Semi-Automatic Suggestion Generation for Young Novice Programmers in an Open-Ended Context

Michelle Ichinco and Caitlin Kelleher
Washington University in St. Louis
St. Louis, USA
{michelle.ichinco, ckelleher}@wustl.edu

ABSTRACT
Independent novice programmers in open-ended contexts rely on help systems to support their learning. These help systems are often laboriously hand-authored by experts. This paper describes a semi-automatic process for the creation of a suggestion-based help system. We demonstrate and evaluate the potential utility of our approach within a blocks-based programming environment for children. With less human effort per suggestion, our approach generated a set of suggestions comparable to a hand-authored set and a set of original suggestions. We ran a study to explore the number and types of suggestions children received, accessed, and used. In 30 minutes, children on average received 9 suggestions, accessed 2.6 suggestions, and inserted 0.8 new concepts from suggestions.

CCS Concepts
•Human-centered computing → Interactive systems and tools; •Applied computing → Interactive learning environments;

Author Keywords
Novice programming; recommender systems; code examples

INTRODUCTION
Many children begin to learn programming independently. These children often start coding in motivating open-ended contexts where they can create games [34], apps [25], and animations [10]. *Open-ended* means that the novices choose what project to create based on their interests, rather than working on a task with a specified solution. In order to gain skills while pursuing open-ended projects, independent novices must seek out hand-authored resources, like tutorials [34] and documentation. Despite the availability of these resources, novices coding without tasks often plateau [40]. One way to reduce this plateau may be through context-sensitive suggested examples, which encourage novice children to explore new skills [15, 17]. Creating suggestions with minimal human effort would enable generation of suggestions for large systems without significant expert effort.

We describe an approach for semi-automatically generating content within a suggestion-based help system for open-ended programming. In this approach, a system can generate candidate suggestions by grouping code examples. One group of examples could include characters moving in multiple directions at the same time, leading to a suggestion about making a character jump diagonally. To group code, an expert defined two metrics: 1) types of objects, like characters or props, and 2) types of actions the objects take, such as changing position or changing size. Using these metrics for code similarity, the system can then generate any number of suggestions with the following steps: 1) system extracts code snippets from a repository, 2) system groups code examples using heuristics, 3) human moderates, and 4) system generates a script that checks whether novice code should receive the suggestion.

We evaluated our approach for semi-automatically generating suggestions in two ways: 1) we compared the semi-automatically generated content to a hand-authored set created for a separate study, and 2) we ran a study in which children had access to suggestions for 30 minutes in an open-ended 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, accessed 2.6, and used close to one suggestion in just 30 minutes.

RELATED WORK
The overarching ideas in this approach relate most closely to: support for learning at scale, recommendation systems for programmers, and methods for finding similar code.

Automated Support for Learning
Researchers have designed ways to help learners of content and software at a large scale, both in task-based and open-ended contexts. In task-based contexts like intelligent tutoring systems, researchers have worked to develop content, hints [31, 29, 37], and feedback [14, 18] for users in automated and semi-automated ways. However, systems automatically generating hints for open-ended *tasks*, like the Hint Factory [29], can still leverage the similarity of student work and knowledge of the solution to generate tips. Automatically generated tutorials can also help novices learn programming [12], as well as similar types of technical skills like photo-manipulation [9, 6] with minimal expert effort. These systems all require a known task

IDC '18, June 19–22, 2018, Trondheim, Norway

ISBN 978-1-4503-5152-2/18/06…$15.00
DOI: https://doi.org/10.1145/3202185.3202762

and solution in order to generate hints toward that solution. Our approach generates hints for programmers without pre-specified tasks, like reuse and recommendations in unbounded contexts. Some systems provide support for reusing others' programs, like remixing in Scratch [34], or Looking Glass [11]. There are also some systems that recommend commands based on how communities often use them [21, 23]. Our approach is unique in semi-automatically generating motivating suggested examples for novice programmers who have designed their own projects, rather than for novices working on tasks given to them.

