Evaluation of a Frame-based Programming Editor

Thomas W. Price
North Carolina State Univ.
Raleigh, NC, USA
twprice@ncsu.edu

Neil C. C. Brown
University of Kent
Canterbury, Kent, UK
nccb@kent.ac.uk

Dragan Lipovac
North Carolina State Univ.
Raleigh, NC, USA
dlipova@ncsu.edu

Tiffany Barnes
North Carolina State Univ.
Raleigh, NC, USA
tmbarnes@ncsu.edu

Michael Kölling
University of Kent
Canterbury, Kent, UK
mik@kent.ac.uk

ABSTRACT

Frame-based editing is a novel way to edit programs, which claims to combine the benefits of textual and block-based programming. It combines structured 'frames' of preformatted code, designed to reduce the burden of syntax, with 'slots' that allow for efficient textual entry of expressions. We present an empirical evaluation of Stride, a frame-based language used in the Greenfoot IDE. We compare two groups of middle school students who worked on a short programming activity in Greenfoot, one using the original Java editor, and one using the Stride editor. We found that the two groups reported similarly low levels of frustration and high levels of satisfaction, but students using Stride progressed through the activity more quickly and completed more objectives. The Stride group also spent significantly less time making purely syntactic edits to their code and significantly less time with non-compilable code.

Keywords

Frame-based editing, Syntax, Evaluation, Greenfoot, Novice programming

1. INTRODUCTION

Programming syntax and syntax errors represent a fundamental, common and difficult challenge for novices learning to program [5, 10, 30]. Programming environments and courses designed for novices often feature innovations to ease the burden of syntax, such as improved error messages [9, 11], more intuitive programming languages [23, 30] and block-based editors which avoid syntax errors altogether [8, 12, 29]. However, improved error messages are not always effective [9, 26], and block-based editors may have unforeseen consequences on programming behavior [25] and be perceived as less authentic by students [27, 33].

Frame-based editing is a novel way to edit code, which attempts to reduce the burden of syntax, while maintaining useful elements of textual programming [6]. It combines structured 'frames' of preformatted code, similar to code blocks, with 'slots' that allow for efficient textual entry of expressions. Previous work argues that frame-based editing can ease the transition from blocks to text [18] while leveraging the benefits of both [6] to reduce syntax errors and increase efficiency [3]. This paper seeks to evaluate some of these claims empirically by comparing two groups of novices working on a short programming activity in Greenfoot, an IDE designed to teach programming [17]. One group used Java with Greenfoot's original text editor, and the other used Stride, a new frame-based language offered by Greenfoot. All other aspects of the environment were identical for the two groups, allowing us to directly measure the impact of using the Stride editor. We investigated the following research questions: Compared with the original Java editor, how will the use of the Stride editor affect students':

RQ1 Frustration and satisfaction with the activity?

RQ2 Performance on the activity?

RQ3 Programming behavior during the activity?

RQ4 Incidence of syntax errors?

ICER '16, September 8–12, 2016, Melbourne, VIC, Australia.

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4449-4/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2960310.2960319

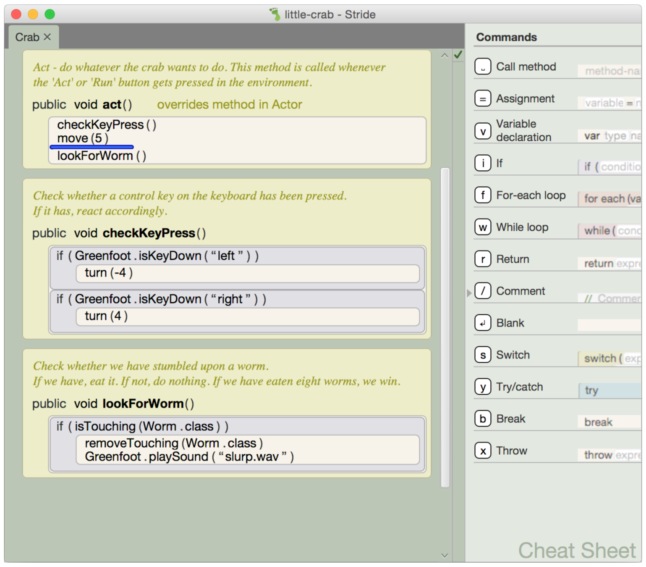
2. RELATED WORK

2.1 Syntax Errors

Syntax errors can be a very challenging [19] and time consuming [3] aspect of programming for novices. Programming syntax can be quite obtuse, and, as others have noted [26], programming languages and compilers are largely designed for professionals, not novices. Stefik and Siebert [30] compared novices' ability to comprehend and then reproduce a program in a variety of languages and found that some languages, such as Java, fared no better than one that used random ASCII characters as keywords. Researchers have attempted to identify the most problematic syntax errors made by novices in common programming languages such as Java [2, 10, 13, 14]. Though results only agree partially across data sources, some of the most common syntax errors in Java are missing semicolons; mismatched parentheses, quotes or brackets; and references to unidentified variables.

Difficulties with syntax may also impact students' ability to learn other, more fundamental principles of Computer Science. Lahtinen et al. [20] analyzed a survey of students' perceived difficulty with various aspects of computing and found that reported difficulty with syntax errors correlated with other, more complex tasks, such as understanding how to design a program and decomposing it into classes and procedures. Ahadi et al. [1] found that students' incidence of syntax errors was predictive of their overall success on a programming assignment. Jadud [14] defined the Error Quotient (EQ) as a measure of students' struggles with syntax errors, based on the incidence of repeated errors in consecutive compiles. Jadud found a weak but significant correlation between the EQ and both assignment and exam grades. Interestingly, Jadud and Dorn [15] later calculated the EQ for a much larger dataset of student work and found that a student's EQ only weakly correlated with the number of days that the student had spent programming.

