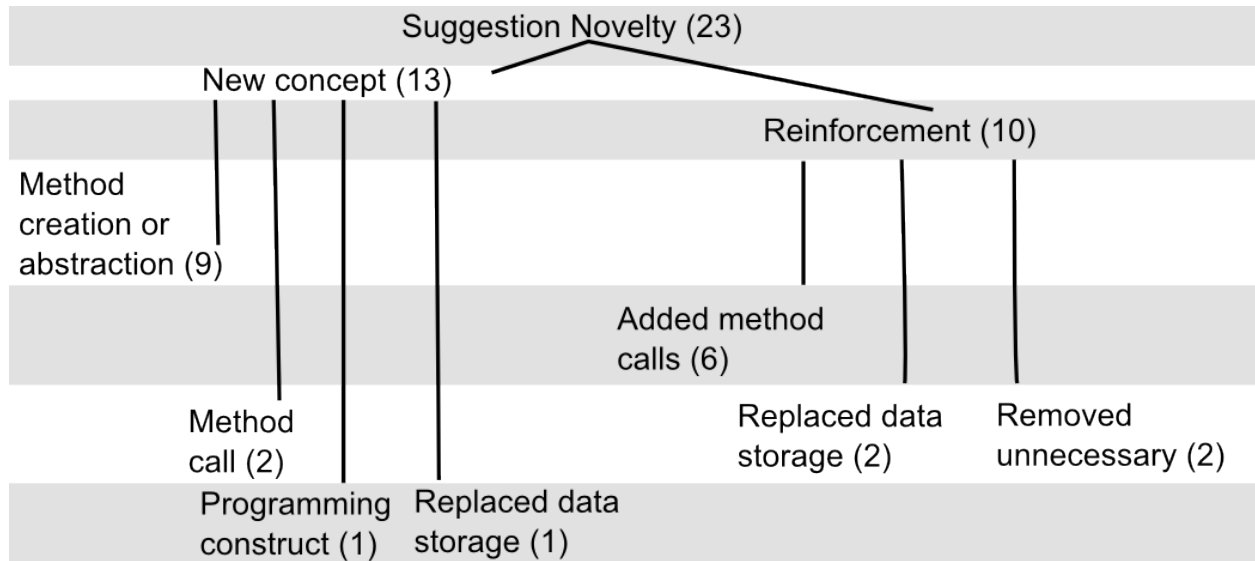


Figure 5.3: Suggestion novelty category hierarchy.

Other participants considered changing the animations in a program, but stated that they did not believe it would apply to other programs. These results suggest that experts will be more likely to improve novice code than animations.

Suggestion novelty

Thirteen of the 21 suggestions utilized explicitly new concepts or constructs, while 10 reinforced constructs or concepts already present in the example program. The majority of suggestions are either: “Method creation or abstraction” or “Added method calls”. In the suggestion novelty classification in Figure 5.3, a method call refers to a provided method, like *walk* or *move*. Programming constructs refer to loops and conditional logic. Method creation involves restructuring a sequence of methods into a custom method. Data storage refers to variables and parameters.

Both new concepts and reinforcement can be valuable feedback for novice programmers, as using a concept once does not imply mastery. However, the intended use of the Example

Guru is to suggest previously unused concepts. A crowdsourced system would thus need to instruct and remind the expert to make suggestions that introduce new skills or API methods. Especially for the API methods, this would likely be challenging because most expert programmers will not have expertise in novice systems or many full APIs.

5.3.2 Rule pseudocode

Seventy percent of pseudocode lines were labeled as either iteration or comparison. Figure 5.4 shows an example of a rule that has the typical pseudocode structure: line 1 iterates through each of the methods and line 2 compares the name of the current method call to the name of the next method called.

Most of the 24% of pseudocode lines identified as neither iteration nor comparison were either matching functions, attempts to access dynamic information, or count functions. A few participants created template matching functions that defined a set of constraints and then checked whether any lines met those constraints. Other participants checked information only available at runtime, such as the location of characters. However, rules with access only to the static code cannot check runtime information. Several participants used functions to count the number of times a line occurs. Because participants rarely used these types of functions, we focus on iteration and comparison.

```
1: foreach(allMethods as k=> method)
2:     if(method->name == allMethods[k+1]->name)
3:         return true;
```

Figure 5.4: Example of a rule

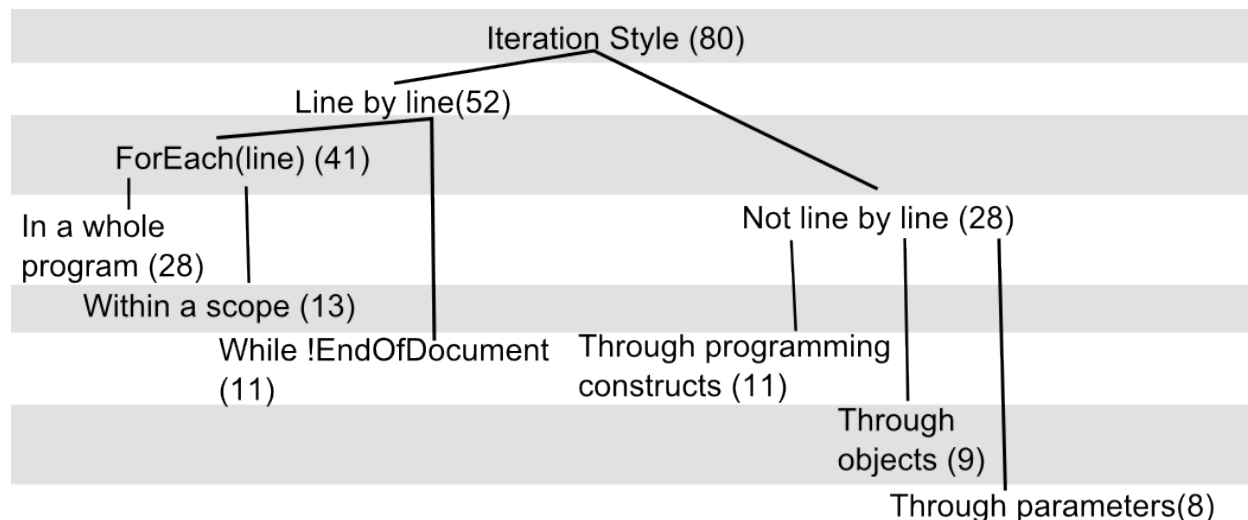


Figure 5.5: Iteration style category hierarchy.

Iteration

The iteration pseudocode lines fall into six categories: three ways of iterating through lines of code, and iterating through parameters, sets of objects, and programming constructs. The number of pseudocode lines for each of the categories is shown in Figure 5.5. When iterating through lines of code, some participants assumed access to a set of lines, in a *ForEach(line)* style, for the whole program or within a certain scope. Surprisingly, a number of participants parsed the program as strings in a *While!(EndofDocument)* style. By far, the most used iteration style was through each line in the program, but providing support for checking conditions in a certain construct or in a custom method is also likely to be useful to experts authoring rules.

Comparison

Participants often used comparison to determine whether a line or group of lines contain a certain issue. Comparisons fell into three high-level groups: comparison of multiple lines of code, whole lines or methods, and part of a line or method, as shown in Figure 5.6. The

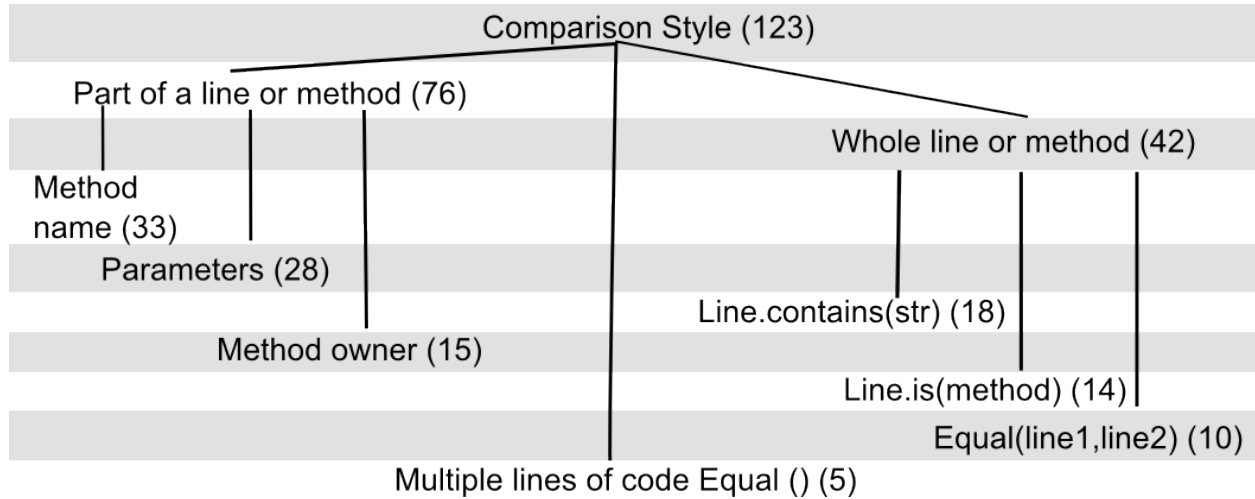


Figure 5.6: Comparison style category hierarchy.

differences result from participants either envisioning the code as a string or as a structure. Eight participants treated programs as strings, parsing and pattern matching with functions like *line.contains("methodName")*. The other 13 participants assumed that programs have a structure that they could query for information, such as accessing the method name with a function like *line.getName()*.

5.3.3 Rule implementation

We ran each rule to determine whether it would find programs that could benefit from the associated suggestion. Table 6.2 summarizes the rules, grouped by quality, and reports the percentages of programs in the mixed and novice groups that fit the rules. We initially hypothesized that these percentages might indicate whether a rule is too general or too specific. In practice, other, unrelated factors such as skill level of the user affect the percentage, making percentage only a weak indicator of rule quality. Our analysis suggests that rules range in quality from unfixable to immediately applicable. Our quality labeling resulted in four groups of rules: Good Code(GC), Fixable Code (FC), Bad Code (BC), and Unfixable (UF).

Good code (8 rules)

Eight of the rules found appropriate sets of programs for their associated suggestions with no improvement necessary. Some of the rules with good code are straightforward programming concepts. For example, GC2 looks for unnecessary *DoTogether* code blocks that contain fewer than two statements. GC3 replaces repeated actions by a set of characters with a *ForEach* loop.

Other rules in this category were less straightforward. GC1 looks for a sequence of color changes. Initially, we were skeptical that this would be generally useful. However, programs that repeatedly change colors are often implementing a flash behavior. While this suggestion may not be offered with great frequency (only 2% of programs matched), it is likely to be a valuable and relevant suggestion when it is offered. This is the type of suggestion that we predict to be the most motivating, as it helps the novice to learn something new while also improving the output of their program.

Fixable code (6 rules)

Fixable code rules are very close to finding appropriate programs for their suggestions, but the code neglects to check one or more conditions that could improve their results. To get a sense for how these rules might perform if corrected, we also created fixed versions of them. For example, FC1 suggested creating a new method if a character performs three actions in a row. A sequence of actions by the same character is often a reasonable place to suggest creating a new method. We modified this rule to check for at least six actions instead of three and to ensure that those actions are not already in a custom method. This dramatically reduced the number of programs matching the rule, reducing the amount of potential false

positives. Table 6.2 reports the original matching percentage followed by the fixed matching percentage for the mixed and novice groups.

Bad code (2 rules)

Based on their English descriptions, the two rules in this category are inspired by valid observations. However, the pseudocode rules do not match these descriptions. BC1 intended to look for programs in which a character performed two actions that can be condensed into a single action. For example, a new user might not realize that parameter values can be modified and use a string of *move* statements to position a character, rather than changing the distance moved. In practice, the rule searches for a sequence of two identical lines. While two *move forward* statements can be easily combined, not all actions have that characteristic.

Unfixable code (3 rules)

Unfixable rules have major issues such as presenting suggestions that are a poor example or irrelevant to the user's program. For example, UF2 looks for sequences of *say* statements and creates a new method. It then passes the text strings in as parameters. In an object-based context, this is an unrepresentative example of using a new method. In most cases, Looking Glass programmers use new methods to create and name a cohesive set of actions for a character.

Rule Quality	Short Description of Rule Intent	Fig.2 Category	Rule Quality Issue	Mixed %	Novice %
Good Code	GC1. Sequence of color changes	New Method	N/A	2	2
	GC2. DoTogether with < 2 statements	Remove Unnecessary	N/A	7	4
	GC3. Characters perform same statement in a row	Remove Rep.	N/A	16	15
	GC4. Custom method called more than once contains move	Default Position	N/A	14	0
	GC5. A character turns backward 1/4	Improve Action	N/A	0	0
	GC6. Program contains moveTo	Improve Action	N/A	21	40
	GC7. Something moves up, never moves down	Default Position	N/A	26	6
	GC8. Repeated use of a value	Generalize	N/A	41	0
Fixable Code	FC1 Character performs 3 actions in a row	New Method	Too Aggressive/ Incomplete	68→8	70→0
	FC2. My First Method has DoInOrder child	Remove Unnecessary	Incomplete	8→13	0→0
	FC3. Object moves more than once in a row	Remove Rep	Incomplete	27→2	7→2
	FC4. Say statement contains > 8 words	Dialog	Incomplete	46→39	56→9
	FC5. Custom method for a specific character	Generalize	Incomplete	36→33	5→7
	FC6. Repeated use of same duration value in a method	Local Variable	Incomplete	64→64	18→15
Bad Code	BC1. 2 consecutive identical lines	Remove Rep.	Idea-Code Mismatch	11	0
	BC1. Single character performs action multiple times	Improve Action	Idea-Code Mismatch	82	95
Unfixable Code	UF1. Reversible action	Default Pos.	Too Aggressive	83	71
	UF2. Sequence of single character say statements	New Method	Poor Example	26	36
	UF3. DoTogether contains several items	New Method	Poor Example	63	9

Table 5.1: Rules, rule issues, and percentages of programs receiving suggestions for participant rules

5.4 Threats to validity

We had a relatively small sample size of 21 participants with highly varying programming experience, all of whom had signed up for a science-focused mailing list. This likely biased our results in several ways: 1) their skills may not be representative of the typical experienced programmer population, and 2) their motivation to participate may be stronger than the typical programmer population.

5.5 Discussion

We discuss the quality of the content participants created and whether the content participants created is likely to be relevant to novice programmers.

5.5.1 Quality in expert created content

Nearly three quarters of the rules experts wrote were either strong or readily fixable, which provides some support for the approach of crowdsourcing suggestion-based help for novice programmers. In fact, our study may underestimate the percentage of good and readily fixable rules. Issues in quality likely arose from two factors: coding rules in a word document, and variance in experience.

Participants wrote rules in a Word document to prevent any influence of the IDE on their pseudocode. However, this also prevented participants from receiving the feedback they normally would receive when programming. The ability to test rules prior to submission would likely substantially improve their quality.

A large variance in programming experience could also explain some of the variety in rule quality. Our study attracted an enthusiastic pool of participants from people who self-identify as programmers. Participants included software engineers with decades of experience, students, self-taught web developers, and people who programmed on a regular basis decades ago. However, there appeared to be little relationship between rule quality and participant programming experience. This suggests that there will always be variance in rule quality. We will need to design crowdsourcing approaches with the high quality variance in mind. For example, to reduce the number of poor quality rules, a system could require vetting before use in a live system. The system would not recommend suggestions to novices until at least two other experts verify its quality. Experts could either edit inappropriate rules or vote them *Unfixable*, in which case they would eventually disappear from the system. This is likely necessary regardless in systems designed for children, where a crowdsourced technique without vetting could risk providing inappropriate content to children.

5.5.2 Relevance of content for novices

There are two potential issues with the types of suggestions experts made to novice programs: 1) whether the suggestions will always be relevant to novices, and 2) whether the suggestions will be motivating.

The nature of the rules participants created indicates that a system would need to use care when offering suggestions to novice programmers. While some rules identify matches with 100% certainty, others do not. *DoTogethers* with fewer than two statements (i.e. GC2) can always be simplified. In contrast, GC4 is inspired by the observation that methods called repeatedly should be free of side effects, such as ending in a location different from the start position. This rule finds methods called multiple times that contain move animations. While this may be appropriate for certain programs, some methods appropriately need the

ending position to differ from the starting position. Additionally, this rule looks only for a *move* method, which is not a perfect signal that a method has side effects. Consequently, systems that offer crowd-sourced suggestions will need to be designed such that suggestions are unobtrusive and are not offered endlessly. The development of this suggestion system will also enable evaluation of whether rules and suggestions actually teach novice programmers new programming concepts.

While we found that experts can find and make beneficial changes to programmers' code, they often focus on improving the style of the code, rather than the output of the code. Suggestions that fix the style of code may be motivating in a task with a specific solution, where improving the style might be important in order for the task to be marked as correct. However, the code quality suggestions often do not change the output of the code. This means that if they receive a code quality suggestion and modify their code to match, the output animation will remain unchanged. For participants in artifact-based contexts, this type of suggestion will likely be less exciting to a novice programmer than one that also improves their animation. For example, improving a novices' *fall down* animation while also introducing a new skill through the example would likely be more motivating than a suggestion that only makes their code cleaner or more maintainable.

5.6 Implications for the design of the Example Guru

The results of this study provide insight into the types of suggestions and rules a system can and should provide to novice programmers working in an artifact-based context.

5.6.1 Suggestions

Because the the Example Guru aims to encourage novice programmers to explore new code and skills, we based the Example Guru suggestions off of the animation suggestions in this study. The animation suggestions make novices' animations better and can also be suggested specifically when a programmer has not yet used a certain skill. We hypothesize that this type of suggestion will encourage novices to explore new code.

5.6.2 Rules

Based on this study, it seems like the rules in artifact-based contexts may not apply to novice programs 100% of the time. In contexts where learners are working toward a specific goal or programmers need to fix a specific error, hints provided by the system likely need to correctly lead the programmer toward the solution. When a system suggests an idea to a programmer whose task has a vague goal and no specific solution, the hint may not always need to apply to exactly what the programmer is working on. Instead, a hint may give the programmer a better idea to improve their program, or provide tangential information that could be useful at a later time. Thus, the rules in the Example Guru follow a similar pattern to many of the rules in this study, using heuristics to determine when to trigger suggestions. Further evaluation is needed to determine whether irrelevant suggestions annoy novices or discourage them from exploring further suggestions.

5.7 Conclusion

This chapter describes a study exploring experienced programmers' ideas for suggestions and rules. Experienced programmers often made suggestions that improved novices' code

quality, but on occasion made suggestions that improved the animation. Participants also wrote pseudocode scripts that would often find valuable opportunities to provide suggestions, but may not trigger suggestions perfectly. Since experienced programmers often provide feedback to peers and less experienced programmers, their suggestions and rules provide intuition about the kinds of feedback a suggestion system can provide. The Example Guru will initially provide animation-type suggestions, but including code-based suggestions as programmers become more experienced would likely be valuable.

Chapter 6

Designing and Evaluating the Example Guru for Suggesting API Methods

Note: Portions of this chapter were published in the Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems [90].

6.1 Introduction

This thesis hypothesizes that novice programmers would benefit from suggested examples because they do not know which skills they lack. This chapter describes the design of the Example Guru suggestion tool and an implementation of the suggestions for API methods. It then evaluates whether young novice programmers in artifact-based contexts choose to access Example Guru suggestions more than static documentation when programming on their own. This chapter addresses hypothesis 2: that suggesting example code to novice programmers in

artifact-based contexts will increase the number of new API methods novice programmers explore and use in their programs more than traditional forms of support.

I chose to evaluate Example Guru suggestions initially for API methods in order to focus the evaluation on whether suggestions can be beneficial. The example code required for API methods is often one code block and is concrete, which minimizes potential problems novices could have using abstract examples. API methods have concrete, visible effects in the output animations, like making an object move, change size, or change color. This reduces any possible confusion that could result from the examples, enabling us to evaluate the efficacy of suggestions.

In addition to reducing the complexity of presented examples, learning APIs is a common challenge for all levels of programmers. Research has shown that programmers often struggle to learn and use Application Programming Interfaces (APIs) [181]. These issues learning APIs stem from a variety of causes, including insufficient resources, confusing API structure, lack of programming experience, and unawareness of API methods [180]. The Example Guru aims to reduce API method unawareness. While unawareness of API methods affects all programmers, those with less programming experience, such as children learning programming or end-user programmers, often find barriers to learning APIs insurmountable [112].

To illustrate why using a large and unfamiliar API can be especially challenging for non-expert programmers, imagine an end-user programmer, Julie, who needs to analyze data from a biology study quickly. She decides to use Ruby to write a CSV file of results, but does not realize that an API method exists to automatically format an array correctly with commas. Instead, she loops through her data, adding commas where they seem appropriate. Existing commas within her data make this task even more complex. Imagine instead that while Julie

writes her array output code, her IDE offers a tip that introduces Julie to the method used for array formatting along with examples that illustrate its use.

We are unaware of existing research in API support or computer science education that has fully addressed the awareness problem in learning APIs. Instead, researchers have created systems for helping experienced programmers use APIs that improve: code completion [10], search [215], and available documentation [206]. These support systems require users to query a method of interest, so they do not help programmers identify new applicable API methods or incorrect usages of API methods.

6.2 The Example Guru design

I first designed the Example Guru to suggest API usages to novice programmers based on their code in Looking Glass [121]. This chapter describes the initial design of the Example Guru for API suggestions, which we modified and improved for code concept suggestions in the next chapter. As a reminder, the Example Guru has three main features: 1) rules, which parse code, looking for opportunities to suggest API methods, and 2) suggestions, and 3) the selected code for the examples. This section describes the rationale for the interface design and the content for each of these three elements.

6.2.1 System design methods

We used two methods to design the Example Guru: 1) formative studies, and 2) program analysis. We used an iterative design process in a formative study with 48 participants aged 10-15 for the Example Guru suggestion interface. To design the rules and suggested concepts and examples, we used two sets of programs not created for this study. One set contained

107 programs created by Looking Glass API experts. The second set contained 600 programs shared to the Looking Glass website by non-experts [121].

Suggestion interface

We will first describe how a user would interact with the suggestion interface, followed by the rationale for the design. When a user is programming in Looking Glass and executes their code, the system uses the rules to analyze their code and determine if any suggestions should trigger. Only one suggestion triggers at a time and appears in two locations: as an annotation on the code and in a list near the code block menu. If they want to know more about the suggestion without opening it, they can hover over the suggestion to read a description or watch a video of the example code animation. When a user chooses to access a suggestion, they click on it, which opens a window showing the example code. The suggestions do not require that a user accesses them and enable the user to reference them at a later time.

We designed the suggestions through an iterative process with one-on-one study sessions with the following goals: 1) to not interrupt or overwhelm the user, 2) to be easily accessible, and 3) to demonstrate the relevance of the suggestion to the code. Formative user testing indicated that programmers were most open to new ideas and improvements around the time they decide to test their code, so the Example Guru presents new suggestions after code execution. A list of suggestions allows the user to return to a suggestion at any point (see Figure 6.1-A). Code annotations connect suggestions to the relevant code (see Figure 6.1-B). Hovering over a suggestion in the suggestion list provides a preview of the example and hovering over an annotation in the code shows a text description of the suggestion. These previews provide a hint of what the suggestion would show if opened, similar to surprise, explain, reward [231].

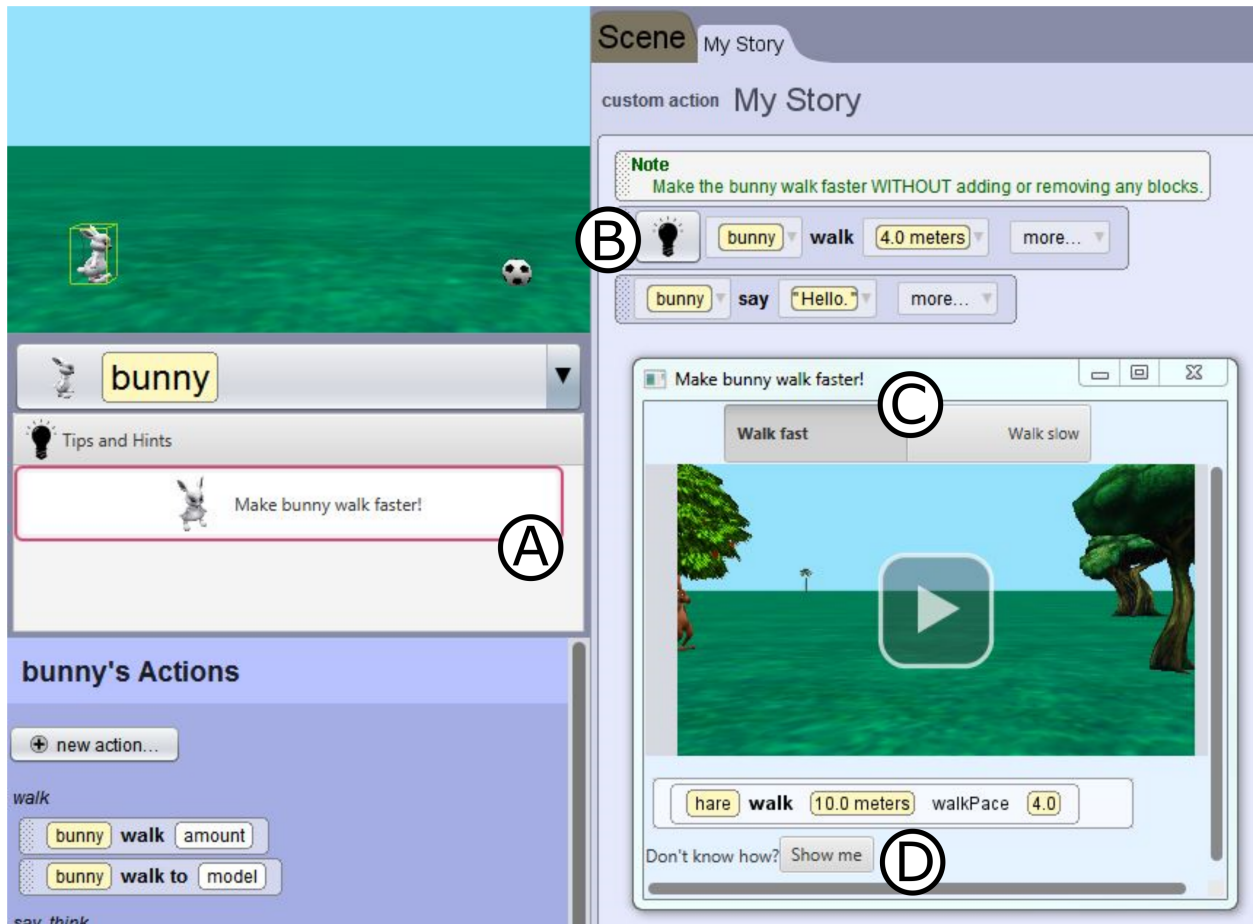


Figure 6.1: The Example Guru implemented within Looking Glass. (A) List of all suggestions. (B) Code annotation button to open the most recently added suggestion. (C) Contrasting examples such as ‘walk fast’ and ‘walk slow’. (D) ‘Show me’ button that users can click to see the location of the suggested block.

The examples presented within suggestions differ from those in other systems [79, 100, 159] in two critical ways: 1) they emphasize how the API method works using two contrasting examples and one-line code examples (see Figure 6.1-C), and 2) they provide support for finding the relevant code block in the interface (see Figure 6.1-D).

We developed the idea for contrasting examples through formative testing, where participants often did not know which argument values to use in blocks of code. The two contrasting examples either show different values or two API methods that work similarly in order to

highlight the differences. The goal of the contrasting examples was to encourage novices to perform self-explanation, which has been shown to be effective for learning [41]. The single-line code examples also support understanding of the examples not including extraneous information that could confuse novices. This version of the Example Guru slightly differs from the final version, in which the two examples always both include the abstract concept. For concrete API methods where the output is clearly visible, contrasting examples can be effective. For abstract concepts, contrasting examples are hard to tell apart and confuse novices.

Within a blocks environment, understanding how a block works does not necessarily mean a novice can use it. Formative and related work indicated that novices sometimes have trouble finding a code block from an example [91]. One study [75] found that providing a ‘show me’ button that, when clicked, highlighted the location of the block in the programming environment, helped programmers find necessary blocks (see Figure 6.1-D). This is slightly different from the final version, which includes a tool tip description of where to find code blocks, rather than a button. We changed this for the final version for two main reasons: 1) as the examples get larger, saving space in the suggestion window is important, and 2) as the concepts get more challenging, we wanted novices to have to think about which code block they needed, rather than having the ‘show me’ button as a way to cheat.

In the results, we answer whether participants used the features provided in our design. Table 6.1 details designs we tested and found to be ineffective.

Rule, suggestion, and example content design

For API suggestions, we designed and implemented the rules by hand. In Chapter 7, we address how a system could automate this process. Our manual process had three main steps:

Design	Issue
Suggestions appeared alongside the execution view.	Users did not focus on suggestions while executing their code, but instead returned to the code before considering what to do next.
Suggestions only appeared as buttons next to the code.	Users did not always want to access suggestions immediately, but displaying many suggestions crowded the editor.
Suggestions only contained one example.	Users did not understand the impact of argument values relative to their code.
Examples had text along with the code to explain how the example worked.	The text made the example view crowded and made it hard for users to focus on the critical elements. Users rarely read the text.

Table 6.1: Unsuccessful design attempts in formative testing

1) compare novice and expert API use to select the API methods to suggest and the priority order, 2) consider the types of animations experts created with specific API methods and find simpler or related animations novices make where new API methods could be useful, and 3) author the rules within the system.

In order to select which API methods to suggest and the priority with which to suggest them, we compared our sets of novice and expert programs. We wanted to suggest API methods that novices were likely to be unfamiliar with, but also that they were likely to find useful. The set of API methods the Example Guru suggests contains API methods that experts used more often than novices and that experts used more than 5% of the time. These API methods are likely unfamiliar, but also used frequently enough by experts to be useful. Selecting the API methods to suggest based on expert usage helps to prevent the suggestions from over-fitting to the first hour of programming. Since the Example Guru only presents one suggestion at a time, we designed a priority ordering for selecting one of the triggered rules to suggest. Rules suggesting ways to correct API usages have the highest priority. The

Example Guru then suggests API methods with the largest difference in expert and novice use and that experts used more often. The lowest priority suggestions are for API methods that experts and novices used with similar frequencies or that experts rarely used.

The design of rules is similar to code smells [57] and anti-patterns [100], but instead of focusing on checking for poorly composed code, most of the rules in the Example Guru look for opportunities to introduce new concepts. Essentially, rules recommend ways to improve animations through the use of previously unused API methods. Our formative work showed that it is important for the rules to find opportunities to improve novices' animations because suggestions that only improved the code quality were less exciting to novices creating animations. This is because novices in Looking Glass are focusing primarily on their animations, rather than on trying to learn new programming concepts. In order to decide when to recommend a particular unused API method, we manually checked how experts used API methods for complex animations. In many cases, novices create similar, more basic animations with more commonly used API methods. For instance, one rule checks programs for characters turning multiple rotations, as the programmer may be attempting to make characters dance. The rule has a suggestion that demonstrates how to animate joints to make a more realistic dancing animation. Each rule has an associated suggestion that introduces the API usage to the programmer.

Finally, we implemented the rules within the Example Guru. Rules contain a specification of how to parse code for opportunities to improve. Specifications use an internal API designed to simplify querying the abstract syntax tree.

API call	Type	Rule Checks For
Walk speed	New	Walking far, no speed
Joint animation	New	Dancing, no joint animation
Straighten out joints	Incorrect	Straighten joints method, no joint movements

Table 6.2: Rules, suggestions and examples

6.3 Evaluation

We ran a study to evaluate the effectiveness of the Example Guru’s suggestions in encouraging new API method use by comparing them to an in-application documentation control condition. We will call the two conditions the ‘suggestions’ condition and the ‘documentation’ condition. In working towards reducing the unawareness problem for novice programmers learning new APIs, we tested the following two hypotheses:

H1: Novice programmers will access suggestions more frequently than documentation. We hypothesize that the suggestions will expose novice programmers to API methods that they likely would not have realized existed.

H2: Novice programmers using suggestions will improve their API usage more as a result of suggestions than novice programmers will from API documentation. Here, we want to compare the number and type of API methods participants add to their code after accessing suggestions or documentation.

6.3.1 Documentation condition

Currently, the best practice for supporting use of unfamiliar API methods is providing easy-to-access documentation containing example code. A few systems use suggestions within

a programming context, but focus on violations of proper programming. Errors provide natural motivation to use suggestions, but in the case of API use, we cannot assume that novices will be motivated to apply a nonessential suggestion. We believe comparing to the best practice, documentation, is an appropriate first step toward evaluating the Example Guru.

We designed the documentation based on two common forms of API support: online API documentation and code completion. We wanted the documentation to have full information like online documentation, while making it easily accessible like code completion. Thus, the user can access a doc (documentation for a specific API method) by clicking a ‘?’ button beside the code block the user is interested in (see Figure 6.2-A). Having the documentation available in the palette with the code blocks and in the parameter menu on code blocks aims to mimic the availability of information through code completion. Upon opening a doc, the user can view descriptions and examples of how the API method works, along with all of the available parameters for that API method (see Figure 6.2). Docs first show only examples of the API method and an option to show more information to view the parameters (see Figure 6.2-D).

6.3.2 Participants

We recruited participants who had never used Looking Glass because this study investigates novice programmers exploration and use of the Looking Glass API. We recruited 81 participants aged 10 to 15 from a local STEM mailing list. Two participants had used Looking Glass in the past and a third skipped the first phase of the study, so we analyzed the data from the remaining 78 participants. The 78 participants had an average age of 11.8 (SD=1.6), were 46.2% female, 52.6% male, and 1.2% unspecified gender. We compensated each participant with a \$10 gift card to Amazon.com.