Recommendations for Programmers

Recommendation systems support non-expert programmers in overcoming barriers and experienced programmers in use of APIs and software maintenance. Recommendations for non-experts typically help them fix bugs, like making suggestions when users are following the wrong path in a task [20]. A set of systems recommend code examples or web responses based on errors [2, 7, 13, 27, 30]. Recommendations for professional programmers center around library functions, or code maintainability. Several systems recommend code examples to support API learning based on the structure of code [16], sets of methods commonly used together [42], or a programmer's comments [41]. Other systems support maintainability, like suggesting alignment with the style of a code-base [1], suggesting code that likely needs to be modified based on previous modifications [43], and encouraging 'best-practice' use of a programming language [8, 39]. Our suggestions focus on how to increase novices' exposure and use of programming constructs through motivating examples.

Selecting Related Code

Our approach finds groups of related code to form suggestions and then finds novices' code that would likely benefit from that suggestion. Existing processes for selecting relevant code use varying levels and types of extra context, like short queries [5, 24, 32, 38], broader code structures [3, 16, 26], frequency of terms [28, 35], the language and framework [4], or timing [42]. One system determines the behavior of code using information retrieval techniques on the code and description text [33, 36]. Our approach uses only the categories of code and the methods and objects, rather than using all of the exact code.

PROGRAMMING ENVIRONMENT & EXAMPLE GURU

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

Looking Glass Programming Environment

Looking Glass is a blocks-based programming environment for children aged 10-15 to make 3D animations [10]. It provides objects to create scenes and blocks that control the objects (see Figure 1). A scene contains characters (i.e. people and animals), props (i.e. trees, couches, snowboards), and the scene objects (i.e. the ground, the camera). To create animations, users can drag and drop two types of code blocks: actions (i.e. *move*, *say*, *resize*, *disappear*), or programming constructs (i.e. simple parallel execution *Do together*, or a *loop*). Looking Glass's suggestion system called the Example Guru suggests examples based on novices' code [17].

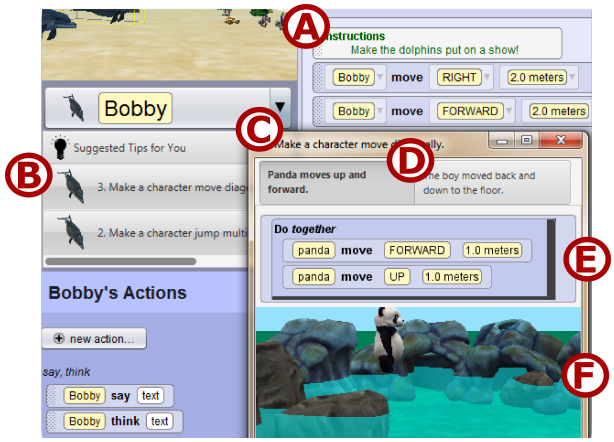

Figure 1. (A) Looking Glass. (B) List of suggestions. (C) An accessed suggestion. (D) The two code examples. (E) The code for this *Do together* suggestion. (F) Preview execution of the code.

The Example Guru

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. When a user executes their program, rules statically analyze the novice's program. For Figure 1, the rule checks for two move actions in a row with the same character (person, animal, or creature). It triggered the suggestion 'Make a character move diagonally'. Each rule has an associated suggestion that triggers based on certain types of code. Suggestions include a title and a pair of examples that demonstrate how to use a code block. When a suggestion is triggered, the title appears in a list near the code block menu (see Figure 1-B). Each suggestion only appears once, but remains in the list throughout the session. When a user accesses a suggestion, a window opens with two examples and the option to execute them (see Figure 1-C).

