# The Cost of Syntax and How To Avoid It: Text versus Frame-Based Editing

Amjad Altadmri
University of Kent
Canterbury, Kent, UK
aa803@kent.ac.uk

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

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

***Abstract*—Plain text has always been the predominant medium for writing and editing programs for expert users. Text is powerful and flexible, but requires more careful manipulation than structural editors, such as those found in block-based environments. In addition, in textual editors programmers are responsible for managing detailed orthography and layout – when beginners work with text, significant time is spent managing syntax problems, indentation and spacing. Frame-based editing is a new editing paradigm that combines the structural editing of block-based systems with the flexibility and keyboard-focus of text editing. In this paper, we empirically examine how much time and effort is spent by beginners on managing syntax errors and indentation, which can be automatically saved by switching to frame-based editing. The data is obtained using the Blackbox dataset; the results predict a clear advantage of frame-based editing over traditional text editors.**

## I. Introduction

Programming is an act of problem-solving: formulating and debugging algorithms. All too often, beginners spend their time solving the incidental problems: syntax errors and incorrect code layout. Educators want to teach universal underlying concepts, yet are forced to spend time correcting use of semicolons, brackets and indentation. Learning these latter details is necessary for practical work, but carries no intrinsic value. If the tools were different, the fundamental concepts would remain, but these syntactic details would not. Having to track down and fix simple syntactic problems typically occupies a significant amount of learners' time; we consider this time and cognitive effort wasted.

Modern teaching tools (and tools for professionals) should aim to reduce the time spent on accidental complexities, and free the learner's mind to think about the important aspects of program development – those that cannot easily be handed to a machine. One of the aspects that should be avoided is the unnecessary struggle with syntax. The time spent managing syntax and layout varies substantially depending on the expertise of the programmer, the programming language and the editing tool. Previous work has explored which syntax errors are problematic [2], [3], [4] and how to alleviate them via additional tools [5]: it is clear that all text-based editing tools suffer from variations of this issue due to the program structure representation as raw text.

Block-based editors, such as Scratch [1], eliminate many of these concerns, but instead introduce weaknesses in readability and navigation, restricting such systems to smaller programs and early learning. They also leave a significant conceptual gap in semantics to full-text programming, which typically features additional concepts. Thus, it is difficult to determine how much of Scratch's simplicity (compared to full text programming) is due to syntax and how much is due to semantics.

In this paper, we aim to quantify time spent on dealing with syntax errors and formatting code in a text-based system and – more importantly – evaluate how much of this time and effort can be saved by using frame-based editing. Frame-based editing is a new editing paradigm that borrows the structured editing of block-based systems but is much closer to text-based programming [6]. We compare editing Java in the BlueJ system [7] to editing Stride in Greenfoot, BlueJ's sister system [8]. Stride is a new frame-based language that is semantically equivalent to Java. This makes the comparison particularly interesting: the semantics are held constant between the two languages, but the editing paradigm is varied. Thus the comparison provides an insight into the cost of dealing with syntax, and to what extent it can be avoided with a frame-based editor.

For this study, we use Blackbox [9], a large dataset that records BlueJ usage data. We analyse edit and compile experiences of all users who have opted in to data collection over one year. This includes more than a quarter of a million users, with 37 million separate compilation events and nearly 200 million edit interactions. The key contributions of this paper are:

- Quantification of the time that novice (Java) programmers spend fixing syntax errors and managing indentation; and
- An examination of how frame-based editing can prevent some of the most common syntax errors by design.

In Section II, we provide background on block- and text-based programming, and discuss prior work. Section III discusses frame-based editing, including the aspects that result in the reduction of errors. The core contributions are in Sections IV and V, where we describe an analysis of errors novice programmers make in a text editor. The paper concludes in Section VI.

## II. Related Work

Previous studies have examined the cost of syntax. Stefik and Siebert [10] compared accuracy rates among syntax styles

in different text-based languages. Denny et al. [11] found that syntax errors in text languages occupied an unexpectedly high amount of novice programmers' time. Recently, studies have examined blocks vs. text in more detail. Price and Barnes [12] analysed block- vs. text-editing for students performing tasks in textual/tiled Grace, finding that students were faster and more accurate when writing code with blocks.

Other studies examined the relative ranking of syntax problems within Java, which is partly replicated in this study. Most researchers in the past used compiler error messages to classify mistakes. Jackson et al. [13] identified the most frequent errors among their novice programming students. McCall and Kölling [4] analysed errors based on semantic categories. Going beyond analysis of frequency, Denny et al. [3] evaluated how long students need to solve different errors. Altadmri and Brown [2] provided more robust data about student mistake frequencies and time-to-fix, using a much larger number of students than other studies (hundreds of thousands). Their study combined parsing detectors and compiler error messages, which are based on a survey of educators asking for the most common mistakes they saw among their students.