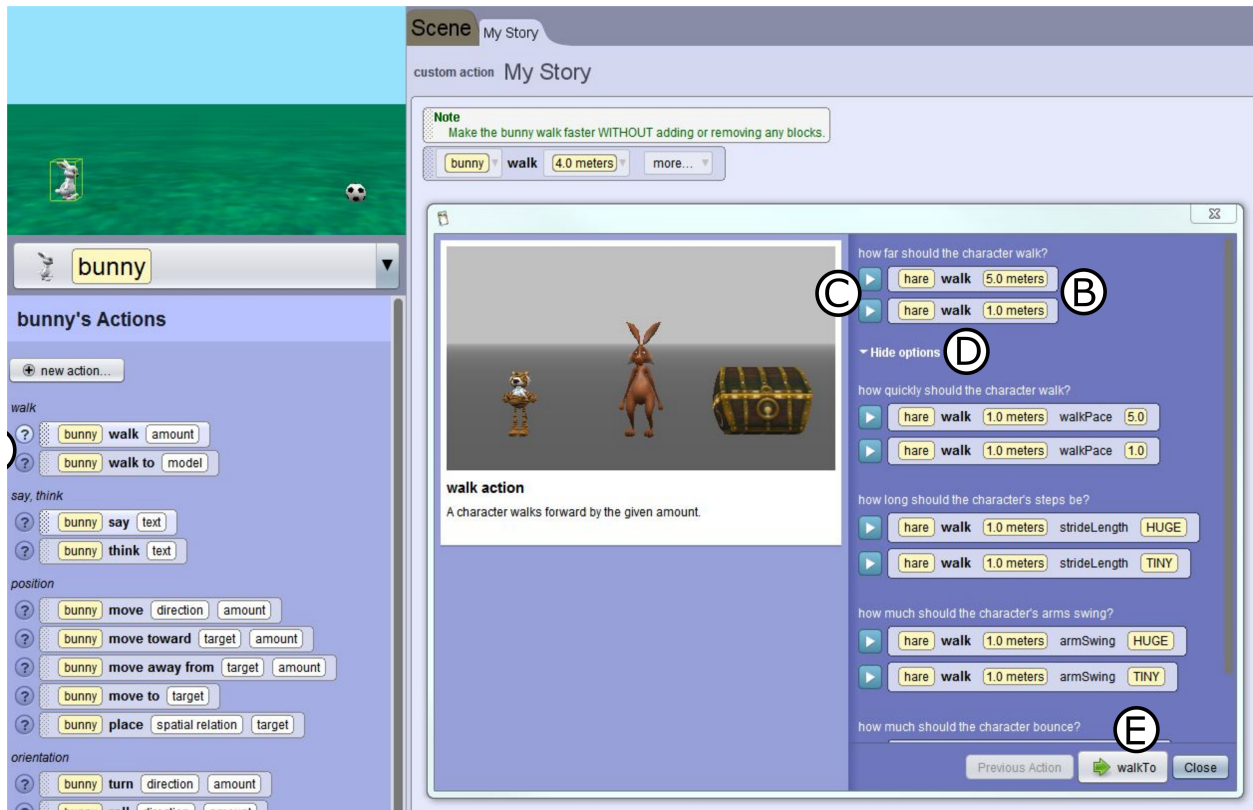


Figure 6.2: In-application API Documentation condition. (A) Users can access documentation using the ‘?’ button available beside APIs. (B) Examples with different values and the description. (C) The play button can be used to execute the code. (D) Button to expand or collapse the parameters information. (E) Users can navigate to other doc using these buttons.

6.3.3 Methods

We created materials in order to measure API information access, API usage, and participant features that could influence how participants use API information.

API information access and usage

In order to evaluate whether participants accessed suggestions more than documentation, we needed to ensure two things: 1) that participants were equally familiar with the API information formats, and 2) that participants actually received suggestions while working

on artifact-based programs. In order to familiarize participants with the API information, we created two training tasks. For the first training task, participants had to make a bunny walk faster by adding an optional argument value to a ‘walk’ action. For the second, they had to make a shark swim around an island by adding an optional argument value to a ‘turn’ code block. We provided instructions on a sheet of paper that directed participants to use the API information provided.

In order to improve the odds that participants would receive suggestions during artifact-based programming, we created and tested scenes with props and characters for participants to use in creating their animations. For instance, complex movements and rotations trigger suggestions, so one scene was designed for a ‘Seaworld’ show animation. This type of scene enabled novice programmers to try to create complex animations with dolphins. We selected five scenes for this study based on popular scenes from formative work because children were most excited to create animations with those scenes. During the study, novices had the choice to create whatever animation they wanted with any of the scenes, so they could always choose whether or not they wanted to create complex animations or use suggestions.

Participant features

In designing the Example Guru, we wanted novice programmers to benefit from suggestions regardless of their age, gender, or programming experience. Because the suggestions are context-relevant, we hypothesized that the suggestions would interest users with very little programming experience, as well as novices with more exposure. In order to capture information about programming experience, participants filled out a demographic and computing history survey.

We also thought that the way participants like to learn might affect how they use API documentation and explore new API methods. In order to capture this, participants also filled out an exploring and learning technology survey on paper before the study (see section D.4.2), modeled after the survey about trying new technology in [30]. Additionally, to better understand participants' motivations in using new API methods, we created dynamic surveys for participants to fill out on-screen after completing artifact-based programs during the study. The surveys asked questions about why participants used new API methods for the first time and why they used or did not use API information during the just-completed program.

6.3.4 Study procedures

There were three phases of this study: 1) baseline project, 2) training, and 3) supported project. The study was between subjects with two conditions: suggestions and documentation. Because work has demonstrated that gender plays into exploration and learning in software [30], we randomly assigned participants to either use suggestions or documentation keeping gender balanced. Participants worked individually on all tasks and were allowed to move onto the next task if they felt they had completed the current one.

Baseline project phase

We wanted to know how participants would use the API without any support, so participants first created an animation without API support for up to 15 minutes. Because some participants had no programming experience, the instructions for the first phase gave information about how to drag blocks into animations and execute the animations (see section D.1). This phase involved artifact-based programming, which means that there was no correct or incorrect answer and that participants were allowed to freely create their animations. We

assigned participants a specific scene for this task and balanced the assignments of scenes across participants to limit any effect of specific scenes on API usage. In order to find out more about why participants added new API methods, participants completed an on-screen survey at the end of this task that asked about: 1) one new API method participants added and executed, and 2) one that they added, executed and deleted, if these existed.

Training phase

Due to time constraints for a controlled study, we wanted participants to become quickly comfortable with using either suggestions or documentation. To do this, we had all participants complete two training tasks. In both cases, the instructions showed how to access the suggestion or documentation that would help them complete the task. The researchers checked the participants' code to make sure they successfully completed the task and helped participants if needed. If participants completed a task without a suggestion or documentation, the researcher demonstrated how they could have used it to ensure that all participants were exposed to suggestions or documentation.

Supported project phase

Finally, we wanted to evaluate how participants used the API information and API methods when working on their own projects. During the supported project phase, participants created animation projects with either suggestions or documentation available to them. We first asked participants to create a program based on the idea of a Seaworld show. The purpose of providing an idea was to give participants a goal to work towards, but not to constrain what code they should use. Next, participants were assigned a scene in which they could create any animation or use a provided story prompt if they did not have an idea. Participants had up to fifteen minutes to work on each of the two animations. If participants finished early,

they could select a scene they had not yet used and create another animation. The template scenes are all available in section D.2. At the end of each of these animations, participants also typed answers to questions onscreen about why they added or removed certain API methods and why they accessed or did not access API information (see section D.4.3).

6.3.5 Data collection and analysis

We logged all actions participants took and survey answers to analyze suggestion, documentation, and API usage.

Time on task

We did not require participants to spend the full amount of time provided on each task, so some participants spent less than the standard amount of time. Most participants (76%) spent the full amount of time on the baseline (15 min.) and supported project (30 min.) phases. We stopped analyzing participants' data after 30 minutes in supported project. We will report the results for the set of participants who spent the full amount of time (59 participants), as well as the results for all participants.

# of participants who spent:	Suggestions	Doc.	Total
<15 min. (baseline open)	1/39	2/39	3/78
<30 min. (supported open)	6/39	7/39	13/78
<15 and <30	1/39	2/39	3/78
full time	31/39	28/39	59/78

Table 6.3: Time participants spent on the tasks

Accessing API information and API usage

We analyzed logs in order to measure which suggestions and documentation items participants accessed, meaning that they clicked to open the API information. To determine whether participants used new methods from the API information in their programs, we measured which API methods participants inserted into their programs for the first time after accessing related API information. When comparing the number of accesses and API usage, we used t-tests to compare the aggregate numbers because participants received different numbers of suggestions. Additionally, participants in the API documentation condition could access docs many more times than the number of suggestions available. We use Cohen's d to measure effect size (small: .3, medium: .5, large: .8). We also report the percentages of participants who accessed API information and used API methods and compare this using Chi-squared tests. We use the odds ratio to measure effect size (small: 1.5, medium: 3.5, large: 9).

For both API information access and API usage, we describe the kinds of API methods participants were accessing information for and inserting into their programs. We believe the best way to do this is to group the API methods based on how much novice programmers generally use them. We base the frequency of novice use on the set of 600 non-expert programs described earlier. We will discuss the API methods in terms of 4 groups: those that the Example Guru did not suggest, APIs suggested that were used least frequently by novices (the bottom third of usage frequency), those suggested that were sometimes used (the middle third), and those suggested that were most often used by novices (the top third of API method usage frequency).

Participant qualities

We analyzed participant qualities to try to understand the types of novice programmers who will benefit from suggestions or documentation. We collected gender, age, programming experience, and learning style data from the surveys. We captured programming experience using two survey questions: ‘Have you programmed before?’ and ‘Have you programmed for more than 3 hours in your whole life?’ Those who had less than 3 hours of programming experience likely only programmed once or twice without much instruction or practice. Nine participants in the suggestion condition (23%) and eight participants in the documentation condition (21%) had 0-3 hours of programming experience. We also intended to capture personal preferences about using API documentation using the on-screen end-of-task surveys for both conditions. Due to a technical error, the survey questions asking participants about why they did or did not access documentation did not appear for the study participants, so we report quotes from pilot users who completed the same study and received these questions.

6.4 Results

We hypothesized that participants who received suggestions would: 1) access suggestions, and 2) use API methods from the suggestions more frequently than participants would access and use in-application documentation. In this section, we first explore these two hypotheses and then delve into how different participants used the API information and the features they used.

6.4.1 Access and use of suggestions and documentation

Ideally, suggestions should encourage API exploration when novice programmers are pursuing their own projects. We evaluated this through the number of times participants accessed API information and how many new API methods they used after accessing API information.

Accessing suggestions and documentation

We found that more participants accessed suggestions than accessed documentation: 82% of suggestion participants and 41% of documentation participants accessed at least one entry. The difference in the number of participants who accessed suggestions verses documentation was significant with a medium effect size ($\chi^2(1) = 12.19, p < 0.001$, odds ratio = 6.4). Participants in both conditions described using suggestions and documentation to gain additional information about API methods that seemed potentially relevant. A participant in the documentation condition described opening an API method that changed a character’s appearance because: “... I wondered what it was. It turned out to change Alice.” One participant in the suggestion condition sought additional information about a new method based on the tip offered as part of the suggestion: “I opened the tip for ‘setTransparency’ because I thought it was a good way to make an object disappear”.

We found that participants accessed more total suggestions, on average, than documentation. For all 78 participants, participants accessed significantly more suggestions ($M=3.3$, $SD=2.7$) than documentation ($M=1.4$, $SD=2.7$), $t(76) = 31, p < 0.01$, $d = 0.69$. Since some participants spent less than the full task time, we also confirmed that this difference existed for the set of participants who used the whole task time. The results were very similar: participants accessed suggestions ($M=3.0$, $SD= 2.7$) significantly more than documentation ($M= 1.1$, $SD = 1.7$) with a large effect size ($t(50) = 3.3, p < 0.01, d = 0.85$). Simply accessing more

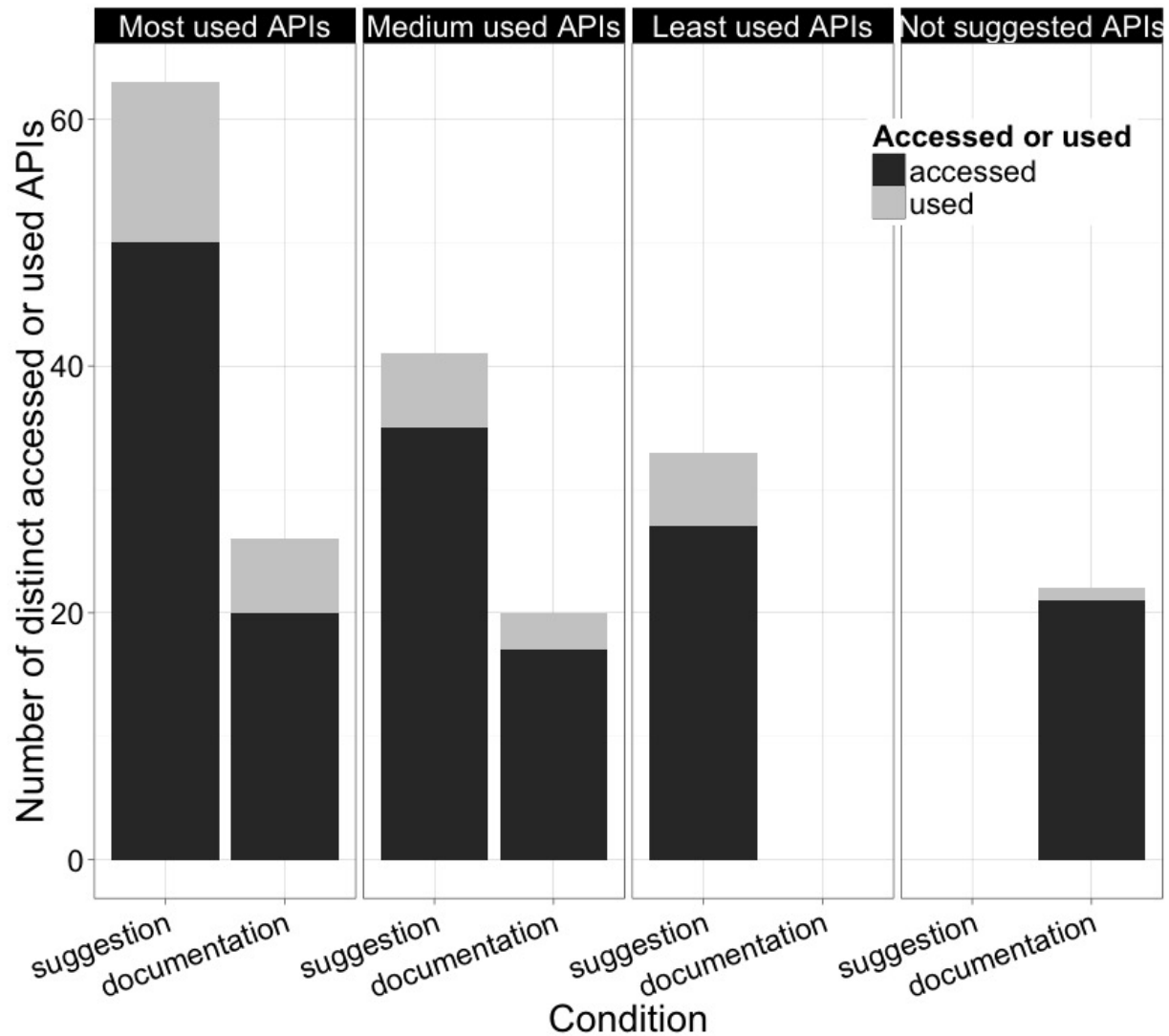


Figure 6.3: API information accessed and used grouped by frequency of API use by novice programmers.

suggestions is a potential benefit to novice programmers because the suggestions expose them to broader range of API methods that may be useful either immediately or in the future. In this study, we could not measure whether a participant used a suggestion based on reading the tip without opening it, but survey responses indicate that some participants did this: Participant S33 did not need to access a suggestion because reading it was enough: “I did not

open the tip for Turn to Face because I read the outline for the Tip and used it in my code.” Similarly, participant S70 said: “I did not open the tip because I saw it from the outside and felt like I could figure it out and I think I did.”

Our goal was to encourage novice programmers to use API methods they would not necessarily use on their own. To evaluate this, we analyzed the information access and API use based on how often novice programmers in our sample set of programs used API methods. We split the API use based on one group of API methods that the Example Guru did not suggest and three groups that the Example Guru did suggest: the top third of methods that novices generally use most frequently, the middle third, the bottom third. The set that was not suggested includes API methods that novices use more than experts or that experts use in less than 5% of programs. In all three groups of API methods, the API information was accessed and used more frequently by participants in the suggestion condition (see Figure 6.3). While the largest use of suggestions was for API methods novices generally use the most, increasing use is beneficial because the average percentage of novices using those API methods is less than 50%. Furthermore, only participants with suggestions accessed information for the least used API methods.

The survey results provide additional insight into the reasons participants chose to access or not access API content. Due to a technical failure, participants in the documentation condition did not receive questions about their documentation access or use. Since questions about usage might encourage some users to increase their API usage, we looked for an increase in suggestion access and usage following the survey, which was administered after the first supported animation project. However, we see little evidence of this. Ten participants used suggestions only during the first animation project, an additional seventeen accessed suggestions throughout, and only five participants accessed zero suggestions during the first

project task, but one or more in the following animations. Thus, we do not believe that the survey questions influenced suggestion use.

Overall, participants described accessing suggestions to gain additional information about API methods that seemed potentially relevant (see the top section of Table 6.4). Often, this was because participants thought the suggestion could improve their animation (50% of statements), like one participant who received a suggestion about setting the color of the sky. They said: “The dark sky was sooo boring, so I looked at the tip and used it.” Other participants wanted to learn about the information for their general knowledge (21% of statements), like one participant who stated “... I did not have a need for it in my current animation, but wanted to know how to use it in the future.”

Participants often accessed suggestions but did not use the API methods or did not access suggestions at all because the suggestions did not fit with their vision of their animation (see the third section of Table 6.4). This was the largest reason for not using API methods from suggestions, explaining 53% of statements about not using API methods. This was also a major reason why participants did not access suggestions in the first place- they read the description and already knew that the API method would not fit in their animation (29% of statements about not accessing suggestions). The other major reason participants did not access suggestions was that they claimed that they already knew how to do what was suggested (23% of statements about not accessing suggestions). Since the Example Guru only suggested new API methods, this may indicate that the titles did not effectively communicate the features of the API methods. In other cases, participants wanted to access all of the suggestions, but ran out of time, like one participant: “I didn’t open all of the tips yet.” Finally, some participants were focused on other suggestions and missed ones that would have been of interest. One participant described missing a suggestion “because I was looking at other tips and didn’t realize there was a tip [to] make only [the] alien’s head turn.”

<i>Label: description or example</i>	<i>Accessed, used</i>	<i>Accessed, did not use</i>	<i>Did not access</i>
Wanted to learn: they wanted to learn or that they wanted to see what it would do	7 (21%)	5 (16%)	0
It improved the animation: description of how they used it to improve their animation	17 (50%)	0	0
As a reminder: explicit statements of using it as a reminder or talking about having forgotten how to do something	3 (9%)	0	0
Experimenting: experimenting, testing, or trying things out	1 (3%)	1 (3%)	1 (2%)
Could figure it out from the suggestion title: ‘... I read the outline for the Tip and used it in my code.’	0	0	1(2%)
Wanted to figure it out on their own: wanted to or did figure out how to do it by themselves	0	0	4 (6%)
Did not fit with animation: why the suggestion did not improve their animation	0	17 (53%)	19 (29%)
Already knew how: participants already knew that API method	0	0	15 (23%)
Animation already had it: their animation already used that type of animation	0	1(3%)	0
Could not figure it out: ‘I couldn’t figure it out.’	0	0	1 (2%)
Accidentally opened: ‘because i accidentally opened it’	0	1 (3%)	0
Meant to: planned to open tip or use code	0	1(3%)	1 (2%)
Ran out of time: they would have accessed or used if there had been more time	0	3 (9%)	1 (2%)
Did not notice it: did not notice it or pay attention to hints	0	0	8 (12%)
No reason: did not give a reason or the reason did not answer the question	6 (18%)	3 (9%)	14 (22%)

Table 6.4: Categories of responses from suggestion participants about why they accessed and used, accessed and did not use or did not access suggestions.

Participants in the documentation condition similarly described a desire for additional information as a motivation for opening documentation: “I wanted to know what it was and I used it because I thought it would be pretty cool to begin and end abruptly.” We unfortunately cannot report on their decisions around documentation they did not access.

Using suggestions and documentation

Since increased access to API information may help to support the use of a new API method, we also wanted to explore the use of new API methods after information access. We found that more participants used new API methods after accessing suggestions than after accessing

documentation. About three times as many participants in the suggestions condition used an API method after accessing the API information as in the documentation condition, 38% vs. 12.8% ($\chi^2(1) = 5.4466, p < 0.05$, odds ratio = 4.17). Additionally, participants added more new API methods after accessing the suggestions ($M = .59$, $SD = .82$) than after accessing the documentation ($M = 0.15$, $SD = 0.43$). This was significant for all 78 participants ($t(57.6) = 2.94, p < 0.01, d = 0.67$) and for the 59 participants who used the full task time ($t(49.3) = 2.2, p < 0.05, d = 0.55$).

In addition to frequency of use, it is interesting to explore the diversity of methods participants choose to use. In particular, we designed our rules and suggestions with the goal of introducing API methods that experts use more frequently than novices. Participants in both conditions used more new API methods from the group of API methods that are most commonly used by novices than the other groups. However, we note that participants using suggestions used more new methods from the middle and low use categories combined (see Figure 6.3). Finally, we looked at API methods for which we did not create suggestions. While some participants in the documentation group accessed information about these methods, only two were actually added. This provides some support for our method of selecting API methods for suggestions.

Our survey results suggest that participants in the suggestions condition decided to use an API method based on its potential to improve their animation. One participant explained “I just thought that changing the posture of the dolphins created a more natural feel than just moving its entire body.” In contrast, participants in the documentation condition more often cited goals of understanding. For example, one participant using documentation stated, “I opened it and chose to use it so I could see what it looked like.” We see a similar dichotomy around participants’ explanations for non-use. A participant in the suggestions condition chose not to use an accessed suggestion because it did not mesh with her vision for her story:

“I wanted to have the dolphin to go different distances showing they each do a little more than the last dolphin.” A participant in the documentation condition explained accessing but not using documentation for a duration parameter because “...I wanted to see how it worked.”

Con- di- tion	Action	<3 hours prog.	3+ hours prog.	p	Male	Female	p
Suggestions	Accessed	100%	77%		86%	78%	
		M=3.8 SD=2.3	M=3.2 SD=2.9		M=4.3 SD=3.0	M=2.2 SD=1.8	<.05
	Added API call	44%	40%		48%	33 %	
		M=.56 SD=.72	M=.6 SD=.86		M=.67 SD=.86	M=.5 SD=.79	
Documentation	Accessed	75%	32%	.07	30%	56%	
		M=3.4 SD=4.0	M=.94 SD=2.0		M=1.05 SD=2.8	M=1.9 SD=2.7	
	Added API call	38%	6.5%	.08	10%	17%	
		M=.5 SD=.76	M=.06 SD=.25		M=0.1 SD=.31	M=0.22 SD=.5	

Table 6.5: Participant characteristics and information access and API usage.

6.4.2 Do participants’ demographics affect how they used suggestions and documentation?

In the design of the Example Guru, we hoped to support participants regardless of age, programming experience, and gender. By having suggestions relate to the context of the program and API methods that the programmer had not yet used, we hypothesized that the suggestions should continue to be relevant to programmers as they become familiar with more of the API. Previous studies have found a correlation between age and programming success with the same age range of children [78]. These differences in performance could result from the developmental changes that impact children’s abilities to understand abstraction around the ages of 11-12 [164]. We hoped that the context-relevant approach would support novice programmers of differing ages. We also hypothesized that suggestions might better support participants who liked to learn by accessing information, rather than by tinkering,

since suggestions do not require the user to seek out new API methods and documentation. Since females have been shown to be less likely to learn through exploration in some cases [18], it seemed as though the suggestions might provide better support for female novice programmers. Overall, we found no difference in suggestions usage by age. However, while females accessed suggestions more frequently than documentation, they did so significantly less frequently than males.

Age and programming experience

Our results did not show a relationship between age and accessing and using either suggestions or documentation. Specifically, we found no significant correlation between age and suggestion access ($t(37) = -0.5, p = 0.62, r = -0.08$) nor between age and documentation access ($t(37) = 0.37, p = 0.71, r = 0.06$). Similarly, we found no significant correlation between age and the number of API methods used after accessing suggestions ($t(37) = -0.92, p = 0.36, r = -0.15$), nor between age and the number of API methods used after accessing documentation ($t(37) = -0.22, p = 0.83, r = -0.04$). These results suggest that both documentation and suggestions are used similarly by children ranging in age from ten to fifteen.

Programming experience played a larger role in how much participants accessed and used API information. Those with less than three hours of programming experience were the most likely to access both suggestions (100%) and documentation (75%). Participants with little programming in both conditions added new API methods after accessing them at similar rates: 44% of those in the suggestions condition and 38% of those in the documentation condition added API methods. However, among those with more than three hours of programming experience, we see a trend towards more use of suggestions. Of the participants in the suggestions condition, 77% accessed a suggestion and 40% added a new API method after accessing its suggestion. For participants in the documentation condition, only 32% accessed

API information and 6.5% used a new API method after accessing its documentation. This trend suggests significant promise in the use of context-relevant API suggestions to help programmers continue to explore new API methods.

Gender and learning style

When considering how gender might relate to participants' use of API information, we explored use by reported gender, as well as learning style based on a survey.

We found that both males and females accessed and used the suggestions at higher rates than the documentation. However, male participants accessed suggestions more often than female participants: males accessed an average of 4.3 suggestions as compared to 2.2 suggestions for females ($t(33.5) = -2.7, p < 0.05, d = .83$). Male participants also accessed a larger percentage of the suggestions they received, so the larger number of suggestions accessed by males was not due to a larger number of suggestions received ($t(34.3) = -2.7, p < 0.05, d = 0.84$). While not significant, we note that female participants accessed documentation more often than male participants, averaging 1.9 documentation accesses as compared to 1.05 for males, as shown in Table 6.5. Overall, male participants opened more suggestions, but both genders accessed suggestions. The significant difference in terms of the number of suggestions accessed represents an important avenue for additional research. While suggestions performed better than documentation for both male and female participants, the lower usage by female participants has the potential to create an educational inequity.

One of the main personality traits that often correlates with gender differences in programming is the programmer's learning style: whether they like to learn by tinkering and exploring or using a step-by-step approach, so we also wanted to compare the way participants desired to learn and their behaviors. We created a survey based on the survey in [30] in order to

try to determine whether participants were more likely to explore and tinker as a way of exploring the API or whether they were more reliant on information like tutorials or books. Unfortunately, the survey only had a reliability of $\alpha = 0.65$ for the questions about learning through exploring, and $\alpha = 0.5$ for the questions about learning using process-oriented information, both of which are less than the accepted reliability for surveys (0.7), so we will not report results for the survey.

6.4.3 Do participants take advantage of API information features?

This section presents results on how participants used features in the suggestions and documentation. Due to the structure of this study, we cannot evaluate the impact of specific features, so instead we explore three questions about feature use to provide insight into the value of the system design: 1) how did participants access information, 2) how much did they execute examples, and 3) how much do they use contrasting examples and the ‘show me’ button?

We expected participants to access the suggestions and documentation using all of the mechanisms provided, which we found to be true, as shown in Table 6.6. For the most part, participants accessed suggestions from the suggestion list (see Figure 6.1-A) and ‘?’ buttons (see Figure 6.2-A), which were both in or near the palette where users drag code blocks from. The list of suggestions was designed to help participants return to suggestions, which participants did: “I opened the tip [suggestion] because I had forgotten how to do it.”

We found that the majority of participants who accessed API information also executed examples in both conditions, but did so more with suggestions: 81.3% of participants who accessed suggestions executed an example at least once, while 68.8 % of participants who accessed documentation executed at least one example. Executing examples may suggest

that participants wanted to engage more deeply with the information in order to find out more about it. Participants who executed examples from suggestions executed on average 4.7 examples (SD = 3.5), while participants executed examples an average of 9.7 times from documentation (SD= 9). This may be because suggestions only provided two examples, while documentation often showed eight examples.

Because we designed suggestions specifically to provide contrasting examples and a button to help novices find code blocks, we measured how much they used those features. 44% of participants who accessed suggestions used contrasting examples and accessed the contrasting example for 1.8 suggestions on average (SD= 1.2). 38% of participants who accessed suggestions used the ‘show me’ button, and on average, clicked it 3.2 times (SD= 2.5). Since participants likely will not need these features for every suggestion, having over 30% usage and having participants return to use these features multiple times seems to indicate that participants found the features useful.

6.4.4 Threats to validity

There are two limitations to this study: the population we picked and the length of the study.

Condition	Way of accessing	% of accesses
Suggestions	Suggestion Panel	86.5%
	Code Annotation	11.1%
	Preview from Panel	2.4%
Documentation	‘?’ Button	45.1%
	Expanding Parameters	29.4%
	‘More Examples’	17.6%
	Next/Previous Buttons	7.8%

Table 6.6: Participants accessed the API information all of the different ways in both conditions

We recruited participants from a mailing list focused on STEM which draws from a sample of more interested and self-motivated learners than the general population. This may have meant that participants were more interested in technology and excited to explore the API than the norm. Furthermore, 94% of participants had programming experience of some form, including 82% who had been taught programming, which is above the norm for middle school children in the US.

While the results from this initial study are exciting, it is important to note that this study focused on a relatively short period of time and on API use, rather than learning. While the patterns of use suggest the potential for improved longer term learning, it will be important to explore how novice programmers engage with the Example Guru over a longer period. We need further studies to understand whether the Example Guru improves novices' comprehension of the API methods.

6.5 Discussion

Given that programmers across a broad range of skill sets describe learning or attempting to learn using 'just-in-time' strategies, effective situated support for API learning has the potential to improve programmer success and efficiency, particularly for novices. The results of our study suggest that the Example Guru approach has the potential to better support learning of APIs during artifact-based programming. Yet, there are places where further work is needed. First, our results found that females used fewer suggestions than males, leading to a potential learning inequity. Second, we hand-coded our rules for this study.

6.5.1 Learning APIs

To achieve mastery of a new API, novice programmers must continue to develop their skills over time. Yet, existing research suggests that novices reach a plateau in which they quickly learn to use a subset of the available capabilities within the system and then stop learning new skills [189, 233]. One recent paper [137] found an increase in the number of API methods used with experience. However, the increase was small after the initial period. Although measures of API method use cannot tell us whether the novice programmers actually have a full understanding of how the API methods work, programmers must first gain exposure and experience with the API methods. Thus, this work begins to address issues in API learning by improving the number of API methods that novice programmers explore and use.

We believe that some of the plateau effect may be due to a lack of appropriate learning mechanisms. While users may spend time focused on trying to learn a new system or API, programmers typically spend more time in artifact-based programming and ‘just-in-time’ learning [23]. In just-in-time learning, programmers seek out information that they know they need. The Example Guru approach shows promise in introducing novice programmers to API methods that they may not know exist. Rather than requiring that they know what methods to search for, the Example Guru observes their code and offers potentially relevant information. Participants accessed suggestions and used API methods from suggestions more frequently than documentation, creating more potential learning opportunities. It is important to note that participants accessed suggestions for API methods novices rarely use in common practice, but experts typically do include in their programs. Users in the documentation condition chose to explore API methods more often used by novices. Over a longer period, the increased exposure to and use of API information could lead to substantial learning gains.

Finally, our results suggest the potential for continued usage by those with varying skill levels. In both the documentation and suggestion conditions, participants with fewer than three hours of programming experience accessed and used API information more frequently than those with more than three hours of experience. The difference is much more dramatic in the documentation condition where only 32% of participants with more programming experience accessed the API information at all. In contrast, 77% of those with more than three hours of programming experience accessed the suggestions. Yet, there is still room for improvement. While 40% of the more experienced novices in our sample engaged with API methods from suggestions, 60% did not.

6.5.2 Gender and the Example Guru

Our results showed different usage patterns among male and female participants. Specifically, male participants accessed more suggestions than female participants, averaging 4.3 versus 2.2 suggestions accessed. This is a potentially troubling difference, as over time this can lead to an educational inequity. Based on the results of this study, we have little information about the reasons for this difference. Previous work suggests that females may prefer learning using step-by-step instructions, rather than through tinkering and exploring [102] and that females have a tendency toward comprehensive information processing versus males' tendency toward selective information processing [144, 145]. However, we note that this difference occurs based solely on the tip describing the suggestion and before users are in a position to do much information processing. This is an area where future work is needed in order to understand and address this difference.

