How Block Categories Affect Learner Satisfaction with a Block-Based Programming Interface

Fernando J. Rodríguez, Kimberly Michelle Price, Joseph Isaac Jr., Kristy Elizabeth Boyer,
and Christina Gardner-McCune
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida 32611
Email: {fjrodriguez, kimberlymprice, jisaacjr, keboyer, gmccune}@ufl.edu

***Abstract*—In recent years, block-based programming languages have been employed as learning tools to help students starting out with programming. How we design the layout of the available blocks likely impacts the success of the student. In this study, we compare student performance in three conditions consisting of different layouts of block categories in a block-based language: a grouping based on computer science (CS) concepts, a grouping based on block functionality, and one with no groupings. We measured task completion time, quality of the final code product, and perceived system usability. We found that although time and quality did not differ across conditions, the students in the *functionality* condition reported higher usability scores than the students in the *CS concepts* condition. These results can inform how we design block-based interfaces to improve learner satisfaction without affecting their performance.**

I. Introduction

In an increasingly technological world, having even basic knowledge of computer science (CS) and programming can be very useful. New systems that enable novice programmers to create their own artifacts have become more and more popular. One example is block-based programming languages, interfaces in which the programmer connects blocks together to build programs. Although research has shown the benefits of these languages for learning, there are still open questions about how to best design these interfaces. Presenting a user with every possible construct in a language can increase a learner's cognitive load [1], so organizing these components into coherent categories may improve usability.

One common feature across many block-based languages is the organization of blocks into categories. The reasoning for which categories are presented and how blocks are organized within them is a task for the designers of these languages. For example, Google's Blockly (Figure 1) features categories based on *CS concepts* related to the blocks, such as logic, loops, math, and text strings. [2]. MIT's Scratch, on the other hand, organizes its blocks based on their *functionality*, such as control, operators, sensing, and looks. [3]. In some cases, the locations of these blocks within categories is not intuitive to users, whether novices or experts. To address this issue, we studied alternative organizations of block categories to evaluate the impact block organization has on learners. In particular, we measure the *time* to complete the task, the *quality* of the final code developed for the task, and the *perceived usability* of the programming interface. Our hypothesis is that students using blocks grouped by *functionality* will complete programs in less time, create higher quality code products, and rate the perceived usability of the interface higher than students using the *CS concepts* grouping.

978-1-5386-0443-4/17/$31.00

To test our hypothesis, we implemented three versions of the block-based language Blockly, each with a different arrangement of categories: the default Blockly categories based on *CS concepts*, a modified version with *functionality* categories inspired by Scratch, and a version with *no categories*. We found that task completion time and code quality were not significantly different across the three variations of block organization. However, the usability scores were significantly higher for the *functionality* grouping than for the *CS concepts* grouping. There were no significant differences in usability scores between either of these conditions and the *no category* condition.

II. Related Work

The research and use of block-based programming languages has increased in recent years. One reason for this increase is that block-based languages can be highly motivating for beginners. For example, in a study with MIT's AppInventor [4] in a course for non-computer science majors, students reported more positive attitudes towards computer science and improved motivation to do well in computer science at the end of the course when compared to their attitudes and motivation at the beginning of the course [5]. When compared to text-based languages, users programming in block-based languages were found to spend less time off task and complete more of the subgoals of a given task without affecting their attitudes toward computer science or the task's perceived difficulty [6].