Some systems have tried to reduce the challenges of syntax errors by providing better compiler messages. For example, Denny et al. identified common syntax errors [10] and then designed a system to enhance error messages with explanations and examples of correct code [9]. However, the authors found that these enhanced messages had no impact on students' ability to avoid or resolve errors, which is supported by other findings that more detailed messages are no better than shorter ones [26]. Traver [31] suggests a number of guiding principles for compiler messages, including clarity and brevity, specificity and locality, which also support this notion. Lee and Ko [21] found that personifying compiler feedback using a robot agent improved novices' completion of levels in a programming game, suggesting that some improvement of error messages is possible.



```
Act - do whatever the crab wants to do. This method is called whenever
the 'Act' or 'Run' button gets pressed in the environment.
public void act()      overrides method in Actor
    checkKeyPress ( )
    move (5 )
    lookForWorm ( )

Check whether a control key on the keyboard has been pressed.
If it has, react accordingly.
public void checkKeyPress()
    if ( Greenfoot . isKeyDown ( "left " ) )
        turn (-4 )
    if ( Greenfoot . isKeyDown ( "right " ) )
        turn (4 )

Check whether we have stumbled upon a worm.
If we have, eat it. If not, do nothing. If we have eaten eight worms, we win.
public void lookForWorm()
    if ( isTouching (Worm . class ) )
        removeTouching (Worm . class )
        Greenfoot . playSound ( "slurp.wav " )
```

Figure 1: The Stride editor. Each large, yellow rectangle is a method *declaration* frame. Each method *call* (e.g. `move(5)`) is a separate frame, even though it does not have an explicitly drawn border. The frame cursor is the thin blue bar between the second and third method calls in the act method.

2.2 Comparing Blocks and Text

Block-based programming environments, including Alice [8], Scratch [29] and Snap*!* [12], address the challenges of syntax errors by attempting to reduce or eliminate them altogether. The effectiveness of block editors has been evaluated in a number of comparative studies. Lewis [22] compared Scratch and Logo in a 5th grade summer camp and found that students perceived exercises to be equally difficult with both languages. On assessment questions, Scratch students performed significantly better with conditionals, but not with loops. Booth and Stumpf [4] compared text and block interfaces for adults learning to program Arduino boards and found that the block group perceived the interface to be more friendly and experienced lower perceived workload. McKay and Kölling [24] used a cognitive modelling tool to predict the execution time of programming tasks in a variety of editors, including block, text and an early prototype of the Stride editor. They found that block and text editors were each more efficient at certain tasks, but the Stride editor outperformed both on most tasks.

Others have directly compared blocks and text within the same programming environment or assessment, similar to the work we present here. Price and Barnes [28] compared novices using a block-based and textual programming interface and found that the block group completed significantly more objectives in less time and spent significantly less time idle compared to the text group. Weintrop and Wilensky [33] compared the performance of students mostly familiar with both blocks and text on two versions of an assessment that presented questions using block and text modalities respectively. They found that while the block modality produced increased scores on most CS topics, the exact reasons for this are complex and merit further study. The authors also found that students perceived programming with blocks to be easier, in part due to their being easier to read, with better natural layout, but students also perceived blocks to be less powerful, authentic and efficient to work with [32].

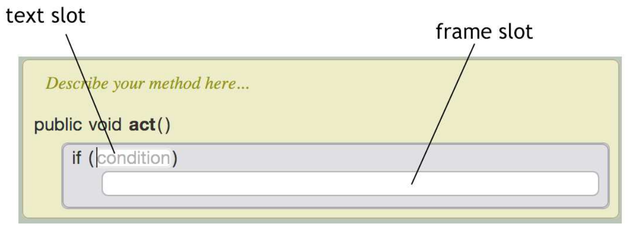
2.3 Greenfoot and Stride

Greenfoot [17] is an integrated development environment (IDE) designed to allow beginners aged 14 and upwards to easily program games and simulations. Originally Greenfoot only allowed programming in Java, but since the release of Greenfoot 3 in late 2015, the Stride language is also provided (see Figure 1). Semantically, Stride is identical to Java: it is only the editing interactions and some syntax that differs. Specifically: Stride uses *frame-based editing*, which tries to merge block-based and text-based editing [6].

Stride frames are first-class interface elements which can be dragged, selected, created or deleted as a whole item. A frame cursor can be placed between frames (see Figure 1), much as a text cursor is placed between characters. The cursor can be moved via keys or placed with a mouse click. Frames can be created at the current frame cursor position by clicking in the sidebar (right-hand side of Figure 1), or more commonly by pressing the indicated command key. For example, pressing 'i' will create an if-frame.

Frames have unalterable structure and labels (e.g. the "while" of a while-loop), with editable *slots* where the user will enter code. There are two types of slots, as shown in Figure 2. Frame slots, like the body of a method or loop, are places where new frames can be entered. Like blocks, but unlike text, it is not possible to have an unterminated body of a method or if-statement. This enforces well-defined scopes, which are drawn as rectangles (see Figure 2). Indentation is drawn automatically according to the frame hierarchy and need not (indeed, cannot) be managed manually.

Text slots, such as the name of a variable declaration or



```
Describe your method here...
public void act()
    if (condition)
```

Figure 2: Slot types: text slots and frame slots.

the condition of a while loop, have a text cursor which allows names and expressions to be entered like in text-based editing. Unlike in block-based editing, expressions are text: it is not "frames all the way down." However, the expression editor also helps students, for example by balancing parentheses: adding an opening '(' always also inserts the closing '),' and they are always deleted together.