6.6 Conclusion

This chapter describes the first design and evaluation of the Example Guru. The Example Guru leverages the previous exploratory studies of suggestions, rules and examples in order to address the main goal of this thesis: to encourage novice programmers to explore and use new programming skills during artifact-based programming through suggested examples. Specifically, this chapter addresses hypothesis 2, that suggesting example code will increase the number of new API methods novice programmers add, compared to existing static forms of support. In this case, novice programmers with suggestions explored and used significantly more new API methods than novice programmers with documentation. The contribution of this chapter is a type of system that can encourage novices to explore new API methods. We implemented the system within an animation context in order to evaluate it, but this system design is not specific to an animation context. This model of suggesting context-relevant unused code blocks can apply in other blocks-based artifact-based programming environments. It could likely also transfer to end-user programming contexts like web development or data analysis. Investigation of adults' perceptions of suggestions would likely be necessary to ensure that adults would prefer suggestions the same way. This chapter however, leaves two open questions that we will address in the following chapter: whether suggested examples can help novices to explore abstract programming concepts, and whether we can create content for this type of system semi-automatically.

Chapter 7

Large-Scale Suggestions: Semi-Automatic Generation

Note: Parts of this chapter will be published in *Interaction Design and Children* 2018 [93].

This chapter describes an approach for semi-automatically generating content for a suggestion-based help system. Chapter 6 showed that context-sensitive suggestions can encourage novices to explore more new API methods than static documentation. If we could create this type of suggestion with minimal human effort and for abstract concepts, we could generate suggestions for many systems and large systems without requiring significant expert time. This chapter addresses hypotheses 2 and 3: that a suggestion system can encourage exploration or programming concepts, and that we can generate the content for this type of system with less human effort than hand-creation.

The core idea behind the semi-automatic approach is that a system can generate candidate suggestions by grouping code examples. For instance, one group of examples might have several camera movements that happen simultaneously, leading to a suggestion about making

the camera zoom out. Another group of examples could include living creatures moving in multiple directions at the same time, leading to a suggestion about making a character jump diagonally. For animation code, we group code examples using two main metrics: 1) types of objects, like characters or props, and 2) types of actions the objects take, such as changing position or changing size.

To semi-automatically generate suggestions, an expert must first define the types of objects and types of methods for the programming context. Once they have defined the types, the system can then generate any number of suggestions through the following steps: 1) system extracts code example snippets from a code repository, 2) system groups code examples using heuristics, 3) human moderates, 4) system generates a script that checks whether novice code should receive the suggestion. This chapter describes each step in detail.

To evaluate our approach for semi-automatically generating suggestions, we perform two preliminary evaluations: 1) we compare the semi-automatically generated content to a hand-authored set created previously for a separate study, and 2) we ran a study in which we compare children who had access to suggestions and children who had access to tutorials for 30 minutes in an artifact-based context. The semi-automatically generated suggestions cover all but two of the hand-authored set and also generated an original set of suggestions. Children, on average, received 9 semi-automatically generated suggestions, accessed 2.6 suggestions, and used 0.8 suggestions in just 30 minutes. They accessed 3.7 times more suggestions than tutorials.

The contribution of this chapter is a semi-automatic suggestion generation approach that creates textual suggestions describing a potential change to improve a child's program, examples that demonstrate how to implement the suggested change, and rules that determine when to offer each suggestion. We describe this system within the context of an animation

programming environment, but the potential to extend this for other programming tools or creative contexts has far reaching implications for children. Our evaluation demonstrates promising access and usage of semi-automatically generated suggestions.

7.1 Related work

The overarching ideas in this chapter relate most closely to existing support for learning at scale.

A variety of researchers have designed ways to help learners of content and software at a large scale, both in task-based and artifact-based contexts. There has recently been a significant push for learning at scale, such as using intelligent tutors and online courses that have specific tasks and solutions. In order to make these useful for large populations of learners and a large number of topics, researchers have worked to develop content, hints [177, 210], and feedback [81, 96] for users in automated and semi-automated ways. Automatically generated tutorials can help novices learn programming [75], as well as similar types of technical skills like photo-manipulation [44, 69] with less effort from an expert. However, these systems require a known solution that they can support learners in working towards.

Our system is most closely related to support for artifact-based contexts, like reuse and recommendations. Some systems provide support for reusing others' programs, like remixing in Scratch [196], or Looking Glass [72]. There are also some systems that recommend commands based on how communities often use them [119, 136]. Our approach is unique in semi-automatically generating motivating suggested examples for programming concepts in an artifact-based context.

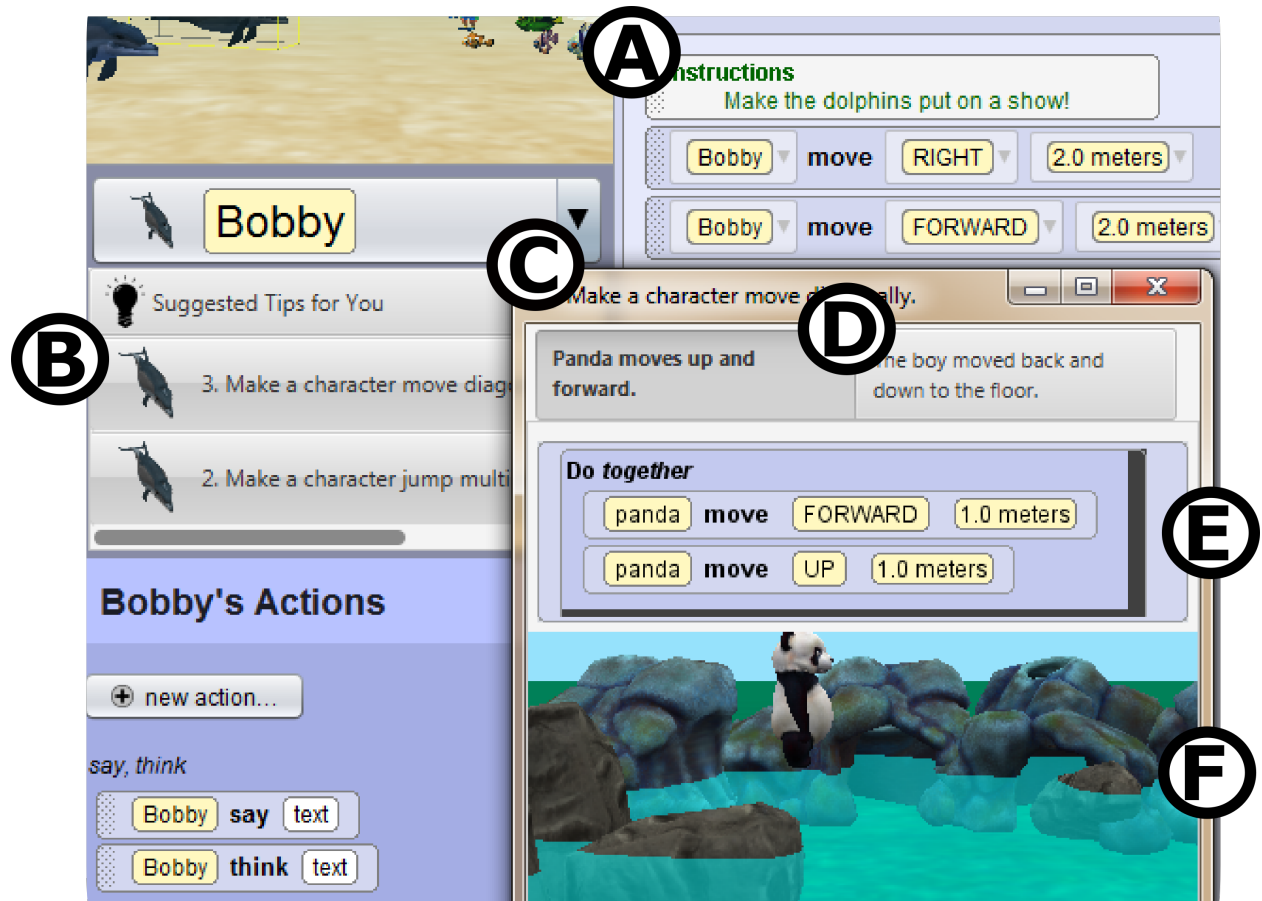


Figure 7.1: (A) Looking Glass programming environment. (B) List of suggestions. (C) An opened/accessed suggestion. (D) The two code examples with the primary one selected. (E) The code for this suggestion, with the *do together parallel execution* block emphasized. (F) Preview execution of the code.

7.2 Programming environment & suggestion system

We implemented the semi-automatic suggestion generation approach for the Example Guru within the Looking Glass programming environment.

7.2.1 Looking Glass programming environment

Looking Glass provides objects to create scenes and blocks that control the objects. A scene contains characters (i.e. people and animals), props (i.e. trees, couches, volleyballs, snowboards), and the scene objects (i.e. the ground, the camera). To create animations, users can drag and drop blocks of code that are either actions (i.e. *move*, *say*, *resize*, *disappear*), or programming constructs (i.e. simple parallel execution *Do together*, or a *loop*).

7.2.2 The Example Guru (final version)

To make suggestions to novice programmers, the Example Guru has three components: 1) rules, which are scripts that analyze novice programs, 2) a suggestion for each rule that introduces a new concept, and 3) a pair of examples that demonstrates the concept. This chapter aims to automate the process for generating suggestion ideas and the rules that check programs for the opportunity to make suggestions. Experts still performed the selection of the concepts to suggest (*Do together* and *Repeat loop*), the specific examples to show, the descriptions of the examples and suggestions.

The version of the Example Guru described in Chapter 5 suggested API methods. For this chapter, we wanted to generate suggestions for two programming concepts: parallel execution (*Do together*) and simple loop (*Repeat loop*). In order to support use of examples for these abstract programming concepts, we iterated on the design of the Example Guru suggestions and examples. We made one minor change to the suggestions: instead of having two contrasting examples to show differences, the suggestions for abstract concepts show two examples with the same concept. We found that this helped users to reinforce how the concepts worked. Contrasting examples for these concepts actually confused users because they had

trouble noticing the differences. We also made a set of minor changes to the suggestion list and suggestions, based on formative one-on-one testing with 33 novice programmers [92].

Based on observation and informal interview, we made two main changes to the suggestion panel: the suggestions are numbered and they do not have a hover preview. When participants access suggestions from the panel, they often want to return to one they saw previously. In the past iteration, the suggestions in the list were colored differently to distinguish them, but that still might make it difficult to find a specific one. Instead, this iteration of the Example Guru has numbers for the suggestions to make it somewhat easier to find them again. This version also removed the hover preview, which enabled users to watch the example preview by hovering over the suggestions. In practice, participants often hovered their mouse over the suggestions but almost never wanted to view the example video at that point. The text descriptions already provide enough information for users to determine if they want to view the suggestions.

We also made several modifications to the opened suggestion: 1) the example code is at the top of the example, 2) the suggested code block has a shadow, and 3) extra information about the code and where to find it appears as a tooltip when hovering over the code example. We moved the code example to the top of the suggestion box because we found that participants were more likely to pay attention to the code snippet at the top. We also explored a variety of code emphasis methods to encourage novices to pay attention to the suggested code block, such as colors, annotations, outlines, and shadows [92]. Participants were often confused if the emphasis method made the code look different than it looked within the programming environment. Adding a drop shadow makes the code block pop out without modifying its appearance. Finally, some novices seemed to benefit from having more formal, written information about the suggested code. However, having it visible initially makes the content overwhelming and makes it less clear where novices should focus. To support this type of

user without adding potentially overwhelming content, we added this information as a tool tip. The tool tip also describes where to find the suggested code block if they cannot find it. While this helped, having an easier way to discover this capability would likely be useful in future iterations of this type of system.

7.3 Suggestion generation approach

The goal of this approach was to generate suggestions and rules for a system like the Example Guru with minimal human effort per suggestion. Our approach requires initial setup by an expert to define the ways the system will group code. After that, the approach can generate any number of suggestions and rules with four phases: 1) example extraction, 2) example grouping for candidate suggestions, 3) human moderation (example selection + labeling), and 4) rule generation. The approach takes a repository of programs as input and outputs suggestions and rules.

7.3.1 Input repository

We used a repository of 1751 blocks programs containing: 1) 780 programs created by expert researchers, and 2) 971 programs created by non-experts. Researchers in our lab created the expert programs previous to this work for other purposes. The non-expert set of programs set contains programs created in past user studies and programs shared to the Looking Glass online community by non-lab members [121].

We split our repository into a design and prototype set in order to be able to avoid over-fitting our approach to a specific set of programs. To design and iterate on our approach we used 25% of each of the novice and expert program sets. We used a small portion of the repository for development of the approach in order to save the majority of the repository for evaluation.

We used the remaining 75% of program sets to generate the 80 suggestions we evaluate in this work. Due to the types of code constructs in our repository, we chose to generate suggestions for: *parallel execution*, *loop*, and nested combinations of *parallel execution* and *loop*.

7.3.2 Initial setup

This approach requires initial setup by an expert in order to select: 1) which code concepts to suggest, and 2) how the system should group code snippets and generate rules. The expert needs to select which code blocks the system will suggest to the novice programmer. Many systems may benefit from generating suggestions for all types of code. We wanted to generate suggestions for young novice programmers, so we chose to generate suggestions for the parallel execution block known as the *Do together* and the loop block called *Repeat*. These will be used in the **Example Extraction** phase, which selects the example code snippets.

In order to group code snippets and generate rules, the system needs heuristics for determining whether code snippets will have similar animations. An expert defines the similarity heuristics. Groups of snippets with similar animations, like snippets that all make the camera zoom, can become suggestions. Rules are generated by finding similar animations that lack the suggested concept, like the *Do together*.

For Looking Glass, we defined types of objects and methods that would have similar animations. Objects have three types: sentient characters, sentient character’s joints, and props, as shown in Table 7.1. Actions have eight groups: communicate, sound, position, orientation, appearance, size, timing, and vehicle. The **Example Grouping** and **Rule Generation** sections below will cover in more detail how those phases use the object and action types.

Set	Type	Specific Examples
<i>Objects</i>	sentient	person, dog
	(sentient's) joint	wing, arm, neck
	prop	sofa, camera
<i>Actions</i>	communicate	say, think
	sound	playsound
	position	move, walk
	orientation	turn, roll
	appearance	setcolor, appear
	size	resize, setwidth
	timing	delay
	vehicle	setVehicle

Table 7.1: Objects and actions used for binning snippets.

7.3.3 Example extraction

First, a system needs to extract snippets from a repository of code. The code examples must contain the code that will be suggested. We generated suggestions for the *Do together* and the *Repeat* code blocks, so our approach extracted only snippets including one of these blocks. We chose to select a snippet containing only the concept, like the *Do together* and the code within the *Do together*, and not any surrounding code. In other contexts, more surrounding code might be useful. The extracted snippets feed into the **Example Grouping** phase.

7.3.4 Example grouping

The Example Grouping phase takes all of the code snippets for a concept and sorts them into groups of code with similar output. These groups are candidate suggestions. The grouping algorithm uses the object and action types defined by an expert in the **Initial Setup**, shown in Table 7.1. The grouping algorithm also uses the number of each type of object and action. The algorithm determines if two code snippets should be in the same group based on the following criteria:

- All code snippets in a group have the ‘same’ number of objects of each type: 0, 1, or 2+. If snippets have two or more objects of the same type, the approach considers them the same. The suggestion in Figure 7.1 has one sentient object, which is a dog in one example and a person in another. A group of examples could also contain an example with three dogs and an example with two people.
- All code snippets in a group have objects of the same type performing either 1 action of the same type, or two or more actions of the same type. The suggestion in Figure 7.1 has one sentient object performing two position actions. The code group for this example could also contain examples with one sentient object performing more than two position actions.

Taking these two criteria together, the system will result in groups of examples where the object types and action types align to create similar animations. In Figure 7.1, the example group contains many examples that have one sentient object moving in multiple directions within a *Do together*, which makes the movements happen at the same time. This is a valuable suggestion because novices who have a character moving in multiple directions sequentially would often rather have the actions happen together.

Other researchers have also worked on finding clones or near-clones of snippets of code, either to detect cheating or maintainability issues [17], or to scalably grade assignments [66]. This is a very specific type of related code search, in which the structure typically needs to be highly similar, with the possibility of certain types of modifications. For maintainability, clone detection is typically based on strings, abstract syntax trees, or metrics [19].

Depending on the number of programs available in the repository and the number of concepts the system is going to suggest, this grouping method may return a large number of groups. The size of the groups gives some indication of how often programmers use concepts in specific

ways. This frequency can be used to prioritize the order in which humans moderate groups or to determine the set that advance to the moderate phase. In our system, this grouping process resulted in 158 groups of examples with more than two code snippets for the *loop*, *parallel execution* and nested combinations. We chose to only use groups with more than two example snippets because many of those with only two had two of almost exactly the same animation.

7.3.5 Human moderation

The main objectives of the human moderation step are to select the two examples that the suggestion will show, label them, and give the overall suggestion a title. In order to select examples with similar outputs and provide motivating descriptions, we need a human to complete these steps. While groups of examples may be large, the moderator does not necessarily need to look through all examples in a group if they quickly find two appropriate examples. We created and iterated on criteria for how an expert should moderate groups of examples, which are listed in Table 7.2.

1. *Select primary and secondary examples.* This approach benefits from having a human select the examples for two main reasons: 1) humans can select examples with the best and most visible animations, and 2) humans can filter out inappropriate animations.
2. *Write titles.* It is important that a human authors the suggestion title and example descriptions because the title should motivate the novice programmer to look at the suggestion. It needs to describe the output animation and how it could improve a program, rather than describing the code itself. For animation, a suggestion title could ideally be something like “Make your character jump multiple times,” rather than a description of the *move up* and *move down* code blocks.

Required	Actions visible in execution preview Correct use of construct No sexual, vulgar, or violent content No errors
Ideally	Code should use minimal extra arguments Two examples should use constructs differently Objects should have intuitive names Two example should have different scenes
Exclude	Examples that do not fit in suggestion Identical examples Group if it does not have two examples

Table 7.2: Human moderation criteria

In our implementation of this approach, we had a non-author researcher perform the human moderation phase. However, we note that the moderator primarily evaluates the output of the code rather than the code itself. A system be able to leverage a crowd or a community of novice programmers to further reduce expert effort for suggestion generation.

Example groups advanced to the generate rules phase if the moderator was able to select a primary and secondary example and give a title to the suggestion. At the end of our human moderation phase, 80 of the 158 groups moved on to rule generation. The excluded groups did not have at least two examples that fit all of the required criteria in Table 7.2. Many of these were due to version issues that caused errors, which systems should check automatically in the future. Our human moderation phase ended with 7 suggestions for the *loop* and 73 suggestions for the *parallel execution* blocks. The difference in the number of suggestions is a result of much more frequent usage of *parallel execution* in the code repository.

7.3.6 Generate rules

Our semi-automatic approach generates rules using the object and action types in Table 7.1, as initially defined by an expert. The approach generates rules that, at a high level, find

novice code with the same object and action types as an example group, but that lacks the suggested concept. For the ‘Make characters move diagonally suggestion’, the rule looks for code that has a sentient object moving in multiple directions that does not use a *Do together*.

In order to generate a rule, the system needs to extract the following information from a suggestion and examples:

- the concept being suggested, so that the rule can look for code that does not contain it
- the number and types of objects to look for
- the number and types of actions for each of the objects

In our implementation, the rule generation process used the full set of examples from the group for each suggestion. It then generated code in Java to fill in the concept, number and types of objects, and number and types of actions that the rule should look for. We used the JavaPoet API to generate Java code for the rules programmatically [99]. The system generated a rule for each of the 80 suggestions.

We next answer two questions about the set of semi-automatically generated suggestions and rules: 1) how do they compare to a hand-authored set of suggestions, and 2) how do young novice programmers interact with them in an artifact-based context?

7.4 Comparison of semi-automatically generated to hand-authored suggestions

We wanted to know how the semi-automatically generated suggestions compared to a set authored by an expert. As a preliminary analysis, we compared the suggestions generated

by our approach to a set of existing hand-authored suggestions for the Example Guru. We focused on comparing the output of the approaches rather than the time and effort of the approaches to get a sense for whether the algorithms will provide reasonable results. Further work should likely use these results to fine tune the semi-automatic approach and then evaluate the savings in expert time.

7.4.1 Comparison methods

To compare our generated suggestions to a hand-authored system, we used a set of manually authored suggestions from another study that suggested programming concepts [83]. The hand-authored set of suggestions included 5 suggestions for *loop* and 6 suggestions for *parallel execution*, as shown in the left side of Table 7.3.

We consider two suggestions to be equivalent if they either 1) involve the same types of objects and actions, or 2) the types of objects or actions in one suggestion are a more specific version of those in the other suggestion. We use this method of comparison because the hand-authored suggestions were designed to be more general, encompassing all objects or a broader set of actions.

7.4.2 Comparison results

Our semi-automated approach generated matching suggestions for all but two of the hand-authored suggestions, and an original set of suggestions beyond the hand-authored set. Table 7.3 shows the the generated suggestions in the right column aligned with their equivalent hand-authored suggestions in the left column.

For the equivalent suggestions, the generated approach often either generated one suggestion for characters, or multiple suggestions with the different object types. For instance, for the

hand-authored ‘make objects turn back and forth multiple times’, the system generated a similar ‘make a character turn back and forth multiple times’. For the hand-authored ‘make objects move in two directions at the same time’, the semi-automatic approach generated three suggestions for a character, a prop, and the camera.

Our semi-automatic approach also generated a new set of suggestions that were not in the hand-authored set. This set contained suggestions about the appearance of characters, props and scenes, and complex joint actions, as shown in the bottom of Table 7.3. The expert likely did not create suggestions about complex joint animations due to the low level of experience of children participating in Looking Glass user studies. However, suggestions about simultaneous turning and moving, as well as simultaneous appearance changes are applicable to young novices and did end up being relevant to novices in a user study, as discussed in the next section.

Hand-authored suggestion: “Make...”	Semi-Automatically Generated: “Make...”	# accessed/received
objects move in two directions at the same time	a character move diagonally. a prop move diagonally. camera move diagonally.	8/18 1/3 1/2
an object bounce multiple times	a prop move back and forth mult. times! a character jump multiple times.	1/3 4/18
objects turn back & forth mult. times	a character turn back and forth mult. times.	5/11
characters talk & walk at the same time	characters move and say at the same time. a character move and talk at the same time.	0/3 1/6
objects move together!	mult. things move at the same time. mult. characters move at the same time. mult. characters move away from something at the same time. 3 more similar suggestions	0/1 7/18 0/1 0/0
something else happen at the same time as resize	a character get bigger or smaller while moving.	0/0
objects flash ... mult. times	N/A	N/A
joint actions happen mult. times	prop’s joints turn back & forth mult. times. a character’s joint turn back and forth many times.	0/0 0/0
an object disappear at the same time as something else	Change multiple things’ visibility at the same time.	0/0
objects do the same thing at the same time!	mult. characters act at the same time.	0/0
simultaneous actions happen mult. times!	N/A	N/A
NA	8 simultaneous turning and moving suggestions: 6/18, 2/8, 2/9, 4/10, 0/3, 0/1, 1/7, 0/0	
NA	7 speaking while moving suggestions: 4/7, 0/1, 1/7, 1/4, 1/7, 0/0, 0/0	
NA	6 appearance suggestions: 0/3, 0/1, 0/2, 1/3, 1/2, 1/2	
NA	15 Simultaneous joint movements suggestions	0/0
NA	9 Jumping and joint movements suggestions	0/0
NA	7 joints turn while a character moves suggestions	0/0
NA	5 movements while talking suggestions	0/0
NA	4 other misc. suggestions	0/0

Table 7.3: Left: Hand-authored suggestions. Right: Semi-automatic suggestions and the numbers of suggestions received and accessed by children in our study.

7.5 User study: novices' interaction with semi- automatically generated suggestions vs. tutorials

We wanted to compare semi-automatic generated suggestions with tutorials through this study. Our comparison with tutorials aimed to understand how novices accessed and used suggestions and tutorials. Over time, we expect that novices who access and use suggestions or tutorials will also gain new understanding. Despite our short study, we also chose to evaluate learning through transfer tasks because our suggested concepts were relatively simple. Specifically, this study aims to answer five questions: 1) how many and which suggestions did participants receive? 2) how did tutorial and suggestion participants differ in the content they accessed, 3) how did tutorial and suggestion participants differ in new usage of concepts after access, and 4) did tutorial and/or suggestion participants show evidence of learning, and 5) did participants have different perceptions of programming?

7.5.1 Tutorial control condition

Many existing programming environments provide tutorials that show the steps to create programs and have text descriptions of how the demonstrated code works, either externally or within the programming environment. We created tutorials based on Scratch [196], which provides a list of 13 tutorials in a collapsible side panel within the programming environment. Figure 7.2 shows our tutorials, which are designed the same way as a study that compared Scratch-like tutorials to code puzzles [77]. Like typical tutorials, ours have screen-captured videos that show where to find code blocks, how to add them, and the execution of the code (see Figure 7.2-C). Written instructions explain how the code in the tutorial works (see

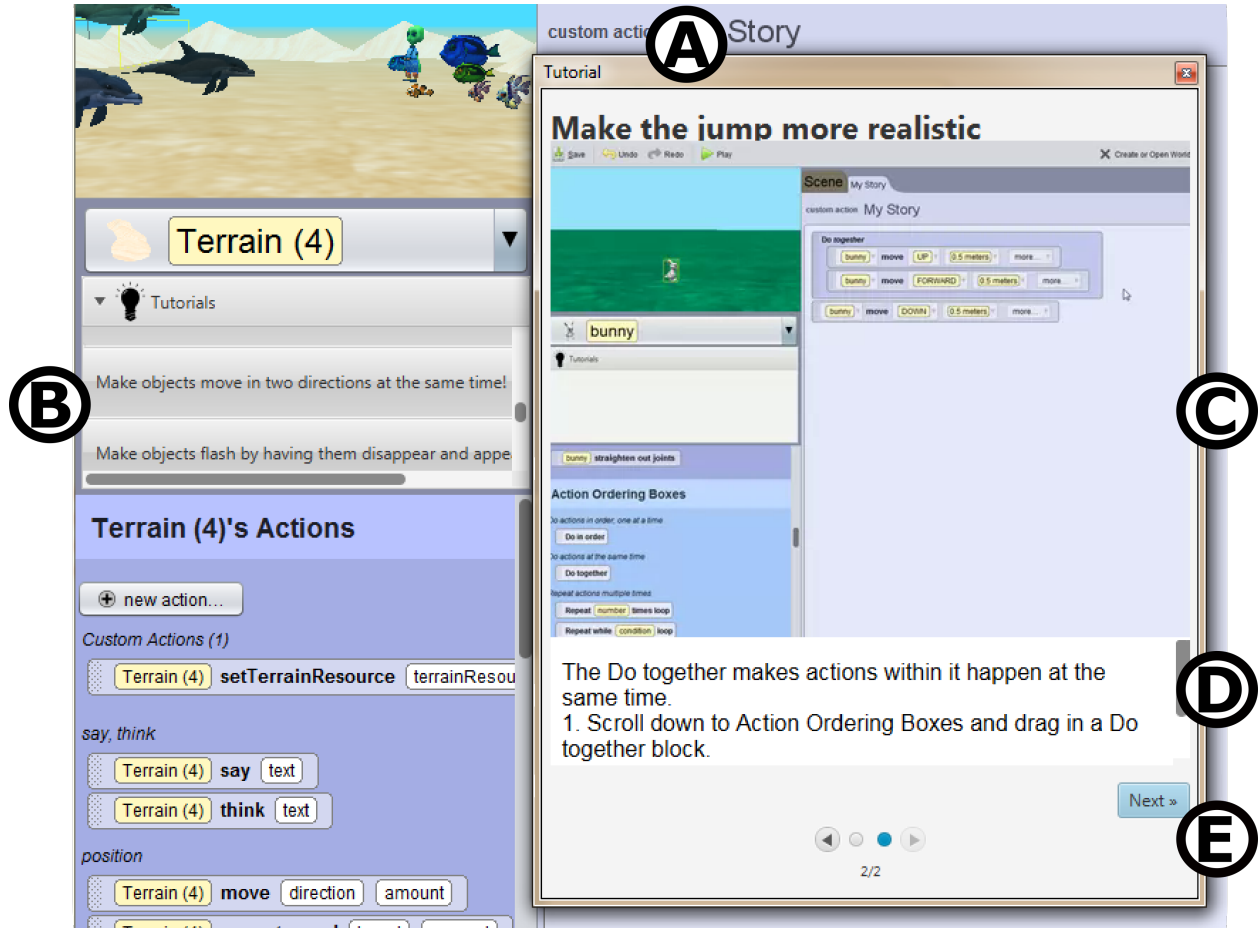


Figure 7.2: (A) An accessed tutorial. (B) List of tutorials, which always has the same set of 13 tutorials for everyone in the condition. (C) A short video that shows how to complete the step. (D) Written instructions. (E) Next button to go to the next step of the tutorial.

Figure 7.2-D). We used content for the tutorials from hand-authored suggestions for *parallel execution*, *loop*, and those concepts nested, from a previous study[83], as listed in Table 7.3.

7.5.2 Study protocol

In this paper we analyze data from: 1) baseline project, 2) system familiarization tasks, 3) supported project, 4) transfer tasks, and 5) surveys.

Baseline project

To measure what novices would explore without support, all participants worked on an animation project for the first 15 minutes of the study. Participants could select one of nine pre-made scenes to code their own animation. Participants did not have access to suggestions or tutorials during this phase. For those with no programming experience, a researcher briefly demonstrated the mechanics of drag and drop programming.

System familiarization tasks

To ensure that participants knew how to access suggestions or tutorials, participants completed two familiarization tasks. Each task lasted 5 minutes and asked the user to modify an API method call using a suggestion or tutorial, depending on the participants' condition. A researcher confirmed that each participant accessed at least one suggestion or tutorial before participants moved on. If participants got stuck, a researcher guided them to the support or helped them complete the task, as the goal of these tasks was system familiarization, not evaluation.

Supported project

To investigate how participants would receive, access, and use suggestions or tutorials during artifact-based programming, participants had 30 minutes to code with access to suggestions. A researcher told participants that they were not required to use suggestions in this phase. Participants first created a performance animation and then could select other scenes to make animations with. This was the main part of the study that tells us where novices access and use the suggestions or tutorials.

Concept transfer tasks

To investigate whether suggestions could be used as support for learning in addition to exploration, participants completed 4 transfer tasks. The tasks tested *parallel execution*, *loop*, *parallel execution* nested within *loop*, and *loop* nested within *parallel execution* understanding. Participants received these tasks in a balanced order using Latin squares and had 3 minutes to complete each task. Participants could not access suggestions or tutorials during this phase.

The tasks were closely based on those from another study evaluating these concepts [78]. Each transfer task included code and instructions for how the programmer needed to change the animation. For each of the tasks, the solution was for the participants to use one or two of the concepts from the suggestions in particular locations. For instance, one transfer task asked the participants to make an iceberg shake and a yeti fall at the same time. The user had to find the *parallel execution* block, add it to the program, and drag the appropriate blocks inside of it.

Surveys

Participants rated their coding experience and experience with suggestions on 6 Likert scales shown in Table 7.5.

7.5.3 Participants

We recruited 44 participants from a local science-focused mailing list. Participants filled out a demographic survey at the beginning of the study that asked about their age, gender, and previous programming experience. We excluded four participants: three because they had

participated in similar studies with our lab and one due to technical issues that prevented them from receiving suggestions. We analyzed the data for the remaining 40 participants: 24 males and 16 females who ranged in age from 8 to 15 ($M = 11.2, SD = 1.6$). Although we recruited participants with less than three hours of programming experience, the surveys revealed that many had more than three hours. We randomly assigned participants to conditions balancing for gender, age, and programming experience, as these factors have in past studies affected early learning in coding [78]. We also use age and programming experience as covariates throughout our analysis.

7.5.4 Data and analysis

To better understand whether the semi-automatically generated suggestions were relevant and useful to young novice programmers, we evaluated the number and type of suggestions participants received, accessed, and used, and how participants performed on the transfer tasks.

Suggestions received

The number and type of suggestions each participant received and the number of generated suggestions novices received tells us whether the generated suggestions applied to novice programs. Participants received suggestions based on the context of their code and whether they had used a concept yet. During the 30 minutes of artifact-based programming, each time a novice programmer executed their code, rules checked novice programs for opportunities to make suggestions.

Suggestion access

To evaluate whether participants wanted to explore suggestions, we measured how many suggestions novices clicked on to open. We count an access as the participant clicking to open a suggestion. We do not count repeat accesses of the same content. While exposure to new content can be beneficial, we also wanted to know if participants used the new code blocks in the suggestions.

New usage of concepts after access

We define ‘use’ of a suggestion as inserting the programming concept from that suggestion for the first time after accessing it. Because participants may have accessed multiple suggestions for a certain concept, we report whether participants added a *Do together* or a *Repeat loop* for the first time after accessing any of the suggestions for that code block.

Evidence of learning

To determine whether participants likely understood constructs they accessed or used in the supported project phase, we measured participants’ success on the concept transfer tasks. We selected our scoring metric to reflect whether participants realized which programming construct they needed to use. Each transfer task received a score of 0% if it did not have any of the correct programming constructs added and 100% if they were exactly correct. For the two tasks where participants only needed to insert one type of concept (either *loop* or *parallel execution*), they received a score of 50% if they inserted that construct but did not fully complete the task. For the other two tasks which required inserting both the *loop* and the *parallel execution* blocks, participants received 33% if they inserted one and 66% if they inserted both but did not complete the task fully correctly.

7.5.5 Study results

Our results address our five questions about the types of suggestions participants received, which suggestions and tutorials participants accessed and used, evidence of learning, and participant perceptions.

How many and which suggestions did participants receive?

Because novices likely will not find all suggestions relevant, in a thirty minute session of artifact-based programming, we would hope that novice programmers would at least receive several suggestions. On average, participants received 9 suggestions ($SD = 4.5$). All participants received at least one suggestion and 80% of participants received at least 5 suggestions. The most suggestions participants received was 17.