Another reason for the increased use of block-based programming in introductory courses is that block-based languages are designed in part to reduce or eliminate syntax errors and promote a focus on concepts, algorithms, and sequencing. Studies have shown that the visual design of block-based languages helps users better understand concepts such as nested statements and return values of functions [7]. Other topics, such as variable initialization, have proven to be difficult to teach with block-based languages, as they may be represented differently with blocks than how they appear in text-based languages [8]. In a recent study that allowed students to switch

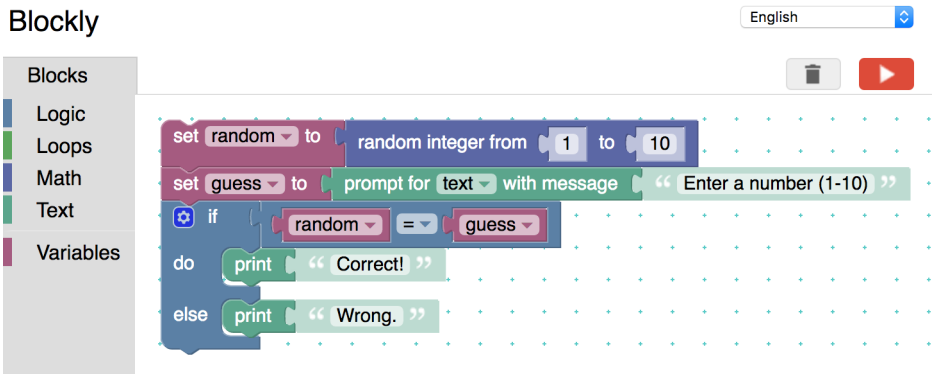

Fig. 1. A screenshot of the Blockly programming interface and a code segment.

between block-based and text-based languages, users would gradually switch to text-based languages as time progressed but would go back to blocks to add new functionality to their code [9]. By providing students with an intuitive organization of related blocks, we may be able to optimize this behavior and reduce the time and cognitive load required for these students to build programs.

An open question in block-based programming research is how to best present the available tools to the user. The developers of AppInventor used the "cognitive dimensions evaluation" framework to review their design, resulting in a more coherent naming scheme than the prior design that reduced errors and complexity [10]. Previous studies in block-based languages with undergraduates demonstrate that students may spend a significant portion of their task time searching for the blocks they need, indicated by frequent block category switching [11]. We have observed in our own studies that students appeared to struggle often in the search for blocks, with category switching making up an average of 16% of their total programming actions. How to best present and organize blocks within block-based programming interface categories to maximize their utility is still an open question.

III. Block Category Design

We have taken an empirical approach to better understand the effects of different block organization approaches in block-based programming interfaces. For this study, we implemented three versions of the Blockly programming language (Figure 2). Each version displayed the same set of blocks but differed in how these blocks were organized into categories. The first version featured block groupings based on Blockly's default categories: "Logic," "Loops," "Math," and "Text." Blocks in this version were organized based on the *CS concepts* represented by the category labels. For instance, "if" statements and relational operators are found in the "Logic" category. The blocks retained their original colors from the default Blockly interface, matching those of their corresponding categories.

The second version of Blockly used block groupings inspired by languages such as Scratch: "Control," "Operators," and "Input/Output." These groupings correspond to the *functionality* of the blocks: "Control" features blocks that define

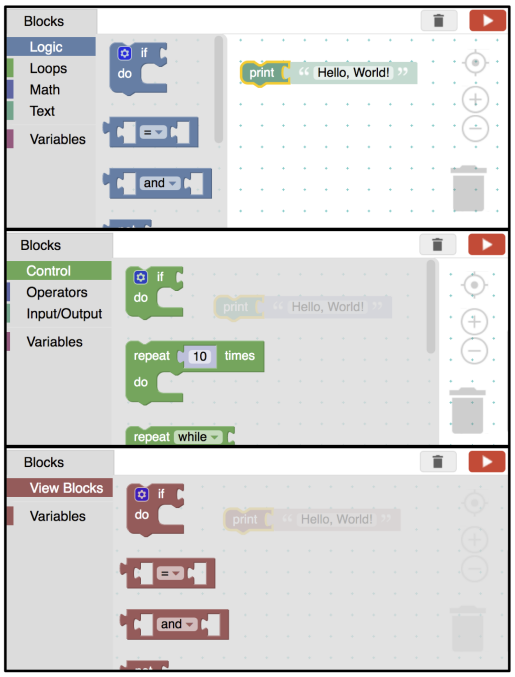

Fig. 2. Test conditions. Top: CS concepts categories, Middle: Functionality categories, Bottom: No categories.