SUGGESTION GENERATION APPROACH

This approach aimed to generate suggestions and rules with minimal human effort per suggestion. It 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 steps: 1) example extraction, 2) example grouping for candidate suggestions, 3) human moderation (example selection + labeling), and 4) rule generation.

Input Repository

We used a repository of 1313 blocks programs containing: 1) 585 programs created by expert researchers, and 2) 728 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 contains programs created in past user studies and programs shared to the Looking Glass online community by non-lab members [22].

Initial Setup

This approach requires initial setup by an expert, who selects: 1) which code concepts to suggest, and 2) how the system should group code snippets and generate rules. The expert

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

Table 1. Objects and actions used for binning snippets.

selected 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 extremely novice programmers. The expert chose to generate suggestions for the parallel execution block called *Do together* and the loop block called *Repeat*.

In order to group the extracted code snippets and generate rules, the system needs heuristics to determine whether code snippets have similar animations. Groups of snippets with similar animations, like snippets that all make the camera zoom, can become suggestions. For Looking Glass, the expert defined types of objects and methods that would have similar animations, as shown in Table 1. Objects have three types: sentient characters, sentient character's joints, and props. Actions have eight groups: communicate, sound, position, orientation, appearance, size, timing, and vehicle.

Example Extraction

In order to generate groups of examples, the system needs to extract snippets from a repository of code that contain the code that will be suggested. We generated suggestions for the *Do together* and *Repeat* code 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.

Example Grouping

The Example Grouping generates candidate suggestions. It does this by sorting the code snippets for a certain concept into groups of code with similar output. The grouping algorithm uses the object and action types defined by an expert in the **Initial Setup**, shown in Table 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 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 1 has one sentient object performing two position actions. This code group could also contain examples with one sentient object performing more than two position actions.

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 2. Human moderation criteria

The example grouping results in groups of examples where the object types and action types align to create similar animations. In our system, this grouping process resulted in 158 groups 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.

Human Moderation

The main objectives of the human moderation step are to select two examples, label them, and give the suggestion a title. This approach benefits from having a human select the examples because humans can select examples with the best and most visible animations and humans can filter out inappropriate animations. The criteria for moderating groups are listed in Table 2. It is also important that a human writes the suggestion title because the title should motivate the novice programmer to look at the suggestion. Descriptions of examples need to describe the output animations, rather than the code itself. A suggestion title would ideally be something like "Make your character jump multiple times," rather than a description of the *move up* and *move down* code blocks. 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 had a non-author researcher perform the human moderation phase. Our moderator sent 80 of the 158 groups to rule generation. The excluded groups did not have at least two examples that fit all of the required criteria in Table 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.

Generate Rules

Our approach generates rules using the object and action types in Table 1, as initially defined by an expert. The approach generates rules that 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 and does not use a *Do together*.

In order to generate a rule, the system needs to extract the following information from a set of examples: 1) the concept

Hand-authored suggestion: "Make...."	Semi-Automatically Generated: "Make...."	# participants who accessed/ received
objects move in two directions at the same time	a character move diagonally.	8/18
	a prop move diagonally.	1/3
	camera move diagonally.	1/2
an object bounce multiple times	a prop move back and forth mult. times!	1/3
	a character jump multiple times.	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.	0/3
	a character move and talk at the same time.	1/6
objects move together!	mult. things move at the same time.	0/1
	mult. characters move at the same time.	7/18
	mult. characters move away from something at the same time.	0/1
	3 more similar suggestions	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.	0/0
	a character's joint turn back and forth many times.	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, NA, NA, NA, NA	15 Simultaneous joint movements suggestions, 9 Jumping and joint movements suggestions, 7 joints turn while a character moves suggestions, 5 movements while talking suggestion,4 other misc. suggestions	0/0

Table 3. Left: Hand-authored suggestions. Right: Semi-automatic suggestions and the number of suggestions received and accessed in our study.

being suggested, 2) the number and types of objects to look for, and 3) the numbers and types of actions for each of the objects. The system then generates 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 [19].

