AN ABSTRACT OF THE DISSERTATION OF

Islam M. Almusaly for the degree of Doctor of Philosophy in Computer Science presented on August 17, 2017.

Title: Custom Keyboards for Inputting Programs on Touchscreen Devices

Abstract approved: _____

Carlos Jensen Ronald Metoyer

Soft keyboards come in different shapes, sizes, and layouts. Each layout is designed to help the users in different inputting tasks. Most of these layouts, however, focus on general text entry as opposed to computer programs. This dissertation addresses the problems with current input mechanisms on touchscreen devices. The dissertation presents the design and evaluation of different custom keyboards for inputting programs on touchscreen devices. It shows the advantages of using the soft custom keyboards over the default input techniques. The contributions include (1) *Syntax-Directed Keyboard Extension*—a novel keyboard design for inputting Java source code efficiently and accurately; (2) *Evolution and Evaluation of the Syntax-Directed Keyboard*—a longitudinal evaluation of an enhanced design of the Syntax-Directed Keyboard using Java source code statistics; and (3) *Evaluation of A Visual Programming Keyboard*—a soft keyboard alternative to the drag-and-drop method to input block-based programs and its evaluation. These

custom keyboards were evaluated with empirical user studies involving human participants.

Custom Keyboards for Inputting Programs on Touchscreen Devices

by

Islam M. Almusaly

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 17, 2017
Commencement June 2018

Doctor of Philosophy dissertation of Islam M. Almusaly presented on August 17, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Islam M. Almusaly, Author

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to both of my advisors Professor Carlos Jensen and Professor Ronald Metoyer. You have been tremendous mentors for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my courses have been priceless. I would also like to thank my committee members, professor Eric Walkingshaw, professor Kenneth Funk, and professor Michael Pavol for serving as my committee members even at hardship. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

A special thanks to my family. I am grateful to my siblings Kumail, Rayan, Sumana, and Afnan, who have provided me through moral and emotional support in my life. Words cannot express how grateful I am to my mother Maasoma and my father Mohammed for all of the sacrifices that you've made on my behalf. Your prayer for me was what sustained me thus far. I am also grateful to my other family members and friends who supported me and encouraged me to strive towards my goal.

I am very grateful to my god who have bestowed me with kindness beyond what is due. I am grateful to my daughter Rodeen who gave me joy and happiness. At the end I would like express appreciation to my beloved wife Bashayr who spent sleepless nights and was always my support in every moment.

Thank you for all your encouragement!

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

Chapter 1 : Introduction

With the increasing availability of powerful touchscreen devices, which now outnumber all other computing platforms, they are becoming an integral part of our lives. Mobile is increasingly the sole computing platform for large parts of the world's population. They are used in schools, houses, and the shops. Nevertheless, they are not being used to write programs as laptops or desktop computers. This is because there are many problems when it comes to inputting programs on a touchscreen. One of the reasons that make inputting programs a difficult task on touchscreen devices is the lack of physical keyboards. Not extending coding paradigms to mobile excludes large populations from engaging in critical skill development and personal growth. A world where touchscreen devices are commonly used to write computer programs rather than watching videos and playing video games is a better world.

To extend programming access on touchscreen devices, domain-specific soft keyboards were developed. These custom keyboards allow programs input faster, more efficiently, and more accurately than the default input method. QWERTY keyboard is used to write text-based programs like Java. To input block-based programs like Blockly, drag-and-drop is used. However, these input means are not designed to take full advantage of touchscreen devices, especially when it comes to input programs. The custom keyboards, on the other hand, were designed to

minimize the input effort, time, and errors on touchscreen devices. They do that by minimizing the fingers' travel time and the keystrokes.

The design, development, and evaluation of the custom keyboards are discussed in the following chapters. Chapter 2 discusses the design of a custom soft keyboard for writing Java source code. It shows how it was designed using syntax-directed approach and the use of Cognitive Dimension framework to enhance its usage. In this chapter, the custom keyboard was evaluated by conducting a formal user study. The results showed how the custom keyboard outperformed the standard input method (the QWERTY keyboard) when it comes to inputting Java source code. The custom keyboard has better efficiency and accuracy. On top of that, the custom keyboard reduces the mental, physical, and temporal demands when compared to that standard QWERTY keyboard.

In chapter 3, the evolution of the custom keyboard is presented with step by step on how user feedback and Java source codes statistics influenced the design decisions. The enhanced custom keyboard then evaluated with a longitudinal study to measure the performance and user feedback over eight sessions in a period of two weeks. The results of that evaluation showed the long-term advantages of using a custom keyboard to input programs on touchscreen devices. Advantages like the reduce error rate, the enhanced input speed, and increased efficiency. The study showed an average speed of 30.29 WPM by the eighth session which is 16.5% faster than the theoretical top speed of an expert QWERTY keyboard user when inputting Java source code.

By showing that the custom keyboard on a textual programming language like

Java performs better, we wanted to extend the idea further to visual programming languages. Chapter 4 presents a visual programming soft keyboard to input blocks as an alternative to the drag-and-drop method. In this chapter, the design decisions to create a custom keyboard for blocks input are explained. A user study to evaluate this keyboard is presented in this chapter. The user study showed how the custom keyboard for blocks input exceeded the standard blocks input methohd (the drag-and-drop) in terms of speed, accuracy, and efficiency. In addition to these advantages, the custom keyboard was preferred over the drag-and-drop when it comes to blocks input on touchscreen devices.

Finally, chapter 5 represents a general conclusion about custom soft keyboards for programs input on touchscreen devices. It discusses the advantages of the custom keyboard with a summery of their performances.

1.1 Contributions

1.1.1 A Syntax-Directed Keyboard Extension for Writing Source Code on Touchscreen Devices

The custom keyboard idea started with some experimentation guided by Prof. Ronald Metoyer. I contributed to this work by designing and implementing the keyboard. In addition, I designed and conducted the user study with the help of Prof. Ronald Metoyer. Prof. Metoyer and I wrote the paper together. We got feedback from our colleagues.

1.1.2 Syntax-Directed Keyboard Extension: Evolution and Evaluation

My contribution to this work was by enhancing the design of the custom keyboard. I collected and analyzed the Java statistic. With the help of Prof. Carlos Jensen and Prof. Ronald Metoyer, I designed and conducted the user study. I wrote this paper with the cooperation of Prof. Jensen and Prof. Metoyer.

1.1.3 Evaluation of A Visual Programming Keyboard on Touchscreen Devices

I contributed to this research by designing and implementing the blocks keyboard. In addition, I designed and ran the user study. I wrote the paper for this research. Prof. Carlos Jensen and Prof. Ronald Metoyer helped me throughout the entire research. From the design of the keyboard to enhancing the paper.

Pages#195-202

A Syntax-Directed Keyboard Extension for Writing Source Code on Touchscreen Devices

Islam Almusaly and Ronald Metoyer

IEEE Symposium on Visual Languages and Human-Centric Computing

Atlanta, Georgia, USA

2015

Chapter 2 : A Syntax-Directed Keyboard Extension for Writing Source Code on Touchscreen Devices

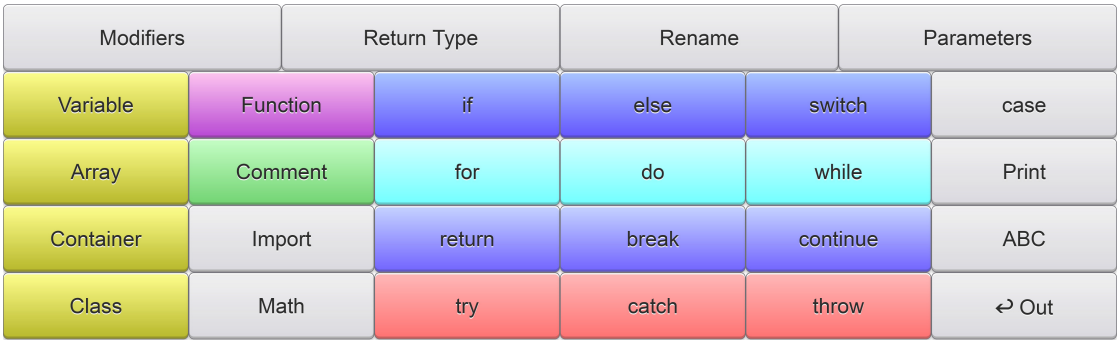

Figure 2.1: Our keyboard extension for Java program input. The extension serves as the top-level keyboard and provides entry to the standard soft keyboard when necessary.

2.1 Abstract

As touchscreen mobile devices grow in popularity, it is inevitable that software developers will eventually want to write code on them. However, writing code on a soft (or virtual) keyboard is cumbersome due to the device size and lack of tactile feedback. We present a soft syntax-directed keyboard extension to the QWERTY keyboard for Java program input on touchscreen devices and evaluate this keyboard with Java programmers. Our results indicate that a programmer

using the keyboard extension can input a Java program with fewer errors and using fewer keystrokes per character than when using a standard soft keyboard alone. In addition, programmers maintain an overall typing speed in words per minute that is equivalent to that on the standard soft keyboard alone. The keyboard extension was shown to be mentally, physically, and temporally less demanding than the standard soft keyboard alone when inputting a Java program.

2.2 Introduction and Related Work

Touchscreen devices such as smart-phones and tablets are making tremendous gains in usage in the United States and around the world. According to Pew Internet Research, 58% of American adults own smartphones and 42% own a tablet device and growth is expected to continue [1, 2]. International Data Corporation forecasts that between 2013 and 2017, desktop sales will decrease 8.4% while laptop, smartphone, and tablet sales will grow 8.7%, 71% and 79% respectively [3].

While touchscreen devices have many strengths, text input using the standard QWERTY soft keyboard, from here on referred to as the standard soft keyboard, is not one of them. Unlike physical keyboards, mobile device soft keyboards are generally small and typically require switching between character and numerical/symbol input screens [4]. While the research community has explored options for improving text input on soft keyboards and physical keyboards as well, these approaches tend to focus on general text entry tasks [4, 5, 6, 7, 8, 9].

Many mobile device tasks, however, are carried out in contexts with very spe-

cific, structured language and require entry of domain-specific text. For example, consider the use of mobile devices in a physical therapy setting, where exercise prescriptions may be input on a tablet device. Therapists are trained to use a particular protocol based on Frequency, Intensity, Time, and Type (FITT) to specify prescriptions. For example: Do 3 sets of 4 repetitions (Intensity) of squats (Type), every other day (Frequency), for one week (Time). Or, consider the domain of computer programming where the programming language is naturally structured by the grammar. Domain-specific contexts such as these present unique opportunities to take advantage of this structure to develop more efficient means for text input on touchscreen devices.

Computer programming presents a particularly interesting domain for touchscreen device text input. First, as mobile device use grows, developers will eventually seek to write code on them [10, 11]. Second, mobile touchscreen devices are heavily used in K-12 settings where there are also many efforts to introduce programming [12, 13]. While many drag and drop solutions exist for teaching programming, there are few tools designed for more traditional text-based programming on mobile devices [14, 15, 16].

In this paper, we present a soft keyboard extension designed specifically with Java developers in mind. Our goal was to reduce input errors while also improving speed and efficiency. Our design utilizes frequently used domain primitives instead of characters as the input unit, spatially groups primitives according to function, and employs a syntax-directed approach to reduce errors (See Fig. 2.1).

We make two specific contributions in this paper. First, we present our soft

keyboard extension designed specifically for Java program input on touchscreen devices. Second, we present empirical results that indicate that users perform programming input tasks with fewer errors and more efficiency when using the keyboard extension as compared to the standard soft keyboard alone. Moreover, using the keyboard extension is mentally, physically, and temporally less demanding.

In the following section, we will present background information to structure our discussion of keyboard design and evaluation. We will then present our soft keyboard extension and a laboratory study designed to analyze the effectiveness of this keyboard as compared to the standard soft keyboard alone. We then discuss the results of the study and its implications, and we conclude with a discussion of several avenues of future work.

2.3 Background

In this section we discuss previous work related to keyboard design and evaluation. We then present several metrics that will be used to evaluate our soft keyboard extension and we discuss the syntax-directed approach that we have built upon in our keyboard.

2.3.1 Keyboard Design

There are many keyboards designed to improve input efficiency or reduce errors in general text input tasks, such as writing email, word processing, and texting on physical keyboards [17], as well as on soft keyboards [4, 5, 9, 18, 19]. On the other hand, gesture-based entry methods have an acknowledged speed disadvantage and they are not designed for speed/error improvement, but rather ease of input on a standard keyboard [20]. In this paper we focus on soft keyboard design for input in particular domain contexts, as opposed to general text input. Our specific goal is to demonstrate the efficacy of domain-specific keyboards for reducing errors, as well as improving typing efficiency and speed for users writing source code.

2.3.2 Programming on Tablet Devices

There are some attempts to ease programming on touchscreen devices [21, 22, 23]. However, these tools focus only on editing *existing* code. To our knowledge, TouchDevelop, by Microsoft, represents the only attempt to focus on text-based code input as opposed to editing of *existing code* on touchscreen devices [10]. TouchDevelop represents a completely new language and integrated development environment (IDE) for writing computer programs on touchscreen devices as opposed to working with existing languages. They employ a soft keyboard as part of this IDE. In order to write a program in TouchDevelop, users must move their fingers between the soft keyboard and other elements of the IDE. With our design, however, users can input an entire program without lifting their fingers from the

keyboard area. While it does not specifically target novice users, TouchDevelop is advertised as being a platform for both teaching and learning programming. The TouchDevelop keyboards and interface, however, have not been evaluated for usability or efficiency. Our goal was to evaluate the use of such keyboards for program input.

2.3.3 Syntax-Directed Editing

Syntax-directed editors were introduced in 1981 to improve programmer efficiency by taking advantage of the hierarchical composition of computational structures in programs [24]. In doing so, syntax directed editors enforce proper syntax at all times. For example, a syntax-directed environment may require that the user type commands in order to generate template code with assignments and expressions that have to be completed before moving on [24]. A programmer in this environment, therefore, cannot begin to efficiently use the system before memorizing the commands. Our keyboard extension also utilizes the hierarchical components of computational structures, however, we avoid the above problem by encoding the commands *visually* in the soft keyboard design as primitives and augmenting the keyboard with a dynamic component.

2.3.4 Typing Performance Metrics

There are three primary metrics for measuring text input on a keyboard: accuracy, efficiency, and speed. Accuracy is measured in terms of errors [25]. Error metrics include the minimum string distance error rate (MSD) and the total error rate (TER) [26]. MSD is a measure of the total number of errors (i.e., omissions, substitutions, and insertions) in the resulting typed text. TER, on the other hand, reflects these same errors in the final typed text, as well as corrections that are made during the typing of the final text. Keystrokes are categorized into four classes within an input stream: *Correct* (C), *Incorrect Fixed* (IF), *Fixes* (F), and *Incorrect and Not Fixed* (INF) [26]. We will use all four keystroke classes to compute the TER.

Keystrokes per character (KSPC) measures the average number of keystrokes required to enter a single character [27]. The KSPC on a standard physical QWERTY keyboard is approximately 1.00 [27] but has been shown to reach up to 1.21 when correcting errors [26].

Text entry speed is typically measured in words per minute (WPM), where a word is assumed to consist of five characters on average. This metric has been used to compare various hard and soft keyboard designs [28, 29]. We use TER, KSPC, and WPM to evaluate the efficiency, speed, and accuracy of input on our soft keyboard extension design as compared to the standard soft keyboard native to iPad tablet devices.

2.4 Syntax-Directed Keyboard Extension

Our Syntax-Directed extension is designed to reduce typing and syntactic errors while increasing source code writing speed and user efficiency. The general design philosophy is to provide the user with the most commonly used programming constructs as primitives on the keyboard extension and to support a syntax-directed editing approach.

Many keyboard designs have been informed by word and letter frequencies for "common English" [4, 8]. Our design was informed by analysis of Java programs. In particular, we performed a frequency analysis to produce a ranking for common keywords and constructs. We also consulted the Java language grammar to leverage the hierarchical nature of the language.

The keys on the extension represent the most commonly used programming keywords (e.g., if, for, return, etc.) and programming constructs (e.g., variable, function, comment, etc.). These keys make up the bottom four rows of the soft keyboard. The top row represents the "options" row of the keyboard extension and is dynamically updated with options that correspond to the previously selected key (See Fig. 2.1). Fig. 2.2 shows three versions of the the options row as it would appear after pressing the function, variable, and modifiers keys respectively. Some keys, such as the "try" key, have no options.

Keys are placed to facilitate search. This is done by grouping related keys together spatially and encoding them with the same background color. The spatial and color encoding utilizes gestalt principles to help users visually group elements

Figure 2.2: The options row is empty until a key is pressed. The top, middle, and bottom images show the appearance for the options row after pressing the "function", "variable", and "modifiers" keys respectively.

that correspond to similar programming constructs, supporting visual working memory [30]. For example, keys that represent the keywords of *conditionals* are grouped in a row and colored similarly. Likewise, all *looping* related keys are grouped together and colored accordingly.

Variables and functions are used most frequently, so we placed them in the beginning of the first row. Conditional statements are used more than looping statements. Therefore, we placed them in the first row after the "Variable" and "Function" keys. We used the language hierarchy information to place high level, frequent constructs in the bottom four rows and to present deeper language constructs via the options bar. We considered various hierarchy depths and key sets for the design. More keys, however, require more search and/or memorization by the user. In addition, a shallower hierarchy decreases the cognitive load associated with hidden dependencies.

Most of the keys represent the top level elements in the Java language grammar. When a key is pressed, the keyword and boilerplate syntax for that key are either inserted into the code or the user is prompted to provide additional information via the options bar or via the standard soft keyboard. For example, the standard

keyboard may appear to prompt the user to enter a variable name, or the options area may change to require further specification of a hierarchical construct (e.g. a variable type). The options associated with a key are determined by the grammar and will go three layers deep at most (e.g. variable → type → int). Table. 2.1 shows an example of inputting a Java function. The code in the second column resulted from pressing the keys shown in the first column of the same row.

