Sketching as a Strategy to Solve Problems in Introductory Programming

Sophie Blouin
Faculty of Computer Science
Dalhousie University
Halifax, Canada
sophie.blouin@dal.ca

Brent Crane
Faculty of Computer Science
Dalhousie University
Halifax, Canada
brentcrane@dal.ca

Eric Poitras
Faculty of Computer Science
Dalhousie University
Halifax, Canada
eric.poitras@dal.ca

I. Abstract

While prior work has investigated the role of sketching in reading code, the role of flow diagrams has received little attention. In this paper we contribute a technique for sketching flow diagram that serves as a strategy for novice programmers to improve performance in solving procedural programming problems. We then analyze sketches in terms of the degree of ambiguity in pseudocode descriptions of operations, and any description of changes in data resulting from operations. Sketches were collected in the context of an introductory programming course throughout a semester where students had the opportunity to receive formative feedback on their designs prior to implementing code for course assignments. Our results indicate that whilst novices who sketched flow diagrams were found to perform better, these sketches were seldom completed and did not necessarily lead to efforts to mentally simulate program execution. We discuss the implications of these findings for instruction to promote strategic knowledge in introductory programming.

II. Introduction

The term "sketching" has been defined as the act of creating illustrations or diagrams, typically using pen and paper, while solving programming problems [1]. Over the last twenty years, researchers in computing education have established that sketching contributes to better performance when students solve code reading and understanding problems. "Tracing" in this case refers to the act of recording intermediate states to mentally simulate the execution of a program [2]. Findings consistently show that students who choose to create detailed and complete sketches while tracing through code perform better on problems that require an understanding of how a piece of code works, especially in cases where the operations performed on data, as well as the order of these operations, are more complex.

Prior research has focused on sketching as a technique for teaching code reading skills. Our work, however, examines the use of sketching for facilitating cognitive processes used for designing solutions to code writing problems. In this study, a control flow diagram (see Figure 1) is defined as a type of sketch depicting the operations that must be performed on data in a program, and the order in which these operations must occur. Techniques for creating control flow diagrams include writing pseudocode statements and explicitly acknowledging intermediate states that will occur during program execution.

Novice programmers often face difficulties in solving programming problems [3]. To what extent do novices sketch while designing solutions? Does sketching lead to improved performance? We predict that sketching complete and detailed flow diagrams contributes to better student performance in code writing tasks. This is justified by two design principle that differentiate between more and less effective flow diagrams. The first of these principles is the generativity principle, which states that including semantic elements, rather than just syntactical ones, contributes to the writing of more accurate solutions. The second is the specificity principle, which is that solutions are more correct when syntactical elements of flow diagrams are presented unambiguously. Examples of this include specifying data types and exact operations in writing.

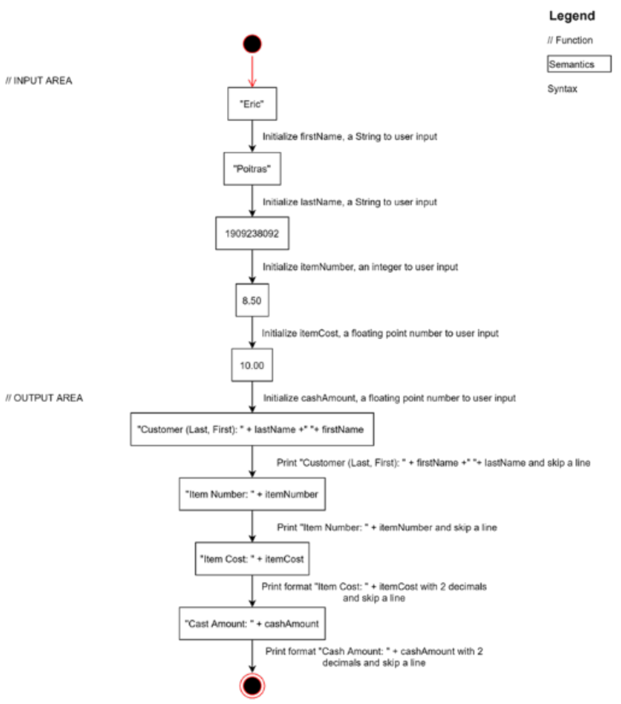
Legend
// Function
Semantics
Syntax
// INPUT AREA
"Eric"
Initialize firstName, a String to user input
"Poitras"
Initialize lastName, a String to user input
1909238092
Initialize itemNumber, an integer to user input
8.50
Initialize itemCost, a floating point number to user input
10.00
// OUTPUT AREA
Initialize cashAmount, a floating point number to user input
"Customer (Last, First): " + lastName +" "+ firstName
Print "Customer (Last, First): " + firstName +" "+ lastName and skip a line
"Item Number: " + itemNumber
Print "Item Number: " + itemNumber and skip a line
"Item Cost: " + itemCost
Print format "Item Cost: " + itemCost with 2 decimals and skip a line
"Cast Amount: " + cashAmount
Print format "Cash Amount: " + cashAmount with 2 decimals and skip a line

Figure 1: An example flow diagram including pseudocode and trace depictions of operations designed as a solution to a programming problem.