We next evaluate the generated suggestions and rules in two ways: 1) how do they compare to a hand-authored set of suggestions, and 2) how do young novices interact with them?

EVALUATING SEMI-AUTOMATIC VS. HAND-AUTHORED

We wanted to know how the semi-automatically generated suggestions compared to a set authored by an expert. 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.

Comparison Methods

To compare our generated suggestions to a hand-authored system, we used a set of manually authored suggestions from a previous study [15]. The hand-authored set of suggestions has 5 suggestions for *loop* and 6 suggestions for *parallel execution*, as shown in Table 3. We consider two suggestions to be equivalent if they either involve the same types of objects and actions, or the types of objects or actions in one suggestion are a more specific version of those in the other suggestion.

Comparison Results

Our semi-automated approach generated more suggestions overall, including 1) equivalent suggestions for all but two of the hand-authored suggestions, and 2) an original set of suggestions. Table 3 shows the hand-authored suggestions in the left column, matched with the generated suggestions in the right column. The 61 new suggestions are listed at the bottom.

The generated approach resulted in 19 suggestions equivalent to the 11 hand-authored suggestions. Our approach often generated multiple suggestions equivalent to one hand-authored suggestion, where the generated suggestions are specific to a certain object type (see Table 3). For the hand-authored 'make objects move in two directions at the same time', our approach generated three suggestions for a character, a prop, and the camera. This would require significant extra effort from an expert to hand author, but very little effort in our approach. These specific suggestions are highly valuable for novices, who often need surface-similarity like similar objects to make connections between abstract examples.

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 3. The expert likely did not create suggestions about complex joint animations due to the experience of the target audience. 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. Requiring less human effort per suggestion makes it more feasible to create a larger number of suggestions, some of which may be highly relevant to small subsets of novice programmers.

USER STUDY

We ran a study to explore how novices received, accessed, and used semi-automatically generated suggestions. In this paper, we only discuss and analyze data from participants in a suggestion condition in a multi-condition study.

Study Protocol

In this paper we analyze data from the first three phases of the study: 1) baseline open-ended, 2) system familiarization tasks, and 3) supported open-ended.

Baseline Open-Ended. To measure what novices would explore without support, all participants worked on an open-ended animation for the first 15 minutes of the study. Participants could select one of 9 pre-made scenes to code their own animation. Participants did not have access to suggestions. 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 and what information the suggestions contained, participants completed two familiarization tasks. Each task lasted 5 minutes and asked the user to modify an API method call using a suggestion. A researcher confirmed that each participant accessed at least one suggestion 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 Open-Ended. To investigate how participants would receive, access, and use suggestions during open-ended 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 an open-ended performance animation and then could select other scenes to make animations.

Participants

We recruited 24 participants from the Academy of Science of St. Louis mailing list. 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 20 participants: 11 males and 9 females who ranged in age from 8 to 15 ($M = 11.2, SD = 1.3$).

Study Results

We report the number and types of suggestions participants received, accessed, and used.

How many suggestions did participants receive?

The number of suggestions participants received tells us whether the generated suggestions applied to novice programs. Novices likely will not find all suggestions relevant. In a 30 minute session, we would hope that novice programmers would at least receive several suggestions. On average, participants received 9 suggestions (SD =4.5). Participants received between 1 and 17 suggestions and 80% of participants received at least 5 suggestions.

Participants received 29 of the 80 available suggestions, as shown in Table 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 many suggestions did participants 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. On average, participants accessed 2.6 suggestions ($SD = 2.6$). Fifteen participants (75%) accessed at least one suggestion and thirteen participants (65%) accessed two or more suggestions. Most participants accessed suggestions for both concepts: *parallel execution* and *loop*. Of participants who accessed suggestions, 14/15 accessed *parallel execution* suggestions and 12/15 accessed *loop* suggestions.