For example, when a programmer touches the "Function" key, our keyboard produces the boilerplate code and displays the QWERTY keyboard for the user to type the function's name. When he is done typing the name (Table 2.1, second row), he can press the "Done" key to see the function options (Fig 2.2, first row). He can insert the "public" modifier by selecting the "Modifiers" key then the "public" key (Table 2.1, third row) from the options row (Fig 2.2, third row). The "void" return type can be inserted using the same method. The keyboard inserts the modifiers and return type in their correct place.

Table 2.1: Example of creating a new function. The code in the second column resulted from the key presses in the first column. The length column shows the number of characters in the code including the new line character and the final columns shows the total number of keystrokes required.

Key pressed	**Code**	**length**	**Total Keystrokes**
Function	`(){` `}`	5	1
s , e , t , shift , X , Done	`setX(){` `}`	9	7
Modifiers , public	`public setX(){` `}`	16	9
Return Type , void	`public void setX(){` `}`	21	11

Our keyboard extension supports a limited form of syntax-directed editing

which presents both advantages and disadvantages. One disadvantage is loss of flexibility [31]. For example, every inserted statement has to preserve syntactic correctness, which is not always appreciated by programmers who may wish to temporarily violate syntactic correctness as they input their code. To mitigate this problem, the keyboard extension allows the insertion of arbitrary text at any place in the code by pressing the "ABC" button (See Fig. 2.1) to show the standard soft keyboard.

Finally, our design does not enforce the order in which options are selected when multiple are available nor does it enforce the programming language grammar. The programmer can add modifiers, add a type, or rename a variable's identifier in any order. Moreover, the programmer can skip any or all of these option keys to add the next line of code. This approach helps overcome the inflexibility of syntax directed editors while still building on the structure they provide.

2.4.1 Syntax-Directed vs. Prediction

The majority of modern integrated development environments and mobile device text input keyboards employ some form of word prediction and/or autocompletion methods that attempt to reduce the KSPC by reducing the total number of keys typed. In these approaches, the user begins typing and the system infers possible endings and suggests completions. A syntax-directed or structured approach, on the other hand, uses the grammatical structure of the language to fill in boilerplate details; no inference is necessary. Brackets, colons, commas,

and semicolons, for example, can be inserted into their correct place with this approach. Structured approaches are particularly useful in domains that have a highly structured grammar. These are exactly the domains in which we suspect domain-specific keyboard extensions will be effective and thus we have taken a syntax-directed approach as opposed to a predictive approach.

2.4.2 Cognitive Dimensions Analysis

The design of the keyboard extension layout and interaction went through several iterations. We used the Cognitive Dimensions framework to inform our iterations [32] and in the process, we identified several tradeoffs that led to design modifications. Below we discuss some of the dimensions and the associated design changes.

Consistency In a consistent notation, the functionality of an element can be inferred based on what is known about the functionality of other elements. In early designs, many of the keys originally contained several options and these options were different for each key. To simplify, we chose a limited subset of options that are consistent across all similar keys and moved the additional options deeper in the hierarchy.

Hidden dependencies A hidden dependency exists when the relationship between two dependent components of a notation is not fully visible. Our original design contained such dependencies through the use of constraints. For example,

the "else" key originally was visible only after selecting the "if" key, preserving the semantics of an *if/else* statement. This and all similar constraints were removed to eliminate such hidden dependencies, but done so at the cost of language support. While the user can now violate the programming language grammar, many of the hidden dependencies no longer exist. The "options area," however, still contains some dependencies. For example, the list of modifiers can be only viewed after selecting the "modifiers" option. The modifiers options can only be seen when creating or selecting text to edit. We suspect that such relationships can be quickly learned, especially by users with programming experience.

Premature commitment Premature commitment refers to the strong constraints on the order that tasks must be accomplished in a notation. Some syntax-directed editors demand correct program structure at all stages of development. In this case, a programmer needs to have a full hierarchical perspective of the program from the beginning. Our original keyboard extension design enforced such order constraints. In the final design, these constraints were removed, allowing a user to input grammatically incorrect code. This simplified the keyboard by removing the need for look ahead and reduced premature commitment at the cost of allowing input of illegal code.

Role-expressiveness Role-expressiveness refers to the degree to which an element of the notation indicates its role in the system. In the keyboard extension, the design has evolved such that the label for each key indicates its role clearly. This, however, is only true for users with programming experience who are familiar

with the generic labels used to describe imperative programming concepts.

2.5 Study Design

We now describe a formal user study designed to compare input performance with the soft keyboard extension to performance using a standard soft keyboard alone. We chose a copying task as opposed to a programming task to avoid the confounding factors of the cognitive aspects of programming.

We use iPad 2 machines for our study and we implemented the soft keyboard extension in JavaScript. To ensure comparable performance, we also reimplemented the standard soft keyboard in JavaScript. The syntax-highlighting feature was included for both the standard soft keyboard and the soft keyboard extension. We disabled auto-correction and prediction for both keyboards. We decided to not include auto-completion for several reasons. First, we cannot control the effectiveness of the auto-complete algorithms being used. To produce results that generalize beyond the operating system and algorithm, we decided it was best to exclude it. Second, in order to make a fair comparison, we would have to enable auto-completion for both keyboards, however, we cannot force participants to use the feature on either keyboard. We therefore control for this variable to eliminate this threat to validity.

Both keyboard applications were instrumented with JavaScript code to measure and log the KSPC, WPM, and TER. The keystroke classes presented in Section 2.3.4 (e.g., *C*, *IF*, and *F*) are automatically calculated by the instrumented

JavaScript code. Missing keystrokes in the final typed text are classified as *INF* and added to the total *INF* if they prevent compilation.

2.5.1 Tasks

There is no standard code sample in the literature to test program input performance in terms of KSPC, WPM, or TER. Different Java corpus have different percentages of language elements, therefore, there is no single representative corpus. In addition, some language elements are seldom used. For example, there are 0.76 conditional statements per method and 0.11 Try/catch statements per method [33]. To create a representative program where this is true, we would have to create a very long program that would not be doable in the amount of time available for the study. We had to come up with a task that is close to the average program but short enough to reduce task entry time for the purposes of the study. Our programs have 3.5 methods per class which is similar to reported statistics [33]. However, they have 1 class field instead of 1.9 and 0.43 local variable per method instead of 0.87 [33].

We therefore chose two different Java programs that exercise use of the keys with the most options to test the lower bound performance of the keyboard extension. These two programs do not use every key in the keyboard extension because many of the keys, such as all loops, “break”, “continue”, and “return” keys have no options. A coding task that required all of these keys would give an unfair advantage to the keyboard extension. The two programs are quite different in terms

of the constructs used.

2.5.2 Participants

The study participants consisted of 27 males and 5 females all with Java programming experience. All of the participants volunteered for the study in response to an email message circulated to the students in the computer science department at Oregon State University. Fourteen participants were graduate students, 2 participants were recent graduates, and 16 participants were undergraduate students. All but two participants had *never* used a tablet device to write code. Eighteen of the participants considered themselves touch typists (typing without using the sense of sight to find the keys on a physical keyboard).

2.5.3 Experimental Design and Procedure

We used a within-subjects design with repeated measures. The independent variable was the soft keyboard used to complete the programming tasks and the study consisted of two treatments: the standard iPad 2 soft keyboard design and the soft keyboard extension. We asked each participant to enter two different Java programs. Both programs were entered using each keyboard. We balanced the order of the treatments using a Latin Square. The dependent variables were the KSPC, WPM, and TER. We measured these variables for each keyboard and each program independently. We measure and report omission, substitution, insertion,

and spacing errors with the unified error metric, TER [26].

We ran the study in a lab setting in groups of 2-4 participants. After signing an informed consent document, each participant was randomly assigned to one of the two experimental conditions as described above. The group was given a 10-minute tutorial on how to use the keyboard extension and then allowed 5 minutes to practice with it. We encouraged the participants to ask any questions that they might have during the course of the study. The participants then carried out two tasks using the first treatment. After a short break, they carried out the same two tasks using the second treatment. After each task, we asked the participants to complete a NASA Task Load Index (NASA-TLX) questionnaire for assessing subjective mental workload [34]. When all tasks had been completed, we asked participants to complete a post session questionnaire about their experience with the two keyboards.

2.6 Results

Our initial hypothesis was that users would input the Java programs faster, more efficiently, and with fewer errors when using the soft keyboard extension as compared to the standard soft keyboard alone. Thus, our null hypothesis for all analyses is that there is no significant difference between the distributions of corresponding performance measures across the two keyboard designs. For all measurements we use a paired t-test analysis. Fig. 2.3 summarizes performance on each metric for each keyboard design.

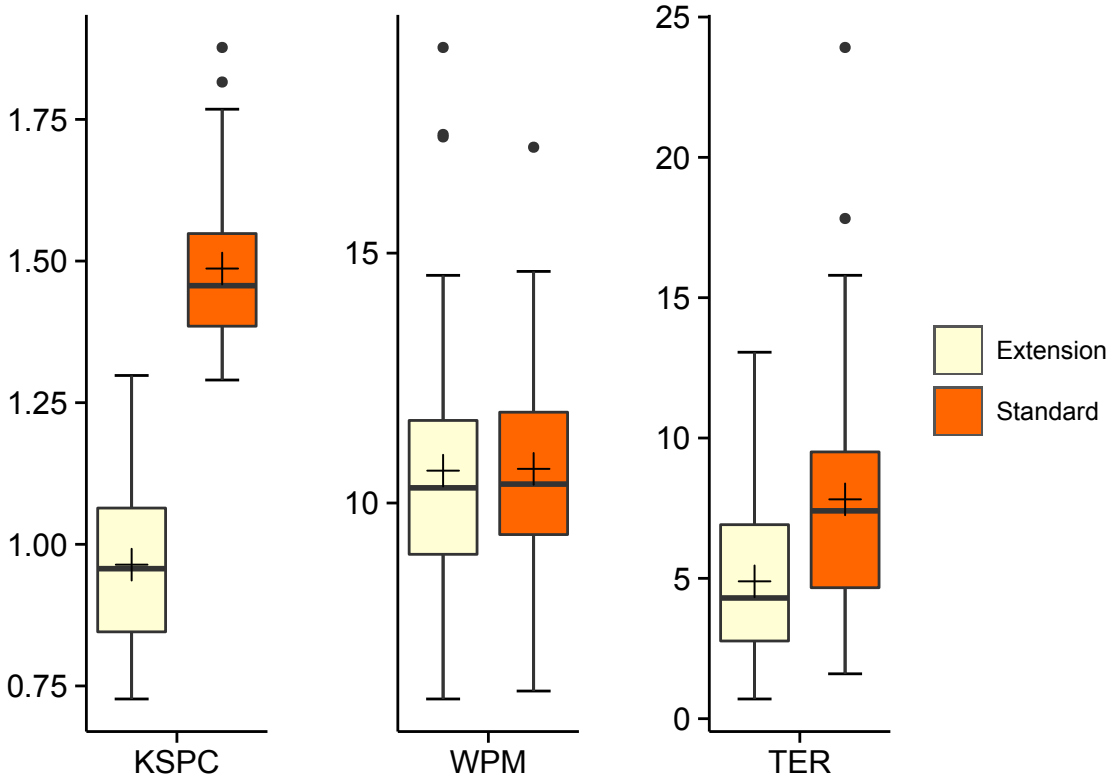

Figure 2.3: KSPC, WPM, and TER for both the standard and extension keyboards. The mean is shown with the "+" sign.

2.6.1 TER

Participants' mean total error rate for the standard soft keyboard and soft keyboard extension was 7.81% (SD: 3.74%) and 4.89% (SD: 2.54%), respectively. There was convincing statistical evidence for an effect of keyboard design on TER ($t_{(31)}$ = -4.59 , p<.0001). See the third column in Fig. 2.3.

2.6.2 KSPC

Participants used fewer keystrokes per character when typing programs with the soft keyboard extension (0.97, SD: 0.09) as compared to the standard soft keyboard (1.49, SD 0.11). This represents a 34.9% reduction in KSPC with the soft keyboard extension. In fact, there is convincing statistical evidence for an effect of keyboard on KSPC ($t_{(31)}$ = -24.66, p<.0001). This improvement is expected because our design replaces many key presses with single keys that represent entire words or constructs and the corresponding syntax elements.

2.6.3 WPM

Participants typed on average 10.68 WPM (SD: 1.78) and 10.64 WPM (SD: 2.31) on the standard and extension keyboards, respectively. Pairwise t-tests show no significant differences in WPM between the two keyboards ($t_{(31)}$ = -0.1348, p>.05). Fig. 2.3, however, gives us a clearer picture of performance with respect to WPM. In particular, note that the two largest WPM values for the soft keyboard extension are both higher than the largest WPM value for the standard soft keyboard, while the worst case performance for both are similar (7.27 and 7.55 for the standard and extension keyboards respectively).

2.6.4 NASA-TLX

Table 4.2 shows the mean response values for the NASA TLX questionnaire measures. While there was no statistical evidence for perceived difference in performance, there was convincing statistical evidence for an effect of keyboard on all other TLX measures including mental demand ($t_{(31)}$ = -2.655187 , p<.01), temporal demand ($t_{(31)}$ = -4.024615 , p<.001), physical demand ($t_{(31)}$ = -5.574217 , p<.0001), effort ($t_{(31)}$ = -5.574217 , p<.0001), and frustration ($t_{(31)}$ = -3.256373 , p<.01). Fig. 4.5 summarizes the TLX questionnaire results.

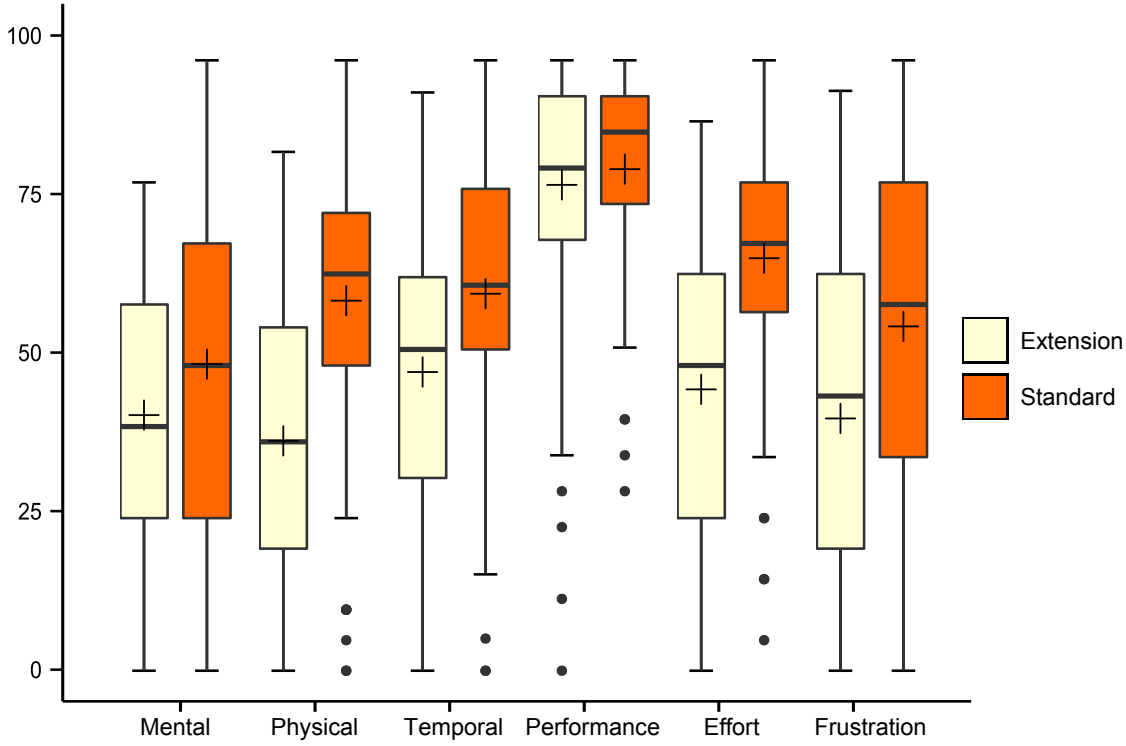

Figure 2.4: Boxplot summary for the NASA-TLX measures

Table 2.2: NASA-TLX measures comparison (mean response) between the standard iPad keyboard and the keyboard extension.

TLX Measure	**Extension**	**Standard**
Mental Demand	41.88	50.23
Physical Demand	37.66	60.63
Temporal Demand	46.48	58.67
Performance	82.66	84.84
Effort	46.09	67.58
Frustration	41.33	56.41

2.6.5 Participant Feedback

At the end of the study, participants filled out a questionnaire about their experience with the two keyboards. They were asked to rate the helpfulness of the two keyboards from 0 (Not helpful) to 100 (Very helpful) when writing a Java program. Participants on average ranked the keyboard extension (70.2) to be more helpful compared to the standard keyboard (36.1). When asked to rank ease of use of the two keyboards on a scale from 0 (Very difficult) to 100 (Very easy), they ranked the keyboard extension to be easier (68.0) on average to use when compared to the standard keyboard (42.5). 47% of the participants preferred the keyboard extension while 31% preferred the standard soft keyboard. 22% had no preference.

