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

Islam Almusaly
Oregon State University
almusali@onid.oregonstate.edu

Ronald Metoyer
Oregon State University
metoyer@eecs.oregonstate.edu

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.

I. 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 specific, 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. 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.

II. 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.

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.
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.

![Table](image)

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Return Type</th>
<th>Rename</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Function</td>
<td>if</td>
<td>else</td>
</tr>
<tr>
<td>Array</td>
<td>Comment</td>
<td>for</td>
<td>do</td>
</tr>
<tr>
<td>Container</td>
<td>Import</td>
<td>return</td>
<td>break</td>
</tr>
<tr>
<td>Class</td>
<td>Math</td>
<td>try</td>
<td>catch</td>
</tr>
</tbody>
</table>

Fig. 1. There are three primary 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) Programming on Tablet Devices: There are some attempts to ease programming on touchscreen devices [21], [22]. 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.

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 [23]. In doing so, syntax directed editors enforce proper syntax at all times. For example, a syntax-directed editor may require that user type commands in order to generate template code with assignments and expressions that have to be completed before moving on [23]. 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.

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 [24]. Error metrics include the minimum string distance error rate (MSD) and the total error rate (TER) [25]. 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) [25]. 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 [26]. The KSPC on a standard physical QWERTY keyboard is approximately 1.00 [26] but has been shown to reach up to 1.21 when correcting errors [25].

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 [27], [28]. 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.

III. 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], [9]. 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. 1). Fig. 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.

<table>
<thead>
<tr>
<th>1</th>
<th>Modifiers</th>
<th>Return Type</th>
<th>Rename</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Modifiers</td>
<td>Type</td>
<td>Rename</td>
<td>Assign</td>
</tr>
<tr>
<td>3</td>
<td>private</td>
<td>public</td>
<td>protected</td>
<td>static</td>
</tr>
</tbody>
</table>

Fig. 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.

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 that correspond to similar programming constructs, supporting visual working memory [29]. 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 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 I, second row), he can press the “Done” key to see the function options (Fig 2, first row). He can insert the “public” modifier by selecting the “Modifiers” key then the “public” key (Table I, third row) from the options row (Fig 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>
<thead>
<tr>
<th>Key pressed</th>
<th>Code</th>
<th>length</th>
<th>Total Keystrokes</th>
</tr>
</thead>
</table>
| Function    | {}
| s.e.t.      | shift
| X.          | Done
| public      | setX{}
| public      | setX{}
| Return Type | void       | public void setX{}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Type</td>
<td>void</td>
<td>public void setX{}</td>
</tr>
<tr>
<td>Key pressed</td>
<td>Code</td>
<td>length</td>
</tr>
<tr>
<td>Function</td>
<td>}</td>
<td>5</td>
</tr>
<tr>
<td>s.e.t. shift X. Done</td>
<td>}</td>
<td>9</td>
</tr>
<tr>
<td>public setX{}</td>
<td>}</td>
<td>16</td>
</tr>
<tr>
<td>public void setX{}</td>
<td>}</td>
<td>21</td>
</tr>
</tbody>
</table>

Our keyboard extension supports a limited form of syntax-directed editing which presents both advantages and disadvantages. One disadvantage is loss of flexibility [30]. 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. 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.

A. 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 auto-completion 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.

B. 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 [31] 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.
1) 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.

2) 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.

3) 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.

4) 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.

IV. 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 II-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.

A. 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 [32]. 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 [32]. However, they have 1 class field instead of 1.9 and 0.43 local variable per method instead of 0.87 [32].

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.

B. 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).

C. 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, insertions, and spacing errors with the unified error metric, TER [25].

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 [33]. When all tasks had been completed, we asked participants to complete a post session questionnaire about their experience with the two keyboards.

V. 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. 3 summarizes performance on each metric for each keyboard design.

![Fig. 3. KSPC, WPM, and TER for both the standard and extension keyboards. The mean is shown with the “+” sign.]

A. 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. 3.

B. 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.

C. 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. 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).

D. NASA-TLX

Table II 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 summarizes the TLX questionnaire results.

E. 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.

VI. 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 [34]. 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. 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 I 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 [35], [36].

1) 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 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].

2) 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.

3) 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.

A. 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.

B. Limitations

The keyboard extension is not without limitations. Some limitations arise from using a syntax-directed editing approach [30]. 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.

VII. 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.

ACKNOWLEDGMENT

We would like to thank the participants in our study and those who have given extensive comments on earlier versions of this paper.

REFERENCES