Participants received 29 of the 80 available suggestions, as shown in Table 7.3. Participants received 11/19 of the suggestions that aligned with hand-authored suggestions. Almost all (18/20) participants received three of the suggestions: make a character move diagonally, make a character jump multiple times, and make multiple characters move at the same time. Interestingly, these often focused on positional actions. Of the 61 suggestions generated that did not align with the hand-authored set, 18 were suggested to at least one participant. Six of the eighteen were about changing the appearance, seven were about speech or speech while turning or moving, and the remaining eight were about combinations of turning and moving.

The types of suggestions participants did not receive primarily focused on characters moving their joints. Of the 51 suggestions that no participants received, 34 focused on complex joint animations. While many programs in our repository included complex joint movement

animations, most novice programmers in our study did not reach this skill during the short lab session.

How did tutorial and suggestion participants differ in the content they accessed?

Participants accessed significantly more unique suggestions than tutorials with a large effect ($F(1, 35) = 9.5, p < 0.01$, partial $\eta^2 = .21$). Suggestion participants on average accessed 2.6 suggestions ($SD = 2.6$), while tutorial participants on average accessed 0.7 tutorials ($SD = 1$). There was no significant effect of age or programming experience. More suggestion participants accessed at least one suggestion: fifteen suggestion participants (75%) accessed at least one suggestion, while only 9 tutorial participants (45%) accessed at least one tutorial, though this difference was not significant ($\chi^2(1, N = 40) = 2.6, p = .11$). Taking into account only those who accessed at least one suggestion or tutorial, suggestion accessors still accessed significantly more unique suggestions than tutorial accessors with a large effect size ($F(1, 19) = 4.9, p < 0.05$, partial $\eta^2 = .2$).

We looked at how often participants accessed support for the two concepts: *parallel execution* and *loop*. Of tutorial participants who accessed tutorials, 7/9 accessed *parallel execution* tutorials and 3/9 accessed *loop* tutorials. Of suggestion participants who accessed suggestions, 14/15 accessed *parallel execution* suggestions and 12/15 accessed *loop* suggestions. This indicates that participants accessing suggestions seem to have accessed information for more concepts than tutorial users. This may be because most of those who accessed tutorials only accessed one tutorial, as shown in Figure 7.3, while most suggestion accessors accessed at least two suggestions.

To provide more in-depth information about users' behavior, we compared which of the available tutorials and suggestions participants accessed. Table 7.3 shows the number of

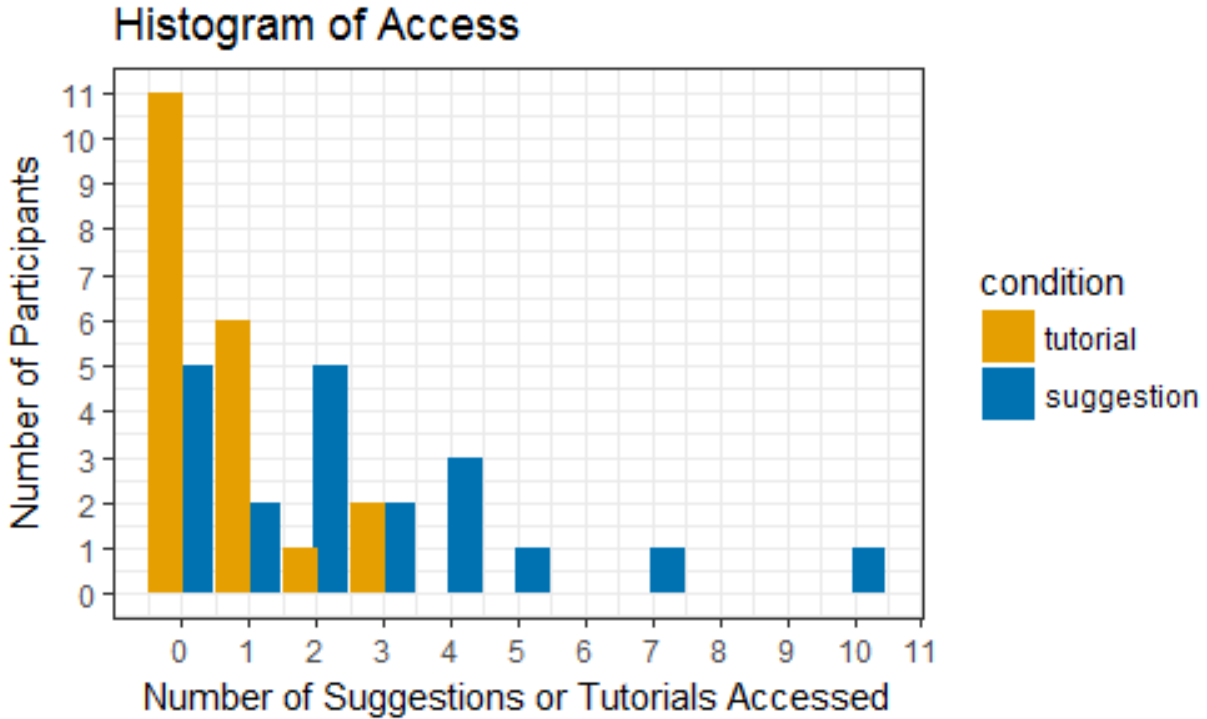


Figure 7.3: The number of suggestions and tutorials participants accessed.

participants who accessed tutorials and generated suggestions. It also shows the hand-authored suggestions, which we will discuss later. Tutorial participants received the 11 hand-authored suggestion ideas statically as a set of tutorials they could access throughout the project phase. Of those 11 tutorials that they could access, 7 (64%) of them were accessed by at least one participant. 30 suggestions were suggested to participants. 21 (70%) were accessed by at least one participant (5 *loop* and 16 *parallel execution*). The types of suggestions and tutorials accessed did not always align. Further exploration is needed to understand more about why participants accessed these specific types of information.

How did tutorial and suggestion participants differ in new usage of concepts after access?

Suggestion participants generally added more new programming concepts than those in the tutorial condition. Overall, suggestion participants used more new constructs for the first time after accessing suggestions ($M = .8, SD = .9$) than tutorial participants ($M = .25, SD = .4$). This difference was significant with a large effect size ($F(1,35)=6.1, p < 0.05$, partial $\eta^2=.15$). There were no significant effects of age or programming experience.

There was not a significant difference in the number of participants who used new concepts after accessing suggestions and tutorials: five of nine tutorial participants and ten of fifteen suggestion participants inserted a programming concept into their program for the first time after accessing a suggestion or tutorial for it ($\chi^2(1, N = 40) = .01, p > .1$). Participants also inserted new programming constructs into their programs after accessing suggestions or tutorials at similar rates. Suggestion participants inserted 60% of new concepts they accessed suggestions for ($SD = 50\%$), while tutorial users inserted 50% of concepts they accessed tutorials for for the first time ($SD = 50\%$) ($F(1, 19) = .15, p > .1$, partial $\eta^2 = .01$). Since the rates of new usage after accessing are similar, the larger number of overall new concepts inserted is likely related to participants' accessing more suggestions than tutorials.

We looked at participants usage of each of the concepts for the first time after access. Of the 14 participants who accessed suggestions for *parallel execution*, 10 of them subsequently used a *parallel execution* block for the first time (71%). Of the 7 participants who accessed *parallel execution* tutorials, 5 then added them for the first time (71%). Suggestion participants on average added 3.4 *parallel execution* blocks after accessing ($SD=2$), while tutorial users inserted on average 8 *parallel execution* blocks ($SD = 6.3$). Of the 12 suggestion participants who accessed *loop* suggestions, 6 added *loops* for the first time afterward (50%). None of the

3 tutorial users who accessed *loop* tutorials inserted loops for the first time. Suggestion users on average added 1.8 new *loop* blocks (SD=1).

Did tutorial or suggestion participants show evidence of learning?

We wanted to know if there was an overall effect of having access to tutorials or suggestions on ability to solve the transfer tasks. Participants did not significantly differ on transfer task scores with condition ($\Theta = .15, F(1, 35) = 1.2, p > .1$). There was a marginally significant effect of age on performance ($\Theta = .34, F(1, 35) = 2.7, p = .05$). There was also no difference for the subset of participants who accessed suggestions or tutorials ($\Theta = .13, F(1, 19) = .52, p > .1$). There was a no effect of age in this case. Table 7.4 shows the average scores for the transfer tasks. Because the overall MANCOVA was not significant, we did not do follow-up tests for the individual transfer tasks. We also repeated this comparison for participants with not more than three hours of programming (our original intended participant pool) and found a similar result ($\Theta = .11, F(1, 26) = .66, p > .1$). The biggest differences are for the *loop* task, which is the construct where few users accessed tutorials and no tutorial participants used *loop* blocks for the first time after accessing tutorials.

As post-hoc follow-up tests, we explored whether accessing or using suggestions or tutorials may have had a relationship with success on transfer tasks. There was a significant moderate correlation with *using* new programming concepts after accessing suggestions or tutorials and average transfer task score ($r = .5, p < .01$). Using a MANCOVA with Pillai's trace and with whether participants added *parallel execution* or added *loop* blocks to their programs as independent variables, we found that both were significant with relationship to score ($V=.47, F(1, 33) = 6.7, p < .001$ and $V=.27 F(1, 33) = 2.8, p < .05$ respectively). We did not find a significant correlation between *accessing* suggestions or tutorials and the average transfer task score ($r = .3, p = .12$).

Task	All Tutorial	Accessed Tutorial	All Suggestion	Accessed Suggestion
<i>Parallel</i>	50%	61%	58%	67%
<i>Loop</i>	18%	22%	33%	40%
<i>Parallel{loop}</i>	32%	37%	42%	49%
<i>Loop{parallel}</i>	26%	37%	43%	51%

Table 7.4: Transfer task scores

Did participants have different perceptions of programming?

We surveyed participants about two main things: 1) how they perceived programming in the study, and 2) their perceptions of suggestions and tutorials (see Table 7.5). Participants gave high ratings for all questions, with averages ranging from 4.9 to 6.5 on a scale from 1-7. Participants did not significantly differ in their responses to these questions based on condition. The covariates of age and programming experience were also not significant. Suggestion participants did find the study marginally more exciting on the *dull*→*exciting* scale than tutorial participants with a medium effect size ($F(1, 36) = 2.89, p = .1$, partial $\eta^2=.08$).

Question	Tutorials	Suggestions
disgusting→enjoyable	M=6.1, SD=1.3	M=6.5, SD=.8
dull→ exciting	M=5.3, SD=1.8	M=6.1, SD=1.1
unpleasant→pleasant	M=5.9, SD=.9	M=6.3, SD=1
boring→interesting	M=6, SD=1.5	M=6.4, SD=1.1
not→very useful†	M=5.6, SD=1.5	M=4.9, SD=2
very confusing→ very understandable †	M=4.9,SD=2.3	M=5.5, SD=1.6

†Participants who did not use them were not included

Table 7.5: Participants responded to Likert scales from 1-7. $\wedge p = .1$

7.6 Threats to validity

Our population contained primarily children whose parents have invested in their education by signing them up for a STEM-focused mailing list. The participant population also had more programming experience than expected, though they were still novices. Because we recruited participants with extremely limited programming experience, we expected to be able to assume that they had not mastered the programming concepts suggested. While some participants reported more extensive programming experience, their behavior did not always support this. Most participants still did not use *parallel execution* or *loop* blocks before accessing suggestions or tutorials. The majority of our participants likely still fit into our target group: novices who would not explore and use new programming concepts on their own. However, we cannot be certain that our participants had not been exposed to the suggested concepts prior to this study. Though our participants had a larger range of programming experience than anticipated, the range of skills was still small relative to the general programming audience. This study may not generalize to adult or experienced programmers.

7.7 Discussion

We discuss: 1) how our approach generalizes, 2) the potential of suggestions to increase use and learning, 3) the effect of the code repository on suggestion generation, and 3) the potential of this approach to support personalization.

7.7.1 How our approach generalizes

We believe this approach can apply to a variety of programming contexts. The two main criteria for this type of approach are that the user is motivated by the outcome of their code and that it is possible to group code by output. We can imagine many programming contexts where both conditions apply, such as web programming, phone App development, and game development. In order to be able to find and group the snippets, an expert programmer has to define the types of objects and methods used to group snippets. However, defining the types of objects and methods is much less open-ended than trying to imagine suggestions and create examples. Furthermore, once these are defined, they likely will not need to change and can be used to continually find new suggestions and examples as programmers' usages change over time. When thinking about a context as large as web programming and the frequency that new contexts and APIs for web programming arise, the amount of setup effort required will be minimal in relation to the amount of work it would require to design and create suggestions for full coverage.

7.7.2 Potential of suggestions to increase use and learning

Overall, participants accessed almost three suggestions in a thirty minute session. This implies that our semi-automated approach to generating suggestions created relevant suggestions that participants received at appropriate points in programming. Although past studies [89, 183] have found that novices have trouble re-appropriating information from examples, this study showed that novice children were often able to use the new concepts they saw in the suggestions. Over a longer period of time and with more programming constructs, semi-automatic generated suggestions would likely encourage more use of the programming

concepts. This could lead to more learning in artifact-based contexts with less human effort required.

7.7.3 Effect of the code repository on suggestion generation

There are two interesting challenges of using code repositories for suggestion generation: the generated content is limited by the code in the repository, and we may not always know the expertise of the snippet creator. From fewer than 2,000 code examples, our approach generated an impressive number and quality of suggestions and examples. Many programs in the repository contained *parallel execution* code snippets, but many fewer included *loop* or nesting block snippets. This approach relies on at least some subset of users to be effectively using programming constructs. Because there are often at least a subset of programmers who do explore and seek out ways to learn new programming concepts on their own, we believe it is reasonable to assume that there will be at least a small number of usages of most programming concepts or API methods, though this would be interesting to evaluate.

We decided to include both novice and expert programs in our repository for generating suggestions because novices might create programs more similar to those that the novices would create in their first 30 minutes. This seems to have succeeded in generating suggestions with a wide range of complexity. Furthermore, the moderate phase should prevent low-quality code or suggestions from being created. When selecting code snippets from programs, the script could also be designed to more effectively filter out poor code. Furthermore, depending on the type of repository, there might be ways of measuring expertise that could be used to give higher priority to certain examples or suggestions. A few possibilities are the frequency of use of certain programming constructs and the number of programs or code contributions a user has made.

7.7.4 Personalization

Automated approaches for creating learning material means that there is more potential for personalization. Our approach generated many suggestions for each concept. Some of the suggestions were also highly similar to each other. This could enable systems to further personalize the suggestions provided based on a broader set of information about the novice programmer. In this study, some participants paid less attention to the suggestions and some had more trouble understanding the suggestions. Personalization might enable a system to better support a broader range of children. This would be hard to do using expert-created content because the expert would need to create much more content and determine how it applies to different types of children.

7.8 Conclusion

Existing systems require significant human effort to generate support to help programmers. As programming becomes more prevalent, there are more and more systems for children to begin learning programming. With each of these new systems comes the need for more documentation and support, which is often static and outside of the novice's context. Being able to generate support semi-automatically for these systems provides a substantial advantage over requiring human effort to author the support. Semi-automatic suggestion generation also provides a way to keep support up to date, such as with themes in pop culture that can be highly motivating for children. This chapter demonstrates an approach that produces equivalent suggestions to an expert set, as well as a novel set of suggestions. Our approach could ease the human costs associated with creating and updating help resources. While future work should explore how to automate this process even further, this work supports hypothesis 3, which claims that we can reduce the human effort required. Combined with

an effective suggestion system for artifact-driven programming and example support, this semi-automatic approach for generating content has the potential to help children overcome plateaus by encouraging exploration of previously unused skills, ideally leading to learning and continued interest in programming.

Chapter 8

Summary and Future Work

This section first summarizes the contributions of this thesis and their potential implications for supporting artifact-based learning. It then discusses future directions that build upon our contributions.

8.1 Summary

This thesis makes three main contributions: 1) a better understanding of how novices use and focus on example code, 2) the design of a system that encourages novice programmers to explore new skills in a artifact-based context, and 3) an approach for generating suggestions and rules semi-automatically for artifact-based novice programming. These contributions advance our ability to design systems that can expose novice users to new skills as they work toward their own project goals.

This work contributes to the gap in knowledge about what makes example code use difficult for novice programmers through two studies looking at: 1) novices' processes using examples and

2) how they memorize and recall code. These studies showed several important results: novices need help knowing what to focus on in examples, novices benefit from comparing example code, novices need support for finding code blocks in blocks-based environments, and novices need examples designed to reduce distraction. At a narrow scope, these findings provide important guidelines for designing support for example use in blocks programming environments. Blocks programming environments should provide multiple short code example snippets in which the critical elements are early in the example and other unknown elements are limited. The examples should also emphasize the critical elements and provide support to help the novice programmer find those blocks in the programming environment. These recommendations have the potential for high impact, as most blocks programming environments do not currently include example support beyond reuse of entire programs. Beyond novice programming in blocks programming environments, this body of knowledge can also apply to other programming environments, complex software systems, or digital learning environments.

With this better understanding of novice example use, we designed the Example Guru, which encouraged novice programmers to explore programming skills during artifact-based programming more than common static support. The Example Guru provides multiple examples to encourage comparison and self-exploration. The suggestions are initially available as annotations on the code that demonstrate the connection of the suggestions to their code. Suggestions are also available in a list to make them accessible throughout the programming process. Finally, the suggestions provide a button or directions to help novices find the code blocks. Evaluations of the Example Guru support the success of these suggestions in encouraging exploration of new skills in a artifact-based context. Artifact-based learning is common for novice programming, with the popularity of blocks-based programming environments like App Inventor, Scratch, Alice, and Kodu. When novices use these contexts outside of a classroom, they select the type of app, animation, or game that they want to

create. This type of suggestion system can then encourage these programmers to explore new skills as they work. This is particularly useful for children in blocks programming environments, which typically lack even basic in-context support. These results have broader implications for more experienced programmers, as well as artifact-based novices in other contexts. Even intermediate to experienced programmers will likely lack knowledge of all of the useful API methods or APIs that could improve their programs.

Finally, our approach for semi-automatically generating suggestions and rules for artifact-based novice programming produced an impressive and effective array of suggestions. The approach generated suggestions that aligned with expert-created suggestions and a novel set of suggestions not generated by an expert. Our study showed that these suggestions often triggered for novice programmers and encouraged them to explore abstract programming concepts. This approach is not specific to Looking Glass. Similar types of suggestions and rules could be generated with other code repositories and defined grouping metrics. In combination with the evidence that novice programmers gain significantly more exposure to new programming concepts with suggestions than other support, this approach for generating suggestions has the potential for significant impact in the context of larger systems. This could enable programmers to discover skills at appropriate times and make better use of available programming resources. It could also help newer programmers feel more empowered. Empowering artifact-based novice programmers could increase retention of diverse populations in computer science.

8.2 Future work

The contributions of this thesis imply that making suggestions to novices in artifact-based contexts may encourage novices to explore new relevant skills. However, the scope of this

thesis has focused only on children in a blocks programming environment and on a small subset of skills. The designs and approaches of this work have the potential to have a large impact beyond novice programming if the human effort can be further reduced and if suggestions can be generated equally well for other types of information. Beyond improving the scope of this type of system, enabling user interaction with the system could improve the relevance of suggestions. Enabling users to interact with the content could allow researchers to better evaluate suggestion relevance. It could also improve learning outcomes by encouraging self-explanation and community improvement of suggestions. Future work should address the resulting open questions about the generalizability, scalability, and interactivity of this type of system: 1) how can we apply large-scale, in-context, suggested content to help people learn other topics, 2) how can we automatically generate context-relevant support at a large scale, and 3) what types of user interaction can help users learn and improve suggestion relevance?

8.2.1 How can we apply large-scale in-context suggested content to help people learn other topics?

Systems like the Example Guru have the potential to be highly effective in helping users explore more complex programming skills and skills in artifact-based areas beyond programming. As programming and computing skills become more complex, they may require more complex suggestions and rules. Beyond programming, a variety of other skill-sets have software in which users likely learn as they work on projects, such as data analysis, engineering, architecture, and design. If we could generate suggestion content like we did for the Example Guru in these other contexts, on-the-fly feedback could be available for a wide variety of topics. As an example, let's consider how suggestions could apply to data analysis and the new challenges in suggesting content outside of programming.

Many researchers learn data analysis methods somewhat on-the-fly. Learners define their own project goals and may not realize there are more correct ways to analyze their data or parameters they should apply in certain cases. This is an artifact-based context like programming, but unlike programming, relevant information for data analysis may exist in the data itself as opposed to the code. Thus, rules would need to be able to analyze imported data sets and suggestions would need ways to link or refer to data. Furthermore, many questions and answers are often more conceptual and text-based rather than code-based. Tags or key words may still help to link textual information to specific concepts, but examples may need to be drawn from more formalized instructional information rather than crowdsourced online content.

The method for generating content will need adaptation for other topics because the body of examples online for data analysis and similar topics is much smaller than the population of general programmers. Much of the information about data analysis is in the form of expert knowledge, documentation, websites, forums, slides, and books. If we could generate suggestions from these varied sources by extracting examples and related text and linking them, it would enable much further generalization of this type of system beyond programming. In order to be able to generate suggestions for topics other than programming, systems may need to be able to generate suggestions and rules at a large scale without repositories of examples. The next section discusses future work in this direction and also the possibility of generating suggestions with less human effort than our semi-automatic approach.

8.2.2 How can we automatically generate context-relevant support at a large scale?

In order to create suggestions on a large scale, systems need to construct them automatically. My dissertation shows that a system can generate suggestions semi-automatically by drawing from a repository of programs for animation code [93]. This approach still requires human effort to define the ways to group example snippets and for authoring the titles and descriptions of suggestions and examples. We could create systems for suggesting content to artifact-based learners even more efficiently if we could create this content without an expert in the loop and from other kinds of content than code repositories. There are two avenues to address here: 1) can we generate suggestions automatically from other types of content, like forums, and 2) can we generate these suggestions without a human in the loop?

Code repositories often do not provide descriptions or other information about the code or may not even exist. Other resources, such as forums, documentation, or revision histories might provide more context and richer support. One way to integrate this type of content into a suggestion system might be to use example code in the questions to generate rules and examples in the solutions as the suggestions. The question and answer text could be used as the description information for the suggestions. An approach like this would need to filter out poor questions, answers, and examples. It would also need to determine which text and example code is relevant and useful, either algorithmically or using crowdsourcing. Another possibility would be to use the question and answer information to design the rules, but to find more reliable content for the suggestion and examples, such as from high quality open-source projects or documentation.

Using sources of examples that also contain text descriptions may also help to remove the expert in the loop in order to generate suggestions and rules completely automatically. Using

resources like forums or documentation that include text descriptions or using multiple types of information could alleviate some of the effort of experts writing the descriptions themselves. The experts' other main role was to identify the ways to group code snippets. Machine learning techniques may also enable us to cluster code examples with less human intervention. Another way to reduce expert effort may be to have the learners select which examples are the most helpful and author the suggestion titles and example descriptions. This is just one of several interaction techniques that could improve this type of suggestion system.

8.2.3 What types of user interaction can help users learn and improve suggestion relevance?

Current systems that support programmers allow for little to no feedback or input from the user. By interacting with a suggestion system, we believe learners could: learn through contributing descriptions, improve the suggestion model, and provide feedback that could enable better evaluation of effectiveness.

One form of interactivity that will likely improve users' learning from content is involving them in authoring the descriptions for suggestions or examples. As discussed, self-explanation can be very helpful for learners. Suggestions could start off with automatically generated suggestions about the code included, such as "Based on your usage of X, others have used Y". They could then ask users to provide a better title to the suggestion. Descriptions of examples could come from content like revision histories or forum text and ask users to improve the descriptions. Once several users have started adding titles and descriptions, users could have the option to select the best description, mark incorrect descriptions, or author their own. Research on learners labeling video content suggests that having learners label examples will

likely support deep understanding of the suggested concepts and also generate valuable labels [109].

Integrating feedback into how these systems provide information would likely make them better in two main ways: the system could use feedback to improve when suggestions get triggered and information about users' perceptions of suggestions could enable a system to provide more support. Promptor, which suggests Stack Overflow content within Eclipse has a thumbs up or thumbs down option [168]. This is a starting point, but is still very limited because it does not provide any information about why the information is not useful to the programmer. For example, content might be irrelevant, incorrect, or confusing. A user could also alert the system that they planned to try to use a suggestion. If a system collected this feedback, it could augment rules with relevance ratings, remove incorrect feedback, and provide more support for users who find a suggestion confusing. If a system knows that a user is trying to implement a suggestion but having difficulty or a user does not understand a suggestion, it could provide more information or suggestions with the information broken down. In an educational or evaluation setting, this information could also indicate whether example content effectively enables users to use new skills and how often automatically generated suggestions are triggered incorrectly.

Overall, this thesis demonstrates the potential of a suggestion system to encourage novice programmers to explore new programming skills in an artifact-based context. It also inspires new directions to establish the effectiveness of this system more generally and to improve its value to users through interactivity.

References

- [1] ISO/IEC 9126-1 2001. *Software Engineering- Product Quality – Part 1: Quality Model*. 2001.
- [2] Beth Adelson. “Problem solving and the development of abstract categories in programming languages”. In: *Memory & cognition* 9.4 (1981), pp. 422–433. URL: <http://www.springerlink.com/index/F4656R1818X225R2.pdf>.
- [3] *Alice Community*. URL: <http://www.alice.org/community/>.
- [4] *Amazon Mechanical Turk - Welcome*. URL: <https://www.mturk.com/mturk/welcome>.
- [5] Nancy Anderson and Ben Shneiderman. “Use of peer ratings in evaluating computer program quality”. In: *Proc. 15th annu.SIGCPR conf. SIGCPR '77*. New York, NY, USA: ACM, 1977, pp. 218–226. DOI: 10.1145/800100.803247. URL: <http://doi.acm.org/10.1145/800100.803247>.
- [6] *Anybody can learn / Code.org*. URL: <http://code.org/>.
- [7] D. Arnow and O. Barshay. “WebToTeach: An interactive focused programming exercise system”. In: *Frontiers in Education Conf. FIE'99. 29th Annual*. Vol. 1. 1999, 12A9–39. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=839303.
- [8] C. Artho and A. Biere. “Applying static analysis to large-scale, multi-threaded Java programs”. In: *Software Engineering. Proc. Australian*. 2001, pp. 68–75. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=948499.
- [9] Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. “PARC: Recommending API methods parameters”. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE. 2015, pp. 330–332.
- [10] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. “Csc: Simple, efficient, context sensitive code completion”. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE. 2014, pp. 71–80.
- [11] Dimitar Asenov, Otmar Hilliges, and Peter Müller. “The effect of richer visualizations on code comprehension”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM. 2016, pp. 5040–5045.

- [12] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. “Learning from examples: Instructional principles from the worked examples research”. In: *Review of educational research* 70.2 (2000), pp. 181–214. URL: <http://rer.sagepub.com/content/70/2/181.short>.
- [13] Alan Baddeley. “Working memory”. In: *Science* 255.5044 (1992), pp. 556–559.
- [14] Omar Badreddin, Andrew Forward, and Timothy C Lethbridge. “Model oriented programming: an empirical study of comprehension”. In: *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp. 2012, pp. 73–86.
- [15] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. “Leveraging usage similarity for effective retrieval of examples in code repositories”. In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 157–166. URL: <http://dl.acm.org/citation.cfm?id=1882316>.
- [16] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. “Sourcerer: a search engine for open source code supporting structure-based search”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 681–682.
- [17] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. “Clone Detection Using Abstract Syntax Trees”. In: *Proceedings of the International Conference on Software Maintenance*. ICSM ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. URL: <http://dl.acm.org/citation.cfm?id=850947.853341>.
- [18] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. “Tinkering and gender in end-user programmers’ debugging”. In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, pp. 231–240. URL: <http://dl.acm.org/citation.cfm?id=1124808>.
- [19] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. “Comparison and evaluation of clone detection tools”. In: *IEEE Transactions on software engineering* 33.9 (2007). URL: <http://ieeexplore.ieee.org/abstract/document/4288192/>.
- [20] R E. Berry and B A.E. Meekings. “A style analysis of C programs”. In: *Commun. ACM* 28.1 (Jan. 1985), pp. 80–88. DOI: 10.1145/2465.2469. URL: <http://doi.acm.org/10.1145/2465.2469>.
- [21] Michael Blumenstein, Steven Green, Ann Nguyen, and Vallipuram Muthukkumarasamy. “An experimental analysis of GAME: a generic automated marking environment”. In: *SIGCSE Bull.* 36.3 (June 2004), pp. 67–71. DOI: 10.1145/1026487.1008016. URL: <http://doi.acm.org/10.1145/1026487.1008016>.

- [22] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. “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. URL: <http://dl.acm.org/citation.cfm?id=1753402>.
- [23] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598. URL: <http://dl.acm.org/citation.cfm?id=1518944>.
- [24] Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. “Measuring static quality of student code”. In: *Proc. 16th annu. joint conf. on ITiCSE*. ITiCSE ’11. New York, NY, USA: ACM, 2011, pp. 13–17. DOI: 10.1145/1999747.1999754. URL: <http://doi.acm.org/10.1145/1999747.1999754>.
- [25] Ruven Brooks. “Towards a theory of the comprehension of computer programs”. In: *International journal of man-machine studies* 18.6 (1983), pp. 543–554. URL: <http://www.sciencedirect.com/science/article/pii/S0020737383800315>.
- [26] Marcel Bruch, Martin Monperrus, and Mira Mezini. “Learning from examples to improve code completion systems”. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 213–222.
- [27] Peter Brusilovsky. “WebEx: Learning from Examples in a Programming Course.” In: *WebNet*. Vol. 1. 2001, pp. 124–129. URL: <http://www.pitt.edu/~peterb/papers/WebNet01.html>.
- [28] Peter Brusilovsky and Gerhard Weber. “Collaborative example selection in an intelligent example-based programming environment”. In: *Proceedings of the 1996 international conference on Learning sciences*. International Society of the Learning Sciences, 1996, pp. 357–362. URL: <http://dl.acm.org/citation.cfm?id=1161185>.
- [29] *Build software better, together*. en. URL: <https://github.com> (visited on 03/27/2018).
- [30] Margaret Burnett, Scott D. Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. “Gender differences and programming environments: across programming populations”. In: *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. ACM, 2010, p. 28. URL: <http://dl.acm.org/citation.cfm?id=1852824>.
- [31] Raymond PL Buse and Westley Weimer. “Synthesizing API usage examples”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 782–792.

- [32] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. “Eye movements in code reading: Relaxing the linear order”. In: *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. IEEE. 2015, pp. 255–265.
- [33] Teresa Busjahn, Carsten Schulte, and Edna Kropp. “Developing Coding Schemes for Program Comprehension using Eye Movements”. In: *PPIG, University of Sussex* (2014). URL: http://www.mi.fu-berlin.de/inf/groups/ag-ddi/members/wimis/Busjahn-et-al_---2014---Developing-Coding-Schemes-for-Program-Comprehensio.pdf.
- [34] Jill Cao, Yann Riche, Susan Wiedenbeck, Margaret Burnett, and Valentina Grigoreanu. “End-user mashup programming: through the design lens”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 1009–1018. URL: <http://dl.acm.org/citation.cfm?id=1753477>.
- [35] William M Carroll. “Using worked examples as an instructional support in the algebra classroom.” In: *Journal of Educational Psychology* 86.3 (1994), p. 360.
- [36] Richard Catrambone. “The subgoal learning model: Creating better examples so that students can solve novel problems.” In: *Journal of Experimental Psychology: General* 127.4 (1998), p. 355.
- [37] Richard Catrambone and Keith J Holyoak. “Overcoming contextual limitations on problem-solving transfer.” In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 15.6 (1989), p. 1147.
- [38] Kerry Shih-Ping Chang and Brad A. Myers. “WebCrystal: understanding and reusing examples in web authoring”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 3205–3214. URL: <http://dl.acm.org/citation.cfm?id=2208740>.
- [39] William G. Chase and Herbert A. Simon. “Perception in chess”. In: *Cognitive psychology* 4.1 (1973), pp. 55–81. URL: <http://www.sciencedirect.com/science/article/pii/0010028573900042>.
- [40] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. “Sniff: A search engine for java using free-form queries”. In: *Fundamental Approaches to Software Engineering* (2009), pp. 385–400.
- [41] Michelene TH Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. “Self-explanations: How students study and use examples in learning to solve problems”. In: *Cognitive science* 13.2 (1989), pp. 145–182. URL: <http://www.sciencedirect.com/science/article/pii/0364021389900025>.
- [42] Michelene TH Chi, Paul J. Feltoovich, and Robert Glaser. “Categorization and representation of physics problems by experts and novices”. In: *Cognitive science* 5.2 (1981), pp. 121–152. URL: http://onlinelibrary.wiley.com/doi/10.1207/s15516709cog0502_2/abstract.