Greenfoot automatically compiles student code whenever the user stops editing for more than one second. Errors are highlighted with a red underline similar to spell-checking errors in other software. If the user hovers over the error with the mouse or places the text cursor in the region, the error message will be shown with a pop-up. Java errors only originate from the Java compiler, whereas Stride errors can have two distinct sources. One is very similar to Java: syntactically valid Stride code is trivially transformed into Java and sent to the Java compiler, with any compilation errors shown on the original frames. However, Stride also performs syntax checking before generating Java code, which shows more targeted error messages. Most commonly, if a slot is left blank (e.g. an if condition, or the parameter in a method call) then an error is shown indicating that the slot "cannot be blank," rather than the consequent Java error "Illegal start of expression".

Stride offers a number of additional navigational features (e.g. a fold-out display of inherited superclass methods), as well as better context for code completion. However, the programming activity used in this study did not emphasize these features, so we focus here on the aspects of Stride which scaffold students' interaction with program syntax.

3. METHODS

To evaluate our research questions regarding the effectiveness of Stride's frame-based editor, we designed a controlled experiment in which we compared students working with Greenfoot's original Java editor to those working with the new Stride editor. All other aspects of programming activity and environment were kept constant, allowing us to directly measure Stride's impact.

3.1 Materials

We designed a Greenfoot tutorial and activity in which students create the Asteroids video game, adapted from a lesson in the Greenfoot textbook [16]. The core mechanics of the game were already implemented in a separate Java class file, and the students were tasked with responding to user input and calling existing methods to create game functionality. The activity was broken up into 9 steps, which introduced method calls, conditionals, parameters, variables, numeric comparisons and while-loops. We intentionally included a broad range of programming concepts with varying difficulty in the activity to avoid ceiling or floor effects.

Instructions for each step included an explanation of any new programming concepts and explicit objectives for the student. The instructions included pictures and analogous example code but never any code which could be directly used in a student's program. We created two versions of the activity and instructions, written in Java and Stride respectively. The instructions were identical across versions except for the programming language used in the example code and occasional language-specific explanations. For example, the Stride instructions identified which keyboard shortcuts would insert certain frames, and the Java instructions explained semicolons.

The instructions for the activity were provided on a website, and the website was instrumented to keep logs of how students progressed through the instructions. When students finished a step, they were required to click a button acknowledging its completion to proceed, which was logged to a database. Additionally, we instrumented the Greenfoot environment using the Blackbox logging framework [7] to keep detailed logs of student work over time, including regular snapshots of student source code and records of compilation errors.

Additionally, we designed a pre- and post-survey to give to students. The pre-survey contained four 5-point Likert items to assess students' self-efficacy and three to assess their interest with respect to computing, as well as a number of questions on past computing experience and demographic information. The post-survey repeated the self-efficacy and interest questions and additionally asked students to rate how much they enjoyed various aspects of the activity on a 5-point Likert-type scale. Additionally, it asked students to rate on a 5-point Likert-type scale how satisfying and frustrating they found four parts of the activity: creating the Asteroids game, figuring out what to do next, writing code in Greenfoot and figuring out what was wrong with their code when it had an error.

3.2 Procedure

The study took place during a middle school CS outreach program, which students attend voluntarily on Saturdays for 2-4 hours. We worked with three classes of students. The first two classes consisted of underserved students in a pre-college program. The first class ($n = 13$) was randomly assigned to the Java condition and the second was assigned to the Stride condition ($n = 9$). The third class consisted of students from local middle schools, and students were each randomly assigned to either the Java condition ($n = 5$) or the Stride condition ($n = 5$). Students in all three classes had been exposed to some block-based programming in 1 or 2 previous workshops but not textual programming. In total, we worked with 32 students (27 male; 5 female), with 18 in the Java condition and 14 in the Stride condition.

The Asteroids activity was led by the first author, though there was an unaffiliated graduate student in charge of each class as a whole. The whole activity lasted approximately 100 minutes. Students first took a pre-survey (5m), and then the instructor led them through setting up Greenfoot, loading the instructions and first two steps of the activity together (25m). During this time, students were encouraged not to work ahead, though a few did. Afterwards, students worked independently, using the instructions to complete as

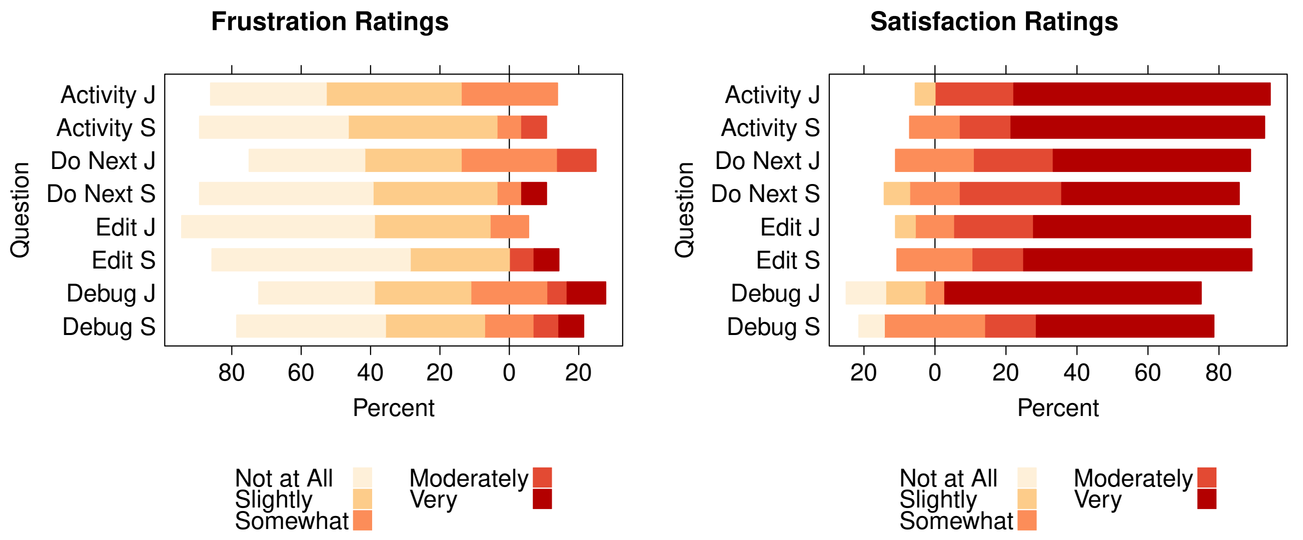


Figure 3: Post-survey ratings compared between the Java (J) and Stride (S) groups. The horizontal bars are aligned by the neutral "Somewhat" rating. Students rated how frustrating/satisfying they found "Creating the Asteroids game" (Activity), "Figuring out what to do next" (Do Next), "Writing code in Greenfoot" (Edit) and "Figuring out what was wrong with my code when it had an error" (Debug).