2.7 Discussion

The results of the study indicate that users performed code input tasks better or no worse when using the soft keyboard extension as measured by TER, KSPC and

WPM. There are several explanations for this result. The keys in our keyboard are large and according to Fitts' law, this will positively affect the typing speed [35]. Larger targets will also result in fewer typing errors. Most importantly, because we design to the domain and take advantage of the constraints imposed by the domain, we can insert much of the boiler-plate syntax for the user, hence improving KSPC and reducing errors. Fig. 2.5 shows an example of a partially input program. In this example, a programmer switched to the standard keyboard 10 times to type the highlighted code. The rest of the program was typed using presses strictly from the keyboard extension. The reduction in the number of keys pressed, due to the use of keywords as opposed to characters as input units, results in a lower KSPC. For example, Table 2.1 shows the total keystrokes and the length of the code. In this example, it took only 11 keystrokes to type 21 characters. The combination of these factors results in improvements even after using the keyboard extension for only a short period of time. It is reasonable to suspect that users would perform better after using it for longer periods, an observation made in the use of other keyboards [8, 36].

Efficiency and Errors We found that in terms of KSPC, participants were much more efficient with the keyboard extension than with the standard soft keyboard alone. In fact, for every 100 characters in the code, a participant pressed approximately 149 keys when using the standard soft keyboard compared to 97 keys on average when using the soft keyboard extension. Although participants switched from the extension to the standard soft keyboard to write 55% of all

```
public class Point{
    public int x;
    public int y;
    public Point(int nx, int ny){
        x = nx;
        y = ny;
    }
    public int getX(){
        return x;
    }
    public void setX(int nx){
        x = nx;
    }
}
```

Figure 2.5: The standard keyboard was only used to type the highlighted code.

characters in the input tasks, KSPC and TER were still improved when using the soft keyboard extension.

Errors are quite common on soft keyboards due to the small size and because they produce no tactile feedback. This results in an increase in typing mistakes such as unintentionally pressing keys adjacent to the intended typed key [37]. This mistake is magnified as the size of the keys decreases [7]. In our study, the standard soft keyboard has 12% more keys than our keyboard extension and the keys are therefore smaller.

In addition, the keyboard extension uses complete word primitives as opposed to individual characters. These two factors combined result in a lower error rate for

the soft keyboard extension. In fact, the total error rate was decreased by 37.38% when using the soft keyboard extension. While this finding is important in general for this particular domain-specific case of Java program input, it is particularly important for the large class of users with motor disabilities. These users could benefit greatly from the use of a soft keyboard that reduces the overall number of keys required to input code and thus the likelihood of mistyping errors [38].

Speed A slightly lower or equivalent WPM using the keyboard extension is not surprising, in hindsight, due to the need to learn the layout of the keyboard. For those who have not memorized the layout, a visual search process is necessary to find the key to be typed. Our participants did not have the benefit of time to learn the keyboard extension layout. After only 5 minutes of practice, however, the code entry speed of participants using the soft keyboard extension was as good as that using the standard soft keyboard.

NASA-TLX: User Perceptions Participants on average felt that our keyboard is not as mentally, physically, or temporally demanding as the standard keyboard alone. A programmer does not have to remember to close parentheses or braces because the syntax-directed keyboard inserts them at their correct place. By doing so, our keyboard reduces the mental load. Moreover, it reduces the physical load by decreasing the keystrokes. The participants also did not feel that they were rushed with our keyboard. This could be due to the keystrokes reduction. By reducing the mental and physical demand, participants felt that the syntax-directed keyboard requires less effort and little frustration compared to the standard keyboard by

itself.

However, participants perceived that they performed *worse* using our keyboard extension despite their better KSPC and TER measures using our keyboard. When asked about their performance, some participants said that they would perform better if they were given more time to practice. Even without a significant learning phase, participants reported a lower mental, physical and temporal demand, as well as lower perceived effort and frustration when using the soft keyboard extension. These lower perceived workload measures support the significant advantages found for the keyboard extension with regards to TER and KSPC.

2.7.1 Threats to Validity

Construct validity: Our JavaScript implementations were instrumented with code to collect the number of keystrokes, the input program text, and the time participants spent typing the programs. This code produces an accurate timing of participants, with the exception of the very last keystroke, eliminating most potential human timing errors. We therefore obtain accurate KSPC, WPM, and in-situ error measurements (errors that were fixed during entry). All remaining errors in the entered programs, that would prevent compilation, were then counted by the researcher to compute the TER. While efforts were taken to accurately count errors, there is the possibility of human error and thus a threat to construct validity. The task that the participants entered could be another threat because the lack of a comprehensive statistics about Java source code.

External validity: The tasks that we designed were intended to use keys with the highest number of options in order to produce a lower bound on performance with the keyboard extension. While we could have included more programming constructs, such as loops and conditionals, these constructs contain a significant amount of boilerplate syntax and few "options" and therefore may bias the results in favor of the keyboard extension. We have confirmed this bias with sample test cases that included these constructs. Nonetheless, while we took efforts to design fair tasks, there is room for human bias and thus a potential threat to external validity. In addition, the two programs do not cover a wide variety of programming concepts. Different program types might produce different results and thus our choice of program may be a threat to external validity.

2.7.2 Limitations

The keyboard extension is not without limitations. Some limitations arise from using a syntax-directed editing approach [31]. However, these limitations were mitigated by allowing arbitrary text insertion in the code. This requires that the user switch from the extension to the standard keyboard, which is done with an additional key press. Another limitation is that the keyboard extension is designed for input of the Java programming language only. This can be addressed by adding support for other languages or by modifying the keyboard design to apply to a more general class of coding such as "imperative programming." This is left for future work. Finally, the keyboard extension represents a completely

new layout. This may cause significant concern for users who are comfortable with the standard keyboard layout and not willing to put in the time to learn a new keyboard layout, primitive set, and functionality.

2.8 Conclusions and Future Work

Our study indicates that a soft keyboard extension for input of Java programs on touchscreen devices has many benefits including improvements in efficiency and error rates, while not degrading input speed. In addition, users perceive the soft keyboard extension to be less mentally, physically, and temporally demanding, as well as less frustrating, and requiring less effort to use.

The benefits of improved keyboard input on touchscreen devices can impact programmers as such devices become more ubiquitous. This can have a particularly important impact on users with motor disabilities or those who simply lack fine motor skills, like children and older adults. This is especially timely as mobile touchscreen devices become more prevalent in schools as a computing device.

While our results apply to Java program input only, we suspect that such benefits would be observed for keyboard extensions in other domains that use highly structured input language. An important avenue of future work is to study alternative application domains such as exercise prescription and medication prescription.

The soft keyboard extension presented in this paper was designed based on manual statistical analysis. For many domains, there is a wealth of data to draw upon in order to inform the design of a soft keyboard extension. We intend to

explore machine learning techniques to automate this process for the domains of interest (e.g., large scale Java programs or exercise prescriptions) to determine the appropriate set of keyboard primitives and to exploit spatial and temporal locality in the domain language.

Finally, all study participants learned to use the soft keyboard extension in very little time. We suspect that additional time to learn the interface will significantly improve users' efficiency. We plan to study how much experience with the keyboard is necessary to reach maximum typing speeds.

Accepted

Syntax-Directed Keyboard Extension: Evolution and Evaluation

Islam Almusaly, Carlos Jensen, and Ronald Metoyer

IEEE Symposium on Visual Languages and Human-Centric Computing

Raleigh, North Carolina, USA

2017

Chapter 3 : Syntax-Directed Keyboard Extension: Evolution and Evaluation

3.1 Abstract

The syntax-directed keyboard extension presented by Almusaly et al. in 2015 allows programmers to input Java source code with fewer errors and keystrokes compared to the soft QWERTY keyboard and it supports a comparable typing speed [39]. While these results were obtained after only 10 minutes of practice, it is unclear how long-term use affects performance. In this paper, we present an updated design for the original syntax-directed keyboard extension, replicate the original results, and evaluate the evolved design with Java programmers over eight sessions in a period of two weeks. Our results indicate that a programmer using the new keyboard extension for two weeks can input Java programs 16.5% faster (words per minute) than an expert QWERTY keyboard typist. In addition, we demonstrate that the efficiency and accuracy for inputting Java source code improves with repeated use over time and that perceived mental, physical, and temporal demands of the keyboard extension decrease over time.

3.2 Introduction

Per Pew Internet Research, half of American adults ages 18-29 report owning a tablet device, 86% of whom also own smart phones [40]. These touchscreen devices are becoming more powerful each year and their adoption rates are on the rise. Furthermore, these devices can be used as application creation environments. Many popular programming languages like Java, Python and C++ are text-based languages. However, text input on these predominantly touchscreen driven devices, a key interaction task across many popular apps, can be difficult.

There are many factors that contribute to the awkwardness of touchscreen device soft keyboards, key among them being the device and screen size. While devices come in different screen sizes, their soft keyboards are usually small (when compared to standard physical keyboards), especially when devices are used in portrait mode. Another factor is the lack of tactile feedback. Without the help of tactile feedback, users of soft keyboards are more prone to typing errors where adjacent keys are accidentally pressed [37]. The lack of tactile feedback also contributes to a decrease in typing speed [41]. Moreover, symbols and numbers typically require switching between character, numerical and symbol keyboards, another source of error and slowdown. This is particularly troublesome when considering the use of touchscreen devices to write source code, which often contains symbols and numbers. This leads to an increase in the amount of keyboard switching and therefore an increase in the number of keystrokes required to accomplish the task.

While programming full-fledged applications on mobile devices may seem an

edge-case, there is a significant drive to deliver educational apps, including many that introduce to programming (e.g. Swifty, Hopscotch, and Swift Playgrounds among many), using these devices [42, 43, 44]. In addition, Tillmann et al. predict that mobile device will be used as application creation environments and propose the teaching of programming using them. They presented their experience with school students which shows that programming directly on mobile devices is accessible to students who are beginning to learn programming [45]. Therefore, it is reasonable to expect coding of short modules, or at least the editing of existing code on mobile devices to soon be a mainstream task on touchscreen mobile devices. Although programming languages are highly structured, general purpose soft keyboards do not take advantage of these domain-specific constraints.

The soft keyboard extension of Almusaly et al. was designed to help developers write Java source code on touchscreen devices [39]. It reduces the number of keystrokes and errors compared to the soft QWERTY keyboard while maintaining a comparable typing speed. Professional and novice programmers can benefit from a domain-specific keyboard like the Syntax-Directed Keyboard Extension. This keyboard enables professional programmers to input source code when they do not have access to a physical keyboard in a fast, efficient, and accurate way. For example, professional programmers may find the ability to write or edit code on a mobile device useful in commuting situations where a laptop is sometimes inconvenient, especially because ideas and algorithms emerge at any time. Additionally, the keyboard extension could prove useful for novice programmers, such as K-12 children, learning to program in classroom environments on mobile devices. The

study presented by Almusaly et al, however, looked at short-term learning effects only; the long-term impact of using the keyboard extension was not studied [39]. It is therefore not clear how users will adapt over time to using the Syntax-Directed Keyboard, whether long-term, habitual use will lead to improvements in performance, or how users feel about this keyboard once the novelty wears off.

We therefore seek to make two specific contributions in this paper. First, we present enhancements to the previously designed soft keyboard extension for Java program input on touchscreen devices, aimed at further improving performance. These enhancements include reducing the number of keystrokes, increasing input speed, and reducing the total error rate. Second, we replicate the results of Almusaly et al. after a single use session, and present results of an eight-session empirical study that shows that the speed, efficiency, and accuracy of users of our keyboard extension increase over time. The results of this study provide insight into the change in user performance over time when using a domain-specific keyboard. We also discuss lessons and best practices for designing domain-specific soft keyboards for programming languages.

3.3 Related Work

In this section, we examine previous work related to keyboard design and evaluation, especially as related to programming tasks.

3.3.1 Text Entry Methods

Keyboards remain the most common mechanism for text input and come in various sizes and layouts. Some keyboards are designed to enhance general text input (e.g. writing email and word processing) efficiency or to reduce input errors in general. They can be physical keyboards [17], or soft virtual keyboards [4, 5, 9, 18, 19].

Gesture-based entry methods have also been introduced for general text as well as source code input [20, 23], however, these either have an acknowledged speed disadvantage [20] or were not well received by users in evaluation studies [23].

Speech is yet another way to provide text input. SPEED is a program editor that enables programming by voice and supports writing, editing, and navigating source code [46]. Nevertheless, programmers face difficulties when the speech recognizer misinterprets their speech (not unreasonable given special characters and syntax used in programming), and speech is not always an appropriate option (e.g. in public or quiet spaces).

Despite the variety in ways to input text, few of these were designed with source code entry in mind. In this paper, we focus on the design of a soft keyboard optimized for the writing and editing of Java source code, as opposed to general text. Our specific goal is to demonstrate the potential impact of a syntax-directed keyboard extension, *over time*, for reducing errors as well as improving input efficiency and speed.

3.3.2 Programming on Tablet Devices

Writing or editing programs on touchscreen devices is arguably not as easy as doing so on desktop computers. Nevertheless, tools have been developed to ease the task and support such complex tasks as refactoring [21, 22]. Other tools target computer science education. Tools like Catroid, Hopscotch, ScratchJr, and YinYang [14, 43, 47, 48] all focus on making programming accessible to leaners. GROPG enables programmers to debug their applications on their phone [49].

TouchDevelop, for instance, was designed to ease writing programs on touchscreen devices (e.g. smart-phones) for use on touchscreen devices by creating a new programming language [10]. Another approach has been to develop hybrid solutions allowing the tablet to work in concert with a desktop IDE. Domain-specific gesture languages, performed on a tablet device, have been used to input or manipulate text on a connected desktop IDE [23]. Deverywhere is a tool that uses templates and allows its users to input with speech and touch [50]. However, none of these tools focus on editing or writing code for conventional text-based programming languages (e.g. Java, C, or Python) on touchscreen devices efficiently, accurately, and fast.

3.3.3 Syntax Directed Keyboard Extension

A keyboard extension is a keyboard that serves as the top-level keyboard for the soft QWERTY keyboard. In this study, we build on the Syntax-Directed Keyboard Extension of Almusaly et al., which was designed to ease the task of inputting Java

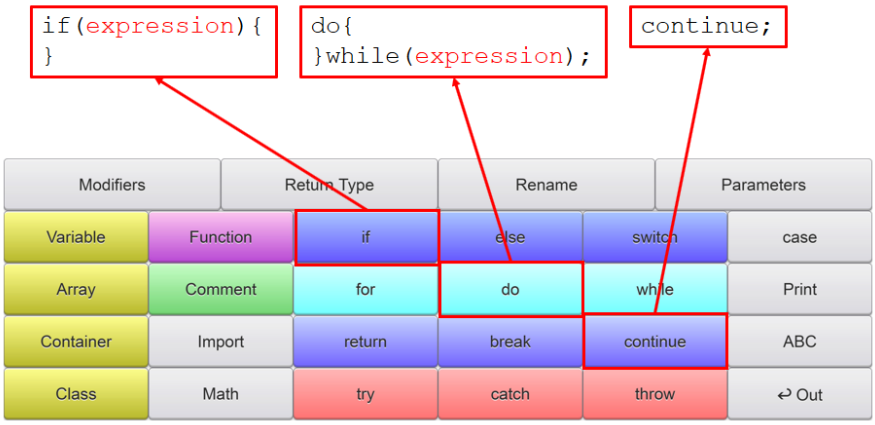

Figure 3.1: The Syntax Directed Keyboard Extension for Java program input. Call-outs give examples of text inserted when button is pressed.

programs on touchscreen devices [39]. This design employs frequently-used domain primitives as the unit of input, instead of individual characters. These primitives are spatially grouped per their function, and the keyboard uses a syntax-directed approach to reduce errors (See Figure 3.1).

Complete statements are inserted by pressing a single key. The call-outs in Figure 3.1 show examples of the statements inserted when the "if", "do", and "continue" buttons are pressed. After pressing one of these keys, the keyboard switches to a standard QWERTY keyboard to let the programmer complete the missing parts of the expression. Additionally, the top row of the keyboard - called the "option area" - presents the programmer with context (statement) dependent key options. Examples of these include modifiers, types, or parameters.

The original keyboard design was evaluated with Java programmers and showed that programmers using this keyboard could input Java source code with fewer er-

rors and fewer keystrokes than when using a soft QWERTY keyboard. Moreover, this keyboard was shown to be less demanding both mentally and physically than the standard soft keyboard for inputting source code on a mobile device. Additionally, programmers maintain a typing speed that is comparable to that on the standard soft QWERTY keyboard. In this paper, we present an enhanced design for the syntax-directed keyboard extension, and perform a more longitudinal study to examine performance and user receptiveness after extended use.

3.4 Improving the Syntax-Directed Extension

We started from the ad hock design of the keyboard extension developed by Almusaly et al. to determine what the long-term impact or potential associated with using a syntax-directed keyboard extension in writing programs on a mobile device [39]. This extension had already been shown to be reasonably effective and accurate. However, the design process of this extension was largely driven by an iterative conversation with programmers, that while well-grounded, might have been biased by what developers thought would be most useful/effective rather than what would prove to be most efficient from a statistical perspective. The original design was developed with the cognitive dimensions in mind [32]. We introduced enhancements upon the existing keyboard without violating these cognitive dimensions. We improved the Syntax-Directed-Keyboard extension by leveraging a statistical analysis of general purpose Java source code, and the terms most frequently used. Users' feedback of the original keyboard were also used to drive the

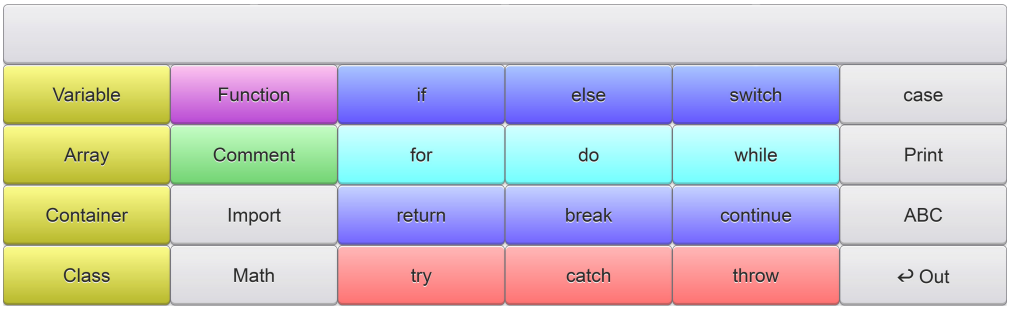

Figure 3.2: The original keyboard extension for Java program input. The extension serves as the top-level keyboard and provides entry to the standard soft keyboard when necessary.