- [43] Michelene TH Chi, Robert Glaser, and Ernest Rees. *Expertise in problem solving*. Tech. rep. DTIC Document, 1981. URL: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA100138>.
- [44] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. “MixT: automatic generation of step-by-step mixed media tutorials”. In: *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 2012, pp. 93–102. URL: <http://dl.acm.org/citation.cfm?id=2380130>.
- [45] *Code Analyzer for Java*. URL: <http://qjpro.sourceforge.net/index.html>.
- [46] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. “Context-based recommendation to support problem solving in software development”. In: *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*. IEEE. 2012, pp. 85–89.
- [47] Davor Čubranić and Gail C Murphy. “Hipikat: Recommending pertinent software development artifacts”. In: *Proceedings of the 25th international Conference on Software Engineering*. IEEE Computer Society. 2003, pp. 408–418.
- [48] Jácome Cunha, João Fernandes, Hugo Ribeiro, and João Saraiva. “Towards a catalog of spreadsheet smells”. In: *ICCSA 2012* (2012), pp. 202–216. URL: <http://www.springerlink.com/index/41084U52236674V5.pdf>.
- [49] Simon P. Davies, David J. Gilmore, and Thomas R. G. Green. “Are Objects That Important? Effects of Expertise and Familiarity on Classification of Object-Oriented Code”. In: *Human–Computer Interaction* 10.2-3 (June 1995), pp. 227–248. DOI: 10.1080/07370024.1995.9667218. URL: <http://www.tandfonline.com/doi/abs/10.1080/07370024.1995.9667218>.
- [50] Adriaan D. De Groot and Adrianus Dingeman de Groot. *Thought and choice in chess*. Vol. 4. Walter de Gruyter, 1978. URL: https://books.google.com/books?hl=en&lr=&id=EI4gr42NwDQC&oi=fnd&pg=PR5&dq=thought+and+choice+in+chess+de+groot&ots=5ESZLGvbmT&sig=_CBrlhCbXPJ6GLbvlQvvFLzEkPA.
- [51] Uri Dekel and James D Herbsleb. “Improving API documentation usability with knowledge pushing”. In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE. 2009, pp. 320–330.
- [52] *DiffLib — Helpers for computing deltas — Python 2.7.13 documentation*. URL: <https://docs.python.org/2/library/difflib.html>.
- [53] *diff-match-patch*. URL: <https://www.npmjs.com/package/diff-match-patch>.
- [54] Christian Dörner, Andrew R Faulring, and Brad A Myers. “EUKLAS: Supporting copy-and-paste strategies for integrating example code”. In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM. 2014, pp. 13–20.

- [55] Stephen H. Edwards and Manuel A. Perez-Quinones. “Web-CAT: automatically grading programming assignments”. In: *Proc. 13th annual conf. on ITiCSE*. ITiCSE ’08. New York, NY, USA: ACM, 2008, pp. 328–328. DOI: 10.1145/1384271.1384371. URL: <http://doi.acm.org/10.1145/1384271.1384371>.
- [56] The CSTA Standards Task Force. *CSTA K-12 Computer Science Standards*. Tech. rep. CSTA, 2011.
- [57] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. en. Addison-Wesley Professional, 1999.
- [58] C Ailie Fraser, Mira Dontcheva, Holger Winnemöller, Sheryl Ehrlich, and Scott Klemmer. “DiscoverySpace: Suggesting Actions in Complex Software”. In: *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*. ACM. 2016, pp. 1221–1232.
- [59] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu. “APOGEE: automated project grading and instant feedback system for web based computing”. In: *ACM SIGCSE Bulletin*. Vol. 40. 2008, pp. 77–81. URL: <http://dl.acm.org/citation.cfm?id=1352163>.
- [60] Dedre Gentner. “Structure-mapping: A theoretical framework for analogy”. In: *Cognitive science* 7.2 (1983), pp. 155–170. URL: <http://www.sciencedirect.com/science/article/pii/S0364021383800093>.
- [61] Dedre Gentner and Cecile Toupin. “Systematicity and surface similarity in the development of analogy”. In: *Cognitive science* 10.3 (1986), pp. 277–300. URL: http://onlinelibrary.wiley.com/doi/10.1207/s15516709cog1003_2/abstract.
- [62] Mary L. Gick and Keith J. Holyoak. “Analogical problem solving”. In: *Cognitive psychology* 12.3 (1980), pp. 306–355. URL: <http://www.sciencedirect.com/science/article/pii/0010028580900134>.
- [63] Mary L. Gick and Keith J. Holyoak. “Schema induction and analogical transfer”. In: *Cognitive psychology* 15.1 (1983), pp. 1–38. URL: <http://www.sciencedirect.com/science/article/pii/0010028583900026>.
- [64] D. J. Gilmore and T. R. G. Green. “Programming plans and programming expertise”. In: *The Quarterly Journal of Experimental Psychology* 40.3 (1988), pp. 423–442. URL: <http://www.tandfonline.com/doi/abs/10.1080/02724988843000005>.
- [65] Barney G. Glaser and Anselm L. Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Aldine de Gruyter, 1967. URL: http://books.google.com/books?hl=en&lr=&id=rTiNK68Xt08C&oi=fnd&pg=PA1&dq=grounded+theory&ots=UUzTWhWL-L&sig=GzQM6hpGR1LGezru_rqbWxaK86Y.
- [66] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. “OverCode: Visualizing variation in student solutions to programming problems at scale”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 22.2 (2015), p. 7. URL: <http://dl.acm.org/citation.cfm?id=2699751>.

- [67] Fernand Gobet, Peter CR Lane, Steve Croker, Peter CH Cheng, Gary Jones, Iain Oliver, and Julian M. Pine. “Chunking mechanisms in human learning”. In: *Trends in cognitive sciences* 5.6 (2001), pp. 236–243. URL: <http://www.sciencedirect.com/science/article/pii/S1364661300016624>.
- [68] Max Goldman and Robert C. Miller. “Codetrail: Connecting source code and web resources”. In: *Journal of Visual Languages & Computing* 20.4 (2009), pp. 223–235. URL: <http://www.sciencedirect.com/science/article/pii/S1045926X09000263>.
- [69] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. “Generating photo manipulation tutorials by demonstration”. In: *ACM Transactions on Graphics (TOG)* 28.3 (2009), p. 66. URL: <http://dl.acm.org/citation.cfm?id=1531372>.
- [70] Simon Gray, Caroline St Clair, Richard James, and Jerry Mead. “Suggestions for graduated exposure to programming concepts using fading worked examples”. In: *Proceedings of the third international workshop on Computing education research*. ACM, 2007, pp. 99–110.
- [71] *Greenfoot / Latest Activity*. URL: <https://www.greenfoot.org/home> (visited on 03/26/2018).
- [72] Paul A. Gross, Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. “A code reuse interface for non-programmer middle school students”. In: *Proceedings of the 15th international conference on Intelligent user interfaces*. ACM, 2010, pp. 219–228. URL: <http://dl.acm.org/citation.cfm?id=1720001>.
- [73] 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. URL: <http://www.sciencedirect.com/science/article/pii/S1045926X10000431>.
- [74] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. “On the use of automated text summarization techniques for summarizing source code”. In: *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 35–44. URL: <http://ieeexplore.ieee.org/abstract/document/5645482/>.
- [75] Kyle J. Harms, Dennis Cosgrove, Shannon Gray, and Caitlin Kelleher. “Automatically generating tutorials to enable middle school children to learn programming independently”. In: *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 2013, pp. 11–19. URL: <http://dl.acm.org/citation.cfm?id=2485764>.
- [76] Kyle J. Harms, Jordana H. Kerr, Michelle Ichinco, Mark Santolucito, Alexis Chuck, Terian Kosciuk, Mary Chou, and Caitlin L. Kelleher. “Designing a community to support long-term interest in programming for middle school children”. In: *Proceedings of the 11th International Conference on Interaction Design and Children*. IDC ’12. New York, NY, USA: ACM, 2012, pp. 304–307. DOI: 10.1145/2307096.2307152. URL: <http://doi.acm.org/10.1145/2307096.2307152>.

- [77] Kyle J. Harms, Noah Rowlett, and Caitlin Kelleher. “Enabling independent learning of programming concepts through programming completion puzzles”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 271–279. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7357226.
- [78] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. “Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 2016, pp. 241–250. URL: <http://dl.acm.org/citation.cfm?id=2960314>.
- [79] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. “What would other programmers do: suggesting solutions to error messages”. In: *Proc. 28th int. conf. on Human factors in computing systems*. 2010, pp. 1019–1028.
- [80] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. “Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 3–12. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7356972.
- [81] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D’Antoni, and Björn Hartmann. “Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis”. In: *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 2017, pp. 89–98. URL: <http://dl.acm.org/citation.cfm?id=3051467>.
- [82] Lars Heinemann, Veronika Bauer, Markus Herrmannsdoerfer, and Benjamin Hummel. “Identifier-based context-dependent api method recommendation”. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE. 2012, pp. 31–40.
- [83] Wint Hnin, Michelle Ichinco, and Caitlin Kelleher. “An Exploratory Study of the Usage of Different Educational Resources in an Independent Context”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 2017, To Appear.
- [84] Raphael Hoffmann, James Fogarty, and Daniel S Weld. “Assieme: finding and leveraging implicit references in a web search interface for programmers”. In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM. 2007, pp. 13–22.
- [85] Reid Holmes and Gail C. Murphy. “Using structural context to recommend source code examples”. In: *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 117–125. URL: <http://dl.acm.org/citation.cfm?id=1062491>.
- [86] Reid Holmes and Robert J Walker. “A newbie’s guide to eclipse APIs”. In: *Proceedings of the 2008 international working conference on Mining software repositories*. ACM. 2008, pp. 149–152.

- [87] Reid Holmes, Robert J. Walker, and Gail C. Murphy. “Approximate structural context matching: An approach to recommend relevant examples”. In: *Software Engineering, IEEE Transactions on* 32.12 (2006), pp. 952–970. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4016572.
- [88] Michael J. Hull, Daniel Powell, and Ewan Klein. “Infandango: automated grading for student programming”. In: *Proc. 16th annu. joint conf. ITiCSE*. ITiCSE ’11. New York, NY, USA: ACM, 2011, pp. 330–330. DOI: 10.1145/1999747.1999841. URL: <http://doi.acm.org/10.1145/1999747.1999841>.
- [89] Ichinco, M., Harms, K.J., and Kelleher, C. “Towards Understanding Successful Novice Example User in Blocks-Based Programming”. In: *Journal of Visual Languages and Sentient Systems* Volume 3. July 2017 (July 2017), pp. 101–118.
- [90] 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*. CHI ’17. New York, NY, USA: ACM, 2017, pp. 1105–1117. DOI: 10.1145/3025453.3025827. URL: <http://doi.acm.org/10.1145/3025453.3025827>.
- [91] Michelle Ichinco and Caitlin Kelleher. “Exploring novice programmer example use”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 63–71. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7357199.
- [92] Michelle Ichinco and Caitlin Kelleher. “Towards block code examples that help young novices notice critical elements”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE. 2017, pp. 335–336.
- [93] Michelle Ichinco and Caitlin L Kelleher. “Semi-Automatic Suggestion Generation for Young Novice Programmers in an Open-Ended Context”. In: *Proceedings of the 17th International Conference on Interaction Design and Children*. ACM. 2018, To Appear.
- [94] Michelle Ichinco, Aaron Zemach, and Caitlin Kelleher. “Towards generalizing expert programmers’ suggestions for novice programmers”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 2013, pp. 143–150.
- [95] Takeo Igarashi and John F Hughes. “A suggestive interface for 3D drawing”. In: *Proceedings of the 14th annual ACM symposium on User interface software and technology*. ACM. 2001, pp. 173–181.
- [96] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. “Review of recent systems for automatic assessment of programming assignments”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 2010, pp. 86–93. URL: <http://dl.acm.org/citation.cfm?id=1930480>.
- [97] Google Inc. Gallup Inc. “Trends in the State of Computer Science in U.S. K-12 Schools”. In: (2016). URL: <http://goo.gl/j291E0>.

- [98] David Jackson and Michelle Usher. “Grading student programs using ASSYST”. In: *Proc. 28th SIGCSE technical symp. on Computer science education*. SIGCSE ’97. New York, NY, USA: ACM, 1997, pp. 335–339. DOI: 10.1145/268084.268210. URL: <http://doi.acm.org/10.1145/268084.268210>.
- [99] *javapoet: A Java API for generating .java source files*. original-date: 2013-02-01T16:56:30Z. Aug. 2017. URL: <https://github.com/square/javapoet>.
- [100] Will Jernigan et al. “A principled evaluation for a principled idea garden”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 235–243. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7357222.
- [101] E. L. Jones. “Grading student programs-a software testing approach”. In: *J. of Computing Sci. in Colleges* 16.2 (2001), pp. 185–192. URL: <http://dl.acm.org/citation.cfm?id=369279.369354>.
- [102] M. Gail Jones, Laura Brader-Araje, Lisa Wilson Carboni, Glenda Carter, Melissa J. Rua, Eric Banilower, and Holly Hatch. “Tool time: Gender and students’ use of tools, control, and authority”. In: *Journal of Research in Science Teaching* 37.8 (2000), pp. 760–783. URL: [http://onlinelibrary.wiley.com/doi/10.1002/1098-2736\(200010\)37:8%3C760::AID-TEA2%3E3.0.CO;2-V/full](http://onlinelibrary.wiley.com/doi/10.1002/1098-2736(200010)37:8%3C760::AID-TEA2%3E3.0.CO;2-V/full).
- [103] N. Jovanovic, C. Kruegel, and E. Kirda. “Pixy: A static analysis tool for detecting web application vulnerabilities”. In: *Security and Privacy, IEEE Symp. on*. 2006, 6–pp. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1624016.
- [104] David Kawrykow and Martin P. Robillard. “Improving api usage through automatic detection of redundant code”. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 111–122. URL: <http://dl.acm.org/citation.cfm?id=1747513>.
- [105] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. “Storytelling alice motivates middle school girls to learn computer programming”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 1455–1464. URL: <http://dl.acm.org/citation.cfm?id=1240844>.
- [106] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. “Design Defects Detection and Correction by Example”. In: *Program Comprehension, IEEE 19th Int. Conf.* 2011, pp. 81–90. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5970166.
- [107] *Khan Academy*. URL: <http://www.khanacademy.org>.
- [108] Md Adnan Alam Khan, Volodymyr Dziubak, and Andrea Bunt. “Exploring personalized command recommendations based on information found in Web documentation”. In: *Proceedings of the 20th International Conference on Intelligent User Interfaces*. ACM, 2015, pp. 225–235. URL: <http://dl.acm.org/citation.cfm?id=2701387>.

- [109] Juho Kim, Robert C Miller, and Krzysztof Z Gajos. “Learnersourcing subgoal labeling to support learning from how-to videos”. In: *CHI’13 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2013, pp. 685–690.
- [110] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. “Supporting end-user debugging: what do users want to know?”. In: *Proceedings of the working conference on Advanced visual interfaces*. ACM, 2006, pp. 135–142. URL: <http://dl.acm.org/citation.cfm?id=1133293>.
- [111] *Klocwork*. URL: <http://www.klocwork.com/>.
- [112] Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. “Six learning barriers in end-user programming systems”. In: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 2004, pp. 199–206. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1372321.
- [113] *Kodu Forum / Discussion*. URL: <https://www.kodugamelab.com/discussion> (visited on 03/26/2018).
- [114] *Kodu / Home*. URL: <http://www.kodugamelab.com/>.
- [115] Jürgen Koenemann and Scott P. Robertson. “Expert problem solving strategies for program comprehension”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1991, pp. 125–130. URL: <http://dl.acm.org/citation.cfm?id=108863>.
- [116] Jochen Kreimer. “Adaptive Detection of Design Flaws”. In: *Electron. Notes Theor. Comput. Sci.* 141.4 (Dec. 2005), pp. 117–136. DOI: 10.1016/j.entcs.2005.02.059. URL: <http://dx.doi.org/10.1016/j.entcs.2005.02.059>.
- [117] Benjamin Lafreniere, Parmit K. Chilana, Adam Fourney, and Michael A. Terry. “These Aren’t the Commands You’re Looking For: Addressing False Feedforward in Feature-Rich Software”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 2015, pp. 619–628. URL: <http://dl.acm.org/citation.cfm?id=2807482>.
- [118] Jill Larkin, John McDermott, Dorothea P. Simon, and Herbert A. Simon. “Expert and novice performance in solving physics problems”. In: *Science* 208.4450 (1980), pp. 1335–1342. URL: https://www.researchgate.net/profile/John_Mcdermott10/publication/6064271_Expert_and_Novice_Performance_in_Solving_Physics_Problems/links/5489c30f0cf214269f1abb55.pdf.
- [119] Wei Li, Justin Matejka, Tovi Grossman, Joseph A. Konstan, and George Fitzmaurice. “Design and evaluation of a command recommendation system for software applications”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 18.2 (2011), p. 6. URL: <http://dl.acm.org/citation.cfm?id=1970380>.
- [120] Dastyni Loksa and Andrew J Ko. “Modeling Programming Problem Solving Through Interactive Worked Examples”. In: (2017).

- [121] *Looking Glass Community*. URL: <https://lookingglass.wustl.edu/>.
- [122] Mark Mahoney. “Storyteller: A New Medium for Guiding Students Through Code Examples”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM. 2018, pp. 1112–1112.
- [123] Yuri Malheiros, Alan Moraes, Cleyton Trindade, and Silvio Meira. “A source code recommender system to support newcomers”. In: *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE. 2012, pp. 19–24.
- [124] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. “Jungloid Mining: Helping to Navigate the API Jungle”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 48–61. DOI: 10.1145/1065010.1065018. URL: <http://doi.acm.org/10.1145/1065010.1065018>.
- [125] M. Mantyla, J. Vanhanen, and C. Lassenius. “A taxonomy and an initial empirical study of bad smells in code”. In: *Int. Conf. on Software Maintenance, 2003. ICSM 2003. Proc.* Sept. 2003, pp. 381–384. DOI: 10.1109/ICSM.2003.1235447.
- [126] M.V. Mantyla. “An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement”. In: *Int.Symp. on Empirical Software Eng.* Nov. 2005, 10 pp. DOI: 10.1109/ISESE.2005.1541837.
- [127] M.V. Mantyla, J. Vanhanen, and C. Lassenius. “Bad smells - humans as code critics”. In: *20th IEEE Int. Conf. on Software Maintenance. Proc.* Sept. 2004, pp. 399–408. DOI: 10.1109/ICSM.2004.1357825.
- [128] Lauren E Margulieux, Richard Catrambone, and Mark Guzdial. “Employing subgoals in computer programming education”. In: *Computer Science Education* 26.1 (2016), pp. 44–67.
- [129] Lauren E Margulieux, Mark Guzdial, and Richard Catrambone. “Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications”. In: *Proceedings of the ninth annual international conference on International computing education research*. ACM. 2012, pp. 71–78.
- [130] Lauren Margulieux and Richard Catrambone. “Using Learners’ Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor”. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM. 2017, pp. 21–29.
- [131] Lauren Margulieux, Briana B Morrison, Mark Guzdial, and Richard Catrambone. “Training learners to self-explain: Designing instructions and examples to improve problem solving”. In: (2016).
- [132] Radu Marinescu. “Detection strategies: Metrics-based rules for detecting design flaws”. In: *Software Maintenance. Proc. 20th IEEE Int. Conf.* 2004, pp. 350–359. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1357820.

- [133] Justin Matejka, Tovi Grossman, and George Fitzmaurice. “Ambient help”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 2751–2760. URL: <http://dl.acm.org/citation.cfm?id=1979349>.
- [134] Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. “Community-Commands: Command Recommendations for Software Applications”. In: *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*. UIST ’09. Victoria, BC, Canada: ACM, 2009, pp. 193–202. DOI: 10.1145/1622176.1622214. URL: <http://doi.acm.org/10.1145/1622176.1622214>.
- [135] Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. “Community-Commands: command recommendations for software applications”. In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 2009, pp. 193–202.
- [136] Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. “Community-Commands: command recommendations for software applications”. In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. UIST ’09. New York, NY, USA: ACM, 2009, pp. 193–202. DOI: 10.1145/1622176.1622214. URL: <http://doi.acm.org/10.1145/1622176.1622214>.
- [137] J. Nathan Matias, Sayamindu Dasgupta, and Benjamin Mako Hill. “Skill Progression in Scratch Revisited”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 1486–1490. URL: https://unmad.in/pdfs/matias-skill_progression_CHI2016.pdf.
- [138] T.J. McCabe. “A Complexity Measure”. In: *IEEE Trans. on Softw. Eng.* SE-2.4 (Dec. 1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.
- [139] Frank McCarey, Mel O Cinneide, and Nicholas Kushmerick. “A recommender agent for software libraries: An evaluation of memory-based and model-based collaborative filtering”. In: *Intelligent Agent Technology, 2006. IAT’06. IEEE/WIC/ACM International Conference on*. IEEE, 2006, pp. 154–162.
- [140] Daniel McFarlane. “Comparison of four primary methods for coordinating the interruption of people in human-computer interaction”. In: *Human-Computer Interaction* 17.1 (2002), pp. 63–139. URL: <http://dl.acm.org/citation.cfm?id=1464475>.
- [141] Katherine B. McKeithen, Judith S. Reitman, Henry H. Rueter, and Stephen C. Hirtle. “Knowledge organization and skill differences in computer programmers”. In: *Cognitive Psychology* 13.3 (1981), pp. 307–325. URL: <http://www.sciencedirect.com/science/article/pii/0010028581900128>.
- [142] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. “Portfolio: finding relevant functions and their usage”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 111–120.
- [143] S.A. Mengel and J. Ullans. “Using Verilog LOGISCOPE to analyze student programs”. In: *Frontiers in Education Conf., 1998. FIE ’98. 28th Annual*. Vol. 3. Nov. 1998, 1213–1218 vol.3. DOI: 10.1109/FIE.1998.738617.

- [144] Joan Meyers-Levy. “Gender differences in information processing: A selectivity interpretation”. PhD thesis. Northwestern University, 1986.
- [145] Joan Meyers-Levy and Durairaj Maheswaran. “Exploring differences in males’ and females’ processing strategies”. In: *Journal of Consumer Research* 18.1 (1991), pp. 63–70. URL: <http://jcr.oxfordjournals.org/content/18/1/63.abstract>.
- [146] George A Miller. “The magical number seven, plus or minus two: Some limits on our capacity for processing information.” In: *Psychological review* 63.2 (1956), p. 81.
- [147] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. “DECOR: A Method for the Specification and Detection of Code and Design Smells”. In: *IEEE Trans. on Softw. Eng.* 36.1 (Feb. 2010), pp. 20–36. DOI: 10.1109/TSE.2009.50.
- [148] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. “Automatic generation of natural language summaries for java classes”. In: *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.* IEEE, 2013, pp. 23–32. URL: <http://ieeexplore.ieee.org/abstract/document/6613830/>.
- [149] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. “Subgoals Help Students Solve Parsons Problems”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* ACM, 2016, pp. 42–47. URL: <http://dl.acm.org/citation.cfm?id=2844617>.
- [150] 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. URL: <http://dl.acm.org/citation.cfm?id=2787733>.
- [151] Matthew James Munro. “Product Metrics for Automatic Identification of”. In: *Software Metrics, 2005. 11th IEEE Int. Symp.* 2005, pp. 15–15. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1509293.
- [152] Sundar Murugappan, Subramani Sellamani, and Karthik Ramani. “Towards beautification of freehand sketches using suggestions”. In: *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling.* ACM. 2009, pp. 69–76.
- [153] N. Nagappan and T. Ball. “Static analysis tools as early indicators of pre-release defect density”. In: *Proc. 27th int. conf. on Software eng.* 2005, pp. 580–586. URL: <http://dl.acm.org/citation.cfm?id=1062455.1062558>.
- [154] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M German. “Quantifying programmers’ mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment”. In: *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM. 2014, pp. 448–451.

- [155] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. “What makes a good code example?: A study of programming Q&A in StackOverflow”. In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 25–34. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6405249.
- [156] Lisa Rubin Neal. “A system for example-based programming”. In: *ACM SIGCHI Bulletin*. Vol. 20. ACM, 1989, pp. 63–68. URL: <http://dl.acm.org/citation.cfm?id=67464>.
- [157] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. “Graph-based pattern-oriented, context-sensitive source code completion”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 69–79. URL: <http://dl.acm.org/citation.cfm?id=2337232>.
- [158] Laura R. Novick and Keith J. Holyoak. “Mathematical problem solving by analogy.” In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 17.3 (1991), p. 398. URL: <http://psycnet.apa.org/journals/xlm/17/3/398/>.
- [159] Stephen Oney and Joel Brandt. “Codelets: linking interactive documentation and example code in the editor”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 2697–2706. URL: <http://dl.acm.org/citation.cfm?id=2208664>.
- [160] Fred GWC Paas, Jeroen JG Van Merriënboer, and Jos J. Adam. “Measurement of cognitive load in instructional research”. In: *Perceptual and motor skills* 79.1 (1994), pp. 419–430. URL: <http://journals.sagepub.com/doi/abs/10.2466/pms.1994.79.1.419>.
- [161] Fred Paas, Alexander Renkl, and John Sweller. “Cognitive load theory and instructional design: Recent developments”. In: *Educational psychologist* 38.1 (2003), pp. 1–4.
- [162] John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. “Studying the language and structure in non-programmers’ solutions to programming problems”. In: *Int. J. Hum.-Comput. Stud.* 54.2 (Feb. 2001), pp. 237–264. DOI: 10.1006/ijhc.2000.0410. URL: <http://dx.doi.org/10.1006/ijhc.2000.0410>.
- [163] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. “Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow”. In: *Georgia Institute of Technology, Tech. Rep* (2012).
- [164] Jean Piaget. “Intellectual evolution from adolescence to adulthood”. In: *Human development* 15.1 (1972), pp. 1–12. URL: <http://www.karger.com/Article/Abstract/271225>.
- [165] Peter Pirolli and Margaret Recker. “Learning strategies and transfer in the domain of programming”. In: *Cognition and instruction* 12.3 (1994), pp. 235–275.
- [166] *PMD*. URL: <http://pmd.sourceforge.net/>.

- [167] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. “Mining StackOverflow to turn the IDE into a self-confident programming prompter”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 102–111. URL: <http://dl.acm.org/citation.cfm?id=2597077>.
- [168] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. “Prompter”. In: *Empirical Software Engineering* 21.5 (2016), pp. 2190–2231.
- [169] Thomas W Price, Yihuan Dong, and Tiffany Barnes. “Generating Data-driven Hints for Open-ended Programming.” In: *EDM*. 2016, pp. 191–198.
- [170] David F. Redmiles. “Reducing the variability of programmers’ performance through explained examples”. In: *Proceedings of the INTERACT’93 and CHI’93 Conference on Human Factors in Computing Systems*. ACM, 1993, pp. 67–73. URL: <http://dl.acm.org/citation.cfm?id=169082>.
- [171] Alexander Renkl. “Learning from worked-out examples: A study on individual differences”. In: *Cognitive science* 21.1 (1997), pp. 1–29.
- [172] Alexander Renkl, Robert K Atkinson, and Cornelia S Große. “How fading worked solution steps works—a cognitive load perspective”. In: *Instructional Science* 32.1-2 (2004), pp. 59–82.
- [173] Alexander Renkl, Robert K Atkinson, Uwe H Maier, and Richard Staley. “From example study to problem solving: Smooth transitions help learning”. In: *The Journal of Experimental Education* 70.4 (2002), pp. 293–315.
- [174] Alexander Renkl, Robin Stark, Hans Gruber, and Heinz Mandl. “Learning from worked-out examples: The effects of example variability and elicited self-explanations”. In: *Contemporary educational psychology* 23.1 (1998), pp. 90–108.
- [175] Lindsey E. Richland, Keith J. Holyoak, and James W. Stigler. “Analogy use in eighth-grade mathematics classrooms”. In: *Cognition and Instruction* 22.1 (2004), pp. 37–60. URL: http://www.tandfonline.com/doi/abs/10.1207/s1532690Xci2201_2.
- [176] Robert S. Rist et al. “Plans in programming: definition, demonstration, and development”. In: *first workshop on empirical studies of programmers on Empirical studies of programmers*. 1986, pp. 28–47. URL: https://books.google.com/books?hl=en&lr=&id=sswoYivNQVUC&oi=fnd&pg=PA28&dq=Plans+in+Programming:+Definition,+Demonstration,+and+Developmen&ots=agQcW1mfUI&sig=P6kY3KdQH13cq_mgfmUE7Bsa0vY.
- [177] Kelly Rivers and Kenneth R. Koedinger. “Automatic generation of programming feedback: A data-driven approach”. In: *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*. Vol. 50. 2013.

- [178] Romain Robbes, Mircea Lungu, and David Röthlisberger. “How do developers react to api deprecation?: the case of a smalltalk ecosystem”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 56.
- [179] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. “Impact of interruption style on end-user debugging”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 287–294. URL: <http://dl.acm.org/citation.cfm?id=985729>.
- [180] Martin P. Robillard. “What makes APIs hard to learn? Answers from developers”. In: *IEEE software* 26.6 (2009), pp. 27–34. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5287006.
- [181] Martin P. Robillard and Robert Deline. “A field study of API learning obstacles”. In: *Empirical Software Engineering* 16.6 (2011), pp. 703–732. URL: <http://link.springer.com/article/10.1007/s10664-010-9150-8>.
- [182] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D’Mello. “Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401. DOI: 10.1145/2568225.2568247. URL: <http://doi.acm.org/10.1145/2568225.2568247>.
- [183] M. B. Rosson, J. Ballin, and H. Nash. “Everyday Programming: Challenges and Opportunities for Informal Web Development”. In: *2004 IEEE Symposium on Visual Languages and Human Centric Computing*. Sept. 2004, pp. 123–130. DOI: 10.1109/VLHCC.2004.26.
- [184] Mary Beth Rosson and John M. Carroll. “Active Programming Strategies in Reuse”. In: *Proceedings of the 7th European Conference on Object-Oriented Programming*. Springer-Verlag, 1993, pp. 4–20. URL: <http://dl.acm.org/citation.cfm?id=679356>.
- [185] 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. URL: <http://dl.acm.org/citation.cfm?id=234530>.
- [186] Naiyana Sahavechaphan and Kajal Claypool. “XSnippet: Mining For Sample Code”. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. New York, NY, USA: ACM, 2006, pp. 413–430. DOI: 10.1145/1167473.1167508. URL: <http://doi.acm.org/10.1145/1167473.1167508>.
- [187] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. “Fully automatic assessment of programming exercises”. In: *Proc. 6th ann. conf. on ITiCSE*. ITiCSE ’01. New York, NY, USA: ACM, 2001, pp. 133–136. DOI: 10.1145/377435.377666. URL: <http://doi.acm.org/10.1145/377435.377666>.

- [188] Nicholas Sawadsky and Gail C. Murphy. “Fishtail: from task context to source code examples”. In: *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. ACM, 2011, pp. 48–51. URL: <http://dl.acm.org/citation.cfm?id=1984722>.
- [189] Christopher Scaffidi and Christopher Chambers. “Skill progression demonstrated by users in the Scratch animation environment”. In: *International Journal of Human-Computer Interaction* 28.6 (2012), pp. 383–398. URL: <http://www.tandfonline.com/doi/abs/10.1080/10447318.2011.595621>.
- [190] Christopher Scaffidi, Christopher Chambers, and Sheela Surisetty. “A Code-Centric Cluster-Based Approach for Searching Online Support Forums for Programmers”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 1032–1037.
- [191] Alan H. Schoenfeld and Douglas J. Herrmann. “Problem perception and knowledge structure in expert and novice mathematical problem solvers.” In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 8.5 (1982), p. 484. URL: <http://psycnet.apa.org/journals/xlm/8/5/484/>.
- [192] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. “Building empirical support for automated code smell detection”. In: *Proc. ACM-IEEE Int. Symp. on Empirical Software Eng. and Measurement. ESEM '10*. New York, NY, USA: ACM, 2010, 8:1–8:10. DOI: 10.1145/1852786.1852797. URL: <http://doi.acm.org/10.1145/1852786.1852797>.
- [193] Rolf Schwonke, Jörg Wittwer, Vincent Aleven, RJCM Salden, Carmen Krieg, and Alexander Renkl. “Can tutored problem solving benefit from faded worked-out examples”. In: *Proceedings of EuroCogSci*. Vol. 7. 2007, pp. 59–64.
- [194] Michael J. Scott and Gheorghita Ghinea. “On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice”. In: *IEEE Transactions on Education* 57.3 (2014), pp. 169–174. URL: <http://ieeexplore.ieee.org/abstract/document/6662493/>.
- [195] *Scratch - Imagine, Program, Share*. URL: <https://scratch.mit.edu/discuss/> (visited on 03/26/2018).
- [196] *Scratch - Imagine, Program, Share*. URL: <https://scratch.mit.edu/> (visited on 03/12/2018).
- [197] *Scratch - Tips*. URL: <https://scratch.mit.edu/> (visited on 03/26/2018).
- [198] Ben Shneiderman. “Exploratory experiments in programmer behavior”. In: *International Journal of Computer & Information Sciences* 5.2 (1976), pp. 123–143. URL: <http://link.springer.com/article/10.1007/BF00975629>.
- [199] Ben Shneiderman and Richard Mayer. “Syntactic/semantic interactions in programmer behavior: A model and experimental results”. In: *International Journal of Parallel Programming* 8.3 (1979), pp. 219–238. URL: <http://www.springerlink.com/index/Q26TJ60786117943.pdf>.

- [200] Elliot Soloway and Kate Ehrlich. “Empirical studies of programming knowledge”. In: *IEEE Transactions on Software Engineering* 5 (1984), pp. 595–609. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5010283.
- [201] *SourceMonitor*. URL: <http://www.campwoodsw.com/sourcemonitor.html>.
- [202] Sai Krishna Sripada and Y Raghu Reddy. “Code comprehension activities in undergraduate software engineering course-a case study”. In: *Software Engineering Conference (ASWEC), 2015 24th Australasian*. IEEE, 2015, pp. 68–77.
- [203] *Stack Overflow*. URL: <http://stackoverflow.com/>.
- [204] M.-A. Storey. “Theories, methods and tools in program comprehension: Past, present and future”. In: *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 2005, pp. 181–191. URL: <http://ieeexplore.ieee.org/abstract/document/1421034/>.
- [205] *StyleCop*. URL: <http://stylecop.codeplex.com/>.
- [206] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. “Improving API documentation using API usage information”. In: *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*. IEEE, 2009, pp. 119–126. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5295283.
- [207] Jeffrey Stylos and Brad A. Myers. “Mica: A web-search tool for finding api components and examples”. In: *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 195–202. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1698785.
- [208] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. “Code comprehension problems as learning events”. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 2012, pp. 81–86.
- [209] Sheela Surisetty, Catherine Law, and Chris Scaffidi. “Behavior-based clustering of visual code”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 261–269. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7357225.
- [210] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D’Antoni, and Björn Hartmann. “Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments”. In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2017, pp. 2951–2958. URL: <http://dl.acm.org/citation.cfm?id=3053187>.
- [211] John Sweller. “Cognitive load theory, learning difficulty, and instructional design”. In: *Learning and instruction* 4.4 (1994), pp. 295–312.
- [212] John Sweller and Graham A. Cooper. “The use of worked examples as a substitute for problem solving in learning algebra”. In: *Cognition and Instruction* 2.1 (1985), pp. 59–89. URL: http://www.tandfonline.com/doi/abs/10.1207/s1532690xci0201_3.