much of the activity as possible. Students worked for 60-70 minutes on the activity, with times varying among classes due to one class starting late (though the same amount of time is used from each class in the analysis). While students were not encouraged to work together, they were in close proximity and did discuss the activity. Afterwards, students stopped the activity and took a post-survey (5m).

The instructor led students through the first 2 of the activity's 9 steps because we wanted to provide adequate support for the challenging activity, while reducing possible bias from the instructor and allowing students to work independently as much as possible. For the Java-only and Stride-only classes, this introduction showed only that language during instruction. For the mixed class, the instructor demonstrated code using both Java and Stride. The first two steps, which were done together, are not included in our analysis, and all times are reported from the start of independent student work.

Along with the instructor, there were 3-4 additional student volunteers who were available to help students throughout the activity. Volunteers received training beforehand on how to handle student requests for help. When a student asked for help, volunteers directed them to look for an answer in the instructions. If the student was still confused, the volunteer then explained the relevant concept or problem with the student's code. Volunteers were instructed to avoid spending too much time with a given student, to avoid giving direct commands ("Now type X") and to never touch the mouse or keyboard. Volunteers were given a log sheet and instructed to record each interaction with a student, including that student's ID (to link to log files), the time and duration of help, and what type of help was given (explaining instructions, general programming concepts, language-specific concepts or debugging).

3.3 Data

We collected 4 sources of data for analysis:

1. Pre- and post-survey data
2. Log data from the instructions website
3. Log data from Greenfoot
4. Instructor logs of help requests

The Greenfoot logs contained the most verbose data. From these logs we extracted complete program traces for each student, showing how their code progressed over time. We also identified successful and unsuccessful compiles for each student and any resulting compiler messages. We used these program traces to determine when students successfully completed each of the objectives outlined in the 9 activity steps. Some steps had multiple objectives (e.g. an if statement with an else statement), and we evaluated these separately for a total of 13 objectives. Most objectives were independent, meaning a student could fail to complete one objective but succeed at the next. Two graders determined the time of completion for each student and each objective. The graders had an initial agreement of 87.8%, and after clarifying objective criteria and independently re-grading this rose to 94.5%. The remaining objectives were discussed to produce final times of completion for each student and each objective.

Due to technical problems, a number of students lost connection to the Greenfoot logging server during the activity. As a result, we only have complete Greenfoot log data for 24 of the 32 students for 50 minutes of programming (measured from the start of independent work), including 13 in the Java condition and 11 in the Stride condition. Of the 8 students with missing data, 3 came from the all-Java class, 1 from the all-Stride class and 2 from each condition of the mixed class. For analyses that involve the Greenfoot logs, we used only data from the first 50 minutes of the 24 students for whom

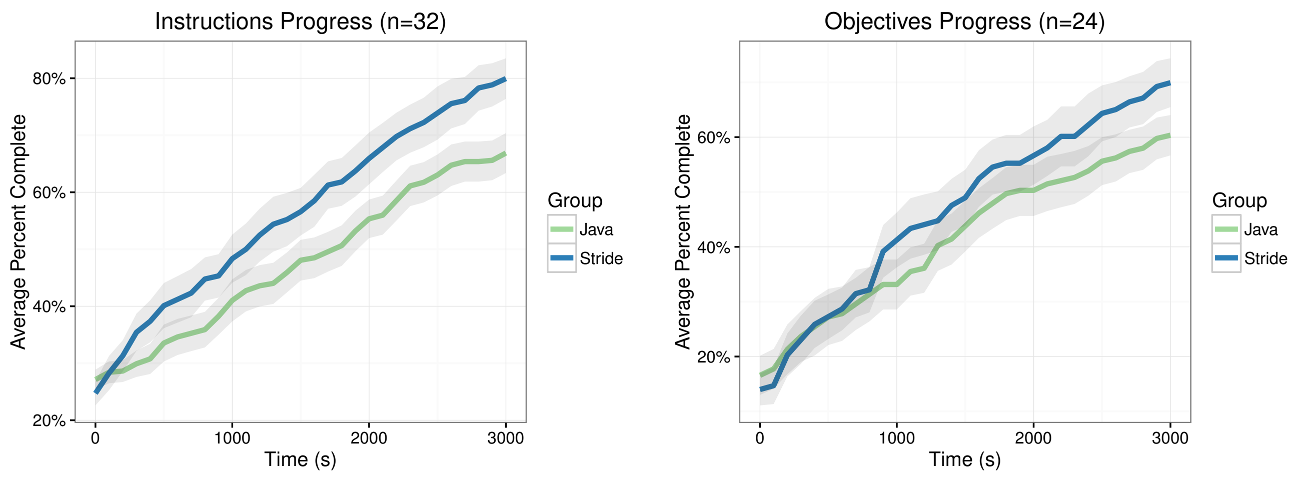


Figure 4: The mean percentage of instructions viewed (left) and activity objectives completed (right) by students in the Java and Stride conditions from the beginning of independent work (time = 0s) to the end of analyzed time (time = 3000s), with shading to indicate standard error.

we have full data. However, for other analyses we used all 32 students' data. Additionally, three students arrived 5-7 late minutes to the all-Stride class. All students arrived at least 10 minutes before independent work started, and the instructor waited for these students to catch up while working together. However, these students may have been slightly disadvantaged. All classes had between 24 and 26 minutes of group instruction before individual work.