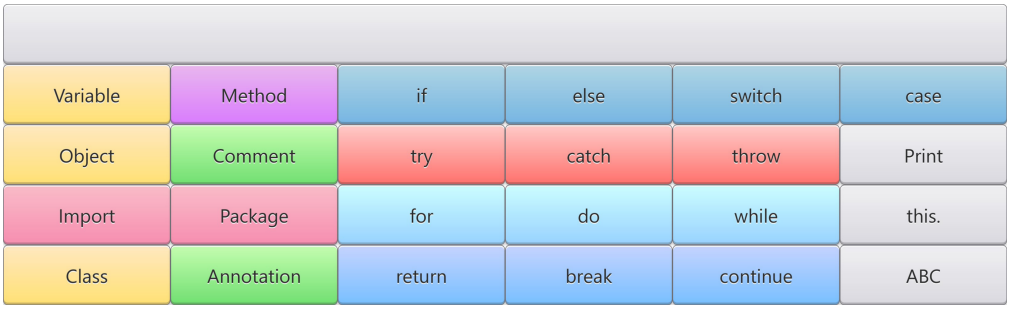

Figure 3.3: The revised keyboard extension for Java program input.

new modifications. We first rearranged a few key locations based on their statistical usage without breaking their functional grouping. In addition, we added more options to the options area and included symbols and data types that are frequently used. These enhancements to the keyboard design are intended to reduce keystrokes and speed Java source code input.

In this section, we describe in detail the statistical analysis and the process that led to the new design and highlight differences between the new and original design.

3.4.1 Java Statistics

Our redesign began with a statistical analysis of the way Java is used in the real world. Our goal was to better understand the usage frequency of Java language constructs so that we could present the most important constructs at the top level of the keyboard. To ensure that our design supports input of typical Java programs, we gathered and analyzed 10,968 Java projects, modeling the recommendations in [51]. Next, we downloaded all the Java files in these projects and used the JavaParser [52] to obtain the abstract syntax tree (AST) for each file. From the AST, we calculated the use statistics of Java primitives and constructs.

Finally, we took this information and modified the design developed by Almusaly et al., removing less frequently used keys, and adding more frequently used keys. These keys were then arranged from the most-used keys (top rows) to the least used keys (bottom rows), while retaining the semantic clustering in the original design. Table 3.1 shows the summary statistics of expressions used in our sample.

Some of the changes we made based directly on the use statistics. For example, the "*try*", "*catch*", and "*throw*" statements are more common than the "*for*", "*do*", and "*while*" statements. For this reason, we switched the location of these elements. However, the "*return*" statement has a higher frequency than all the looping and error handling statements combined. We did not change its location to maintain the grouping of the "*return*", "*break*", and "*continue*" statements. The "Array" and "Container" keys in the previous keyboard were replaced with the

Java Construct	Count per File	Task
Variable Declaration	9.23	7
Method Declaration	7.98	8
Parameter	8.46	7
Import Declaration	7.48	7
Return Statement	5.86	6
If Statement	5.83	6
Object Creation Expression	5.56	5
Field Declaration	4.02	4
This Expression	3.18	3
Line Comment	2.95	3
Class or Interface Declaration	1.21	1
Package Declaration	0.99	1
Switch Entry	0.89	0
Constructor Declaration	0.87	1
Catch Clause	0.84	1
Try Statement	0.81	1
Throw Statement	0.77	1
For Statement	0.60	1
While Statement	0.25	0
Switch Statement	0.14	0
Do Statement	0.03	0
Annotation Declaration	0.01	0

Table 3.1: Java source codes statistics. The second column shows the number of times each construct appeared, on average, over the examined source code files. The third column shows the number of times the construct appeared in the sample Java source code task used in our evaluation study.

"Object" key because the Object declaration is very common in Java source code and therefore needs its own key to reduce keystrokes. We removed the "Math" key because it is a function call that is not used often. Users can input math statement using the "ABC" key to open the standard QWERTY soft keyboard.

As can be seen when comparing Figure 3.2 to Figure 3.3, the changes were

relatively minor, though we hope impactful. We did not perform an evaluation of which keyboard performed best. The reason for performing these optimizations was a desire to avoid annoying our subjects as the novelty of using this system wore off, as well as a desire to create the most optimal conditions for our users.

3.4.2 Interaction Design Updates

The original design, while effective, had several shortcomings [39]. For example, when the "option area" (the top row of the keyboard) displays types (after pressing variable or function, for example), the rest of the keyboard is unused, showing the static top level keys that are disabled until a type key has been selected. The programmer must select the "Custom Type" key to input a type that is not in the option area as it is shown in Figure 3.4. By showing the QWERTY keyboard in place of the disabled static keys, we save a keystroke every time the user wishes to input a custom type that is not in the list. Figure 3.5 shows the revised design that allows for selecting an existing type or creating a new type. In addition, this design contains more primitive types to choose from that are displayed from most common (*int*) to least common (*double*).

Some symbols and operators tend to appear more frequently than others in general-purpose programming. For example, in conditional statements, relational, equality, and logical operators are far more common than the rest of the other operators. Based on our statistical analysis of Java programs, we added the most common operators in the top row "option area" as a shortcut to reduce the number

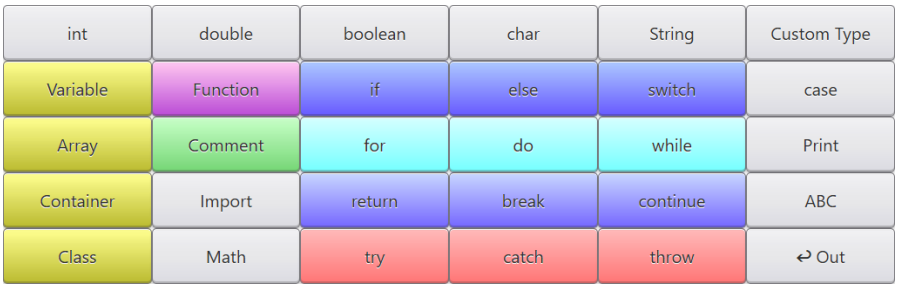

Figure 3.4: The original "types" keyboard view.

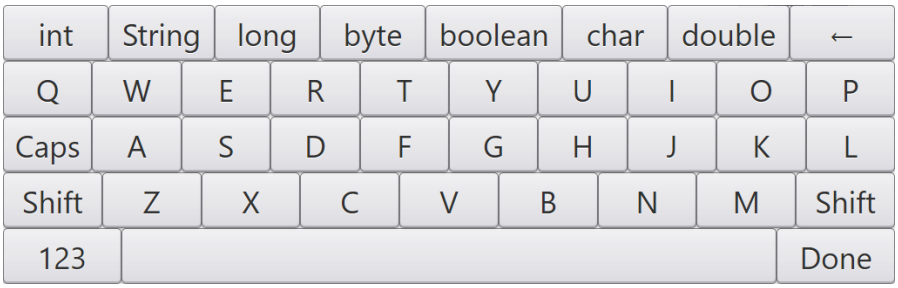

Figure 3.5: The new "types" keyboard view.

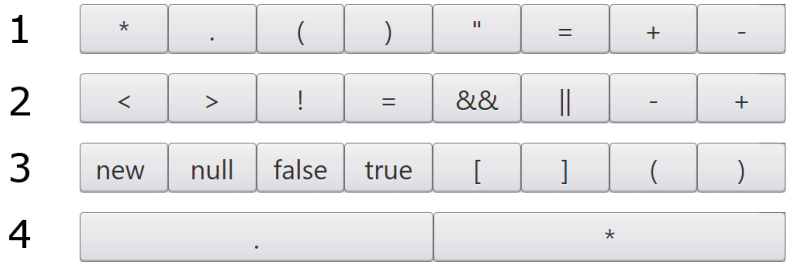

Figure 3.6: 1 default keys, 2 conditional keys, 3 assignment keys, and 4 import keys.

of keystrokes. We display a different operator set depending on the current statement. This approach more consistently eliminates the need to switch keyboards to input an operator. Table 3.2 shows the frequency of the symbols used in Java source code. Figure 3.6, shows the options area that corresponds to the default set,

Symbol	Per File	Symbol	Per File
*	13.9	:	1.2
)	12.9	+	1.2
(	12.9	\	1.0
;	10.1	[	0.8
/	6.9	]	0.8
=	5.9	’	0.7
,	5.8	!	0.6
”	5.0	&	0.5
}	3.7	$	0.4
{	3.7	—	0.3
_	3.5	#	0.3
-	3.0	?	0.2
.	1.8	%	0.1
¿	1.6	ˆ	0.01
¡	1.6	‘	0.01
@	1.6	˜	0.01

Table 3.2: The frequency of symbols in Java source code.

conditional keys, assignment keys, and import keys respectively. Notice that not all the symbols or operators are presented to the user such as the curly brackets and the semicolon. These symbols are inserted by the keyboard automatically in the correct syntactical location.

3.5 Methodology

In this section, we discuss key performance measures for text input and keyboard design, and how we went about evaluating the long-term impact of our prototype.

3.5.1 Evaluating Typing Performance

Among the most important metrics when evaluating a (soft) keyboard are text input accuracy, efficiency, and speed. How prone a user is to committing typing errors is the most commonly used definition of accuracy [25]. Substitution, intrusion, omission, and transposition are the most common typing errors.

There are several ways to measure accuracy, including, but not limited to, the *Minimum String Distance* (MSD) error rate, and the *Total Error Rate* (TER) [26]. MSD is a measure of the minimum number of primitives (insertions, deletions, or substitutions) to transform one string into the other. TER is a measure of the total number of errors in the final typed text, including corrections made during the typing of the final text. We use the TER measure in this paper because it overcomes the weaknesses of the other measures and reflects all errors committed by a participant [26]. Keystrokes are categorized into four classes within an input stream: *Correct* (C), *Incorrect Fixed* (IF), *Fixes* (F), and *Incorrect and Not Fixed* (INF) [26]. We use all four keystroke classes to compute the TER.

To measure how effective our keyboard is in reducing the number of keystrokes required to enter a single character, we used the *Key Strokes Per Character* (KSPC) measure [27]. The KSPC on a standard physical QWERTY keyboard is approximately 1.00 [27]. However, it has been shown to be as high as 1.21 when accounting for the need to correct errors [26].

To measure user input speed, we computed the *Words Per Minute* (WPM) typing speed, where a word is assumed to consist of on average five characters. The

WPM metric has been used to compare various hard and soft keyboard designs [7, 29]. We measure the input speed in WPM using the following equation:

$$WPM = \left(\frac{Number\_Of\_Characters}{Total\_Time}\right) \times \left(\frac{60}{5}\right) \tag{3.1}$$

Total_Time is the time it takes to input the source code. The *60* in equation 3.1 is to convert the seconds to minutes. While the *5* is to convert the *Number_Of_Characters* to the number of words. This number is selected by assuming that the average number of characters in a word is five.

3.5.2 Study Design

We now describe a formal user study designed to evaluate the use of our keyboard extension over an extended period.

We used iPad 4's for our study and implemented the soft keyboard extension in JavaScript. The iPad has a 9.7-inch diagonal screen with 1536 x 2048 pixels. The pixel density is 264 pixels per inch (PPI). Next, we created a sample Java program that met the "average" statistics for Java programs (see Table: 3.1). We decided to use this program as a stand-in for a generic Java program, which our subjects would be asked to key in using our prototype. This was done to remove confounding variables associated with having to design a program independently and on the spot. We designed the source code so the average user would be able to key it in less than 30 minutes (to avoid input fatigue). The auto-completion

capability was disabled to measure the raw performance of the keyboard like the original keyboard study design [39].

We asked our participants to key this program eight times over an eight-session span. This presents an ideal situation, where the developers know what they want to write, and over time potentially become more familiar with their task. This would allow us to set the stage for optimal learning over a short period, while controlling for variables such as programming experience and prowess (while more experienced developers might be better at remembering or anticipating the next commands, less experienced developers would not be at too much of a disadvantage). Our keyboard application was instrumented to log data needed for the KSPC, WPM, and TER measures.

3.5.3 Participants

The study participants consisted of 10 students (2 female and 8 male) all with previous Java programming experience. All the participants volunteered for the study in response to an email message circulated to the students in the computer science department at [omitted for anonymous review]. *No participant* previously used a tablet device to write source code.

3.5.4 Experimental Procedure

We asked each participant to enter the same Java program eight times over the course of eight sessions. We measured the KSPC, WPM, and TER of all participants for each of the eight sessions. We measured and report omission, substitution, insertion, and spacing errors with the unified error metric, TER [26].

We ran the study in a lab setting, each participant working individually. Only in the first session, participants were given a 10-minute tutorial on how to use the keyboard extension and then allowed 5 minutes to practice with it. We encouraged the participants to ask any questions that they might have during the study. Participants then carried out the assigned task using our keyboard. After each session, we asked the participants to complete the NASA Task Load Index (NASA-TLX) questionnaire to assess subjective mental workload [34]. At the end of the study, we asked participants to complete a questionnaire about their experience with the keyboard.

The longest time a participant took to finish the first session was 31 minutes. The average session time was a proximately 17 minutes. Participants spent a total of 2 hours and 15 minutes on average using the keyboard. The eight sessions spanned 10 to 14 days across all participants, with no participant doing more than one session per day.

3.6 Results

We present the speed, accuracy, and efficiency of participants using our keyboard in the following section. In the interest of replicating the results of Almusaly et al. [39] (though our keyboards differed slightly), we also report these measures for the first session separately. We conclude the section with the NASA task load index measures as well as questionnaire results.

3.6.1 Words Per Minute

A one-way repeated measures ANOVA was conducted to compare the effect of (IV) our keyboard's usage time on (DV) the speed of input (WPM) during the 8-session span. There was a statistically significant effect of usage on the input speed (WPM), $F(7, 63) = 83.74$, $p<.001$. The participants' average typing speed in the first session was 15.87 WPM ($SD = 2.82$) and gradually increased to 30.29 WPM ($SD = 5.59$) by the eighth and final session. This represents a 91% increase in average typing speed over 8 sessions (see Figure 3.7). There was no evidence to show that participants had reached their top speed by the end of the experiment. This shows the importance of giving participants time to acclimate and thoroughly learn how to use new input systems.

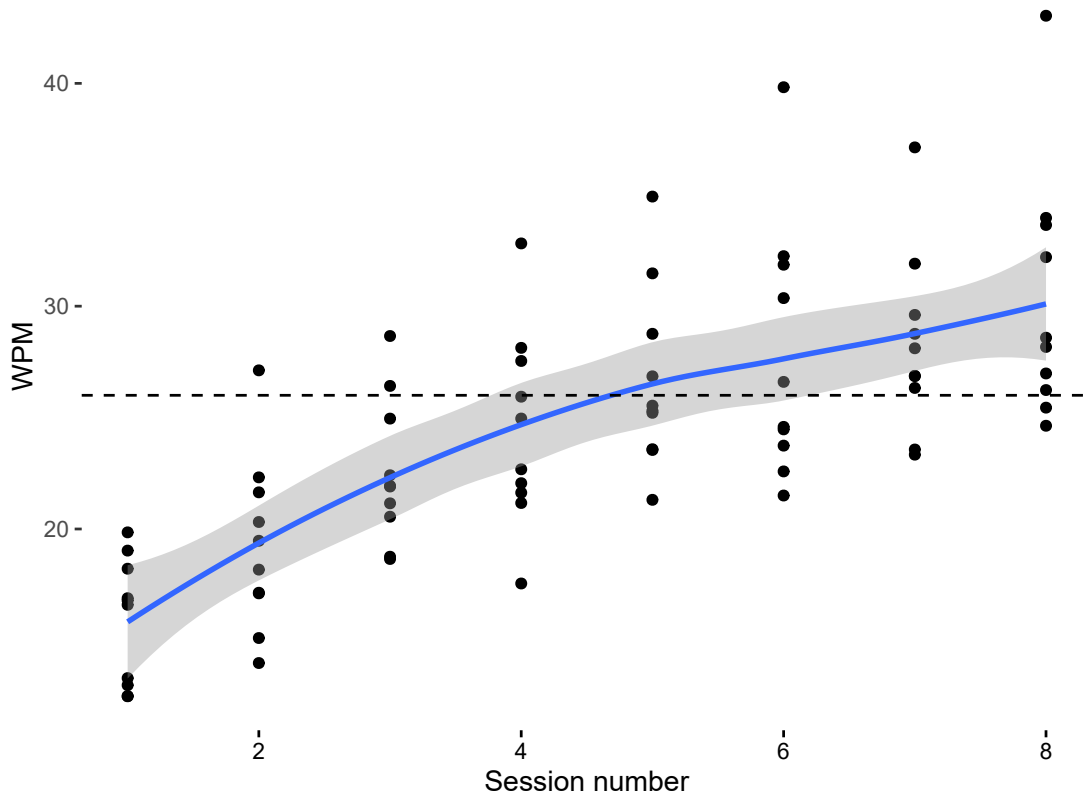

Figure 3.7: The input speed in WPM over the eight sessions. Each dot represents a participant's typing speed for a given session. The gray area is the Confidence Interval of the Loess smoother. The dashed line represents the theoretical speed of an expert QWERTY keyboard typist

3.6.2 Total Error Rate

A one-way repeated measures ANOVA was conducted to compare the effect of (IV) the enhanced keyboard's usage time on (DV) the input accuracy (TER) during 8 session span. There was no statistical significant effect of usage on the input accuracy (TER), $F(7, 63) = 1.568$, $p = 0.162$. The total error rate in the

first session was 3.69% ($SD = 1.71$). This rate was reduced to 2.61% ($SD = 1.32$) by the eighth session; a decrease of 29%. This decrease in total error rate is shown in Figure 3.8.