- [213] MohammadReza Tavakoli, Abbas Heydarnoori, and Mohammad Ghafari. “Improving the quality of code snippets in stack overflow”. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1492–1497. URL: <http://dl.acm.org/citation.cfm?id=2851789>.
- [214] *The MIT App Inventor Library: Documentation & Support | Explore MIT App Inventor*. URL: <http://appinventor.mit.edu/explore/library.html> (visited on 03/26/2018).
- [215] Suresh Thummalapenta and Tao Xie. “Parseweb: a programmer assistant for reusing open source code on the web”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213.
- [216] Suresh Thummalapenta and Tao Xie. “Parseweb: a programmer assistant for reusing open source code on the web”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213. URL: <http://dl.acm.org/citation.cfm?id=1321663>.
- [217] Nghi Truong, Paul Roe, and Peter Bancroft. “Static analysis of students’ Java programs”. In: *Proc. 6th Australasian Conf. on Computing Educ.- Volume 30*. ACE ’04. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 317–325. URL: <http://dl.acm.org/citation.cfm?id=979968.980011>.
- [218] *Tutorials for App Inventor | Explore MIT App Inventor*. URL: <http://appinventor.mit.edu/explore/ai2/tutorials.html>.
- [219] Iris Vessey. “Expertise in debugging computer programs: A process analysis”. In: *International Journal of Man-Machine Studies* 23.5 (1985), pp. 459–494. URL: <http://www.sciencedirect.com/science/article/pii/S0020737385800547>.
- [220] Iris Vessey. “Expert-novice knowledge organization: An empirical investigation using computer program recall”. In: *Behaviour & Information Technology* 7.2 (1988), pp. 153–171. URL: <http://www.tandfonline.com/doi/abs/10.1080/01449298808901870>.
- [221] Camilo Vieira, Alejandra J Magana, Michael L Falk, and R Edwin Garcia. “Writing in-code comments to self-explain in computational science and engineering education”. In: *ACM Transactions on Computing Education (TOCE)* 17.4 (2017), p. 17.
- [222] Arto Vihavainen, Craig S. Miller, and Amber Settle. “Benefits of Self-explanation in Introductory Programming”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, 2015, pp. 284–289. DOI: 10.1145/2676723.2677260. URL: <http://doi.acm.org/10.1145/2676723.2677260>.
- [223] Anneliese Von Mayrhauser and A. Marie Vans. “Program comprehension during software maintenance and evolution”. In: *Computer* 28.8 (1995), pp. 44–55. URL: <http://ieeexplore.ieee.org/abstract/document/402076/>.

- [224] Mark Ward and John Sweller. “Structuring effective worked examples”. In: *Cognition and instruction* 7.1 (1990), pp. 1–39. URL: http://www.tandfonline.com/doi/abs/10.1207/s1532690xcio701_1.
- [225] Michael S. Ware and Christopher J. Fox. “Securing Java code: heuristics and an evaluation of static analysis tools”. In: *Proc. workshop on Static analysis*. SAW ’08. New York, NY, USA: ACM, 2008, pp. 12–21. DOI: 10.1145/1394504.1394506. URL: <http://doi.acm.org/10.1145/1394504.1394506>.
- [226] Christopher Watson, Frederick Li, and Jamie Godwin. “BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair”. In: *Advances in Web-Based Learning-ICWL* (2012), pp. 228–239. URL: <http://www.springerlink.com/index/X52012344853048N.pdf>.
- [227] Gerhard Weber and Antje Mollenberg. “ELM-PE: A Knowledge-based Programming Environment for Learning LISP.” In: (1994). URL: <http://eric.ed.gov/?id=ED388302>.
- [228] David Weintrop and Uri Wilensky. “To block or not to block, that is the question: students’ perceptions of blocks-based programming”. In: *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 199–208. URL: <http://dl.acm.org/citation.cfm?id=2771860>.
- [229] Susan Wiedenbeck. “Novice/expert differences in programming skills”. en. In: *International Journal of Man-Machine Studies* 23.4 (Oct. 1985), pp. 383–390. DOI: 10.1016/S0020-7373(85)80041-9. URL: <http://linkinghub.elsevier.com/retrieve/pii/S0020737385800419>.
- [230] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. “Snipmatch: using source code context to enhance snippet retrieval and parameterization”. In: *Proceedings of the 25th annual acm symposium on user interface software and technology*. ACM, 2012, pp. 219–228. URL: <http://dl.acm.org/citation.cfm?id=2380145>.
- [231] Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. “Harnessing curiosity to increase correctness in end-user programming”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2003, pp. 305–312. URL: <http://dl.acm.org/citation.cfm?id=642665>.
- [232] Benjamin Xiang-Yu Xie. “Progression of computational thinking skills demonstrated by app inventor users”. PhD thesis. Massachusetts Institute of Technology, 2016. URL: <https://dspace.mit.edu/handle/1721.1/106395>.
- [233] Benjamin Xie and Hal Abelson. “Skill progression in MIT app inventor”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE, 2016, pp. 213–217. URL: <http://ieeexplore.ieee.org/abstract/document/7739687/>.

- [234] Seungwon Yang, Carlotta Domeniconi, Matt Revelle, Mack Sweeney, Ben U. Gelman, Chris Beckley, and Aditya Johri. “Uncovering trajectories of informal learning in large online communities of creators”. In: *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 2015, pp. 131–140. URL: <http://dl.acm.org/citation.cfm?id=2724674>.
- [235] Annie TT Ying and Martin P. Robillard. “Code fragment summarization”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 655–658. URL: <http://dl.acm.org/citation.cfm?id=2494587>.
- [236] Annie Ying and Martin P. Robillard. “Selection and presentation practices for code example summarization”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 460–471. URL: <http://dl.acm.org/citation.cfm?id=2635877>.
- [237] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. “Automatic parameter recommendation for practical API usage”. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 826–836. URL: <http://dl.acm.org/citation.cfm?id=2337321>.
- [238] Yan Zhang, Sheela Surisetty, and Christopher Scaffidi. “Assisting comprehension of animation programs through interactive code visualization”. In: *Journal of Visual Languages & Computing* 24.5 (2013), pp. 313–326.
- [239] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. “MAPO: Mining and Recommending API Usage Patterns”. In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Italy: Springer-Verlag, 2009, pp. 318–343. DOI: 10.1007/978-3-642-03013-0_15. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_15.
- [240] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. “MAPO: Mining and recommending API usage patterns”. In: *ECOOP 2009—Object-Oriented Programming*. Springer, 2009, pp. 318–343. URL: http://link.springer.com/chapter/10.1007/978-3-642-03013-0_15.
- [241] Sedigheh Zolaktaf and Gail C. Murphy. “What to learn next: recommending commands in a feature-rich environment”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 1038–1044. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7424457.

Appendix A

Understanding How Novices Use Examples Study Materials

A.1 Computing history survey

1. How old are you?
2. What is your current grade in school?
3. What is your gender? (choose one) a) Female b) Male c) Not specified
4. What kind of school do you go to? (choose one) a) Public school b) Private school c) Home-schooled
5. How good do you think you are with computers? (choose one) a) Very good b) Good c) Fair d) Poor
6. Have you ever written a computer program? (choose one) a) Don't know b) Yes c) No

7. How would you characterize your computer programming experience? (choose one) a) No programming experience. b) I have programmed a few times as part of an activity. c) I enjoy programming in my free time. d) I'm not sure. e) Other:

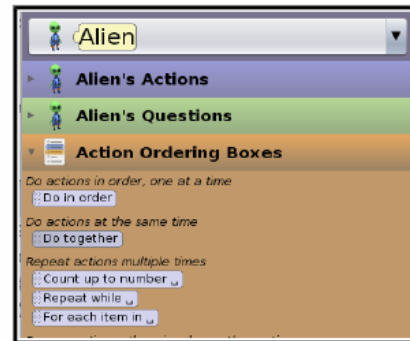
8. Have you ever used the following computer programming software? (circle all that apply)
a) Alice b) LEGO Mindstorms c) Looking Glass d) Scratch e) Robotics f) Programming languages such as: Javascript, Python, C++, Java, Visual Basic, C#, Processing g) Other
Programming Software:

9. Have you participated in the following programming activities? (circle all that apply)
a) Hour of code. (code.org) b) Programming at school as part of classroom activity. c) A programming camp or after school workshop. d) Programming at an event (examples: scouting, academy of science, science center) e) I program a computer at home. f) Looking Glass research study. (Not including today) g) None – I have never participated in a programming activity. h) Other:

A.2 Intro instructions

1. Drag a **'Do in order'** action ordering box into your animation.

Hint: You can find a *'Do in order'* block in Action Ordering Boxes.



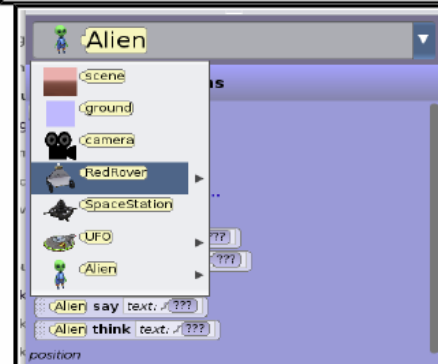
2. Drag a **'Alien turn left 0.25 rotations'** action inside of the **'Do in order'** action ordering box.

Hint: Drag the Alien's *'turn left'* action into the *'Do in order'* block.



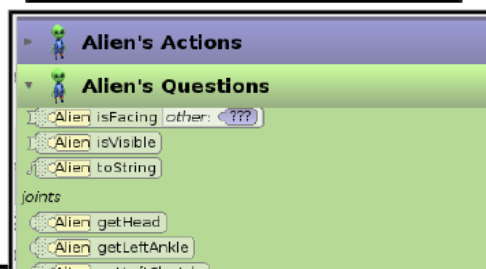
3. Drag a **'RedRover say Hello'** action inside of the **'Do in order'** action ordering box.

Hint: You can switch characters in the drop-down menu and then drag the *'say'* action into the *'Do in order'* block.

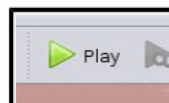


4. Replace the **'Alien'** with the **'Alien getHead'** question in the **'turn left 0.25'** action.

Hint: You can find *'Alien getHead'* question in the Questions and replace Alien with Alien getHead.



5. Play your animation by clicking the Play button.

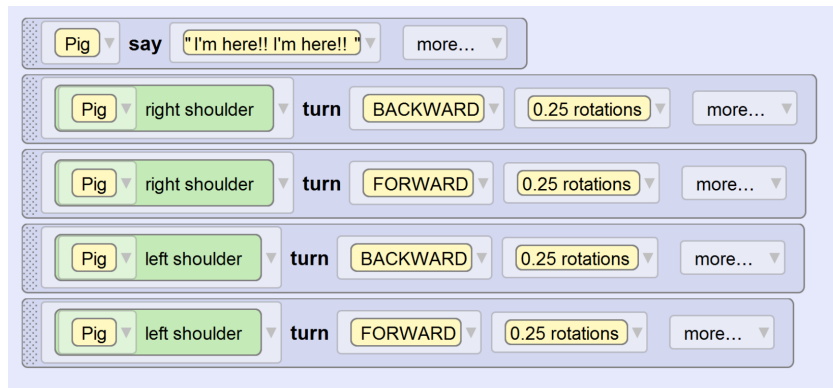


6. When you are finished, STOP and raise your hand.

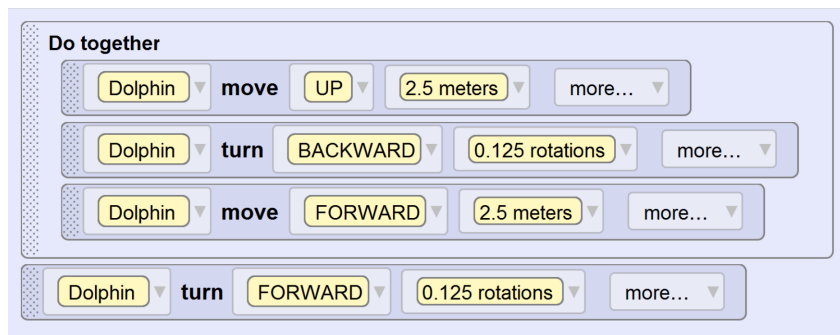
Figure A.1: The intro task sheet.
[210]

A.3 Task programs and solutions

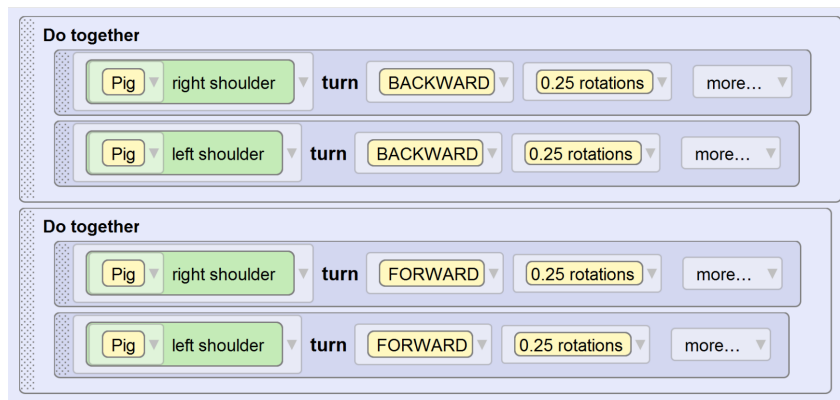
Do Together task instructions: Make the pig wave both of his hands up and then both down to signal the helicopter.



Do together
task code



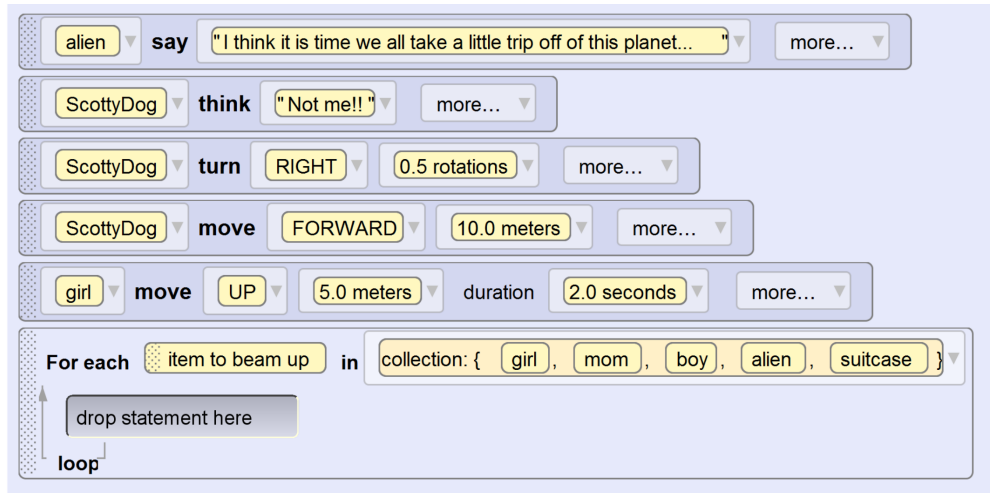
Do together
example code



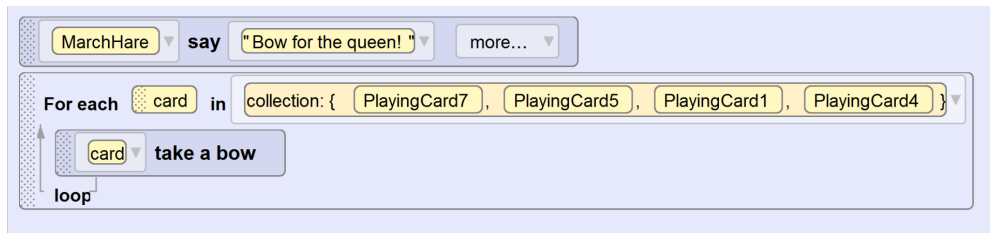
Do together
task solution

Figure A.2: Do together task

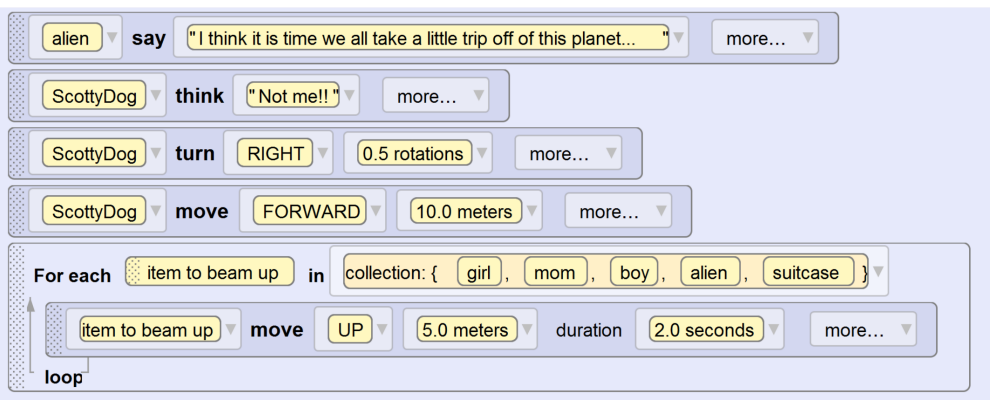
For each loop Task instructions: Make the girl, mom, boy, alien and suitcase fly to the spaceship WITHOUT ADDING OR REMOVING any actions, questions or action ordering boxes.



For each loop
task code



For each loop
example code



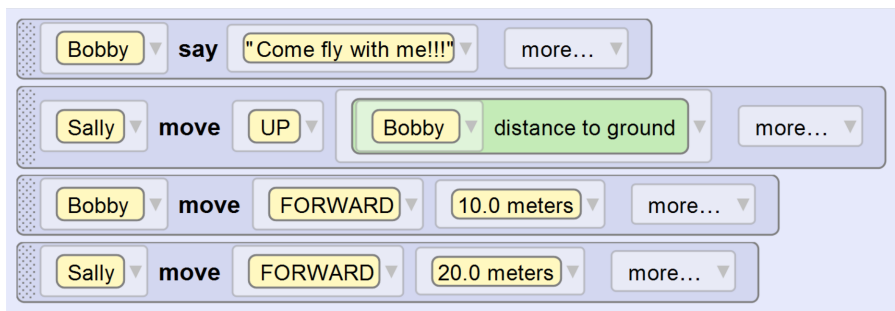
For each loop
task solution

Figure A.3: For each loop task

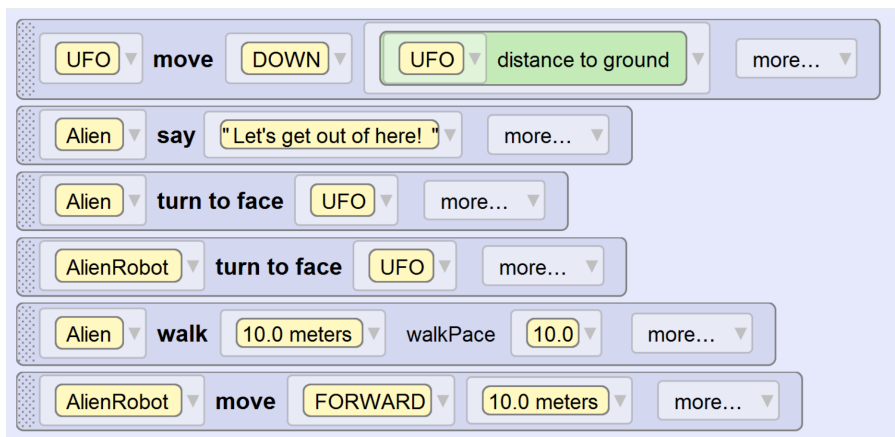
Function task instructions: Make the ufo land exactly on the ground by WITHOUT adding any ‘move’ actions AND WITHOUT creating a new number.



Function
task code



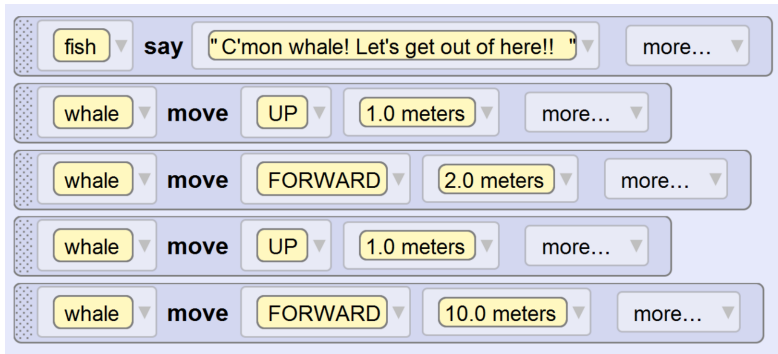
Function
example code



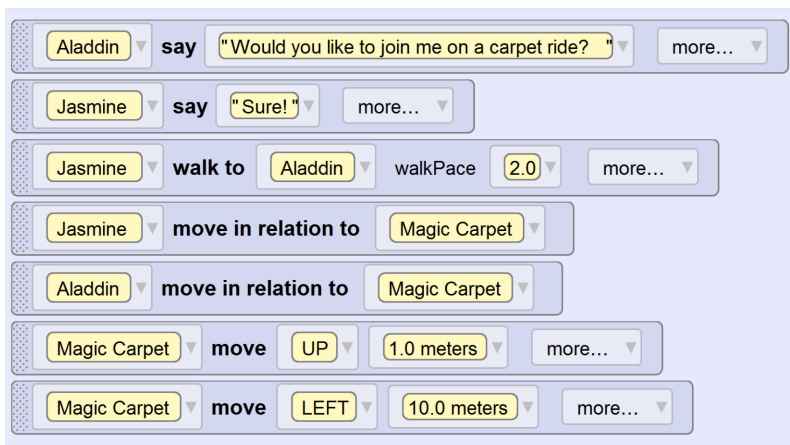
Function
task solution

Figure A.4: Function task

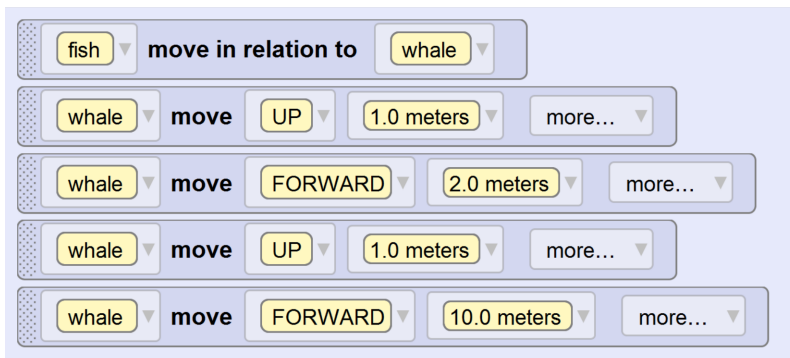
API method task instructions: Make the orange fish stay with the whale as the whale avoids the other creatures by ONLY ADDING ONE action, action ordering box or question and WITHOUT REMOVING. anything.



API method task code



API method example code



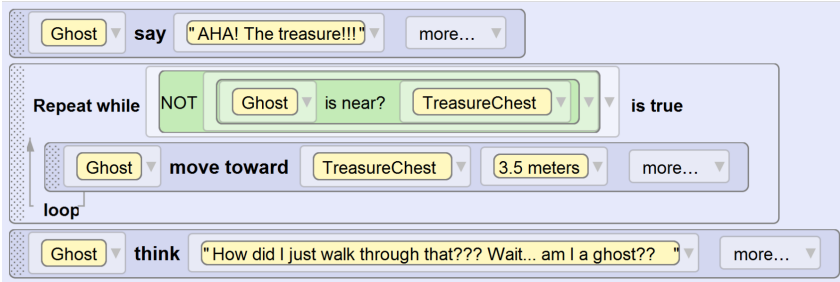
API method task solution

Figure A.5: API method task

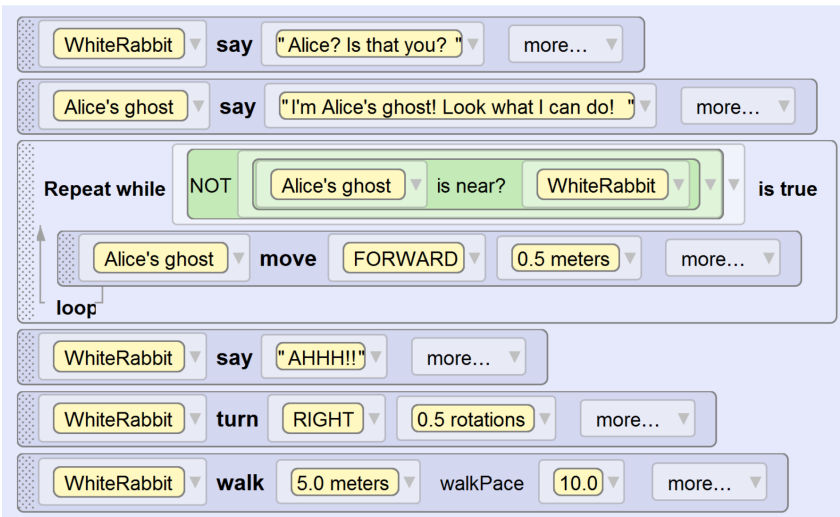
While loop task instructions: Make Alice the ghost go into the same space as the rabbit and then have the rabbit run away by **ONLY MAKING ONE MODIFICATION**.



While Loop
task code



While Loop
example code



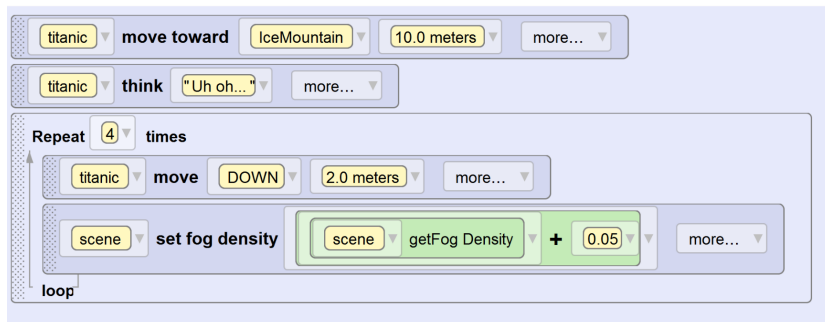
While Loop
task solution

Figure A.6: While loop task

Repeat loop task instructions: Make the bunny do 2 jumping jacks and then hop 3 times before lying down to rest by ONLY ADDING a total of TWO actions, questions or action ordering boxes.



Repeat Loop
task code



Repeat Loop
example code



Repeat Loop
task solution

Figure A.7: Repeat loop task

A.4 Interview questions

Questions asked at the midpoint:

- What have you tried so far and why?
- What would you do next? Please explain it like you are telling your partner how to do it. Why?
- Have you used the example at all? Why or why not?

Questions asked at the end of each task:

- If they correctly completed the task: How did you figure it out?
- If they did not complete the task: Did you have any ideas about how to finish it and how would you do it?
- All participants: What did you try?
- All participants: How does the example work?

Appendix B

Comparing Novices and Experts Study Materials

B.1 Instructions

In this HIT you will be asked to remember and then recall code. You do not need to have any experience with code to complete this task. We only ask that you try to do the best that you can.

Overall HIT structure (approx. 30 mins):

You will have 3 chances to study and recall each piece of code. ****WARNING:** if you copy/paste or screenshot rather than memorizing and recalling, you will not be paid (the logging will notify us of this).**

You will do this for 4 pieces of code. Two pieces of code will have problems.

The code will control an animation. For each piece of code, you will first view the animation scene. Then you will be asked to memorize some code, which you will see on the next page.

B.2 Demographic survey

What is your gender?

- male
- female
- other or prefer not to specify

What is your programming experience?

- I have never programmed or coded before
- I program once in a while (i.e. once a week or month).
- I used to program once in a while or often.
- I program on an everyday basis.

How did you learn about programming?

- I have never learned about programming.
- I have learned about programming or coding informally (i.e. from an online tutorial or Q&A website).
- I have learned about programming or coding from an online course like EdX or Coursera.

- I have learned about programming or coding from a high school course or extra-curricular activity.
- I have learned about programming or coding from a college or university course or courses.

Which languages have you used to program?

- I have never programmed.
- C/C++
- Java
- Javascript
- Python
- PHP
- SQL
- Other

B.3 Tasks

For each task

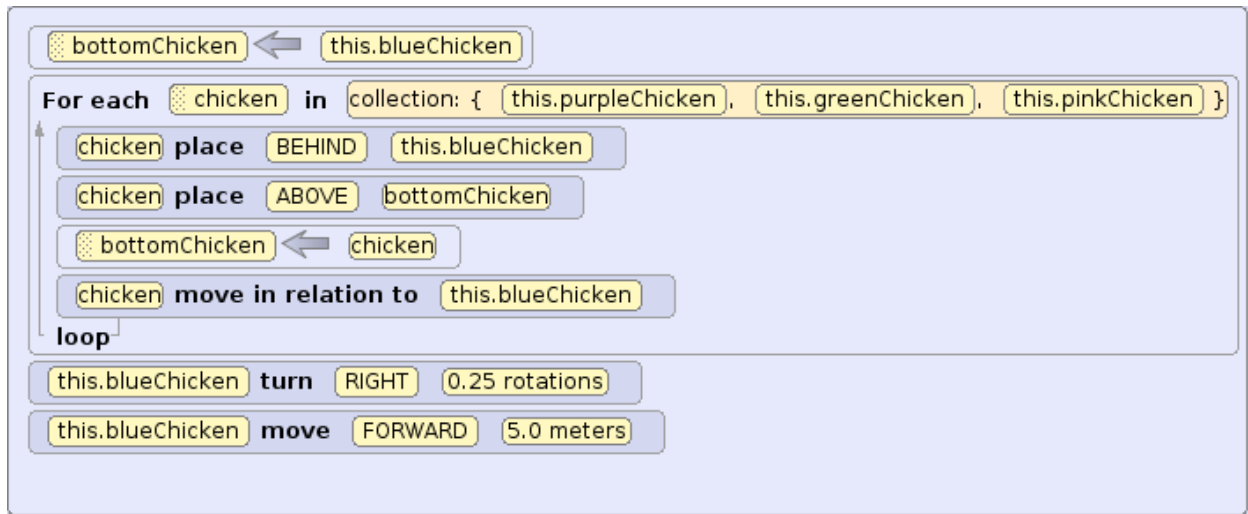


Figure B.1: For each loop block code snippet

```
Chicken bottomChicken = this.blueChicken;
for( Chicken chicken : new Chicken[] { this.purpleChicken, this.greenChicken, this.pinkChicken} ) {
    chicken.place( BEHIND, this.blueChicken );
    chicken.place( ABOVE, bottomChicken );
    bottomChicken = chicken;
    chicken.setVehicle( this.blueChicken );
}
this.blueChicken.turn( RIGHT, 0.25 );
this.blueChicken.move( FORWARD, 5.0 );
```

Figure B.2: For each loop text code snippet

Problem:

Write less than 5 lines of code that make 5 dogs (named sparky, fluffy, rodger, red, and dodger) say 'ruff'. Write the code in the same style as the code shown with this problem.