Of the 29 suggestions received by participants, at least one participant accessed 20 of them. On average, 23% of participants who received a suggestion accessed it ($SD = 19\%$). Table 3 shows that the percentage of participants accessing the different suggestions is relatively similar across suggestion types. The suggestions received by higher numbers of participants commonly have accordingly higher access of suggestions. Most suggestions received by even small numbers of participants were accessed by at least one participant.

How many new concepts did participants use?

Participants often accessed multiple suggestions for the same concept, so we report whether participants used a new concept after accessing any suggestion for that concept. *Use of a suggestion* refers to inserting the programming concept from that suggestion for the first time after accessing it. Ten of fifteen participants inserted a concept for the first time after accessing a suggestion for it. On average, participants used 0.8 new concepts from suggestions ($SD = .9$).

Participants on average inserted 60% of new concepts they accessed suggestions for ($SD = 50\%$). Of the 14 participants that accessed suggestions for *parallel execution*, 10 then used it for the first time (71%). Participants on average added 3.4 *parallel execution* blocks after accessing a suggestion (SD=2). Of the 12 suggestion participants who accessed *loop* suggestions, 6 added *loops* for the first time afterward (50%). Suggestion users on average added 1.8 new *loop* blocks (SD=1).

The results suggest that participants received sets of relevant suggestions and that participants were able to apply the suggestions to their animations. In a thirty minute session where many participants had little or no programming experience, beginning to use one new abstract programming

concept based on suggestions demonstrates the potential of semi-automatically generated suggestions.

THREATS TO VALIDITY

The primary threat to validity is our population, a group of children who primarily have access to a science-focused mailing list. They are likely more interested in programming than the general population. We also had a small sample size.

DISCUSSION

We discuss how our approach generalizes, the effect of the code repository, and the potential of this approach to support personalization.

How our approach generalizes beyond animation

This approach can apply to a variety of programming contexts in which programmers are working toward their own goals and code can be grouped by output. Both conditions apply in many contexts, such as web, phone application, and game development. These open-ended contexts have object and method types that can be grouped. This paper describes suggestion generation for two early programming concepts. The same approach could generate suggestions for any other programming concept block with no extra effort as long as the repository has enough data to support the approach. Future work should explore how to generate these types of suggestions for programming skills that span larger code snippets.

Effect of the code repository on suggestion generation

Our approach produces suggestions from a repository of programs, limiting the generated content by the code in the repository and risking quality issues. From fewer than 2,000 examples, our approach generated an impressive number and quality of suggestions and rules. Many programs in the repository contained *parallel execution* code blocks, but many fewer included *loop* or nested blocks. This approach relies on at least some subset of users to be effectively using programming constructs. There are often at least a subset of programmers who do explore ways to learn new programming concepts on their own. There will likely be at least a small number of usages of most programming concepts and a small number of suggestions could encourage use of skills that would feed into the repository.

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.

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.

CONCLUSION

Existing systems require significant human effort to generate support for programmers. There is a growing number of 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. Our approach could ease the human costs associated with creating and updating help resources. This has the potential to help children overcome plateaus when coding in contexts without tasks, ideally leading to learning and continued interest in programming.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1054587 and 1440996.

SELECTION AND PARTICIPATION OF CHILDREN

We recruited children through the Academy of Science of St. Louis mailing list. All who responded and reported minimal programming experience were invited to participate. Parents and children read and signed consent and assent forms. The forms described what the children would do in the study, that there was minimal risk, and that the children could stop participating at any time and still receive their compensation. Participants received $5 gift cards to Amazon.com.

REFERENCES

1. Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.
2. Vahid Amintabar, Abbas Heydarnoori, and Mohammad Ghafari. 2015. Exceptiontracer: a solution recommender for exceptions in an integrated development environment. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. IEEE, 299–302.
3. Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. 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, 681–682.
4. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: integrating web search into the development environment.