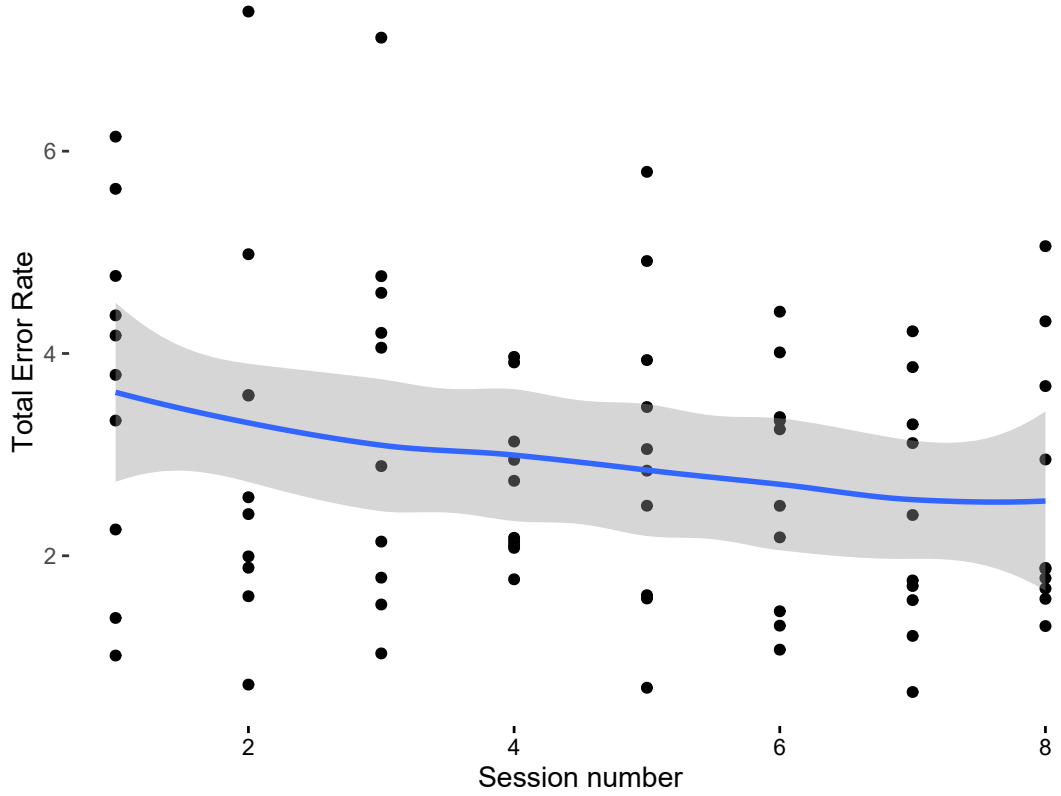

Figure 3.8: The total error rate over the eight sessions.

Because typing errors are somewhat misleading due to the amount of text inserted automatically (and therefore correctly) by the syntax-directed keyboard, we also wanted to look at how often users simply chose the wrong key on the keyboard extension. On average, participants pressed the wrong key in the first

session 20.3 times, and this number decreased to 5.9 times by the final session.

3.6.3 Key Strokes Per Character

A one-way repeated measures ANOVA was conducted to compare the effect of (IV) the enhanced keyboard's usage time on (DV) the input efficiency (KSPC) during 8 session span. There was a statistically significant effect of usage on the input efficiency (KSPC), $F(7, 63) = 4.436$, $p<.0001$. The average KSPC is perhaps a more telling measure than the TER, as it indicates how many keys the user must touch to enter the code. The KSPC for the first session was 0.88. ($SD = 0.05$) By the end of the eighth session, participants reached an average KSPC of 0.81 ($SD = 0.03$). As expected, KSPC reduces slightly over time as participants make fewer errors as measured by TER. Figure 3.9 shows the changes in participants' KSPC over the eight sessions.

3.6.4 NASA-TLX

We used the NASA-TLX measure to obtain participants' perception of workload when using the syntax-directed-keyboard and report the Raw TLX. As seen in Figure 3.10, the average perceived workload of the participants decreased over time for all facets of the measure: mental, physical, temporal, performance, effort and frustration.

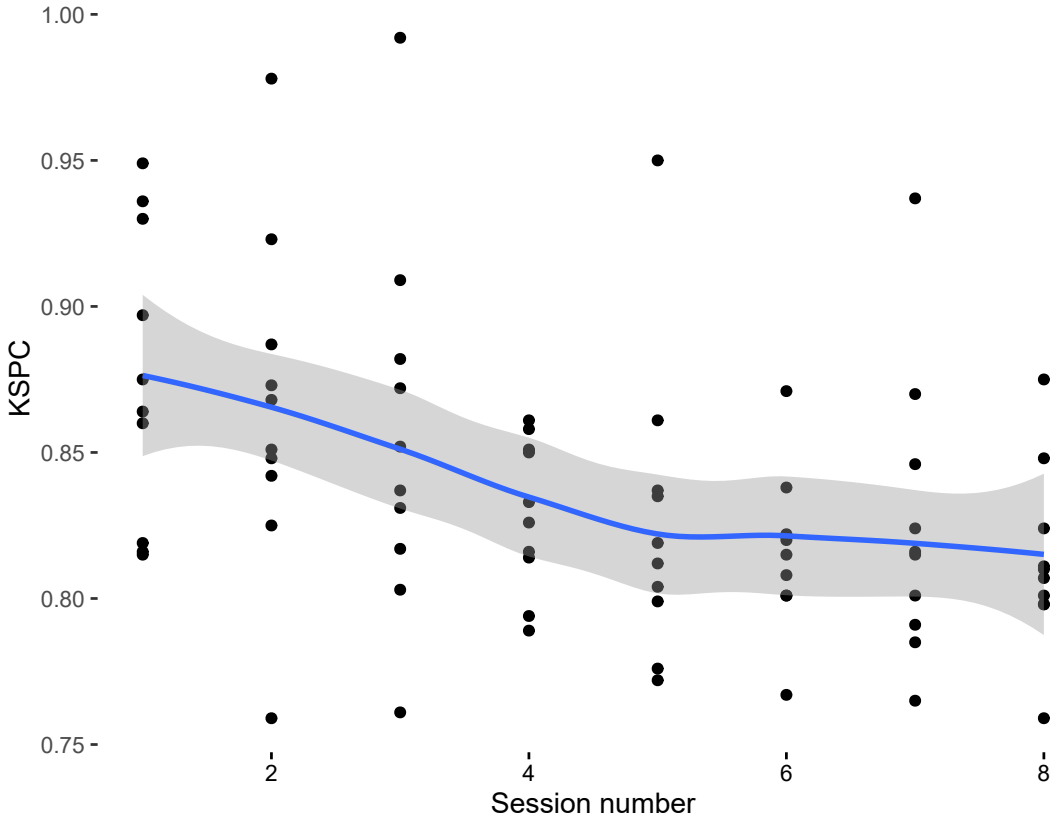

Figure 3.9: Input efficiency in KSPC over the eight sessions.

	First session	Last session	Improvement
Mental	46	27	41%
Physical	48	25	48%
Temporal	43	25	42%
Performance	78	71	9%
Effort	51	29	43%
Frustration	28	19	32%

Table 3.3: The average improvement in the perceived workload.

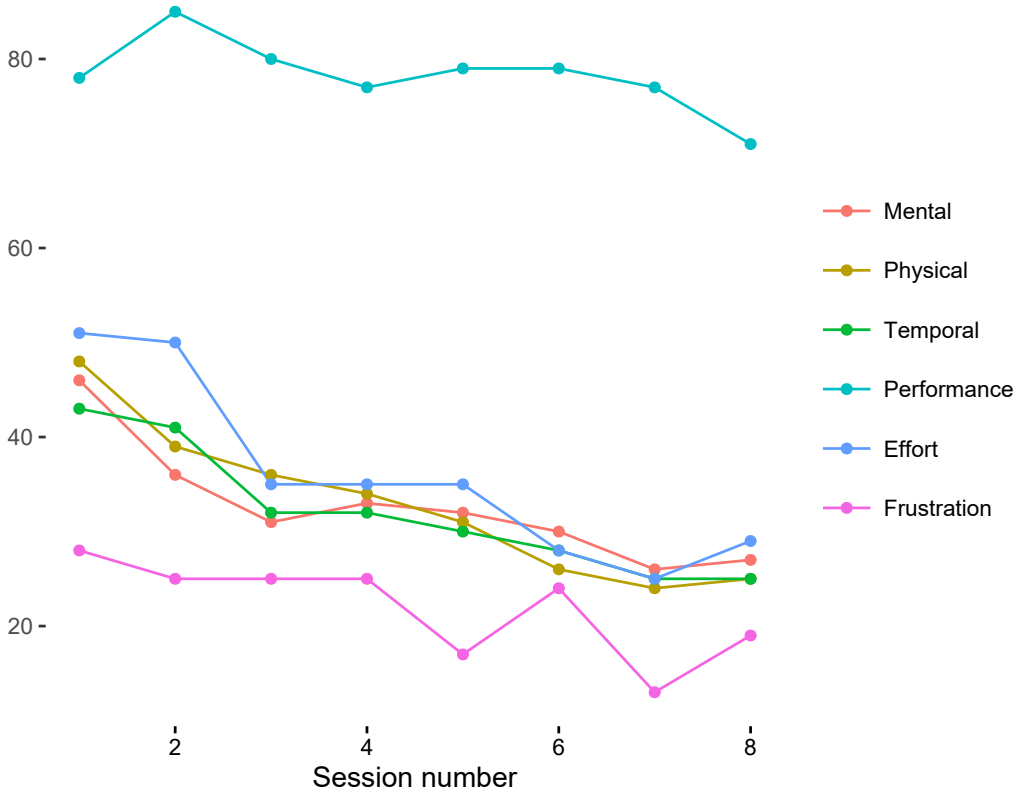

Figure 3.10: The Raw NASA-TLX results over the eight sessions. Each line represents one of the NASA-TLX questions, and each point represents the average response for all participants. Questions are out on a 100-point scale, with higher scores representing higher workloads.

3.6.5 Participant Feedback

After the eighth and final session, participants completed a questionnaire about their overall experience with the keyboard. All participants indicated that they were between neutral and very positive about using our keyboard to write Java source code. None of the participants said they were unlikely to use the syntax-

directed keyboard. Most of the participants found the keyboard to be helpful or very helpful for writing source code. All but one participant thought the keyboard was easy to use. Figure 3.11 shows each question and a box-plot of the participants' average answers to each question.

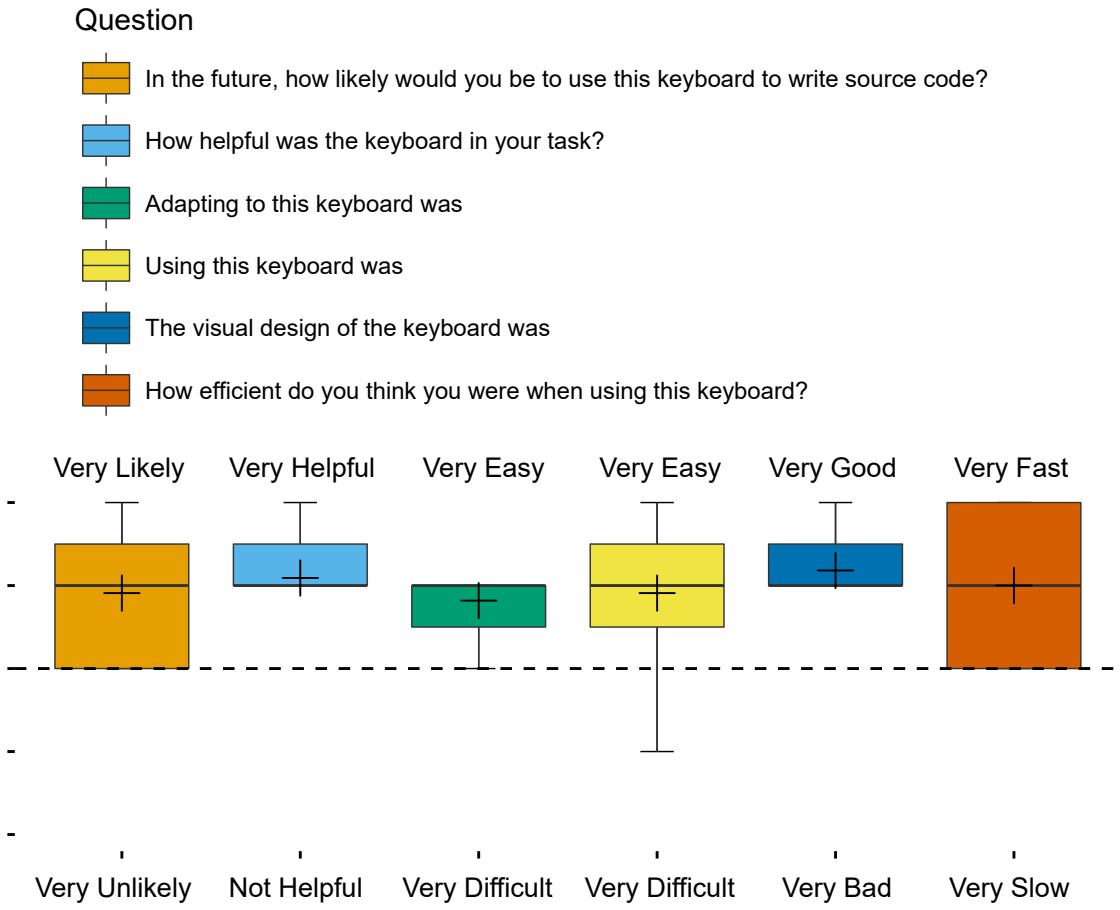

Figure 3.11: The results of the post study questionnaire. Each column represents a question. The Tukey boxplot shows the average response on a 5-point scale.

3.7 Discussion

In this section, we discuss our results with specific attention paid to how participant performance changed over the course of the experiment (8 session) and how the updated keyboard design compares to the original keyboard design of Almusaly et al. [39] after a single session. However, it is important to know that our keyboard was evaluated using a different source code that matches the collected statistics.

3.7.1 Efficiency

The source code keyed in our study task can in theory be input using our syntax-directed keyboard with a KSPC of 0.723 when no mistakes are made. While none of our participants achieved the optimal input rate, their KSPC clearly improves over the 8 sessions, with an average KSPC of 0.81 at the end of the study. This KSPC includes correcting errors while keying the text. In Figure 3.9, we can also see that the performance rate levels off after about 5 sessions, suggesting that users can learn to use a custom keyboard layout effectively after a relatively short use duration.

Assuming no errors, the lowest KSPC of an expert QWERTY physical keyboard user while keying the same source code is calculated to be 1.32. Our participants' average KSPC of 0.81 represents a 35.5% improvement over the best possible KSPC with a physical QWERTY keyboard, even when correcting errors while typing. If no errors are made, our KSPC would be further reduced to an optimal rate that

is 44.55% better than the optimal QWERTY physical keyboard input.

To replicate the results of Almusaly et al. [39], we analyzed the performance of users with our updated design after using it for only one session. Our updated design appears to perform slightly better than the keyboard design of [39]. Looking only at the first session, the average efficiency in our first session was 0.88 KSPC compared to 0.97 KSPC for the previous design. This is a 9.3 % improvement over the original design. Thus, our updated design produced better results, and users still have significant room for improvement over the 8 sessions of the study. While we are happy with this, it is likely that there is still room for improvement, and further study should be devoted to methods for optimizing these types of keyboards.

3.7.2 Accuracy

From Figure 3.8, we can see a steady decrease in the error rate over the course of the 8 sessions. In addition, the average number of times that participants pressed the wrong key on the keyboard also decreased over the 8 sessions. We expect a low error rate because the keyboard extension inserts keywords and many of the symbols for the user and typical Java source code contains many symbols and keywords. This limits the room for error by constraining the input space.

Participant performance using our design also resulted in better accuracy than was reported by Almusaly et al. [39]. Their total error rate was 4.89%, while we saw an average error rate of 3.64% in the first session; a 25.6 % reduction in the

total error rate compared to the previous keyboard design. While these numbers are compelling and encouraging, we caution the reader that we do not have a measure of statistical significance.

3.7.3 Speed

With increased use and familiarization with the keyboard layout and hierarchy, we would expect input speed to increase as well. The average input speed of the eighth session (the highest performance session) was 30.3 WPM. This is comparable to an expert user's theoretical speed on our enhanced keyboard extension as computed using Fitt's Law (42 WPM). The theoretical speed of an expert QWERTY keyboard user, however, is 26 WPM when inputting the same source code. Our participants' average input speed was therefore 16.5% faster than the theoretical top speed of an expert QWERTY physical keyboard user. This is of course an artifact of extensive templating, but does show that such keyboard overlays, though often unfamiliar and requiring some adaptation, quickly pay off.

In the first session, our participants' average input speed was 15.87 WPM, which is 48.6% faster than reported in Almusaly et al. [39] (10.68 WPM).

3.7.4 User Experience

Participants' perceived mental, physical and temporal demands were all somewhat high at the beginning at the study, which is to be expected given the novelty

of the keyboard and task. We would also expect high perceived effort and frustration level scores, as shown in Figure 3.10. Each of these measures, however, steadily declines over the eight sessions as participants became more comfortable with the keyboard design.

Perceived performance, however was somewhat high in the beginning, and contrary to expectations, it declined over time. This means that despite improvements in efficiency, speed, and accuracy, and improvements in all other perceived measures, participants did not believe they were getting faster. In fact, they believed they were getting slower. Perhaps this is due to an equal rise in expectations of oneself as users become more familiar with the keyboard.