control structures, like loops and "if" statements; "Operators" includes all blocks that perform calculations or comparisons, as well as constant values, such as numbers, Boolean values, and text; "Input/Output" only features the "prompt" and "print" blocks, which are used to request input and present output, respectively. This last category differs slightly from Scratch in that both input and output blocks are located in the same category as opposed to in their own category. The equivalent Scratch categories, "Looks" and "Sensing," feature other blocks related to visual components of the program and additional input modalities, such as the mouse. Since these features are not directly supported in Blockly, and there were only two blocks related to input and output ("prompt" and "print"), we chose to collapse them into a single category. Additionally, these concepts are usually associated with each

other in text-based programming packages, such as Java. Block colors were altered such that each block within a category was colored the same, which resulted in recoloring certain blocks from their original colors.

As a control group, the third version of Blockly featured *no category* groupings and presented all blocks within a single category. Blocks were listed in the original order they were presented in the default Blockly interface but were all displayed in the same color.

Several categories of blocks available in the Blockly language were excluded from all versions of our block organizations. The "List" and "Functions" categories were removed because study participants had not been exposed to the related topics in their coursework. The "Color" category, which Blockly uses to represent color values, was omitted to reduce the visual output components of the given task. The "Variables" category remained constant across all versions. Most block-based languages treat variables as their own category, and variables are a challenging concept in block-based languages [8].

IV. Data Collection

We recruited 37 participants across two introductory Java programming courses: 29 students from a course for CS majors (19 male, 10 female; 14 white, 1 Black, 5 Latino, 7 Asian, 2 did not report race; ages 18 - 30) and 8 from a course for non-CS majors (6 female, 2 male; 6 white, 1 Latino, 1 Asian, ages 18 - 22). The study took place halfway through the Spring 2017 semester, resulting in participants with at least half a semester of prior programming experience in Java. Students were offered course credit for their participation and were given the option to complete an alternate assignment to receive the same course credit.

Participants were asked to bring their own laptops and complete the study by accessing the programming environment on their browsers. All participants received the same set of online instructions to create a number guessing game, which was presented to them as four separate subtasks: 1) select a random number between 1 and 100 and ask the user to guess a number, then display whether the user's guess was correct; 2) repeatedly ask the user for a guess until the user guesses correctly; 3) display to the user after each incorrect guess whether it is too high or too low; and 4) keep track of the number of attempts and display it once the user guesses correctly. Students were given 45 minutes to complete as many subtasks as possible.

We randomly assigned each participant to one of three conditions: 1) *CS concepts*, in which the blocks were organized in their default Blockly categories; 2) *functionality*, in which blocks were organized into categories inspired by Scratch; and 3) *no category*, in which blocks were all included in a single category. Out of the 37 participants, 35 of them completed the first subtask and had complete data logs. These 35 participants are considered in the present analysis and were distributed across the three conditions as follows: $n_{CS} = 12$ (9 majors, 3 non-majors; 6 male, 6 female; 7 white, 1 Latino, 3 Asian, 1 did not report race; ages 18-25), $n_{function} = 11$ (9 majors, 2 non-majors; 10 male, 1 female; 6 white, 3 Latino, 2 Asian; ages 18-30), $n_{none} = 12$ (10 majors, 2 non-majors; 5 male, 7 female; 6 white, 1 Black, 2 Latino, 2 Asian, 1 did not report race; ages 18-25).