Almost all of these previous studies have either tried to understand and classify the errors observed, or aimed to make the mistake (and potential correction) clearer to students; some also alter the text-based syntax. Our study evaluates a different approach, frame-based editing, which aims at preventing these mistakes by design. Many low-level syntax errors are caused by a lack of memorisation or simple typing errors, not a misunderstanding of a fundamental concept. Letting students make these mistakes and fix them does not aid their learning of useful programming concepts. We use the analysis method proposed by Altadmri and Brown [2] to study the expected impact of frame-based editing on time spent fixing errors and managing indentation.

## III. Frame-Based Editing

Figure 1 shows the interface of a frame-based editor for a new, Java-derived language called Stride. We will use this implementation as the prototype to discuss the concepts of frame-based editing, focusing on the features of this editor which alleviate dealing with syntax errors and code formatting.

Like block-based systems, frame-based editing has manipulatable graphical representations of program constructs (called "frames"), but the visual appearance is more muted than in most block systems: fewer colours are used, 3D-effects and shadows are avoided, and frames blend into each other more, resulting overall in a more text-like view of a program.

While some specific design decisions are influenced by the concrete system context (novice users), most of the aspects described are independent of this context, and the advantages apply equally to frame editing in professional environments.

In block-based systems a syntactic construct, such as an if-statement, is a block. It provides places to specify the condition and the body. This block is pre-formed; the condition and body

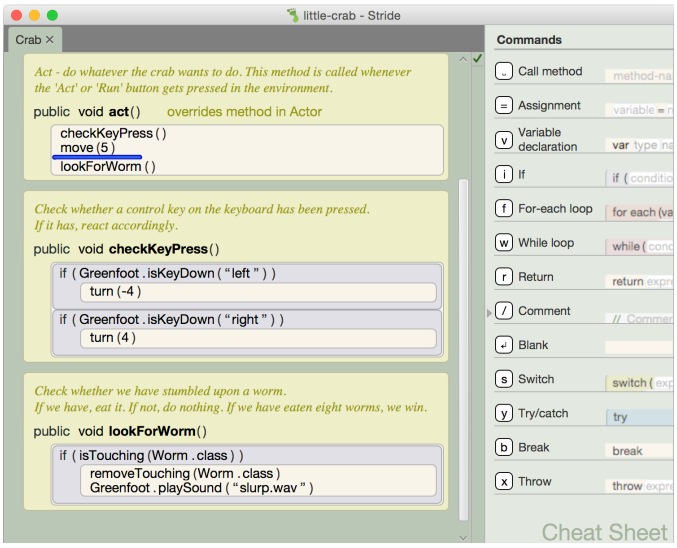


Fig. 1. The interface of a frame-based editor

may be edited, but the construct itself is otherwise unmodifiable. However, in text-based editors, a language construct is made up of smaller syntactic tokens: the `i` and `f` characters are edited separately in the `if` keyword, the condition follows in round brackets and the body in curly brackets. There is much room for error here; mistyping the keyword, mismatching the brackets, and so on.

In frame-based editing, an if-statement is a frame. Similar to a block, the frame has places – *slots* – for condition and body, and it is indivisible. Several frames can be seen in the interface in Figure 1, representing methods, if-statements and method calls. The structure of a frame is fixed and uneditable; no syntax errors relating to the statements' structure are possible. In addition, since frames are intrinsically distinct entities, no separators (e.g. semicolons) are needed to delimit them.

### A. Slots

Frames provide two kinds of slots for entry of additional code into frames: *frame slots* and *text slots*. An if-statement, for example, has a frame slot for its body and a text slot for its condition. Frame slots hold nested frames, while text slots hold expressions or identifiers. Frame slots have the intrinsic advantage of avoiding structural syntax errors, since their content is assembled entirely from nested frames. For text slots, the Stride editor allows structured text entry.

Expressions in block editors are assembled from blocks dragged into place in the same manner used for statements. This has been shown to impose an undesired level of viscosity [15] and is perceived as tedious by many programmers. In text editors, expressions can be entered freely via the keyboard, providing fast entry and flexibility, but allowing more errors. In our frame editor, expressions are typed into the slot in a way similar to text editors, but structure is inferred from the expression in real time. Paired symbol operators, such as quotes and parentheses, are recognised as a single operator. They are always entered and deleted as a pair; deleting one

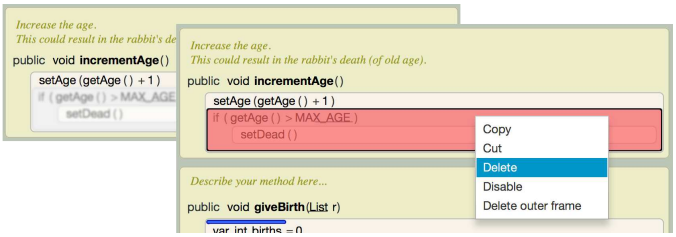


Fig. 2. A disabled frame (left) and preview of a delete operation (right).