Overall, participants' experiences as measured by the post study questionnaire were positive. When combined with the TLX results, the prospect of users adopting domain-specific keyboards, specifically for programming, is promising.

3.7.5 Threats to Validity

Our study is not free from threats to its validity, though we tried to minimize and address them. Our sample size of 10 participants was small, therefore it may not generalize to a much larger sample size, and we were not able to perform tests for statistical significance. The results are applicable to the task that we gave to our participants, however it is not clear how results would generalize to new source code samples. In addition, touchscreen devices come in many shapes and sizes - our study was carried out on one device size only. Finally, while we tested

the keyboard using common Java constructs, not all source code will necessarily contain the same statistics. and performance will vary depending on the frequency of the Java constructs in the source code. This is true, however, for all keyboard designs.

3.8 Conclusions and Future Work

In this paper, we have presented an improved design of the syntax-directed keyboard extension of Almusaly et al. [39], validated their original results regarding the benefits of using a syntax-directed keyboard extension for coding on a mobile device when compared to a regular QWERTY keyboard. We also demonstrated that over time, user performance on such keyboard extensions improve markedly. This suggests that the use of custom-designed keyboard extensions for domain specific situations may be more beneficial than the use of only the standard QWERTY keyboard on touchscreen devices. It also points to the need to adopting more longitudinal formats in the study and evaluation of novel input methods.

While we presented some improvements to the keyboard originally proposed by Almusaly et al. [39], we make no claims to having developed the definitive keyboard extension for Java. There are likely still significant improvements to be made. The syntax-directed keyboard extension can also be improved with additional features such as auto-completion of variable names and function calls. This is left as future work.

Finally, the original syntax-directed keyboard extension was created ad-hoc,

based on Java source code statistics and the language grammar. An interesting avenue of future work is the automatic creation of syntax-directed keyboards for additional programming languages by combining an optimization method with the statistical analysis and the language grammar.

Chapter 4 : Evaluation of A Visual Programming Keyboard on Touchscreen Devices

4.1 Abstract

Block-based programming languages are used by millions of people around the world. Blockly is a popular JavaScript library for creating visual block programming editors. To input a block, users use a drag-and-drop input style. However, there are some limitations to this input style. We introduce a custom soft keyboard to input Blockly programs. This keyboard allows inputting, changing or editing blocks with a single touch. We evaluated the keyboard's speed, the number of touches, and errors while inputting a Blockly program and compared it to the drag-and-drop method. Our keyboard reduces the input errors by 68.37% and the keystrokes by 47.97%. Moreover, it increases the input speed by 71.26% when compared to the drag-and-drop. The keyboard users perceived it to be physically less demanding with less effort than the drag-and-drop method. Moreover, participants rated the drag-and-drop method to have a higher frustration level. The Blockly keyboard was the preferred input method.

4.2 Introduction

Computer science jobs are increasing. More than 50% of all science, technology, engineering, and math (STEM) jobs are projected to be in computer science-related fields by the year 2018 [53]. The Computer Science for All (CS for All) initiative is aimed to enable all American students at K-12 schools to learn computer science. This includes teaching computational thinking skills. However, there are many programming languages. Blocks programming environments are being used by millions of people of all ages and backgrounds. They offer many advantages to the beginner programmer. They eliminate syntax issues because they represent program syntax trees as compositions of visual blocks. Block-based programming is a popular way to teach programming concepts. Alice, Scratch, and Blockly are examples of such block-based languages [15, 54, 55]. Their visual characteristic benefits them because recognition is easier than recall. Despite their advantages, there have their drawbacks. One of these drawbacks is the time and the number of blocks it takes to compose a program in the block-based interface compared to the text-based alternative [56]. Dragging blocks from a toolbox is slower than typing. Some attempts were made to blur the line between blocks and text programming [57, 58, 59, 60]. Few of them did that by allowing blocks to be typed using the keyboard. However, these researches did not focus on blocks input performance. They are trying to ease the transition from blocks to text-based programming while we are trying to ease the input of existing block-based language, especially on touchscreen devices.

Most block-based programming environments rely on the mouse as the primary means of inputting blocks. Using touchscreen devices, blocks can only be input by drag-and-drop. However, using the drag-and-drop input method has its disadvantages when compared to the point-and-click input method. The point-and-click input method is faster, more accurate, and it is preferred over the drag-and-drop for adults and children alike [61, 62]. Drag-and-drop requires careful manipulation of blocks to insert them in the right place. The careful manipulation requirement adds physical and cognitive demands. These demands affect users negatively, especially people with motor disabilities or children. The drag-and-drop interaction also changes as the canvas gets zoomed in or out. A zoomed-out canvas makes connecting blocks more difficult as the blocks' connectors become smaller, making it more difficult to aim for. This drag-and-drop entry method does not take advantage of all fingers for inputting the blocks. In addition, blocks have many options that can be changed. However, these options can be changed using small icons. These factors make block entry a slow and difficult process. Furthermore, these factors might lead to frustrated users, which might affect the performance and adoption of visual languages. That is why we are interested in addressing these issues.

We created a custom soft keyboard to input blocks on touchscreen devices. This keyboard was made specifically as a drag-and-drop alternative. The keyboard enables programmers to input blocks using a point-and-click interaction style. We want to reduce the time required to input programs using the keyboard. In addition, a point-and-click input method like the keyboard should decrease the errors.

Moreover, a faster and more accurate way to input blocks could reduce physical and temporal demands thus reducing frustrations. The keyboard also enables interacting with the blocks of the same base regardless of the canvas zooming because the keys' sizes do not change. Finally, a faster, more accurate, and more efficient input method will make blocks entry more accessible to a wider range of users and more appealing.

We chose Blockly as our keyboard target because it is a popular library for creating block programming editors [55]. Section 4.3 presents the related work. Then, Section 4.4 presents our visual programming Keyboard and its design. After that, section 4.5 presents our user study design. Section 4.6 presents the user study results and section 4.7 discusses the results. Lastly, section 4.8 mentions the conclusions and the future work.

4.3 Related Work

4.3.1 Blocks-based Programming

Blocks-based languages are gradually used to introduce novices to programming. In these languages, programs are constructed by connecting blocks via connectors. Alice, Scratch, and Blockly are examples of such block-based languages and environments [15, 54, 55]. These visual programming languages and environments aim to remove the syntax barrier of textual programming languages. They are engaging millions of kids with programming through drag-and-drop [63].

Moreover, they were designed explicitly with learners in mind [64]. However, our keyboard is different because it is trying to reduce the gap between novice and experts users. It does that with better input efficiency by reducing the number of touches required to input. In addition, increasing the input speed and reducing the errors make the keyboard a better alternative. Novice users might benefit from the reduced physical demand while experts might benefit from the increased speed. Novice and experts will benefit from the reduced errors.

4.3.2 Keyboard Designs

Keyboards are designed to minimize discomfort, speed input, or both. Soft keyboards are no exception. Most keyboards are designed and evaluated for text entry [4]. The design of the blocks keyboard is similar to the syntax-directed-keyboard extension [39]. Instead of using the Java syntax, the blocks keyboard uses the Blockly rules. However, designing a keyboard for blocks input is different. Moreover, blocks are input via dragging which is vastly different from keyboards. In section 4.4.2, we discuss in detail the design of the keyboard.

4.3.3 Input Performance Metrics

There are many measures to evaluate an input method. We used three common input measures for evaluation keyboards' performance. Namely, the input accuracy, efficiency, and speed.

4.3.3.1 Accuracy

Different keyboards have different accuracy. The more prone a keyboard to errors the less accurate it is. Errors will be misspellings or typos when using a text entry keyboard. However, the inputted elements in visual programming are blocks. Thus, error types will be different. We defined errors to be the actions that the user did not intend to do when inputting blocks. These actions are misplacing a block, failing to connect a block to another, selecting the wrong field, and inputting the wrong block. Misplacing a block occurs when the participant inputs a block and connects it to the wrong block. When the participant inputs a block far from another block's connector, the inserted block will not be connected. This is counted as failing to connect a block to another error. Selecting the wrong option occurs when a participant did not choose a block's field correctly. When a participant inputs the wrong block, it is counted as inputting the wrong block error. The error rate is the sum of all the error types.

4.3.3.2 Efficiency

Keystrokes per character (KSPC) is frequently used characteristic of text entry methods [65]. For a given text entry method, it measures the number of keystrokes required, on average, to generate a character of text. However, blocks are not characters. We defined a similar block-based characteristic for block entry methods. The keystrokes per block (KSPB) is a measurement of the number of keystrokes required, on average, to generate a block. Thus, a block entry technique with lower

KSPB is more efficient.

4.3.3.3 Speed

To evaluate the speed of text entry techniques, words per minute (WPM) is used. WPM, as the name implies, is the average number of words that can be inputted in a minute by a text entry method, assuming 5 letters per word. We defined a related measure for evaluating the speed of block entry methods. Blocks per Minute (BPM), equation 4.1, is the average number of blocks that can be inputted in one minute by a block entry technique. A faster entry method is the one with a higher BPM.

$$BPM = \frac{NumberOfBlocks}{Time} \tag{4.1}$$

4.4 The Keyboard Design

Section 4.4.1 explains how the keyboard works while section 4.4.2 describes the main iterations that led to the keyboard's final version. In section 4.4.3, some user-interface design principles that helped to fine-tune the keyboard design are mentioned.

4.4.1 How Does It Work

The keyboard works like text entry keyboards, however, instead of inputting letters it inputs blocks. Each key inputs a block or changes a field. Version 3 in figure 4.1 shows the current layout of the keyboard. Once a block is inserted, the keyboard selects and highlights the first unoccupied input connector. Navigating the blocks through their connectors is available by using the arrow keys. When a connector is selected, it gets highlighted. The keyboard's keys will be enabled or disabled according to the highlighted connector. The "if" block, for example, does not allow numbers to be connected to its "condition" connector. Thus, the "Number" and "Arithmetic" keys will be disabled if the condition connector is selected. The grayed-out keys in figure 4.1 are disabled. The top row will list the options for the current block, the block with a highlighted connector. As a user moves through the blocks, the top row updates the list of options automatically. For example, the right side of figure 4.2 shows how the top row displays the options for the "Compare" block.

4.4.2 Block Frequency

There are many ways to layout the keys in the keyboard. Just like many keyboard designs that utilize the letter or word frequencies to layout their keys, we want to utilize the blocks' frequencies. To do this, we looked for Blockly program sources. Code Studio, a website offering online courses created by Code.org, is used by millions of students [63]. It relies heavily on Blockly to teach program-

ming concepts. Therefore, we chose it as a reference for Blockly programs. We counted the frequency of each block that was asked to be input in all the offered activities. However, if the activity asks the users to input less than 10 blocks we did not include that activity in the statistics. We chose not to include such activities because they have fewer blocks that do not represent a common block-base program. Commonly, an activity with few blocks just teaches how to input blocks rather than teaching programming concepts or problem solving. This left us with 47 Blockly programs from courses 2,3, and 4. We found that the average input task consists of 19 blocks. Table 4.1 shows the frequency of block types. The Code Studio activities did not ask the students to input all the block types in Blockly as can be noticed from the table.

Table 4.1: The frequency of inputted blocks.

Block Type	**Frequency**
function call	34.1%
number	23.7%
get variable	14.4%
repeat time	9.2%
arithmetic	5.5%
set variable	5.4%
for loop	3.2%
if	1.6%
function define	1.3%
color with	0.6%
repeat while	0.1%

The keyboard underwent many iterations and the blocks' statistics served as a guide for placing the keys throughout these iterations. First, we placed block types as keys from most frequent (right of the top row) to least frequent (left

of the bottom row). Then, we placed block categories, that contain a list of blocks with similar functionality, as keys from most frequent to least frequent after the block keys based on the sum of their block usage. By this stage, we had version 1 of the keyboard which is shown in figure 4.1. However, the block types were scattered across the keyboard. We grouped the keys by swapping the next frequent key of the same category for each key with the key underneath it. For example, "Arithmetic" is the next frequent key after the "Number" key from the same category. We swapped "Arithmetic" with "if". After repeating the same process for the rest of the keys we had the second version. For the second version, we manually moved the "repeat while" key underneath its category because it is the least frequent block type to maintain the grouping. After placing the blocks and their categories, three keys were not assigned. We choose to assign them to "Text", "Boolean", and Compare blocks to enable access for other data type blocks and the comparison block. Finally, we listed the predefined functions in the first row because function call blocks have the highest frequency of all. The first row acts like a dynamic placeholder for blocks and their options. It lists the options of the last inputted block alongside the predefined functions. For instance, when the "Arithmetic" block is inputted, the dynamic row lists all of its six options: $+$, $-$, $\times$, $\div$, and $\wedge$.

Figure 4.1: The main versions that our keyboard passed through.

4.4.3 User-Interface Design Principles

While the keyboard was evolving, we adhered to the general user-interface design principles listed by Wickens et al [66]. These general design principles help to ensure the keyboard's ease of use and adaptability.

4.4.3.1 Make invisible things visible

Opposite to the drag-and-drop, the Blockly keyboard lists the block's options making them readily available with one touch. These options are hidden in a drop-down menu. Figure 4.2 shows how the keyboard displays the options for the "Compare" block as opposed to the hidden options of different blocks.

Figure 4.2: An example of how the keyboard makes the options more accessible.

4.4.3.2 Consistency and standards

Each block and its option are expressed as a key with the same height and width except the dynamic top row. The size of a block, yet, changes and its connectors move. In addition, keys have two actions, which are input a block or change an option. This is consistent throughout the keyboard. Furthermore, keys with the same functionality are grouped together.

4.4.3.3 Error prevention, recognition, and recovery

To prevent errors from occurring in the first place, our keyboard disables keys to prevent inputting the wrong block in the wrong place. Without the keyboard, users can connect the wrong blocks together. Blockly then disconnects the wrong blocks without prompting the user, which might be confusing.

4.4.3.4 Memory

The keyboard keys are named and colored which utilize see-and-point instead of remember-and-type. Our keyboard exposes the most common blocks instead of hiding them inside a toolbox and reveals their options. The users do not have to remember the locations of the common block inside the toolbox nor do they need to search for the options from their menu. This reduces the reliance on memory.

4.4.3.5 Flexibility and efficiency of use

The keys and the option keys act as shortcuts and accelerators. Our keyboard gives the users the option to speed up frequent actions by listing the most frequent inputted blocks, the Function call blocks. In addition, the dynamic top row gives shortcuts for the block's options.

4.4.3.6 Simplicity and aesthetic integrity

The keyboard's keys are aligned with uniform width and height to make them look good using a simple design. To make information appear in a natural order, the options and keys are presented from left to right based on their usage frequency.

4.5 User Study

The formal user study was designed to compare the input performance of the keyboard to the drag-and-drop method when inputting a Blockly program. We presented the participants with a letter sized paper that has a Blockly program printed in colors. The participants then were asked to copy the program with both input methods. We chose a copying task as opposed to a programming task to avoid the confounding factors of the cognitive aspects of programming. When the keyboard is shown, the drag-and-drop is disabled to measure the native keyboard performance. We used an iPad 4 for our user study. Both input methods were implemented in JavaScript. The same JavaScript code was used to measure and log the participant's interactions, time, and errors. The error types presented in section 4.3.3.1 were collected by the instrumented JavaScript code.

4.5.1 Task

To see how the input methods were going to perform in a common block-base program, we collected statistics from Code.org website. We asked the participants

to input the one Blockly program, which consisted of 29 blocks. In addition, we designed the task to conform to the collected statistics from Table 4.1. The input task consisted of 10 "function call" blocks, 7 "number" blocks, 4 "get variable" blocks, 3 "repeat" blocks, 2 "arithmetic" blocks, 2 "set variable" blocks, and 1 "for" block. Figure 4.3 shows the input task.

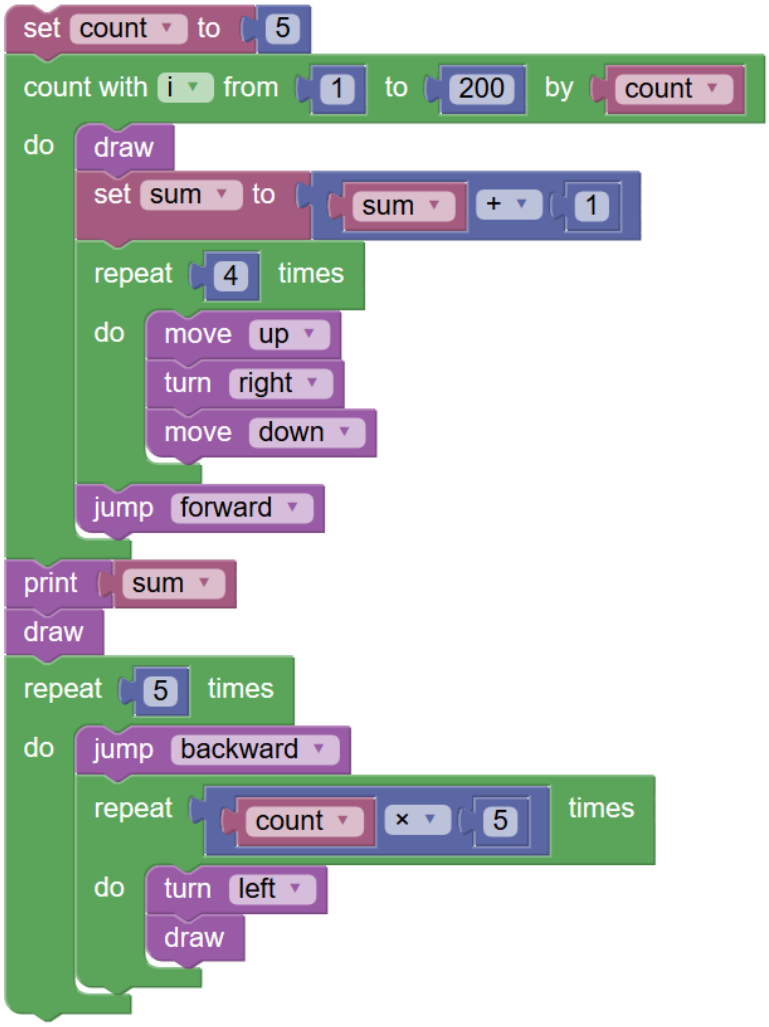

Figure 4.3: The Blockly input task.