4. ANALYSIS

4.1 Surveys

While student self-efficacy and interest with respect to CS are not directly addressed by our research questions, we included these questions in the pre- and post-surveys to assess the appropriateness of the Asteroids activity for CS outreach with our population. For both self-efficacy and interest questions, we averaged students' responses to each of the 3-4 questions in the category to produce a numerical value for both pre- and post-surveys. We performed a Wilcoxon signed-rank test[1] to compare pre- and post-survey response values. There was an improvement from pre to post, and this difference was significant[2] for both self-efficacy ($W = 28.5$; $p < 0.001$) and interest ($W = 22.5$; $p = 0.031$). This suggests that the activity was appropriate for our population and therefore an appropriate context in which to compare Stride and Java.

We also used pre-survey responses to look for potential differences between the Stride and Java groups before the activity. We tested for a difference in the groups' pre-survey self-efficacy and interest scores using a Mann-Whitney U test[1] and found no significant differences for self-efficacy ($U = 154$; $p = 0.293$) or interest ($U = 147.5$; $p = 0.413$). We asked students if they had had previous computing experiences, such as writing a computer program, creating a website or make a video game. We summed the positive responses for each student to produce an experience score and found the difference between groups was not significant ($U = 84.5$, $p = 0.109$). There were more females in the Java group (4 out of 18) than the Stride group (1 out of 14), though there was an equal proportion of underrepresented minority students in the Java (9 out of 18) and Stride (7 of 14) groups. Given these similarities, we determined the two groups came from comparable populations. Additionally, there was no significant difference in the number of minutes volunteers spent helping students in the Java (M=12.4; SD=8.3) and Stride (M=14.0; SD=8.1) groups ($t(28.3) = -0.53$; $p = 0.599$; d[3] $= -0.189$).

To address RQ1, we compared students' post-survey responses to questions regarding their perceived frustration and satisfaction with specific parts of the Asteroids activity. Their responses are shown in Figure 3. The distributions are quite similar for both conditions and show evidence of floor and ceiling effects, with students generally showing low frustration and high satisfaction. Mann-Whitney U tests confirmed that there was no significant difference in frustration or satisfaction ratings between the two conditions.

4.2 Performance

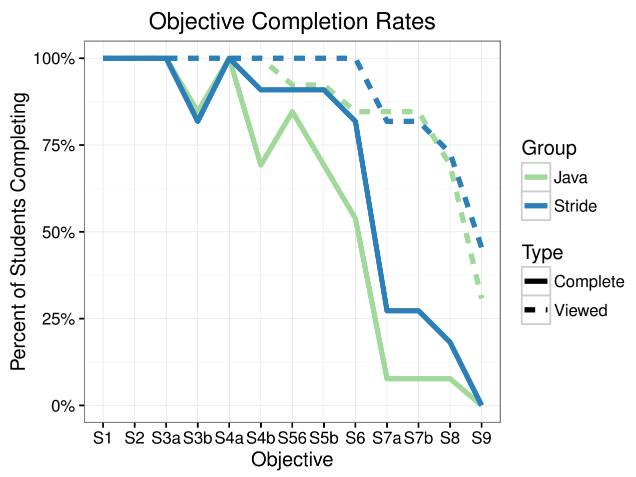
To answer RQ2, we used a number of measures of performance. The most straightforward way to assess how students performed on the Asteroids activity is to compare how far they got in a given amount of time. Because we do not have Greenfoot log data for all 32 students, we can use progress in the instructions as a measure of self-reported progress on the activity. Figure 4 (left) shows the mean percentage of the instructions that each group had viewed, from the start of independent work to the end of the activity. By the end of the activity, the difference between the percentage of instructions viewed by the Java group (M=66.9%; SD=14.9%) and the Stride group (M=79.9%; SD=13.3%) was significant ($t(29.3) = -2.61$; $p = 0.014$; $d = -0.92$).

This self-reported progress is not as meaningful as the completion of assignment objectives, as determined by the

[1]The use of nonparametric tests, including the Wilcoxon signed-rank test and Mann-Whitney U test, indicates that data was either ordinal or not normally distributed as determined by a Shapiro-Wilk test.

[2]All tests were made at a 5% significance level.

[3]We report Cohen's d, an effect size measured in SDs.



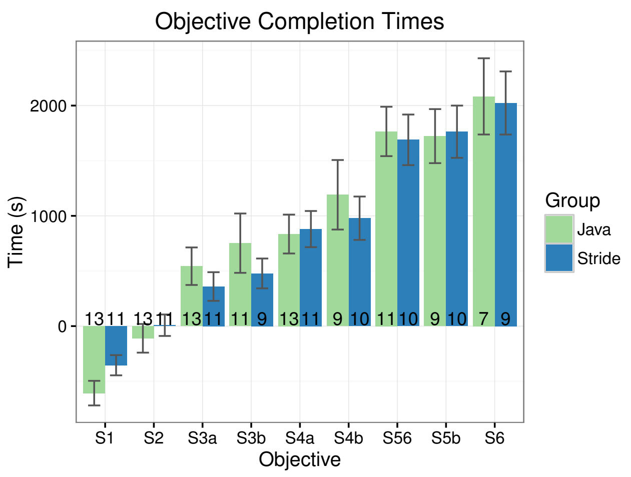


Figure 5: Left, the percent of students who completed and viewed instructions for each objective in each group during the first 50 minutes of independent work. Right, the mean time elapsed (with standard error) before Objectives S1-S6 were completed by students in each group. The number of students who completed the objective is shown below each bar.

graders, described in Section 3.3. Figure 4 (right) shows the percentage of objectives completed by students over time. This counts only objectives that were correctly completed by the student, as some students skipped or improperly completed objectives. This and all following analysis considers only the 24 students for whom we have complete data. Java students completed a mean 60.4% (SD=13.3%) of the 13 objectives, while Stride students completed a mean 69.9% (SD=14.5%). This difference was not significant ($t(20.4) = -1.66$; $p = 0.112$; $d = -0.69$), with a 95% confidence interval that the true difference between the Java and Stride means is between -21.6% and 2.5%.