III. Method

A. Study Design

This study consists of the first round of a design-based research project to guide the design, implementation, and refinement of instruction for design skills in the context of introductory programming [4]. Design-based research is a methodological approach in the learning literature that examines learning processes in authentic contexts through the systematic design and study of instructional strategies and tools. The design worksheets outline the steps involved in designing a solution in terms of the following skills: (1) re-read the problem prompt and take notes; (2) compare test case examples; (3) create a novel test case example; (4) sketch a diagram that describes data and control flow changes during program execution; and (5) write pseudocode to describe operations performed by the program on the data. These steps are completed by students by completing the relevant sections of the worksheet that are progressively faded throughout five course assignments. The course instructor provided feedback to worksheets for students to implement their solutions by writing code. Pre-training was provided during the second week of the course using a series of video-recorded demonstrations to model how to engage in each skill.

B. Sample

The participants in this study were undergraduate students recruited from a single section of a CS1 introductory programming course at Dalhousie University. This course is typically offered to novice programmers who are inexperienced with Java or learning Java as their first programming language. A total of 18 students consented to participate in the study voluntarily to earn additional course credit and 5 of these students completed the demographic survey. The median age category reported by the participants was twenty or younger, and the sample was comprised of 40% male students.

C. Measures

The dependent variables represent a percentage that indicates how many times an error was observed, including both compilation and runtime logical error types, when observing edits made by a student to their solution from the keystroke data recordings. To answer the research questions, we first examined the relationship between compilation errors and the percentage of syntactic elements made explicit in the flow diagram when observing pseudocode statements as the first independent variable. The second research question was addressed by examining the relationship of logical runtime errors with the presence or absence of traces in the operations depicted in the flow diagram as a second independent variable.

IV. Results

The key claims warranted based on our findings can be listed as follows: (1) students seldom sketch when solving code writing problems in the context of introductory programming; (2) sketches often include specific, unambiguous descriptions of operations; and (3) sketches may not necessarily induce efforts to mentally simulate operations by tracing intermediate values during program execution. For the purposes of reporting descriptive statistics, students are referred to by pseudonyms in commenting on specific cases.

A. Students seldom sketch flow diagrams

Despite the opportunity to plan a solution to a programming problem and receiving formative feedback on their own design, it was found that very few students sketched while writing code, with only 3% of students in the course submitting design worksheets. This finding is consistent with prior research that examined sketches made while solving code writing problems during exams, which are limited to re-writing code snippets or writing notes about intended functionality of the solution [5].

B. Sketches often describe operations in a specific manner

Overall, 81% of elements in pseudocode statements included in student flow diagrams were mentioned explicitly. Although Logan wrote the least ambiguous description of operations with 96% of elements identified across each problem of the assignment, Ezra only mentioned 40% of elements by omitting descriptions of common operations such as print statements. A significant linear regression equation was found ($F(1, 35) = 4.37$, $p < .05$, with an R^2 of 11.09%). The percentage of correctness for edits made while writing syntactic statements increased by 33.81% when the pseudocode described an operation in an unambiguous manner, $t(35) = 2.09$, $p < .05$ (95% CI .97%, 66.6%).

C. Sketches may not elicit tracing the results of operations

These sketching effects are attributed to offloading verbal information to a pictorial medium while sketching, therefore student efforts to trace during the forethought phase reduces the likelihood of logic errors (i.e., generative principle). Additionally, efforts taken to write unambiguous descriptions of operations reduces the likelihood of compilation errors (i.e., specificity principle). However, given the limitations of our sample size and design, we cannot conclusively support our claim that this led to better performance in implementing solutions. Our presentation will elaborate further on the implications for future research and limitations to the validity and generalizability of these findings.

V. Conclusion

These sketching effects are attributed to offloading verbal information to a pictorial medium while sketching, therefore student efforts to trace during the forethought phase reduces the likelihood of logic errors (i.e., generative principle). Additionally, efforts taken to write unambiguous descriptions of operations reduces the likelihood of compilation errors (i.e., specificity principle). However, given the limitations of our sample size and design, we cannot conclusively support our claim that this led to better performance in implementing solutions. Our presentation will elaborate further on the

implications for future research and limitations to the validity and generalizability of these findings.

References

[1] Cunningham, K., Ke, S., Guzdial, M., & Ericson, B. (2019, July). Novice rationales for sketching and tracing, and how they try to avoid it. In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (pp. 37-43).

[2] Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., ... & Ko, A. J. (2019). A theory of instruction for introductory programming skills. Computer Science Education, 29(2-3), 205-253.

[3] Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. ACM Transactions on Computing Education (TOCE), 18(1), 1-24.

[4] Collins, A. (1992). Toward a design science of education. In New directions in educational technology (pp. 15-22). Springer, Berlin, Heidelberg.

[5] Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017, August). Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In Proceedings of the 2017 ACM Conference on International Computing Education Research (pp. 164-172).