4.5.2 Participants

The study participants consisted of 14 male and two female students. All participants volunteered for the study in response to an email message circulated to the students in the computer science department at Oregon State University. Two participants were graduate students and 14 were undergraduate students. All participants had never used Blockly. Nine of the participants were not familiar with block-based programming. None of the participants used a tablet device to input any block-based program. 10 participants reported having a tablet device. The participants were compensated for participating in the study.

4.5.3 Experimental Design and Procedure

To study the difference between the two input methods, we used a within-subjects design with repeated measures. The independent variable was the input method used to complete the task and the study consisted of two treatments: the drag-and-drop and the keyboard. We asked each participant to enter the Blockly program using each input method. We counterbalanced the order of the treatments by dividing the subjects into two groups. One group started with the drag-and-drop, followed by the keyboard, and the other started with the keyboard followed by the drag-and-drop. The dependent variables were the time, errors, and the number of touches. We measured these variables for each input method, independently.

We ran the study in a lab setting one participant at a time. After signing

an informed consent document, each participant was randomly assigned to one of the two experimental conditions as described above. Each participant was given a tutorial on how to use the drag-and-drop and the keyboard to input a Blockly program. We encouraged the participants to ask any questions that they might have during the study. The participants then carried out the input task using the two treatments. After each task, we asked the participants to complete a NASA Task Load Index (NASA-TLX) questionnaire for assessing subjective mental workload [34]. When the task had been completed, we asked participants to complete a post-session questionnaire about their experience.

4.6 Results

Our initial hypothesis was that users would input a Blockly program faster, more efficiently, and with fewer errors when using the keyboard as compared to the drag-and-drop. Thus, our null hypothesis for all analyses is that there is no significant difference between the distributions of corresponding performance measures across the two input methods. For all measurements, we used a paired t-test analysis. Figure 4.4 summarizes the performance of each input method.

4.6.1 Errors

Participants' mean errors for the drag-and-drop and the keyboard methods were 6.13% (SD: 4.21%) and 1.93% (SD: 1.84%), respectively. This represents a

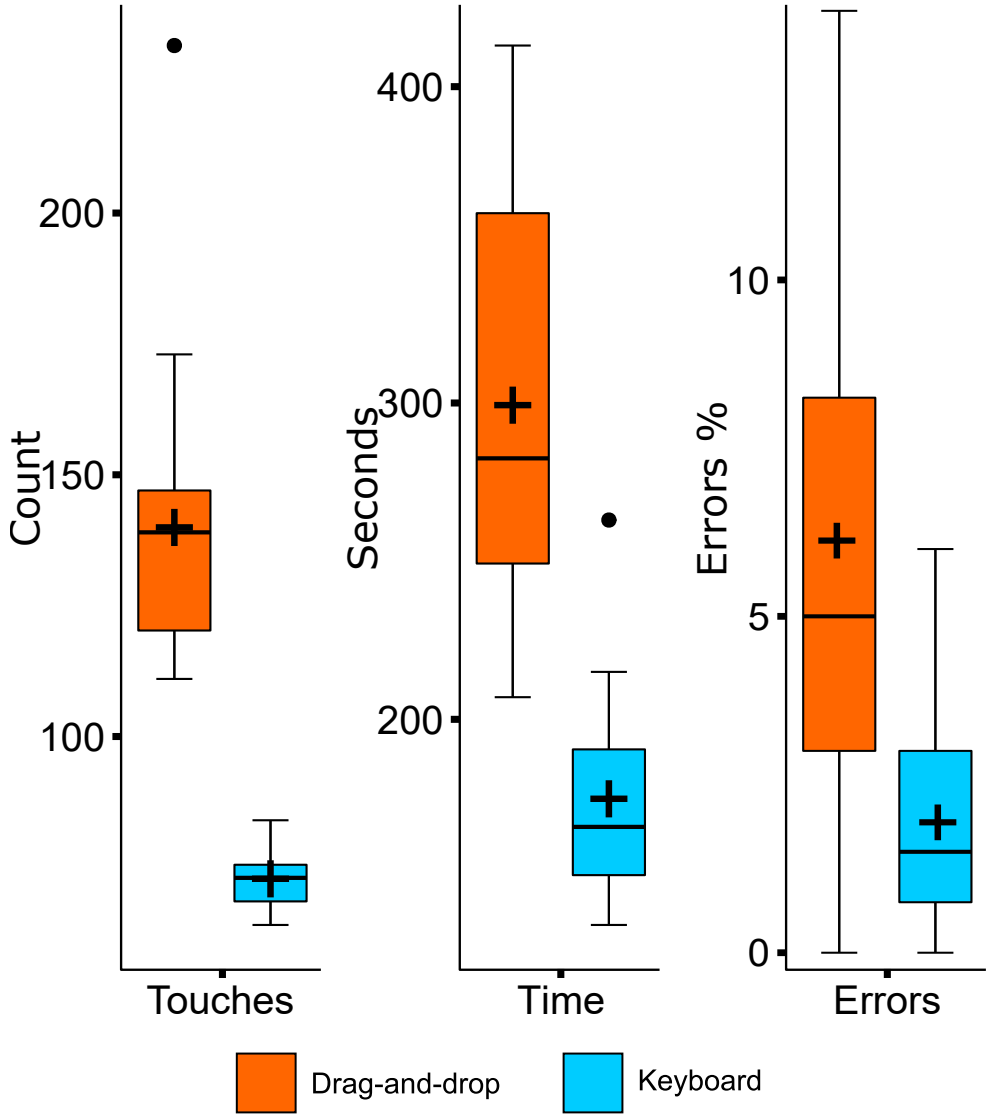

Figure 4.4: The number of touches, time, and errors for both the drag-and-drop and keyboard. The mean is shown with the "+" sign.

68.37% reduction in errors with the keyboard. There was a convincing statistical evidence for an effect of the input method on errors ($t_{(15)} = 3.5564$, $p<.01$). See

the third column in figure 4.4.

4.6.2 Number of Touches

The average touches to input the same Blockly program with the keyboard were fewer, 72.81 touches (SD: 6.19), than the drag-and-drop, 139.94 touches (SD 30.27). This means that the drag-and-drop has a 4.83 KSPB compared to the 2.51 KSPB for the keyboard. This represents a decrease of 47.97% in the keystrokes required to input a block. There is a convincing statistical evidence for an effect of the keyboard on the number of touches ($t_{(15)} = 9.0083$, $p<.01$). The first column in figure 4.4 shows the difference in the number of touches between the two input methods for the exact program.

4.6.3 Time

Participants, on average, took 174.88 seconds (SD: 32.74) and 299.31 seconds (SD: 68.18) to input the program on the keyboard and drag-and-drop, respectively. In another word, the keyboard has a speed of 9.95 BPM while the drag-and-drop has a speed of 5.81 BPM. The keyboard is 71.26% faster than the drag-and-drop method and the pair-wise t-test shows a significant difference in the speed between them ($t_{(15)} = 7.9963$, $p<.01$). The second column in figure 4.4 gives us a picture of the performance with respect to time.

4.6.4 NASA-TLX

Table 4.2 shows the mean response values for the RAW NASA-TLX measures. While there was no statistical evidence for the mental demand, temporal demand, or performance, there was convincing statistical evidence for an effect of input method on the other measures. These are the physical demand ($t_{(15)} = 4.332$, $p<.01$), effort ($t_{(15)} = 2.9929$, $p<.01$), and frustration level ($t_{(15)} = 3.7284$, $p<.01$). Figure 4.5 summarizes the TLX questionnaire results.

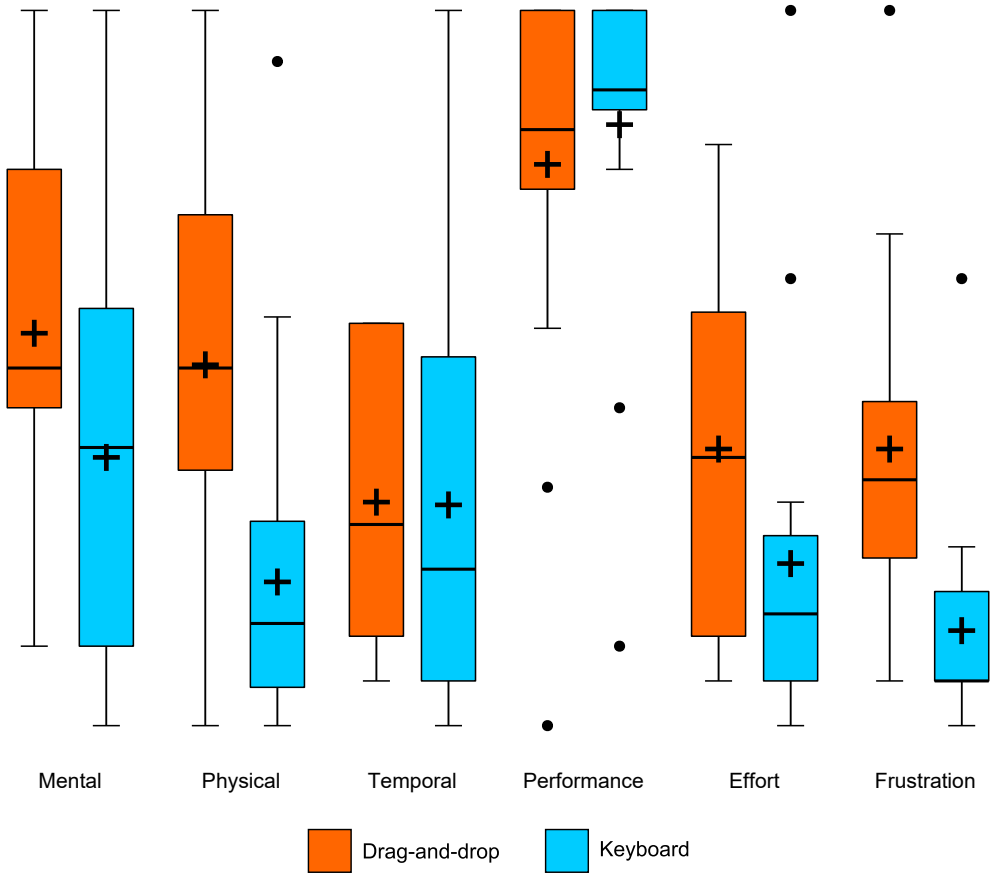

Figure 4.5: A summary of the NASA-TLX measures.

Table 4.2: NASA-TLX measures comparison (mean responses) between the drag-and-drop and the keyboard. The percentage column shows the decrease rate of the keyboard. A negative value indicates an increase.

TLX Measure	**Drag-and-Drop**	**Keyboard**	**Percentage**
Mental Demand	29.69	21.88	26.32%
Physical Demand	40.31	19.06	52.71%
Temporal Demand	30.00	29.68	1.04%
Performance	90.31	92.81	-2.77%
Effort	28.13	18.12	41.41%
Frustration	30.94	10.63	65.66%

4.6.5 Participants' Preference

After inputting the task with both input methods, participants completed a questionnaire about their overall experience with the keyboard. 75% of the participants indicated that they are likely to use the keyboard and 18.75% of them felt neutral. Only one participant indicated that he/she is not likely to use the keyboard in the future. In addition, 93.75% of participants found the keyboard to be helpful for inputting the Blockly program. 87.5% of the participants thought it was easy to adapt to the keyboard and 93.75% thought it was easy to use. 75% of the participants thought that the design of the keyboard is good and the rest felt neutral about the design. All the participants felt that they were efficient when using the keyboard. Figure 4.6 shows each question and a box-plot of the participants' average answers to each question.

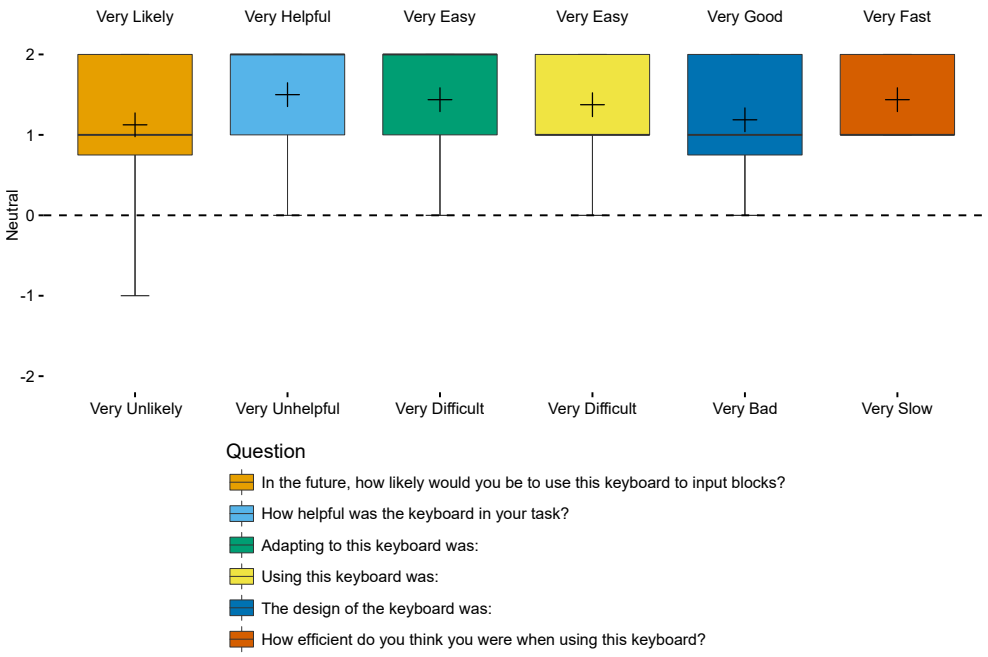

Figure 4.6: The results of the post-study questionnaire. Each column represents a question. The boxplots show the average response on a 5-point scale.

4.7 Discussion

The results of the study show that users performed better blocks input when using the keyboard as measured by BPM, KSPB, and errors. It is worth noting that these results were obtained after only 10 minutes of practice. It is expected to see a better input performance from a point-and-click input style like our keyboard when compared to a drag-and-drop method as was mentioned earlier. One explanation for this result is the shorter distance that fingers must travel when using the keyboard. Moreover, the average key size of the keyboard is larger than the average drag-and-drop touch targets. Per Fitts' law, the shorter distance and the larger

target size will positively affect the speed of the input task [35]. This can be seen in figure 4.7. In this figure, the touch locations are spread across a larger area for the drag-and-drop, by contrast, they are restricted to a smaller area for the keyboard.

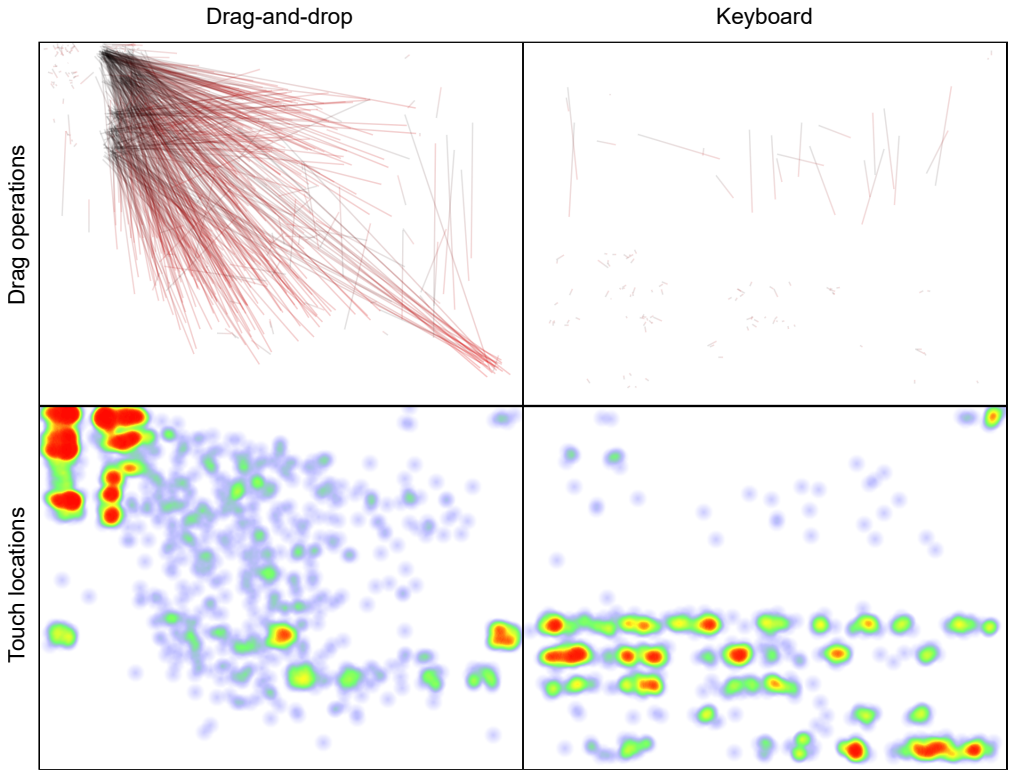

Figure 4.7: A visualization of the dragging and touching locations when inputting the Blockly program for all participants. The drag operation lines start from black and end in red.