We also analyzed students' performance on individual objectives, both in terms of the number of students completing the objective and the amount of time elapsed before the objective was completed. Figure 5 (left) shows the percent of students who completed and viewed instructions for each objective in each group. Objectives S1-S3b have almost identical completion rates for both groups, but the Stride group had a strictly higher completion rate for the next 7 objectives. The difference is especially stark for Objective S5b (Java = 69.2%; Stride = 90.9%) and Objective S6 (Java = 53.8%; Stride = 81.8%). These two objectives came at the end, so presumably fewer students in the Java group had time to complete them. Figure 5 (right) compares the mean amount of time that elapsed before students completed Objectives S1-S6 in each group. Note that some values are negative, as Objectives S1 and S2 were completed before independent work started (time 0). The Java group took slightly longer on most objectives (after time 0), but these differences were not large compared to the standard error. It is important to note that this time comparison considers only students who were able to complete a given objective in the first 50 minutes, so the later objectives have a strong selection bias to include the faster students in a group.

4.3 Programming Behavior

We used the detailed Greenfoot log data, along with the instructions log data, to categorize how users spent their time during the activity. We categorized time as editing code, viewing instructions, running the program or idle time. The first three categories each have specific actions associated with them, such as editing or compiling code, changing the instructions page or clicking the Run button. Each time one of these actions occurred, we labeled the time span until the next action occurred, or until 60 seconds elapsed, with that action's category. Any time beyond 60 seconds was labeled as idle, until the next action occurred. While this estimate is not perfect, it does give a reasonable picture of how students used their time. Table 1 shows the mean amount of time spent in each category for each group.

	Edit	**Instr.**	**Run**	**Idle**
Java	661 (235)	1088 (474)	562 (308)	686 (360)
Stride	585 (177)	1233 (221)	634 (262)	472 (311)

Table 1: The mean amount of time (with SD) spent on each category of action during the Asteroids activity.

Two categories of particular interest are Edit and Idle time. A difference in editing time would show that one group spent longer interacting with the editor. A difference in idle time would indicate a possible difference in engagement between the two groups. Figure 6 shows the percentage of students' time spent idle and editing over 150 slices of time from 0 to 50 minutes, compared between the Java and Stride groups. While there is no apparent difference between the two groups for editing time, the Java group appears to spend much more time idle near the end of the activity, after 2000s have elapsed. The difference in total idle time per student, as shown in Table 1, was not significant ($t(22.0) = 1.57$; $p = 0.132$; $d = 0.63$), with a 95% confidence interval that the true difference between the Java and Stride means was between -70s and 499s. However, a Wilcoxon signed-rank test indicated that there was a significant difference between the proportion of time Java students spent idle before 2000s (Med=15.7%) and after (Med=28.6%) ($W = 15$; $p = 0.033$).

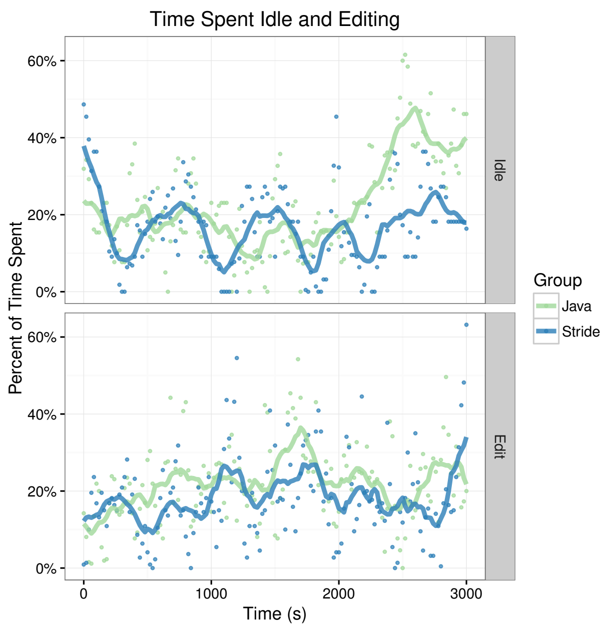


Figure 6: For 150 20-second slices throughout the activity, the average percent of time students in each condition spent Idle and Editing are shown as points. A kernel regression smoothing line is also plotted to show a rolling average.

For the Stride group, the difference between the proportion of time spent idle before 2000s (Med=12.5%) and after 2000s (Med=8.7%) was not significant ($W = 34$; $p = 0.966$).

Though students spent similar amounts of time editing their code in both conditions, we were also interested in the types of edits students made. One of the primary purported advantages of the Stride language is that it avoids the need for the purely syntactic elements of Java, such as brackets, semicolons and whitespace [6]. To investigate this claim, we analyzed each edit students made to their code to determine whether or not that edit changed only these purely syntactic elements (brackets, semicolons and whitespace). We then summed the time spent on these edits for each student, where each edit was assumed to have taken the time since the previous edit, to a maximum of 30s. A Mann-Whitney U test indicated that there was a significant difference between the amount of time spent on these syntactic edits between the Java group (Med=321s) and the Stride group (Med=94s) ($U = 134$, $p < 0.001$).

4.4 Syntax Errors

To get a basic understanding of whether the groups had different incidences of errors, we calculated the time each student spent with non-compilable code. This was done by summing the time between each unsuccessful compile and the next compile. The Java group spent a mean 1447s (SD=672.8s) with non-compilable code, while the Stride group spent a mean 1000s (SD=290.2s). The difference was significant ($t(16.8) = 2.16$, $p = 0.045$; $d = 0.84$). The Stride group spent on average 7.45 minutes less time with non-compilable code than the Java group, which spent on average almost half of the activity with non-compilable code.