In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

5. Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. *Fundamental Approaches to Software Engineering* (2009), 385–400.
6. Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. 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, 93–102.
7. Joel Cordeiro, Bruno Antunes, and Paulo Gomes. 2012. Context-based recommendation to support problem solving in software development. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*. IEEE, 85–89.
8. Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, 2491–2500.
9. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics (TOG)* 28, 3 (2009), 66.
10. Paul Gross and Caitlin Kelleher. 2010. The Looking Glass IDE for learning computer programming through storytelling and history exploration: conference workshop. *Journal of Computing Sciences in Colleges* 26, 1 (2010), 75–76.
11. Paul A. Gross, Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. 2010. A code reuse interface for non-programmer middle school students. In *Proceedings of the 15th international conference on Intelligent user interfaces*. ACM, 219–228.
12. Kyle J. Harms, Dennis Cosgrove, Shannon Gray, and Caitlin Kelleher. 2013. 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, 11–19.
13. B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proc. 28th int. conf. on Human factors in computing systems*. 1019–1028.
14. Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 89–98.
15. Wint Hnin, Michelle Ichinco, and Caitlin Kelleher. 2017. 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, 181–189.
16. Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*. ACM, 117–125.
17. Michelle Ichinco, Wint Yee Hnin, and Caitlin L. Kelleher. 2017. 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)*. ACM, New York, NY, USA, 1105–1117.
18. Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppala. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 86–93.
19. Javapoet 2017. A Java API for generating .java source files. (Aug. 2017). `https://github.com/square/javapoet`
20. Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. 2015. A principled evaluation for a principled Idea Garden. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 235–243.
21. Wei Li, Justin Matejka, Tovi Grossman, Joseph A. Konstan, and George Fitzmaurice. 2011. Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction (TOCHI)* 18, 2 (2011), 6.
22. Looking Glass Community 2016. `https://lookingglass.wustl.edu/`. (2016). Accessed: 2013-02-24.
23. Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. 2009. CommunityCommands: command recommendations for software applications. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09)*. ACM, New York, NY, USA, 193–202.
24. Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.
25. MIT App Inventor | Explore MIT App Inventor 2016. (2016). `http://appinventor.mit.edu/explore/`
26. Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 69–79.

27. Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1295–1298.
28. Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 102–111.
29. Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-driven Hints for Open-ended Programming.. In *EDM*. 191–198.
30. Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 194–203.
31. Kelly Rivers and Kenneth R. Koedinger. 2013. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, Vol. 50.
32. Naiyana Sahavechaphan and Kajal Claypool. 2006. XSnippet: Mining For Sample Code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 413–430.
33. Christopher Scaffidi, Christopher Chambers, and Sheela Surisetty. 2015. 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, 1032–1037.
34. Scratch - Imagine, Program, Share 2016. (2016). `https://scratch.mit.edu/help/videos/` Accessed: 2016-04-01.
35. Jeffrey Stylos and Brad A. Myers. 2006. 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, 195–202.
36. Sheela Surisetty, Catherine Law, and Chris Scaffidi. 2015. Behavior-based clustering of visual code. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 261–269.
37. Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. 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, 2951–2958.
38. Suresh Thummalapenta and Tao Xie. 2007. 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, 204–213.
39. S. J. Whittall, W. A. C. Prashandi, G. L. S. Himasha, D. I. De Silva, and T. K. Suriyawansa. 2017. CodeMage: Educational programming environment for beginners. In *Knowledge and Smart Technology (KST), 2017 9th International Conference on*. IEEE, 311–316.
40. Benjamin Xie and Hal Abelson. 2016. Skill progression in MIT app inventor. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE, 213–217.
41. Yunwen Ye and Gerhard Fischer. 2005. Reuse-conducive development environments. *Automated Software Engineering* 12, 2 (2005), 199–235.
42. Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009 Object-Oriented Programming*. Springer, 318–343.
43. Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.