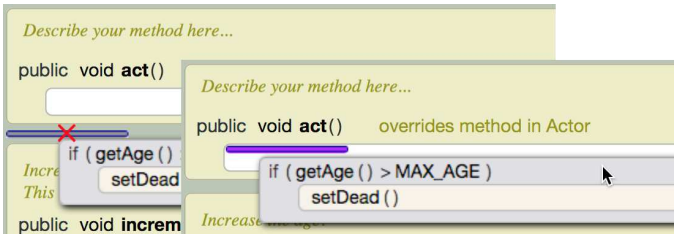


Fig. 3. Dragging a frame, (left) invalid drop target, (right) valid drop target.

bracket automatically removes the other; unbalanced brackets can never exist. Thus, for expression entry, frame editing achieves the flexibility of raw text editing while avoiding some specific potential errors.

Elements requiring further entry are always represented by a slot. For example, an assignment frame has two slots: one for the left-hand destination and one for the right-hand source. Similarly, method declarations have two slots per parameter: one for the type and one for the name. Method calls have one slot per parameter, for the value. Since frames clearly present this structure to the programmer, they aid in the entry of novices' programs. Some common mistakes, such as providing a type and value in a method call, are avoided.

### B. Scope

Scopes are represented as frames: graphical boxes, rather than the customary pair of brackets, parentheses or keywords. This is true for all scopes: classes, methods, and control structures. The frames, like the scopes, may be nested. There are no curly brackets to mismatch, and indentation is automatically determined by the position in the syntax tree.

Recognising the extent (beginning and end) of a scope is much easier and quicker in this representation. Programmers do not need to determine which closing bracket matches which opening bracket, and no additional confusion can be created by misleading indentation.

### C. Inserting, deleting and disabling code

Statements are inserted by inserting an entire frame. Every kind of statement has its corresponding frame, which can be inserted using a single command key, or using the "cheat sheet" sidebar (Figure 1) and the mouse. As with inserting, deleting a frame deletes it in its entirety. Different delete options may be available: An if-statement offers deletion while leaving the body statements intact, or with the body included in the deletion. While a function is selected in the menu, a preview annotation in the source code hints at the effect of the selected function (Figure 2).

Disabling a frame (Figure 2) temporarily treats it as deleted. Traditional systems typically disable a block of text by "commenting out". This is a misuse of the comment symbol, as the purpose here is not a comment; a comment symbol is used merely because of the absence of other mechanisms. However, the richer frame editor interactions and ability to use interface elements other than characters to convey program state open many presentation possibilities. In our implementation, the editor presents disabled frames with blurred appearance.

Unlike in traditional editors it is not possible to comment out part of a statement, for instance missing a closing bracket of a scope, and thus invalidating program structure. The explicit disable function in the frame editor is both visually clearer and less prone to syntax errors.

### D. Moving code

Mouse controlled drag-and-drop functionality for code segments is available in most traditional text editors. However, because the atomic unit of manipulation is a single character, many errors may be introduced by this operation.

In text editors, arbitrary spans of text can be selected and dragged. These may include parts of statements, accidentally selected, and thus the drag operation may invalidate program structure. Avoiding this mistake often requires careful targeting with the mouse to aim the selection at specific characters (including whitespace and line break characters). In the frame editor, only complete frames can be dragged, including simple one-line statements. No fine targeting is required; the frame is a large target, and does not require selecting before dragging.

In text editors, the majority of potential drop locations are syntactically invalid. Yet, editors provide no help in identifying the few valid targets. In the frame editor, frames may be dropped only at locations where they maintain a syntactically correct structure (Figure 3); placing statements outside of a method body, for example, is not allowed.

### E. Relation to Structured Editing

Frame-based editing relates to two independent bodies of prior work. The more recent, and strongest influence, is block-based editing, exemplified by Scratch [1]. The older body of work, originating in the 1980s and early 1990s, is on structure-editing. Inspired by the same observations about text being an imperfect mechanism to edit programs with intrinsic structure, structure editors typically attempted to completely eliminate syntactical errors by preventing the creation of invalid programs. When referencing a variable, for instance, only already-declared variables would be available for entry.