We also counted the occurrences of each type of syntax

Group	Message	Mean
Java	';' expected	13.4 (9.53)
	illegal start of expression	6.31 (7.99)
	(or) expected	4.00 (2.42)
	reached end of file while parsing	2.77 (3.39)
	illegal start of type	2.38 (2.69)
Stride	*Invalid expression*	14.3 (11.3)
	Undeclared variable	13.7 (8.27)
	Expression cannot be empty	8.27 (5.08)
	cannot find symbol - method	4.73 (2.15)
	Method name cannot be blank	4.43 (4.52)

Table 2: The five most common syntax errors for the Java and Stride groups, with the mean (and SD) number of occurrences per student. Stride-specific errors are given in italics.

error viewed by students. While both Stride and Java compile automatically and flag errors with red underlines, we chose to analyze only errors which were actually displayed to the student, when their cursor or mouse enters this underlined code. Since Stride code is converted to Java before being compiled, both groups received errors from the Java compiler; however, Stride introduces a number of new error messages, which replace common Java errors with plain English. Table 2 gives the five most commonly viewed syntax errors for both groups. Many of the most common Stride errors were Stride-specific messages (denoted in italics). Additionally, each of the most common Java errors can be attributed to the presence of characters which Stride adds automatically: semicolons, brackets and parentheses. Notably, none of these top-5 errors are the same in the two groups. The difference between the total errors viewed by the Java group (M=42.3; SD=27.4) and the Stride group (M=55.5; SD=22.1) was not significant ($t(22.0) = -1.30$; $p = 0.208$; $d = -0.52$).

5. DISCUSSION

RQ1 *How did Stride affect frustration and satisfaction with the activity?* Students reported low frustration and high satisfaction with each aspect of the Asteroids activity, but there were no significant differences between the Java and Stride groups. It is possible that a real difference was masked by ceiling and floor effects (i.e. the activity was enjoyed enough that it overwhelmed any differences due to the editor). Previous comparisons between block and textual programming have similarly found no difference in students' perceived difficulty [22, 28], which mirrors our results with respect to frustration. Previous work on block-based programming has also suggested that it is perceived as less authentic than their textual counterparts [32]. While we did not assess perceived authenticity directly, if Stride was perceived as less authentic than Java, we have no evidence that it affected students' satisfaction with the activity.

RQ2 *How did Stride affect performance on the activity?* Taken together, our results suggest that Stride improved students' performance on the one-hour activity. The Stride group progressed through the instructions significantly faster and had a better completion rate than the Java group on the last 7 objectives. While there was no significant difference between the total number of objectives accomplished by the two groups, the first two results suggest that this may be due to the small sample size (24), in part as a result of data loss. There was no significant difference in the amount of time taken to complete a given objective between the groups; however, it is important to remember that this compares only students who completed a given objective within the first 50 minutes of independent work. For example, on Objective 5b, 90.9% of Stride students completed the objective in time, while only 69.2% of Java students completed the objective. The time comparison on this objective possibly excludes the slowest 30% of Java students, compared to only 10% of Stride students. While positive, these results are not as strong overall as previous comparisons between textual and block editors [28], supporting the notion that frame-based editing may be best situated between learning blocks and text [18].

RQ3 *How did Stride affect programming behavior during the activity?* Students' programming behavior in each group offers some explanation of the improved performance observed in the Stride group. Students in both groups spent similar amounts of time idle for the first 30-35 minutes of the activity, after which the Java group's idle time significantly increases and the Stride group's does not. One explanation for this trend is that there comes a point at which some students start to lose interest in the activity (confirmed anecdotally by volunteers), but that this was less common in the Stride group. Additionally, though students in both groups spent similar amounts of time editing their code, Java students spent a median 321s making purely syntactic edits, compared to 94s for the Stride group. This amounts to almost half of the total time the Java group spent editing their code and is twice the proportion found in previous analyses [3]. It is also worth noting that Stride students encountered their own Stride-specific issues, such as accidentally inserting a frame by typing its hotkey, while intending to insert text. While the amount of time wasted as a result of these errors is more difficult to measure, graders report anecdotally that this was a frequent occurrence. Still, these difficulties did not seem to be enough to make up for the advantages that Stride offered.

RQ4 *How did Stride affect incidence of syntax errors?* Taken together, our results support the notion that Stride helps students with syntax errors, which may be another contributing factor to their efficiency. The Stride group spent significantly less time with non-compilable code. This may be due in part to the fact that many of the most common Stride error messages (including four of the top five) were custom messages designed to directly explain the problem in a novice-friendly way. These results contrast with previous work in which enhanced error messages failed to improve students' resolution of syntax errors [9, 26]. The five most common errors encountered by the Java group were somewhat consistent with results from previous work [5, 13, 14]. Each of these errors can be caused by a missing a semicolon, bracket or parenthesis. The Stride editor avoids the need for semicolons and brackets and automatically pairs parentheses, reducing the incidence of these errors. This is evidenced by the fact that none of these top five Java errors were represented in the top five Stride errors (in part because some were replaced by the new Stride messages).

6. CONCLUSIONS

In this paper we have presented a comparison of two groups of novices working on a short programming activity, one using the textual Java language and one using the frame-based Stride language. While we observed no differences between students' perceived frustration or satisfaction, the Stride group did progress through the activity quicker, spending less time with non-compilable code and less time on purely syntactic edits. This suggests that frame-based editing is a useful tool in reducing the burden of syntax for novice programmers and that it may lead to improved performance on programming tasks.