4.7.1 Accuracy

The keyboard allows the users to input blocks with fewer errors than the drag-and drop (68.37%). There are many reasons for this result. First, the keyboard inputs each block to the highlighted connector automatically. Consequently, the errors from connecting a block to the wrong connector are eliminated. Second, the errors are reduced because of the large size of the keys compared to the toolbox or the block's options menu sizes. Finally, as can be seen in figure 4.7, that the participants' fingers travel longer distances while dragging blocks. The keyboard, nevertheless, requires no dragging. Despite that, few participants tried to drag the blocks because they thought that they could drag blocks when using the keyboard even though they were told otherwise. The same figure shows that the touch locations for the keyboard are more confined whereas they are more scattered for the drag-and-drop method. Therefore, the chance of introducing errors is increased with more scattered touches and longer travel distances. These reasons combined make the keyboard a more accurate way to input blocks.

4.7.2 Efficiency

There is a considerable difference between our keyboard and the drag-and-drop method when it comes to the number of touches. Our keyboard allows inputting blocks with almost half the number of touches without dragging (47.97% fewer touches). This reduction happens largely because of two reasons. First, the reduction in errors means less need for corrections. Thus, reducing the number of

touches. The second reason is the ability to change the options without touching the drop-down menus to open them. This requires one less touch each time an option needs to be changed. Many blocks rely on changing options and this impacts the efficiency of the drag-and-drop method negatively.

4.7.3 Speed

The keyboard is exceedingly fast in comparison to the drag-and-drop (71.26% faster speed). The reduced errors and keystrokes lead to this boost in the input speed. In addition, the automatic insertion of the blocks in the right connector without the need to drag is another area that helped the keyboard's speed. The blocks, however, need careful and precise positioning when dragging which slows the input.

Although the keyboard is fast, we suspect that the keyboard will be even faster after a longer period of use. Just like all keyboards, the key locations will be memorized and the visual scanning will take less time resulting in faster input speed. The input speed result in our study is for novice users. An expert user of a keyboard will input with higher speed [4]. However, the same thing cannot be said for the drag-and-drop method. The blocks reshape themselves after being connected to other blocks or after changing the options. For example, renaming the variables or changing the number in blocks will change the size of the block. Figure 4.3 shows how blocks with the same type have different shapes and connector locations, making dragging operations too difficult to memorize. In figure 4.7, we

can see how one program has many ways of dragging operations. Therefore, we suspect that the keyboard will be much faster after practice than the drag-and-drop.

4.7.4 NASA-TLX and Participants' Preference

Preferring a point-and-click style like the keyboard over the drag-and-drop method was shown by different studies [61, 62]. Our keyboard is no different. The NASA-TLX and the participants' feedback demonstrate that participants prefer the keyboard over the drag-and-drop method. Inputting blocks with fewer touches makes the keyboard less physically demanding which was confirmed by our participants' perceived physical demand (53% less physical demand). This may also affect the perceived effort (36% less effort). However, the lower frustration may be caused by the lower errors and the faster input (66% less frustration). From the post-task questionnaire, the majority of the participants preferred to use the keyboard and found it to be easy to adapt and use. The participants' preference is clear from their comments too. For example, participant 1 said, "Keyboard was more easier than drag and drop". While participant 8 said, "The automatic movement of the cursor was better than the drag and drop function it not only reduces the work of properly pairing two parts together but also was easy and smart". Participant 9 said, "The keyboard helps to reduce the dragging time which is helpful". Participants 10 and 11 respectively said, "I prefer the keyboard. It was much faster than moving the blocks around" and "adapting to the keyboard was so easy and

natural and faster than the drag and drop method".

4.7.5 Limitations

Just like other keyboards, there are some limitations for our keyboard. These results were gotten for a specific task. Any Blockly program that does adhere to the collected statistics will perform differently. In that case, however, we can safely assume that the keyboard input performance will not suffer dramatically. We assume that because the low number of touches, the faster speed, and the reduced errors will still hold due to the lack of dragging and the confined keyboard area when inputting different Blockly programs. We tested our keyboard on the original Blockly code. However, there are many derivatives of Blockly. Each one of them will have different input performance. Our keyboard holds a list of commands and their key names. One can change this list to call different or new blocks to accommodate different visual languages if they run on JavaScript.

4.8 Conclusions and Future Work

We presented a drag-and-drop alternative, a keyboard, to input Blockly programs. We talked about the motivation and the design of this keyboard. The user study showed how our keyboard surpasses the drag-and-drop method in terms of accuracy, efficiency, and speed when inputting a Blockly program. In addition, most of the participants preferred the keyboard and found it to be easy to use and

learn. They also perceived it to have less physical demand, less effort, and less frustration level.

This keyboard opens the door to potential future work. The results were obtained after a 10-minute practice. Better results are expected after prolonged use. A longitudinal study of the keyboard will show how far the input performance will go. The keyboard might make blocks input accessible for people with visual impairments because many of them rely on keyboards [67]. It could be beneficial for people with motor skill disabilities to input blocks without dragging. Another area of interest would be to see how a custom keyboard like this performs in other visual programming languages. Although our keyboard was tested on adults, children from different age groups may benefit from using such a keyboard differently because of the variation of their abilities. This makes children potential participants for future work. The keyboard could serve as an intermediate step to learning how to write textual programs because Blockly has a mapping of blocks to JavaScript, Python, PHP, Lua, and Dart. A study of the potential impact of this keyboard as a transitional step toward text-based languages might be useful. Lastly, enabling the two input methods at the same time might bring the positives from both. We are now extending the keyboard capabilities to work in conjunction with the drag-and-drop method. Studying the effects of both input methods at the same time is undergoing.

Chapter 5 : Conclusion

As was pointed out in the chapter 1, touchscreen devices need to extending coding paradigms to engage large populations in critical skill development. To do so, the domain-specific soft keyboards were developed. In Chapter 2, the design of a custom soft keyboard for writing Java source code was discussed. The formal user study showed how fast, efficient, and accurate the custom keyboard in comparison to the QWERTY keyboard. The total error rate was decreased by 37.38% and KSPC by 34.9%. The 10.64 WPM from first time users is encouraging. This is shown by the positive feedback from the users. In addition, the custom keyboard was shown to be mentally, physically, and temporally less demanding.

Chapter 3 showed how fast a typist can go when using the custom keyboard over eight sessions to input a Java source code. The study showed that the custom keyboard outperforms the theoretical limit of the QWERTY keyboard after 5 sessions. The participants' average speed gradually increased to 30.29 WPM by the eighth session. This is 16.5% faster than the theoretical top speed of an expert QWERTY typist. The total error rate was 2.61% and the KSPC was 0.81 by the eighth session. The custom keyboard was rated positively in terms of design, ease of use, and ease of adaptation by the users.

In chapter 4, the visual programming soft keyboard advantages when inputting blocks were discussed. The design of the blocks keyboard was explained. The

blocks keyboard was also compared to the drag-and-drop method using a formal user study. The study has shown that the custom keyboard outperforms the drag-and-drop method when it comes to speed, accuracy, and efficiency. The user study has shown that the keyboard reduced the blocks input errors by 68.37%. It also has shown that the keystrokes reduced by 47.97%. Moreover, the keyboard increased the input speed by 71.26% when compared to the drag-and-drop. The keyboard users perceived it to be physically less demanding with less effort. In addition, the users preferred the custom keyboards over the default input methods.

These custom keyboard designs and their good performance shed the light on program input using touchscreen devices for different programming languages whether it is text-based or block-based. This means that other programming languages with similar syntax or structure would also benefit from such custom keyboards on touchscreen devices. By making program input faster, more accurate, and more efficient, the custom keyboards will make touchscreen devices more appealing to a larger audience. These custom keyboards designs are the right steps toward making program input on touchscreen devices a common reality.

Bibliography

[1] Pew Internet Research, "E-reading rises as device ownership jumps," Pew Research Center's Internet & American Life Project, 2014. [Online]. Available: http://www.pewinternet.org/files/2014/01/PIP_E-reading_011614.pdf

[2] ——, "Smartphone Ownership – 2013 Update," Pew Research Center's Internet & American Life Project, 2013. [Online]. Available: http://www.pewinternet.org/files/old-media//Files/Reports/2013/PIP_Smartphone_adoption_2013_PDF.pdf

[3] International Data Corporation, "Press release: Tablet shipments forecast to top total pc shipments in the fourth quarter of 2013 and annually by 2015," 2013.

[4] I. S. MacKenzie, S. X. Zhang, and R. W. Soukoreff, "Text entry using soft keyboards," *Behaviour & information technology*, vol. 18, no. 4, p. 235244, 1999.

[5] X. Bi, B. A. Smith, and S. Zhai, "Quasi-qwerty soft keyboard optimization," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, p. 283286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1753367

[6] F. C. Y. Li, R. T. Guy, K. Yatani, and K. N. Truong, "The 1line keyboard: A QWERTY layout in a single line," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. New York, NY, USA: ACM, 2011, p. 461470. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047257

[7] A. Sears, D. Revis, J. Swatski, R. Crittenden, and B. Shneiderman, "Investigating touchscreen typing: the effect of keyboard size on typing speed," *Behaviour & Information Technology*, vol. 12, no. 1, pp. 17–22, 1993.

[8] I. S. MacKenzie and S. X. Zhang, "The design and evaluation of a high-performance soft keyboard," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '99.

New York, NY, USA: ACM, 1999, pp. 25–31. [Online]. Available: http://doi.acm.org/10.1145/302979.302983

[9] S. Zhai, M. Hunter, and B. A. Smith, "The metropolis keyboard - an exploration of quantitative techniques for virtual keyboard design," in *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '00. New York, NY, USA: ACM, 2000, pp. 119–128. [Online]. Available: http://doi.acm.org/10.1145/354401.354424

[10] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich, "Touchdevelop: programming cloud-connected mobile devices via touchscreen," in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 2011, pp. 49–60.

[11] Two Lives Left, 2015. [Online]. Available: http://twolivesleft.com/Codea/

[12] D. Leonard, "The iPad goes to school," *BusinessWeek: technology*, Oct. 2013. [Online]. Available: http://www.businessweek.com/articles/2013-10-24/the-ipad-goes-to-school-the-rise-of-educational-tablets

[13] A. Briggs and L. Snyder, "Computer science principles and the CS 10k initiative," *ACM Inroads*, vol. 3, no. 2, p. 29, Jun. 2012. [Online]. Available: http://www.nsf.gov/pubs/2014/nsf14523/nsf14523.htm

[14] A. Strawhacker, M. Lee, C. Caine, and M. Bers, "Scratchjr demo: A coding language for kindergarten," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 414–417.

[15] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-d tool for introductory programming concepts," in *Journal of Computing Sciences in Colleges*, vol. 15, no. 5. Consortium for Computing Sciences in Colleges, 2000, pp. 107–116.

[16] M. Resnick, "Starlogo: An environment for decentralized modeling and decentralized thinking," in *Conference companion on Human factors in computing systems*. ACM, 1996, pp. 11–12.

[17] F. P. Miller, A. F. Vandome, and J. McBrewster, *Keyboard Layout: Keyboard (Computing), Typewriter, Alphanumeric Keyboard, QWERTY, Portuguese Alphabet, QWERTZ, AZERTY, Dvorak Simplified Keyboard, Chorded Keyboard, Arabic Keyboard, Hebrew Keyboard*. Alpha Press, 2009.

[18] S. Zhai and P. O. Kristensson, "Interlaced qwerty: accommodating ease of visual search and input flexibility in shape writing," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 593–596.

[19] D. Langendorf, "Textware solutions fitaly keyboard v1. 0 easing the burden of keyboard input," *WinCELair Review, February*, 1998.

[20] S. Zhai and P. O. Kristensson, "The word-gesture keyboard: reimagining keyboard interaction," *Communications of the ACM*, vol. 55, no. 9, pp. 91–101, 2012.

[21] B. Biegel, J. Hoffmann, A. Lipinski, and S. Diehl, "U can touch this: touchifying an ide," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 2014, pp. 8–15.

[22] F. Raab, C. Wolff, and F. Echtler, "Refactorpad: editing source code on touchscreens," in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2013, pp. 223–228.

[23] M. Bacikova, M. Maricak, and M. Vancik, "Usability of a domain-specific language for a gesture-driven ide," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE, 2015, pp. 909–914.

[24] T. Teitelbaum and T. Reps, "The cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563–573, 1981.

[25] I. S. MacKenzie and R. W. Soukoreff, "Text entry for mobile computing: Models and methods, theory and practice," *Human–Computer Interaction*, vol. 17, no. 2-3, pp. 147–198, 2002.

[26] R. W. Soukoreff and I. S. MacKenzie, "Metrics for text entry research: an evaluation of msd and kspc, and a new unified error metric," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2003, pp. 113–120.

[27] I. S. MacKenzie, "Kspc (keystrokes per character) as a characteristic of text entry techniques," in *International Conference on Mobile Human-Computer Interaction*. Springer, 2002, pp. 195–210.

[28] K. Curran, D. Woods, and B. O. Riordan, "Investigating text input methods for mobile phones," *Telematics and Informatics*, vol. 23, no. 1, pp. 1–21, 2006.

[29] C. L. James and K. M. Reischel, "Text input for mobile devices: comparing model prediction to actual performance," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2001, pp. 365–371.

[30] D. J. Peterson and M. E. Berryhill, "The gestalt principle of similarity benefits visual working memory," *Psychonomic bulletin & review*, vol. 20, no. 6, pp. 1282–1289, 2013.

[31] T. F. Lunney and R. H. Perrott, "Syntax-directed editing," *Software Engineering Journal*, vol. 3, no. 2, p. 3746, 1988. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6888

[32] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a cognitive dimensions framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[33] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 11.

[34] S. G. Hart and L. E. Staveland, "Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research," *Advances in psychology*, vol. 52, p. 139183, 1988.

[35] P. M. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement." *Journal of experimental psychology*, vol. 47, no. 6, p. 381, 1954.

[36] E. Clarkson, J. Clawson, K. Lyons, and T. Starner, "An empirical study of typing rates on mini-qwerty keyboards," in *CHI'05 extended abstracts on Human factors in computing systems*. ACM, 2005, pp. 1288–1291.

[37] E. Hoggan, S. A. Brewster, and J. Johnston, "Investigating the effectiveness of tactile feedback for mobile touchscreens," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1573–1582.

[38] A. Carlberger, J. Carlberger, T. Magnuson, S. Hunnicutt, S. E. Palazuelos-Cagigas, and S. A. Navarro, "Profet, a new generation of word prediction: An evaluation study," in *Proceedings, ACL Workshop on Natural language processing for communication aids*, 1997, pp. 23–28.

[39] I. Almusaly and R. Metoyer, "A syntax-directed keyboard extension for writing source code on touchscreen devices," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 195–202.

[40] Pew Internet Research, "Technology device ownership: 2015," Pew Research Center's Internet & American Life Project, 2015. [Online]. Available: http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/

[41] Y. Kuno, B. Shizuki, and J. Tanaka, "Long-term study of a software keyboard that places keys at positions of fingers and their surroundings," in *Human-Computer Interaction. Towards Intelligent and Implicit Interaction*. Springer, 2013, pp. 72–81.

[42] "Swifty: Learn how to code in Swift," Swifty Team., 2016. [Online]. Available: https://getmimo.com/swifty/

[43] W. A. Fryer, "Hopscotch challenges: Learn to code on an ipad!" *Publications Archive of Wesley Fryer*, vol. 1, no. 1, 2014.

[44] "Swift Playgrounds," Apple, 2016. [Online]. Available: https://www.apple.com/swift/playgrounds/

[45] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt, "TouchDevelop: app development on mobile devices," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 39. [Online]. Available: http://dl.acm.org/citation.cfm?id=2393641

[46] A. Begel and S. L. Graham, "An assessment of a speech-based programming environment," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 116–120.

[47] W. Slany, "Catroid: a mobile visual programming system for children," in *Proceedings of the 11th International Conference on Interaction Design and Children*. ACM, 2012, pp. 300–303.

[48] S. McDirmid, "Coding at the speed of touch," in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 2011, pp. 61–76.

[49] T. A. Nguyen, C. Csallner, and N. Tillmann, "Gropg: A graphical on-phone debugger," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1189–1192.

[50] Y. A. Feldman, A. Gam, A. Tilkin, and S. Tyszberowicz, "Deverywhere: develop software everywhere," in *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. IEEE, 2015, pp. 121–124.

[51] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *The 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 207–216.

[52] javaParser, "Javaparser," 2015. [Online]. Available: http://javaparser.org/

[53] A. P. Carnevale, N. Smith, and M. Melton, "Stem: Science technology engineering mathematics." *Georgetown University Center on Education and the Workforce*, 2011.

[54] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[55] N. Fraser *et al.*, "Blockly: A visual programming editor," *URL: https://code.google. com/p/blockly*, 2013.

[56] D. Weintrop and U. 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.

[57] J. Monig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in gp," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 51–53.

[58] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 445–448.

[59] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, 2015.

[60] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, 2015, pp. 29–38.

[61] I. S. MacKenzie, A. Sellen, and W. A. Buxton, "A comparison of input devices in element pointing and dragging tasks," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1991, pp. 161–166.

[62] K. M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 8, no. 1, pp. 1–33, 2001.

[63] "Learn on code studio," 2017. [Online]. Available: https://studio.code.org/

[64] D. Weintrop and U. Wilensky, "Bringing blocks-based programming into high school computer science classrooms," in *Annual Meeting of the American Educational Research Association (AERA). Washington DC, USA*, 2016.

[65] I. S. MacKenzie, "Kspc (keystrokes per character) as a characteristic of text entry techniques," in *Human Computer Interaction with Mobile Devices*. Springer, 2002, pp. 195–210.

[66] C. D. Wickens, S. E. Gordon, Y. Liu, and J. Lee, "An introduction to human factors engineering," 1998.

[67] S. Ludi, "Position paper: Towards making block-based programming accessible for blind users," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 67–69.