Repeat While Loop task

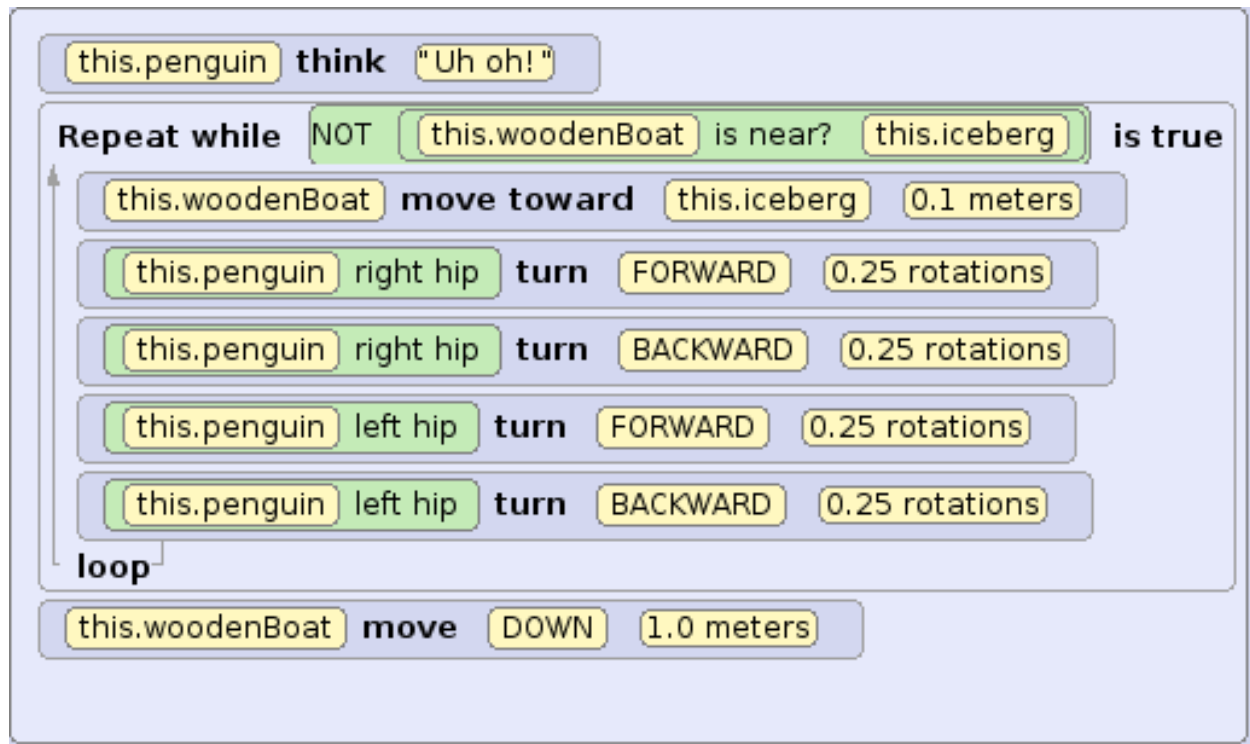


Figure B.3: Repeat While loop block code snippet

```
this.penguin.think( "Uh oh!" );  
while ( !this.woodenBoat.isCollidingWith( this.iceberg ) ) {  
    this.woodenBoat.moveToToward( this.iceberg, 0.1, BEGIN_AND_END_ABRUPTLY );  
    this.penguin.getRightHip().turn( FORWARD, 0.25 );  
    this.penguin.getRightHip().turn( BACKWARD, 0.25 );  
    this.penguin.getLeftHip().turn( FORWARD, 0.25 );  
    this.penguin.getLeftHip().turn( BACKWARD, 0.25 );  
}  
this.woodenBoat.move( DOWN, 1.0 );
```

Figure B.4: Repeat While loop text code snippet

Problem:

Write code that makes a boy named Henry take a step toward a spider until he is right next to it. Write the code in the same style as the code shown with this problem.

Simple Repeat Loop task

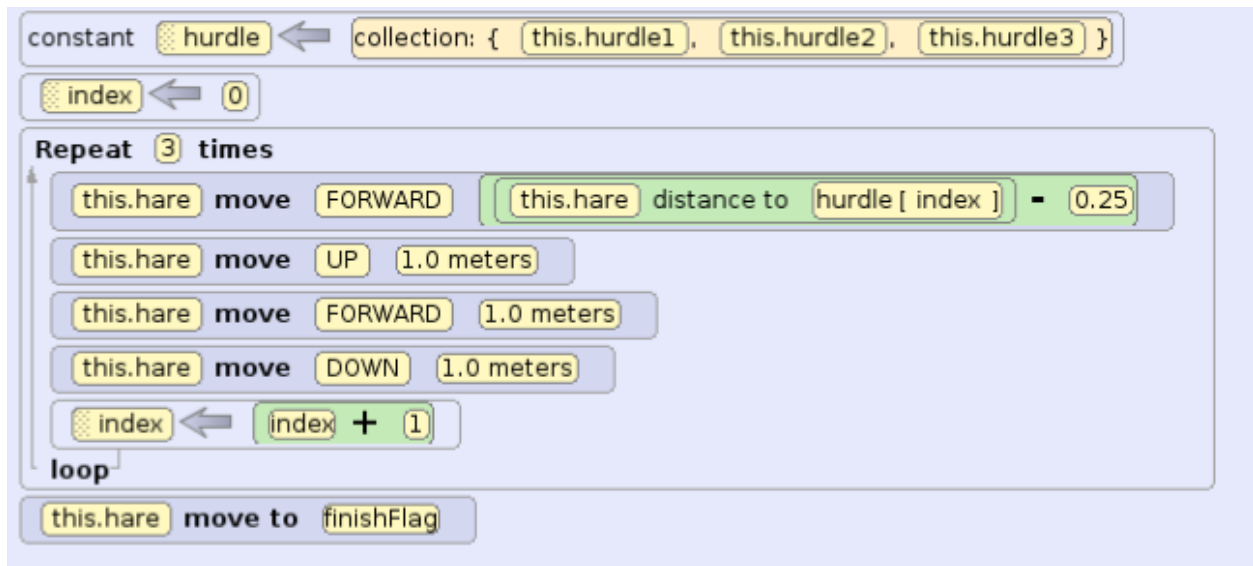


Figure B.5: Simple Repeat loop block code snippet

```
final TrackHurdle[] hurdles = new TrackHurdle[] {this.hurdle1, this.hurdle2, this.hurdle3 };
for( Integer index = 0; index<3; index++ ) {
    this.hare.move( FORWARD, this.hare.getDistanceTo( hurdles[index] )-0.25 );
    this.hare.move( UP, 1.0 );
    this.hare.move( FORWARD, 1.0 );
    this.hare.move( DOWN, 1.0 );
}
this.hare.moveTo( finishFlag );
```

Figure B.6: Simple Repeat loop text code snippet

Problem:

Write less than 5 lines of code that make a dancer do a back-flip 10 times. Write the code in the same style as the code shown with this problem.

Conditional task

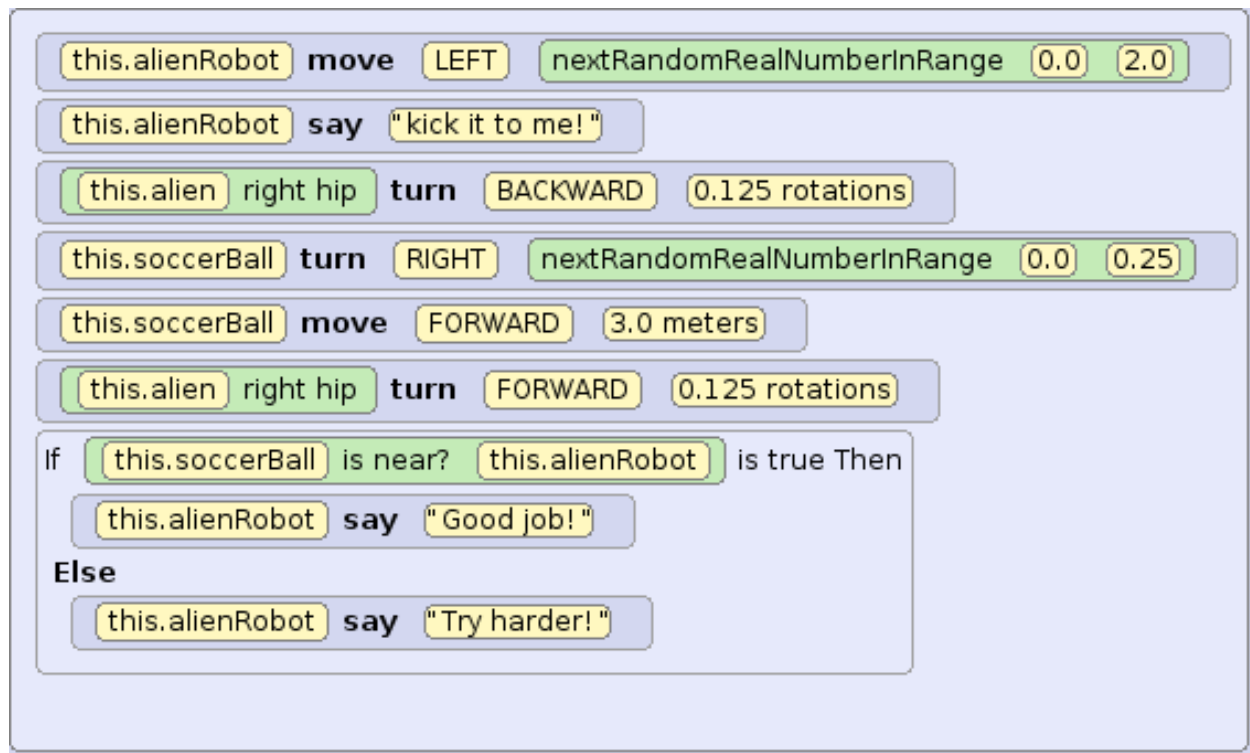


Figure B.7: Conditional block code snippet

```
this.alienRobot.move( LEFT, nextDoubleInRange( 0.0, 2.0 ), duration( 0.125 ) );
this.alienRobot.say( "kick it to me!" );
this.alien.getRightHip().turn( BACKWARD, 0.125 );
this.soccerBall.turn( RIGHT, nextDoubleInRange( 0.0, 0.25 ), duration( 0.125 ) );
this.soccerBall.move( FORWARD, 3.0 );
this.alien.getRightHip().turn( FORWARD, 0.125 );
if( this.soccerBall.isCollidingWith( this.alienRobot ) ) {
    this.alienRobot.say( "Good job!" );
} else {
    this.alienRobot.say( "Try harder!" );
}
```

Figure B.8: Conditional text code snippet

Problem:

Write code that makes 'sportscar' turn if it is near 'minivan' and otherwise, move forward. Write the code in the same style as the code shown with this problem.

B.4 Post-task survey

We asked participants to rate difficulty and mental effort on the following scales:

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

Figure B.9: Difficulty Scale

Very, Very Low Mental Effort	Very Low Mental Effort	Low Mental Effort	Rather Low Mental Effort	Neither Low nor High Mental Effort	Rather High Mental Effort	High Mental Effort	Very High Mental Effort	Very, Very High Mental Effort
------------------------------	------------------------	-------------------	--------------------------	------------------------------------	---------------------------	--------------------	-------------------------	-------------------------------

Figure B.10: Mental Effort Scale

Growth Mindset Scale: “I do not think I can really change my aptitude for programming.”

“I have a fixed level of programming aptitude, and not much can be done to change it.”

“I can learn new things about code, but I cannot change my basic aptitude for programming.”

“I believe I am able to achieve a high level of programming aptitude, with enough practice. ”

“I do believe I can change my aptitude for programming.”

Appendix C

Exploring Types of Suggestions Study Materials

C.1 Skill trees

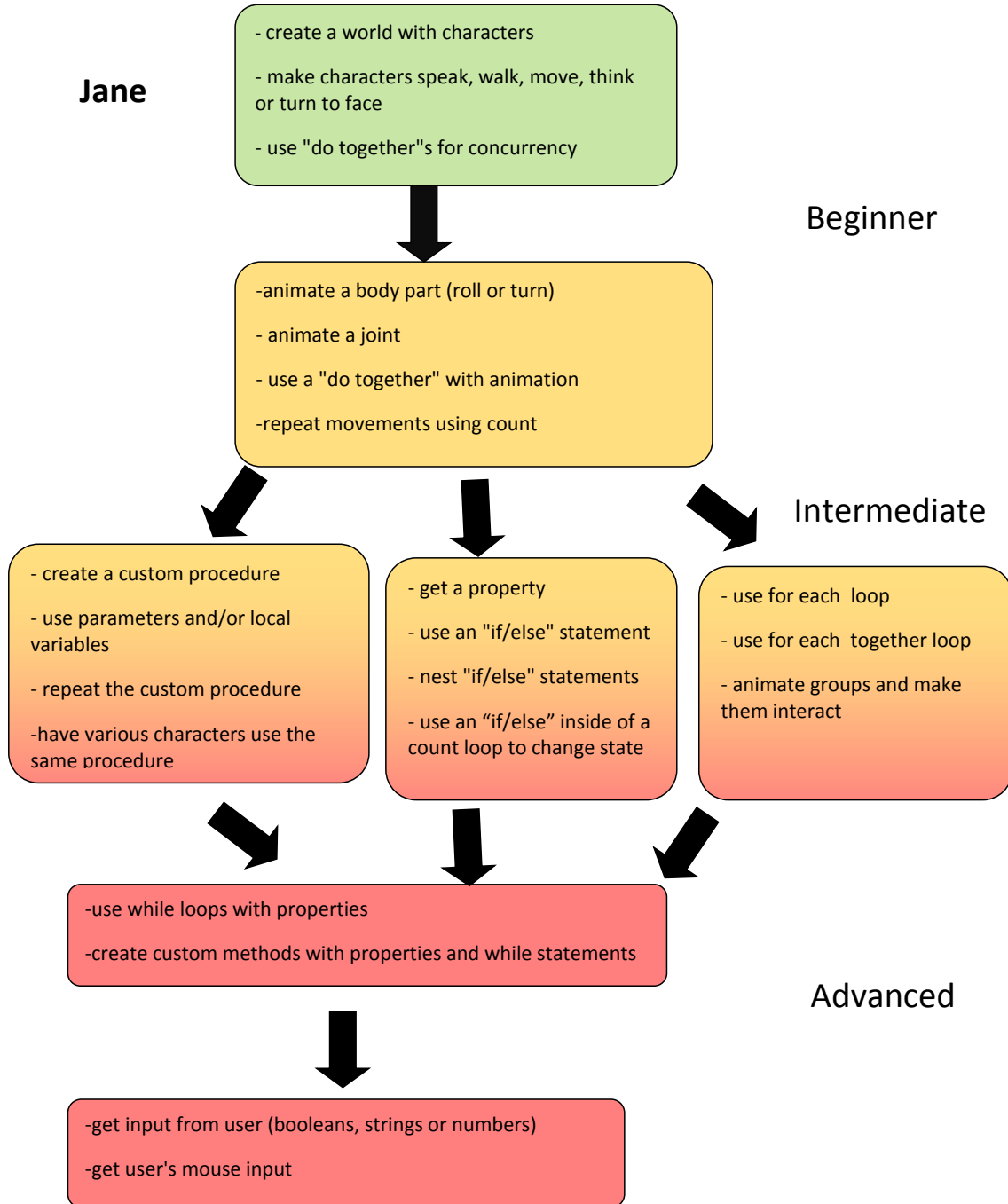


Figure C.1: Skill tree for 'Jane'

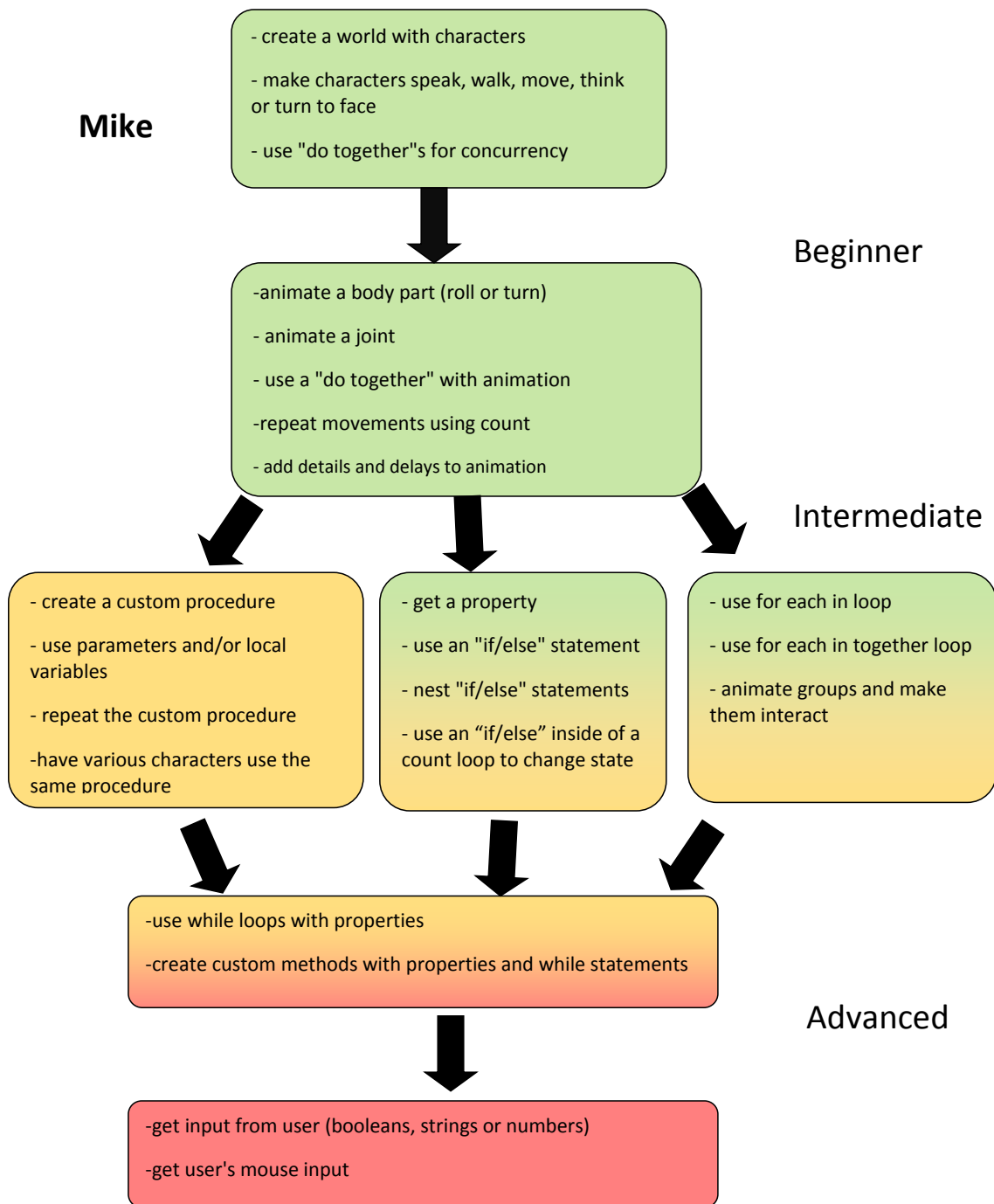


Figure C.2: Skill tree for 'Mike'

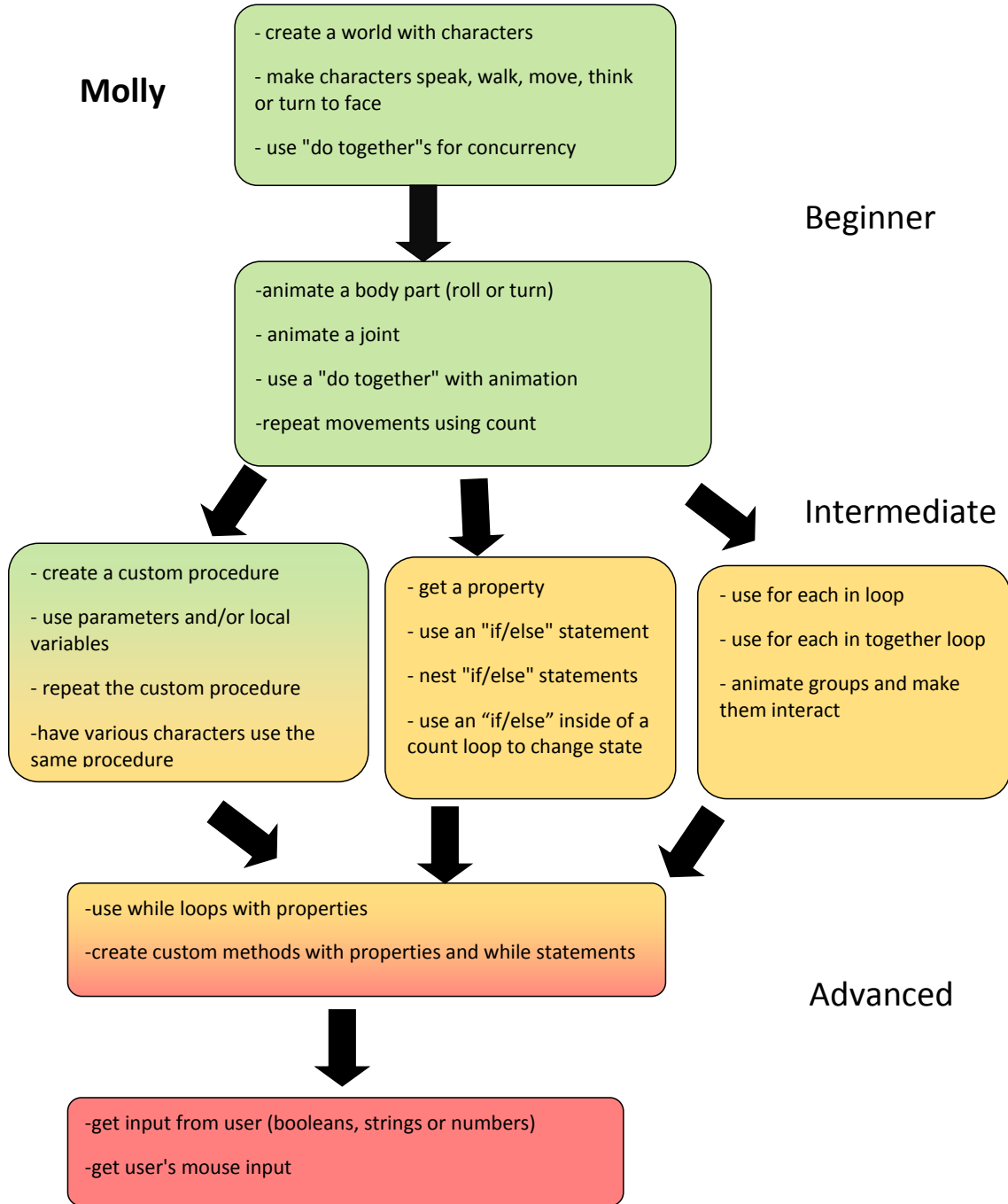


Figure C.3: Skill tree for 'Molly'

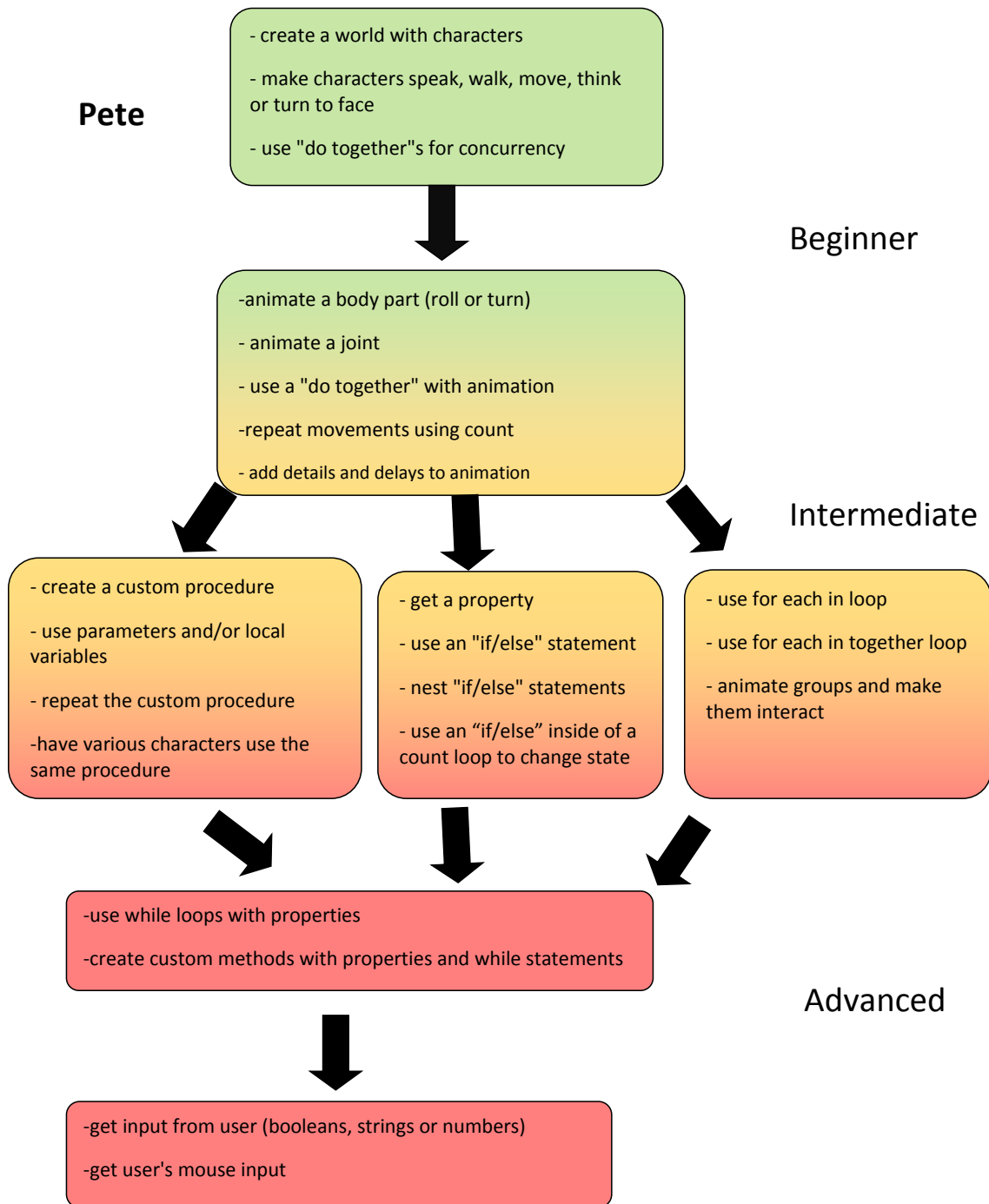


Figure C.4: Skill tree for 'Pete'

C.2 Programs and pre-made suggestions

Jane 1

Modification: In places where characters are talking to each other, in the modified version, they turn their heads, making the animation more realistic.

Student code:

Modified code:

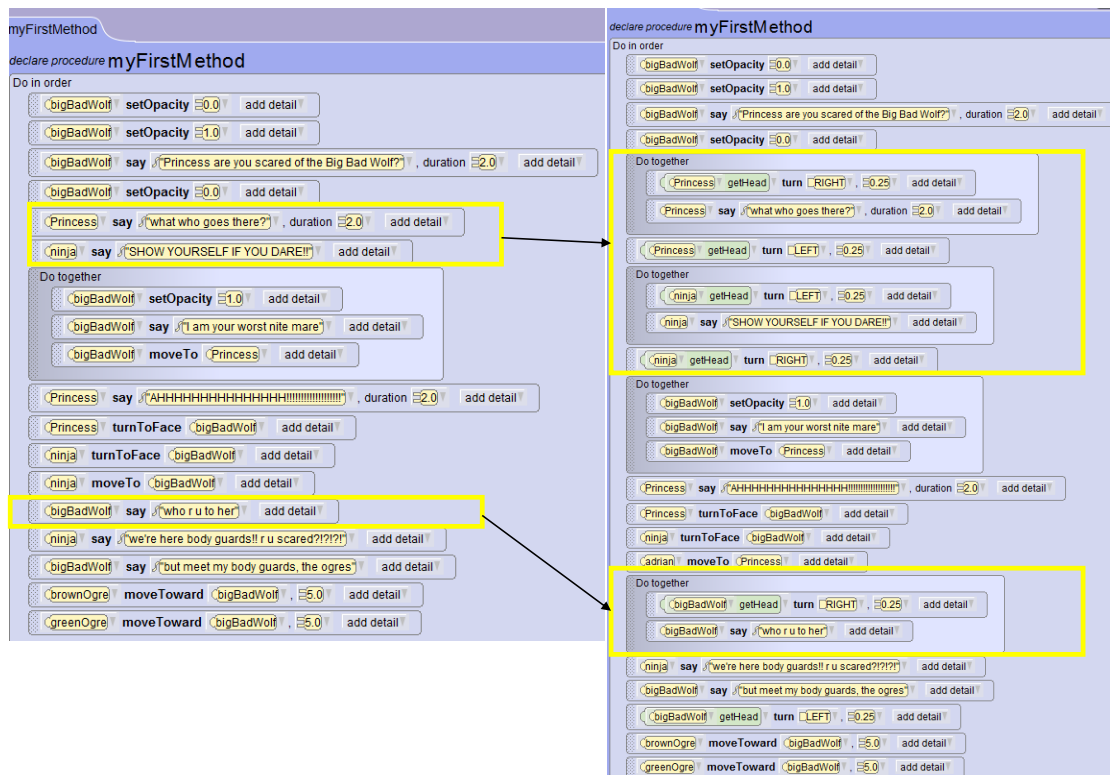
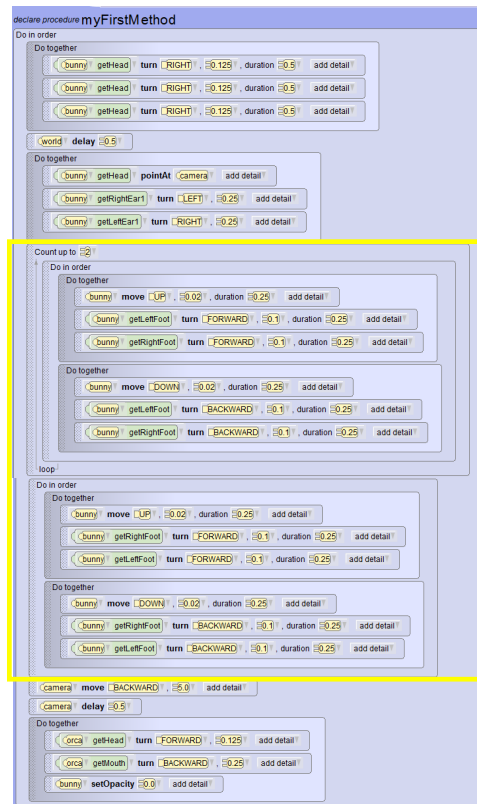


Figure C.5: One of the pre-made suggestions for ‘Jane’

Pete 1

Modification: The three foot wiggles were placed in a custom function called wiggle and consolidated into one count loop that runs wiggle three times.

Student code:



Modified code:

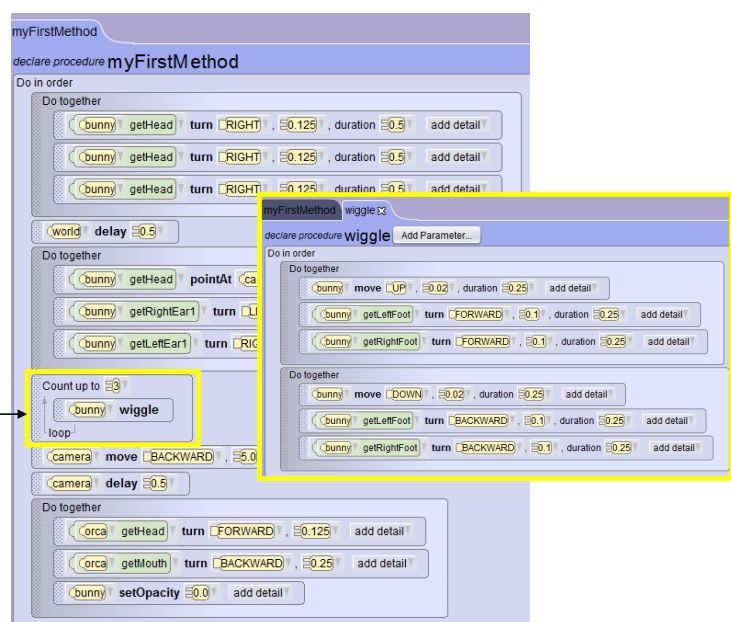
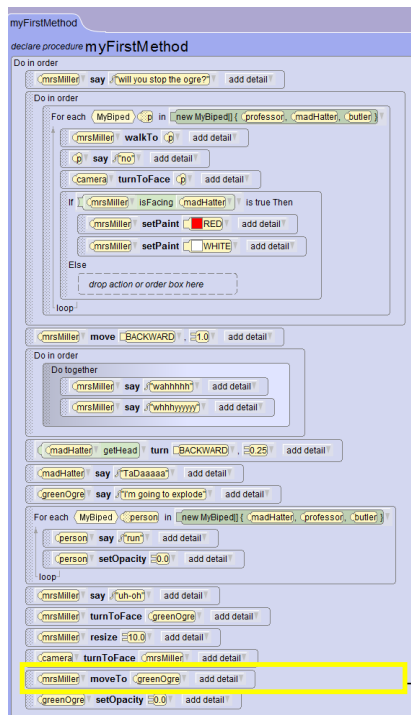


Figure C.6: One of the pre-made suggestions for ‘Pete’

Mike 1

Modification: Changed mrsMiller from “moving to” the ogre to moving only until she is 5 away from the ogre (“moveto” placed mrs Miller at the exact location of the ogre).

Student code:



Modified code:



Figure C.7: One of the pre-made suggestions for ‘Mike’

Molly 1

Modification: Instead of having four separate bow functions that all produced the same motion, they were replaced with one “bow” function that all of the characters can use.