6.1 Limitations

This study took place in a voluntary after school outreach program, which is quite different from a controlled laboratory setting. We endeavored to control as many aspects of the study as possible, but real classrooms come with real confounds, such as interactions between students, possible instructor and volunteer bias and varying student ability. We chose to emphasize the ecological validity of the study, ensuring that it reflected a real-world setting, rather than a perfectly controlled laboratory. Unfortunately, our results also suffer from the loss of data due to technical errors. A number of the differences observed in the 24 students for whom we have full data were sizable but not significant. This only means the differences were not detectable, but not necessarily that they do not exist. For example, for the difference in the number of objectives completed by the Java and Stride groups, reported in Section 4.2, power analysis suggests we only had a 46.3% chance of detecting a large effect size (Cohen's d) of 0.8.

Additionally, this study is limited to a single population (middle school students) and a single, hour-long activity. Our findings with respect to programming performance and behavior will not necessarily generalize to longer, more complex activities, though it is unclear whether the differences observed would increase or decrease in magnitude with more time. These results are also limited to students' *performance* on the activity, which will not necessarily equate to learning gains. Our results with respect to specific syntax errors will likely not fully generalize to other activities, as syntax errors are highly dependent on the programming constructs being used. The activity was also quite popular with the students, who reported low frustration and high satisfaction, and this may have masked any effects that the editor had on student affect.

6.2 Future Work

Frame-based editing has been described as a way to ease the transition from blocks to text [18]. Our own findings support the notion that frame-based editing is more appropriate for novices than text but may not offer the same benefits as blocks. Future work should investigate the relationship between these three types of editors: blocks, frames and text. We can now move beyond the short-term, direct comparisons of student performance that have been presented in this and other work [28] to focus on the long-term effects of the editors, their impact on learning gains and the transitions from blocks to frames and frames to text. Further work on how these transitions might be mediated (e.g. [8]) is needed, along with more in-depth investigations of students' perceptions of these editors (e.g. [32]).

7. REFERENCES

[1] A. Ahadi, J. Prior, and R. Lister. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, pages 401–406, 2016.

[2] A. Altadmri and N. C. C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.

[3] A. Altadmri, M. Kölling, and N. C. C. Brown. The cost of syntax and how to avoid it: Text versus frame-based editing. In *CELT: COMPSAC Symposium on Computing Education & Learning Technologies; part of COMPSAC 2016: The 40th IEEE Computer Society International Conference on Computers, Software & Applications*, June 2016.

[4] T. Booth and S. Stumpf. End-user experiences of visual and textual programming environments for Arduino. In *Proceedings of the 4th International Symposium on End-User Development*, pages 25–39, 2013.

[5] N. C. C. Brown and A. Altadmri. Investigating Novice Programming Mistakes: Educator Beliefs vs Student Data. In *Proceedings of the Tenth International Computing Education Research Conference*, pages 43–50, 2014.

[6] N. C. C. Brown, A. Altadmri, and M. Kölling. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In *Proceedings of the Fourth International Conference on Learning and Teaching in Computing and Engineering*, 2016.

[7] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pages 223–228, 2014.

[8] W. Dann, D. Cosgrove, and D. Slater. Mediated Transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, pages 141–146, 2012.

[9] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 19th ACM Conference on Innovation & Technology in Computer Science Education*, pages 273–278, 2014.

[10] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Conference on Innovation and Technology in Computer Science Education*, page 75, New York, New York, USA, jul 2012. ACM Press.

[11] T. Flowers, C. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through Gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.*, pages 10–13, 2004.

[12] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[13] J. Jackson, M. Cobb, and C. Carver. Identifying Top Java Errors for Novice Programmers. *Proceedings Frontiers in Education 35th Annual Conference*, pages 24–27, 2005.

[14] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Third International Workshop on Computing Education Research*, pages 73–84, 2006.

[15] M. C. Jadud and B. Dorn. Aggregate Compilation Behavior: Findings and Implications from 27,698 Users. In *Proceedings of the 11th International Computing Education Research Conference*, pages 131–139, 2015.

[16] M. Kölling. *Introduction to programming with Greenfoot*. Pearson Education, Upper Saddle River, New Jersey, USA, 2009.

[17] M. Kölling. The Greenfoot Programming Environment. *Transactions on Computing Education*, 10(4):14:1—-14:21, nov 2010.

[18] M. Kölling, N. C. 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*, pages 29–38, 2015.

[19] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. In *Proceedings of the Australasian Computing Education Conference*, pages 105–111, 2003.

[20] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A Study of the Difficulties of Novice Programmers. In *Proceedings of the Tenth ACM Conference on Innovation and Technology in Computer Science Education*, volume 37, page 14, 2005.

[21] M. J. Lee and A. J. Ko. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research*, pages 109–116, 2011.

[22] C. Lewis. How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pages 346–350, 2010.

[23] L. Mannila, M. Peltomäki, and T. Salakoski. What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, 2006.

[24] F. McKay and M. Kölling. Predictive modelling for HCI problems in novice program editors. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, pages 35–41, 2013.

[25] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education.*, pages 168–172, 2011.

[26] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler Error Messages: What Can Help Novices? In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education*, pages 168–172, 2008.

[27] K. Powers, S. Ecott, and L. Hirshfield. Through the Looking Glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin*, 39(1):213–217, 2007.

[28] T. W. Price and T. Barnes. Comparing Textual and Block Interfaces in a Novice Programming

Environment. In *Proceedings of the 11th International Computing Education Research Conference*, 2015.

[29] M. Resnick, J. Maloney, H. Andrés, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[30] A. Stefik and S. Siebert. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*, 13(4), 2013.

[31] V. J. Traver. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction*, 2010:1–26, 2010.

[32] 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 International Conference on Interaction Design and Children*, pages 199–208, 2015.

[33] D. Weintrop and U. Wilensky. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *Proceedings of the 11th International Computing Education Research Conference*, pages 101–110, New York, New York, USA, 2015. ACM Press.