TABLE I
Grading rubric for code quality of first subtask

Item	Points
The program includes a "prompt" block	1
The program includes a "random" block	1
The program includes ONLY ONE "random" block	1
The program includes an "if" block	1
The program uses an "if/else" structure	1
The program compares the random number to the user-input number	1
The program includes a "print(*Correct*)" block	1
The program includes a "print(*Incorrect*)" block	1
The program displays the output correctly	2

V. Data Analysis

For each participant, we collected the time to complete each subtask, the final versions of their code for each subtask, and the perceived usability of the interface upon completion of the main task or after 45 minutes had elapsed. For the analysis described in the next section, we only considered the task completion time and code quality for the first subtask, which was to compare a random number to a number the user entered as input and display whether the numbers matched. The task completion time was measured in seconds and represents how long students spent on the webpage with the first subtask's instructions. Each code submission was graded using a 10-point rubric that took into account the presence of required blocks, the connections between the blocks, and the code's output (Table I). Perceived usability of the interface was measured using the students' responses to the System Usability Scale (SUS) questions [12].

VI. Results

Table II shows descriptive statistics for all three conditions. We performed a Shapiro-Wilk test for normality on each of the dependent variables for each condition. Both the task completion time and code quality did not fit to a normal distribution, but the SUS scores did follow a normal distribution. Therefore, we performed Kruskal-Wallis tests on the task completion time and code quality, and used an ANOVA test for the SUS scores. The Kruskal-Wallis tests on task completion time and code quality did not reveal any significant differences ($p_{time} = 0.7340; p_{code} = 0.4667$), but the ANOVA test on usability scores revealed statistically significant differences in the means of the SUS scores across the three conditions ($p = 0.0062$). We used a Tukey test to determine which two conditions had significantly different mean SUS scores and found that the students in the *functionality* condition reported higher SUS scores than the students in the *CS concepts*

TABLE II
MEAN VALUES FOR EACH DEPENDENT VARIABLE IN THE GIVEN CONDITIONS (STANDARD DEVIATIONS IN PARENTHESES). * INDICATES SIGNIFICANT DIFFERENCES BETWEEN CONDITIONS.

Condition	Time (sec.)	Code Quality (grade)	Usability (SUS score)
CS concepts	731.9 (396.1)	8.3 (2.3)	**45.0 (12.8)***
Functionality	798.3 (513.8)	8.3 (2.0)	**71.1 (17.0)***
No category	932.7 (596.1)	9.1 (1.2)	57.7 (22.9)

condition ($p = 0.0043$). There were no differences between either of these conditions and the *no category* condition ($p_{CS} = 0.2135; p_{function} = 0.1934$).

VII. DISCUSSION

Programming studies with novices tend to use performance metrics, such as the number of tasks completed in a given time and the quality of the final code product [6], as measures of student success. The performance metrics of task completion time and code quality in our study were not significantly different across the study conditions. The students had prior programming experience from their programming course, participating in the study with at least half of a semester's worth of experience in the Java programming language. It is not surprising in this case to see that the results regarding performance metrics were not significantly different. However, students in the *functionality* condition reported higher usability of the interface than students in the *CS concepts* condition, meaning that the *functionality* condition was perceived as more usable, regardless of the similarity in task completion time and code quality across the conditions.

The students in this study were still programming novices. As such, their knowledge of domain vocabulary and terminology was limited. In the *CS concept* interface condition presented in this study, students needed to distinguish between four categories with names like "Logic" and "Text." Based on their limited domain-specific knowledge, it may not have been immediately discernible what kind of blocks can be found within these categories. In particular, we observed in pilot studies that students struggled to find the "print" block, contained within the "Text" category along with other blocks related to text functions. For the *functionality* interface presented to students in this study, the number of categories was reduced from four to three. Reducing the number of categories reduced the amount of domain vocabulary knowledge needed to navigate the interface, which may have contributed to the *functionality* interface achieving a higher usability score.

The task in our study involved receiving user input and displaying output, which can be accomplished with the "prompt" and "print" blocks. In the *functionality* version, these blocks were in their own category, emulating the Looks and Sensing categories in Scratch, which include blocks that alter the visual components of a program and receive input from the user. It is possible that grouping these blocks together in their own category made them more easily accessible, resulting in a more usable interface from the students' perspective.