Student code:

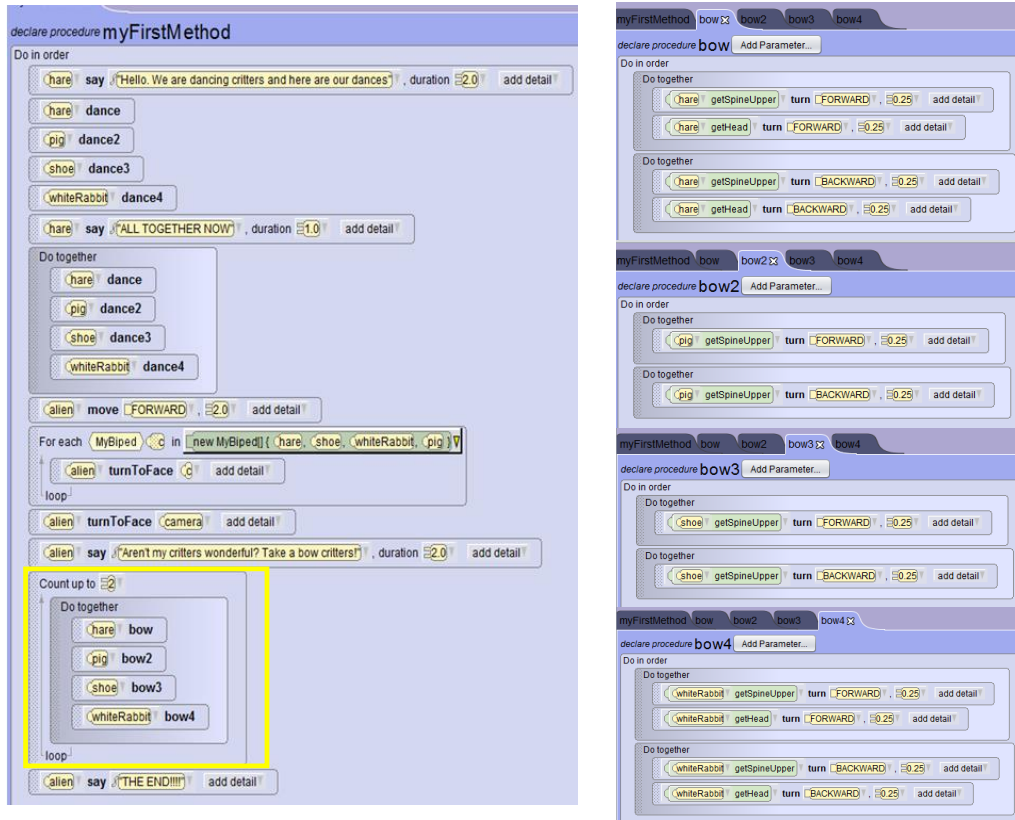


Figure C.8: One of the pre-made suggestions for ‘Molly’, part 1

Molly 1

Modified code:

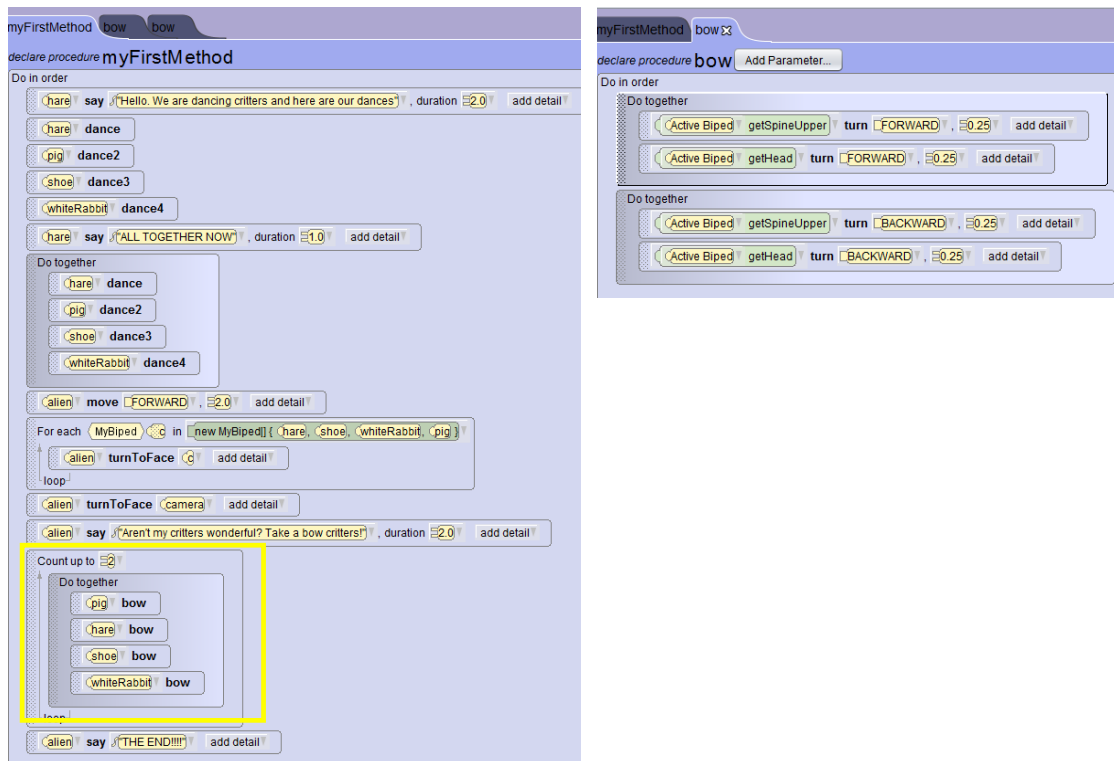


Figure C.9: One of the pre-made suggestions for 'Molly', part 2

C.3 Template

Please describe the rule in English in the form “if
,thenthecodefollowstherule”

Please code the rule in your preferred language’s pseudocode. Return TRUE if it follows the rule and FALSE if it does not

Appendix D

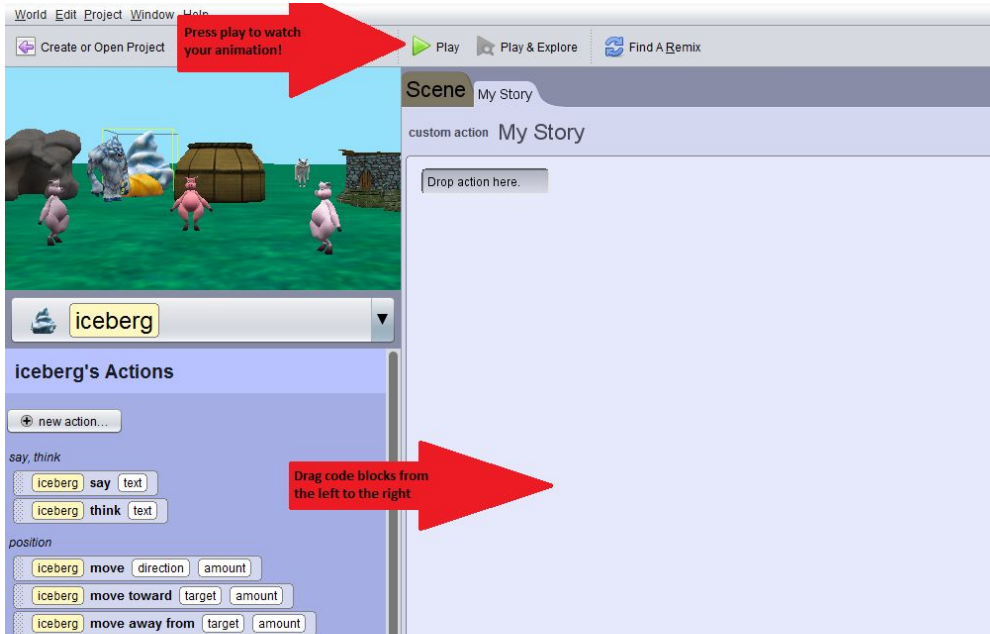
Example Guru for API Methods

Study Materials

D.1 Instruction sheets

1. Double click on 1 and follow the instructions in the 'note' to create an animation!

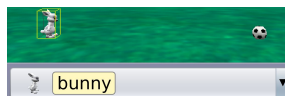
When the time is up, an alert will pop up. Complete the survey and then move on to the next task.
If you've never done this before, look at the description below or raise your hand for help!




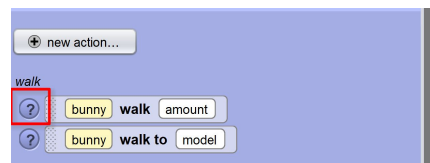
2. Double click 2 and follow the instructions:

Note
Make the bunny walk faster WITHOUT adding or removing any blocks.

FIRST: Select the bunny



THEN: Click the  button next to the bunny walk in the left menu.



THEN: click on 'Show more options' to see more examples.



[238]



When you are done, raise your hand and we will check to make sure it is correct.

D.2 Templates









D.3 Suggestions, rules, and examples

Code Sug- gested	Suggestion Title	Primary Exam- ple Title	Secondary Exam- ple Title	Rule Summary
AnimationStyle argument	Make [...] move smoothly	Use animation style to make movements smooth	Use animation style to create a pause a bit between two moves	Multiple move- ments in a row by the same object (means that they might look better with a specified animation style).
AnimationStyle argument	"Make [...] turn sud- denly or smoothly	Make the ostrich appear to almost fall into the pond by making it turn backward and then forward abruptly.	Make the hippo take a bow by gently turning forward and backward.	One character does multiple turns or rolls without anima- tionStyle.
Animation- Style	Make anima- tions more smooth!	Abrupt ani- mation style changes smoothly be- tween move- ments	Gentle ani- mation style pauses between movements like movements with- out animation style	Sets animationStyle to default so it has no effect.

AsSeenBy argument	Make [...] turn around something other than itself	Make shark turn around the kid	Make the kid turn around the spine	Code uses turn but not asSeenBy.
AsSeenBy	Look how to use 'as seen by' ef- fectively	As seen by ice floe changes the spinning center from ice skater to ice floe	Turning around is already cen- tered at the ice skater	Uses asSeenBy with the same object (which has no effect).
AsSeenBy argument	Improve the turning animation by not going un- derground	Make kid flip by turning around his belly	Make dog go around kid	Object moves for- ward or backward at least one full ro- tation and does not use asSeenBy.
Delay method	Pause before moving to another direction.	The painter turns to look at the man, pauses, and then goes back to painting.	Man turns right, then left without stopping	One character moves in multiple directions sequen- tially but does not use delay.

Duration argument	Make speech display longer!	Use a larger duration value to make speech bubbles display longer.	Use a short duration to make speech bubbles disappear quickly.	Code has a <i>say</i> code block with long text (>50 characters) and no duration argument
Duration	Make [...] move different speeds.	Use a larger duration value to make the submarine move down slower.	Use a short duration to make the fish move down faster.	Object moves long or very short distance (>2 or <.5) but does not have a duration.
Duration	Make animations happen faster or slower using a duration other than 1.0	User a duration bigger than 1 to make things happen for a longer amount of time.	Use a duration less than 1 to make things happen faster.	Default duration of 1 selected.
GetDistanceToGround function	Move [...] exactly to the ground	Use 'getDistanceToGround' to move something exactly to the ground	Use a numerical value to move something a certain amount	If an object moves down and does not use getDistanceToGround.

GetDis- tance- ToGround function	Try to reach something above the ground	Make girl reach to mango on the tree by moving up for mango distance from ground.	Make girl reach to mango on the tree by moving up an estimated distance.	Move up with no getDistance- ToGround function.
GetDis- tanceTo function	Make a character walk the distance to another object	Make yeti walk directly inside of the cave	Make yeti walk for 5 meters	Turn + walk in code without getDistanceTo function.
GetDis- tanceTo function	Find out how much [...] should move to get to [target]	Use getDis- tanceTo to move the snowman into the cave	Use a number to move an object a certain amount	If an object turns and moves and does not use getDistanceTo.
GetJoint function	Change [...]’s pos- ture before spinning	Make ice skater spin with one leg out	Make ice skater spin while stand- ing straight	Code has charac- ter turning without having moved the joints.

GetJoint function	Make [...] jump realistically	The bunny bends over to prepare to jump and then straightens up on the way down	The bunny does a basic jump up and then down.	Move up with no joint movements.
GetJoint movements	Make only [...]’s head turn	Use turn head to make a character just turn their head.	Use turn with a character to make their whole body turn.	If an object turns to face something and does not use joint movements.
Move method	Have you thought about making [...] move?	Move in a certain direction	Move to the same spot as something else	If objects talk about moving, but there are no move code blocks
Move	Make [...] move	Make carp escape by moving into the cave	Make alien move down 2m from the UFO	Characters say or think direction words, but move is not used.

MoveInRelationTo	"Use 'move in relation to' to make objects move together by using it with move	Whenever the snowboard moves, bunny also moves in same direction and same amount	Bunny's movements do not affect the snowboard	MoveInRelationTo used without a following move.
MoveInRelationTo method	Make [...] move together with [target]	Make objects move together using 'move in relation to'	Moving two objects in a row makes them move one after the other	Two objects move the same distance and direction sequentially and do not use moveInRelationTo.
MoveInRelationTo	Make [...] move with [target].	Make helmet 'move in relation to' the alien to make them move together	Without move in relation to, the helmet doesn't move with the alien.	Object moves to a body part of a character, but moveInRelationTo is not used.
OrientToUpright method	Make [...] get back to upright position	Use 'orient to upright' to make an object 'stand upright'	You can use opposite actions to make something stand up again	If an object turns and/or rolls but does not use orientToUpright.

Orient- ToUpright method	Use ‘orient to upright’ to make a charac- ter stand up after rolling or turning	Use ‘orient to up- right’ to get the boy to stand up- right	Use ‘orient to’ to make objects face the same way	Turn or roll without orientToUpright method.
Resize	Make [...] dis- appear by shrinking	Use resize with a very small value to make some- thing disappear	Use disappear to make something fade away	Code uses setOpac- ity to make some- thing disappear but does not use resize.
Resize	Make [...] larger or smaller by using a value other than 1	Resizing 4 times makes the plant grow larger	Resizing by 0.5 makes the ice shrink	Sets resize to 1 without having set it to anything else (which has no ef- fect).
Resize	Make [...] bigger or smaller	Make stuffed tiger bigger	Make lioness smaller	Characters say or think words about size, but resize is not used.

SetAtmo- sphere method	Change sky color	Change sky to dark color for night	Change sky to bright color for morning	If the characters talk about time of day but do not change the atmo- sphere color.
SetPaint method	Make [...] turn color and then disappear	Use 'set paint' to change the color or something be- fore disappearing	Use 'disappear' to make some- thing disappear.	If an object disap- pears but the code does not use the set- Paint method.
SetPaint method	Make objects change color by setting the color to something other than white!	Setting the paint to black makes the moonface go dark.	Setting trans- parency to 50% makes the alien look faded.	Code uses setPaint with white color (which has no ef- fect).
SetTrans- parency method	You can also make [...] see- through!	Make something see-through by setting trans- parency to 50%.	Make something disappear by setting trans- parency to 0.	Code uses dissap- pear, but not set- Transparency.

StraightenOutJoints method	Reset [...]’s joints to standing straight.	Use ‘straighten out joints’ to get joints back to original place	Turn or roll joint in opposite direction to get back to original place	Character turns and moves joints but does not use straightenOutJoints
StraightenOutJoints method	Reset [...]’s joints to original posture	Use ‘straighten out joints’ to get joints back to original place	Turn or roll joint in opposite direction to get back to original place	Code has joints moved and then straightened manually
StraightenOutJoints	Use ‘straighten out joints’ after joint movements to reset.	Use ‘straighten out joints’ to get yeti’s right shoulder back to original position	Use ‘straighten out joints’ to get madhatter’s joints back to original position.	StraightenOutJoints is used, but no joint movements are used.
Turn	Make [...] spin.	Make girl spin by turning around 2 times	Make the girl turn her feet out to dance using roll	Characters use say or think methods with words about dancing, but do not use the turn method.
TurnToFace	Make [...] turn to face before walking	Make girl turn to face the cola before walking toward it	Make girl walk toward the cola bottle	Walking without turning to face method.

TurnTo-Face method	Make [...] turn to face before moving	Make the dragon face the UFO before moving towards it.	The dragon can move toward the alien without turning because it's already facing it.	Move without a turnToFace method.
TurnTo-Face method	Make characters face each other to talk to each other!	Have a character 'turn to face' before speaking so that it looks like a conversation.	Have characters 'orient to' each other to face the same direction.	Say statements with no turnToFace method.
Walk method	Make [...] walk	Use 'walk' to make 2-legged characters walk, like the yeti	Use 'move' to make objects like the sled move.	If a character with a walk method moves and there is no walk method.
WalkPace argument	Make [...] walk faster!	"Use a larger walk pace to make a character walk quicker.	Use a smaller walk pace to make a character walk slower.	Character walks more than 2 meters at a pace <=1.
WalkTo method	Make [...] walk!	Make the tortoise walk to the cake.	"Make the yeti walk 1.0 meters to walk to the cake.	Movement words in say or think statements without walkTo method

D.4 Surveys

D.4.1 Demographic history Survey

See section A.1.

D.4.2 Learning style survey

Rate on a Likert scale from 1-5 (strongly disagree to strongly agree):

I enjoy being taught new things in a classroom.

I enjoy learning new things from books.

I enjoy learning new things from online tutorials or video tutorials.

When I use a new app or technology (phone, computer, or game system), I like to try figure out how all of the different features work.

I only learn the technology I have to know to do my schoolwork.

I enjoy trying out brand new technology that my friends or family might not know how to use yet.

If an app or computer is not working, I will play around with it myself to try to get it to work.

If an app or computer is not working, I will try to find information online about how to fix it.

I prefer to use technology that I am used to or that others already know how to use.

D.4.3 In-task survey questions

“You opened the tip for [...] (shown below) and used it in your program. Why did you open the tip and add it to your animation?”

“You opened the tip for [...] (shown below) but did not use it in your program. Why did you open the tip, but not add it to your animation?”

“Why did you not open the tip for [...] (shown below)?”

Appendix E

Semi-automatic Suggestion Generation Study Materials

E.1 Surveys

E.1.1 Demographic and computing history survey

See section A.1.

E.1.2 Post-study survey

Please circle 1-7 in each row.

Today my coding experience was:

1 (disgusting) - 7 (enjoyable)

1 (dull) -7 (exciting)

1 (unpleasant) - 7 (pleasant)

1 (boring) - 7 (interesting)

Please rate how useful you found the [tips and hints or tutorials] in task 4 (create your own).

0 (N/A: did not look at them)

1 (I looked at them but they were not useful at all) - 7 (I looked at them and found them very useful)

Please rate how understandable you found the [tips and hints or tutorials] in task 4 (create your own).

0 (N/A: did not look at them)

1 (I looked at them and found them very confusing) - 7 (I looked at them and found them very easy to understand)

Please write a few sentences describing why the tips and hints were or were not useful to you in task 4 (create your own).

E.2 Training tasks

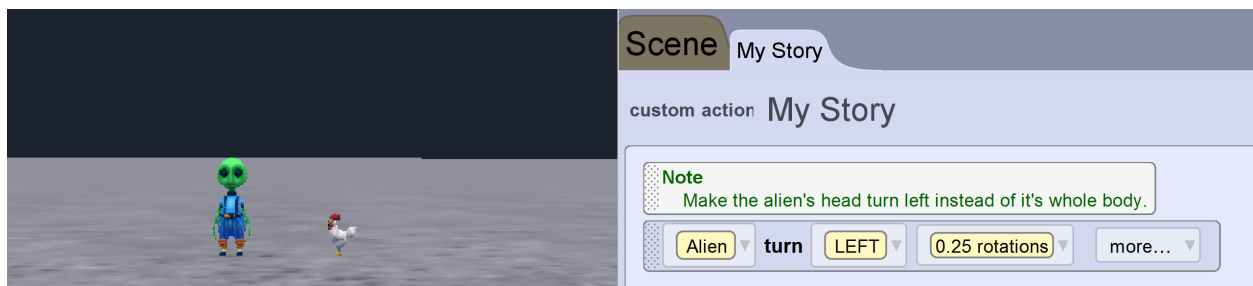


Figure E.1: First Training Task

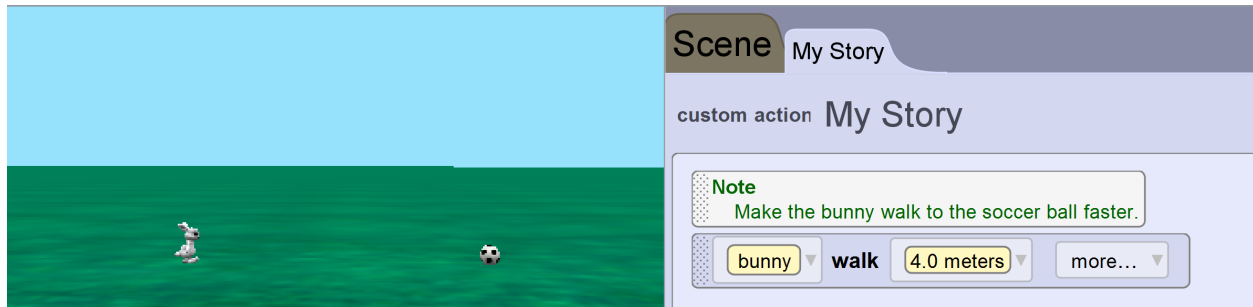


Figure E.2: Second Training Task

E.3 Templates

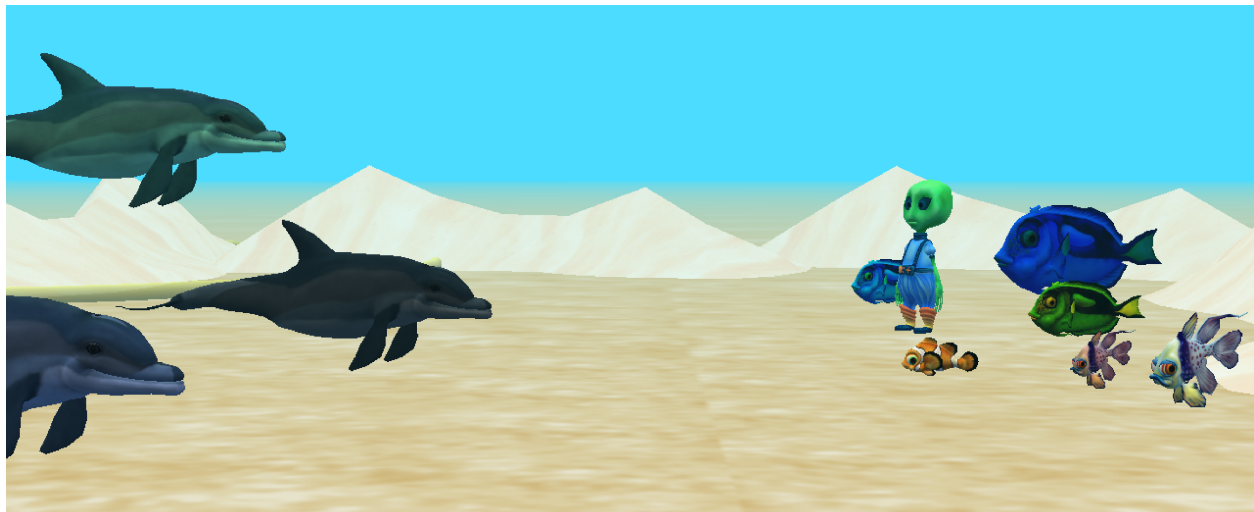


Figure E.3: Seaworld template



E.4 Tutorials

E.4.1 Turn the character's head.

Click on the character drop down to find the head joint.

E.4.2 Increase the walk pace of a character

Make a character walk faster by using 'walk pace' in the 'more' drop-down menu.

E.4.3 Make the flash happen multiple times.

1. Scroll down in the action blocks. 2. Find the disappear block and drag it into your program. 3. Then, drag in the appear block.

The repeat block makes the actions within it happen multiple times. 1. Scroll down to the Action Ordering Boxes and find the repeat block. 2. Drag it into your code and select a number of times you want the flash to repeat. 3. Drag the disappear and appear blocks into the repeat block.

E.4.4 Make actions happen together

Drag in the move action and the disappear action blocks.

The Do together makes blocks within it happen at the same time. 1. Scroll down to the Action Ordering Boxes. 2. Drag in the Do together block.

E.4.5 Make Alice wave three times

First, roll Alice's shoulder. Then turn her elbow left and right to make a wave animation.

The repeat loop makes the actions happen multiple times. 1. Scroll down to Action Ordering Boxes and drag in a repeat loop. 2. Set the number of times it repeats to 3. 3. Drag the roll and turn blocks into the repeat block.

E.4.6 Make a character jump multiple times

Drag in move up and move down blocks.

The repeat loop makes the actions within it happen multiple times. 1. Scroll down to Action Ordering Boxes and add the repeat block. 2. Set the number of times you want the actions to repeat. 3. Drag the move blocks into the repeat block.

E.4.7 Make the two objects move together

Drag in move code blocks for two different objects.

The Do together block makes the blocks within it happen at the same time. 1. Scroll down to Action Ordering Boxes and drag in a Do together block. 2. Drag the move blocks into the Do together block.

E.4.8 Make the set of actions happen multiple times

Scroll in the actions to find blocks like move, turn and say.

The Do together block makes the blocks within it happen at the same time. 1. Scroll down to Action Ordering Boxes and add a Do together block. 2. Add the actions to the do together block.

The repeat block makes the blocks within it happen multiple times. 1. Drag in a repeat block from the Action Ordering Boxes. 2. Drag the Do together block into the repeat block.

E.4.9 Make a character talk and walk at the same time.

Drag in a say block and set the text. Drag in a walk block.

The Do together block makes the blocks within it happen at the same time. 1. Scroll down in the Action Ordering Boxes and drag in a Do together block. 2. Drag the say and walk blocks into the Do together block.

E.4.10 Make an object change size and color at the same time

Scroll down in the actions to find the resize block and the set color block.

The Do together block makes the blocks within it happen at the same time. 1. Scroll down to the Action Ordering Boxes and drag in a Do together block. 2. Drag the resize and set color blocks into the Do together.

E.4.11 Make a character turn and turn back multiple times

Drag in two turn blocks and set the directions to be opposites.

The repeat loop makes the blocks within it happen multiple times. 1. Scroll down to the Action Ordering Boxes and drag in a repeat block. 2. Set the number to the number of times you want the turns to happen. 3. Drag the turn actions into the repeat block.

E.4.12 Make the dolphins flip at the same time

Drag in a turn block for the baby dolphin. Then click on the mom dolphin and drag in another turn action.

The Do together makes actions within it happen at the same time. 1. Scroll down to the Action Ordering Boxes. 2. Drag in a Do together block. 3. Drag both of the turn actions into the Do together block.

E.4.13 Make the jump more realistic

Drag in three move actions. One should have the direction up, the next should have forward, and the last should have down.

The Do together makes actions within it happen at the same time. 1. Scroll down to Action Ordering Boxes and drag in a Do together block. 2. Drag the move up and move forward blocks into the do together to make them happen at the same time.

E.5 Suggestions and examples

Code Sug- gested	Suggestion Title (Number of Ex- ample Snippets in Group)	Primary Ex- ample Title	Secondary Ex- ample Title	Rule Summary
Repeat Loop	Make prop's joints turn back and forth multiple times.(3)	Treasure chest opens and closes 10 times.	Treasure chest opens and closes 2 times.	One character's joints turn back and forth, but no repeat loop is used.
	Make a prop move back and forth multiple times! (3)	The rover shakes 30 times	Camera moves up and down three times.	A character moves in one direction and then back the other way but does not use repeat.
	Make a character repeat turn and move multiple times. (3)	Chicken turns around and jumps 3 times.	Dolphin flips and jumps 3 times.	A character turns and moves multiple times but does not repeat.
	Make a character turn back and forth multiple times. (4)	Baby yeti turns left and right 3 times.	Baby yeti turns right and left twice.	A character turns back and forth but does not repeat.

	<p>Make a character jump multiple times. (6)</p> <p>Make a character's joint turn back and forth many times. (7)</p> <p>Make a character move multiple times. (9)</p>	<p>CheshireCat jumps 3 times.</p> <p>Fox wags its tail left and right.</p> <p>Baby walrus moves 1 meter 3 times.</p>	<p>Seagull moves a little 15 times.</p> <p>Dog wags its tail right and left.</p> <p>Joe moves back twice.</p>	<p>A character moves up and down but does not repeat.</p> <p>A character moves multiple times but does not repeat.</p>
Do together	<p>Make multiple characters say or think at the same time. (105)</p> <p>Make multiple characters move at the same time. (98)</p> <p>Make a character's multiple joints turn at the same time. (97)</p>	<p>Poodle says AHHH while bison thinks the same.</p> <p>Both boy and girl move up at the same time.</p> <p>The girl moves her arms out.</p>	<p>Clown fish says 'a shark' while blue tang say to run.</p> <p>Both boy and girl walk at the same time.</p> <p>Ronalda turns her arms around.</p>	<p>Two characters move one after another with no do together.</p> <p>A character moves multiple joints but not within a do together.</p>

Make multiple characters turn at the same time. (62)	Both lioness and jaguar turn around.	The hippo and camel turn away from each other.	Two characters turn but not in a do together.
Make a character turn and move at the same time. (62)	Bunny turns around as it moves forward.	Alien spins and moves up.	A character turns and then moves, not within a do together.
Make a character move and talk at the same time. (49)	Alien runs away.	Panda walks in while talking to wait.	A character talks and then walks not in a do together.
Make a prop move diagonally. (45)	Plane flies up and forward at the same time.	Camera zooms out of Odin.	An object moves in two directions sequentially.
Make multiple characters move at the same time. (39)	Alien and spaceship move in opposite directions.	Both witch and cauldron move up.	Two characters move sequentially.
Make a character talk while turning. (29)	Baby yeti shouts while falling.	Flamingo shouts while spinning.	A character turns and then talks.

Make a character's head move while talking. (26)	Man drops his head while he speaks.	Pig turns head to left as it talks.	A character's head moves and then it speaks.
Make multiple characters change appearance at the same time. (25)	Ogre transforms into a prince.	Both fish disappear at the same time.	Two characters change appearance sequentially.
Make a character move diagonally. (24)	Panda moves up and forward.	The boy moved back and down to the floor.	A character moves in two directions sequentially.
Make a characters' joints multiple times move while talking. (24)	Baby yeti's mouth moves as he speaks.	The camel's mouth moves as it speaks.	Character's joints move and then it speaks.
Make a character's leg joints move at the same time. (21)	The man's both legs bent forward.	The man stretches the legs.	A character's leg joints move sequentially.
Change a prop's position and orientation at the same time. (19)	Camera moves and faces Morpheus.	Spaceship spin as it moves up.	A prop moves and changes orientation sequentially.
Make a character's arm joints move at the same time. (17)	Rabbit raises its arms at the same time.	Yeti bends his arm towards the face.	A character's arm joints move sequentially.

multiple things move at the same time. (16)	Both curtains open.	Multiple things move to ship.	Two objects move sequen- tially.
Make characters move and say at the same time. (11)	Three stu- dents shout and go up.	Three pigs say as they move back.	
Make a character use their legs more realistically. (12)	The man jumps.	The chicken steps.	
Change multiple things' visibility at the same time. (10)	Cactus dis- appears and turns into a cup of water.	Multiple things disap- pear at the same time.	Multiple objects change visibility, but no do to- gether used.
Make a character's arms and legs move at the same time. (10)	The girl's arms and legs cross.	The boy poses for a jump.	A character's arms and legs move but no do together used.

Table E.1: The 7 suggestions and rules for repeat and the suggestions and rules for do together with 10 or more examples in the cluster.

E.6 Transfer tasks

Note

- All of these actions are already in the correct order.
- You may only add up to 3 additional code blocks.
- First play the animation.

Note

- Make the monkey jump forward twice.

monkey ▾ **jump forward**

monkey ▾ **say** "yummy! yummy!" ▾ more... ▾

Note

- Make the monkey look up and then climb twice.
- After the monkey looks up and climbs twice, the monkey should shake the tree and then the palm tree should drop a coconut, twice.

monkey ▾ **look up**

monkey ▾ **climb**

monkey ▾ **shake** *thing :* palm tree ▾

palm tree ▾ **drop coconut**

monkey ▾ **move** DOWN ▾ 2.0 meters ▾ duration 0.5 seconds ▾ more... ▾

Figure E.5: Transfer task 1

Note

- All of these actions are already in the correct order.
- You may only add up to 3 additional actions or action ordering boxes.
- Before making any changes, first play the animation.

Note

- The big yeti should lose its balance and the tiny ice should shake at the same time.

tiny ice ▾ **shake**

big yeti ▾ **lose balance**

big yeti ▾ **say** "I gotta get off this!" ▾ more... ▾

big yeti ▾ **jump to** *thing :* big ice ▾

Note

- Both pieces of ice should move down at the same time.
- Afterwards, both yetis should fall into the water at the same time.

big ice ▾ **move** DOWN ▾ 1.0 meters ▾ more... ▾

tiny ice ▾ **move** DOWN ▾ 1.0 meters ▾ more... ▾

tiny yeti ▾ **move** DOWN ▾ 1.2 meters ▾ duration 0.5 seconds ▾ more... ▾

big yeti ▾ **move** DOWN ▾ 1.2 meters ▾ duration 0.5 seconds ▾ more... ▾

Figure E.6: Transfer task 2

Note

- All of these actions are already in the correct order.
- You may only add up to 3 additional code blocks.
- First play the animation.

worker ▼

walk

2.0 meters ▼

walkPace

3.0 ▼

more... ▼

Note

Make all of this happen three times:

The worker jump on the button, then push the button,
then both balloons inflate at the same time,
and then the worker jumps off the button.

worker ▼

jump on

thing :

button ▼

worker ▼

push

thing :

button ▼

purple balloon ▼

inflate

blue balloon ▼

inflate

worker ▼

jump off

thing :

button ▼

purple balloon ▼

float away

how high : 2 ▼

blue balloon ▼

float away

how high : 2 ▼

Figure E.7: Transfer task 3

Note

- All of these actions are already in the correct order.
- You may only add up to 3 additional code blocks.
- Before making any changes, first play the animation.

mommy penguin ▾ **say** "Let me examine your fishing abilities..." ▾ duration 2.0 seconds ▾ more... ▾

baby penguin ▾ **flutter**

Note

- The mommy penguin should side step around the hole once, while at the same time the baby penguin dives under, kicks, and surfaces three times.

mommy penguin ▾ **side step around** thing : hole ▾

baby penguin ▾ **dive under**

baby penguin ▾ **kick**

baby penguin ▾ **surface**

mommy penguin ▾ **flutter**

mommy penguin ▾ **say** "No fish!? Better luck next time..." ▾ duration 2.0 seconds ▾ more... ▾

Figure E.8: Transfer task 4