Limitations

One limitation of this work's internal validity is that the given task required a specific set of blocks to complete, which may have biased the results to reflect how well participants were able to find and use this set of blocks. In particular, the "prompt" and "print" blocks were frequently required. These blocks were separated into their own category in the *functionality* condition, which may have made it easier for students to locate them. Generalizability could be improved in future evaluations by using a diverse set of tasks that requires participants to access blocks across more categories.

With respect to external validity, it is unclear whether these results can generalize to programming audiences with varying levels of expertise due to the limited sample size. Advanced users may complete programming tasks more quickly or may report lower usability of the system, given that it is a tool used mostly for teaching novices and does not provide many advanced features.

VIII. CONCLUSION

The features of block-based programming languages make them very practical as teaching tools for novice programmers. However, careful consideration must be taken when designing the layout of these interfaces, particularly the organization of blocks into categories. We conducted a user study to evaluate the performance of students using Blockly with *CS concepts* categories, *functionality* categories, and no categories. Although we found no significant difference across conditions for both task completion time and code quality, the students in the *functionality* condition reported higher usability of their interface than the students in the *CS concepts* condition. These findings imply that alterations to a block-based interface can make it more usable without impacting a learner's performance.

As mentioned in the limitations section, evaluating the interfaces on richer programming tasks could inform how these results hold in general. Additionally, new alternate groupings and even alternate category labels may reveal more about how to best design these interfaces for novices. Since the reduction of interface elements may have contributed to the perceived usability of the system, it may be a good practice to provide minimal interface elements to novices and gradually increase the elements available as they gain experience. Such design principles may yield new and interesting learning experiences that maximize the benefit for programming novices.

ACKNOWLEDGMENTS

We would like to thank the members of the LearnDialogue Group and the Engaging Learning Lab for their helpful input. This material is based upon work supported by the National Science Foundation under grants CNS-1640141 and DUE-1625908. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] A. Renkl and R. K. Atkinson, "Structuring the Transition From Example Study to Problem Solving in Cognitive Skill Acquisition: A Cognitive Load Perspective," *Educational Psychologist*, vol. 38, no. 1, pp. 15–22, 2010.

[2] N. Fraser, "Google Blockly: A Visual Programming Editor," http://code.google.com/p/blockly/, 2013.

[3] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[4] S. C. Pokress and J. J. Dominguez Veiga, "MIT App Inventor: Enabling Personal Mobile Computing," in *Proceedings of the Programming for Mobile and Touch Workshop (ProMoTo '13)*, 2013.

[5] K. Ahmad and P. Gestwicki, "Studio-Based Learning and App Inventor for Android in an Introductory CS Course for Non-Majors," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, 2013, pp. 287–292.

[6] T. W. Price and T. Barnes, "Comparing Textual and Block Interfaces in a Novice Programming Environment," in *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER '15)*, 2015, pp. 91–99.

[7] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," in *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER '15)*, 2015, pp. 101–110.

[8] D. Franklin, C. Hill, H. A. Dwyer, A. K. Hansen, A. Iveland, and D. B. Harlow, "Initialization in Scratch: Seeking Knowledge Transfer," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*, 2016, pp. 217–222.

[9] D. Weintrop and N. Holbert, "From Blocks to Text and Back: Programming Patterns in a Dual-modality Environment," in *Proceedings of the 48th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*, 2017, pp. 633–638.

[10] F. Turbak, D. Wolber, and P. Medlock-Walton, "The Design of Naming Features in App Inventor 2," in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*, 2014, pp. 129–132.

[11] F. J. Rodríguez and K. E. Boyer, "Discovering Individual and Collaborative Problem- Solving Modes with Hidden Markov Models," in *Proceedings of the 17th International Conference on Artificial Intelligence in Education (AIED '15)*, 2015, pp. 408–418.

[12] J. Brooke, "SUS: A 'Quick and Dirty' Usability Scale," in *Usability Evaluation in Industry*, 1996, pp. 189–194.