Structure editors were mostly implemented on text-mode terminal screens at a time when knowledge of human-computer interaction and interface design was in its infancy. Thus, many of them turned out to be too clunky and awkward to use [16]. The work on structure editors largely ceased in the

1990s. A decade later, however, Scratch, Snap!, and similar systems demonstrated that the ideas underpinning structure editors can be successful in modern graphical systems with modern, well-designed user interfaces.

## IV. Analysis of syntax cost

### A. Method

To get an understanding of the cost of syntax, we analysed programming efforts of a cohort of students using a traditional text-based system. The system used was BlueJ, an educational development environment for Java, and data was collected via the Blackbox project [9]. To get a representative sample, we included data for all BlueJ users for whom data was available, for one [academic] year (1/9/2013 to 31/8/2014, inclusive).

Blackbox provides a comparatively large data set, offering the basis for relatively robust analysis. In this study, interactions from 265,979 unique users were analysed, generating 37,158,094 separate compilation events and 197,091,248 tracked edit interactions.

The compile events were analysed to evaluate and categorise the kinds of errors users made, with the goal of identifying the errors that are caused purely by syntactic problems. We can then analyse how many (and which) of these errors would be prevented by Stride's frame-based editor.

We also analysed the time students took to fix the errors, resulting in a quantification of the amount of time that could be saved. In addition to error occurrence, we evaluated edit actions to identify edits that deal purely with indentation or syntactic decorations (such as semicolons or curly brackets). These edits are not needed in frame editors, and again we can quantify the amount of effort that may be saved by use of a frame editor.

### B. Student Mistakes

Altadmri and Brown [2] identified 18 mistakes that serve as a basis for analysing the most frequent and relevant errors. This does not include all errors students make, and thus provides a lower bound in the resulting data. Their set was in turn derived from Hristova et al. [14], who identified 20 mistakes by interviewing educators. For our analysis here, we maintain Altadmri and Brown's labelling of errors from A through R, but exclude error *N* (ignoring the non-void result of a method), as it is not always an error. This leaves us with 17 errors categorised into three groups:

- Semantic errors: 'A', Confusing the assignment operator (=) with the comparison operator (==); 'C', Unbalanced parentheses, curly or square brackets or quotation marks, or using these different symbols interchangeably; 'D', Confusing "short-circuit" evaluators (`&&` and `||`) with conventional logical operators (`&` and `|`); 'E', Incorrect semicolon after an if-, for-, or while-condition before the body of the construct; 'F', Wrong separators in for loops (using commas instead of semicolons); 'G', Inserting the condition of an if-statement within curly brackets instead of parentheses; 'H', Using keywords as method or variable names; 'J', Forgetting parentheses after a method call; 'K', Incorrect semicolon at the end of a method header; 'L', Incorrect spelling of greater-than-or-equal operator or less-than-or-equal operator, i.e. using => instead of >= or =< instead of <=; 'P', Including the types of parameters when invoking a method.
- Type errors: 'I', Invoking methods with a wrong type of argument; 'Q', Incompatible types between method return and type of variable that the value is assigned to.
- Other semantic errors: 'B', Use of == instead of `.equals` to compare strings; 'M', Trying to invoke a non-static method as if it was static; 'O', Control flow can reach end of non-void method without returning a result; 'R', Class claims to implement an interface, but does not implement all the required methods.

These mistakes are detailed with examples in [2].

### C. Error detection

We use the same methodology as Altadmri and Brown [2] for detecting the mistakes: a mix of compiler error messages and custom source analysis code to detect other errors.

We took each source file in the data set, and tracked it over time. At each compilation, we checked for the 17 mistakes. If a mistake was present, we then looked forward in time to find the next compilation where the mistake was no longer present (or until we had no further data for that source file). When the mistake was no longer found, we counted this as one instance of the mistake. Further occurrences in the same source file were treated as further error instances.

The time to fix is calculated in seconds between the first appearance of the mistake and the possible fix, and capped at 300 seconds (5 minutes) for any mistake that took longer than 300 seconds to fix, or is never fixed. The total time taken to fix each mistake was then added up across all errors of all users. The result is the total time this user cohort spent on fixing that particular kind of error over one year.

### D. Editing statistics

For the gathering of edit statistics we tracked each source file over time, as for the error detection, but took snapshots at each edit interaction rather than at compilation events. Blackbox records an edit operation for each line that is changed in the source code; individual character edits within the same line are grouped into one edit interaction.

Time for edit interactions is recorded by calculating the time between distinct edits, or between an edit and the most recent prior user interface interaction. The time is measured in seconds and capped at 60 seconds.

After recording edits and their interaction time, we identified edit operations that concerned solely auxiliary text (whitespace, semicolons and curly brackets). These are the edits that are not required (or even possible) in frame editors and would thus be saved by switching to such a tool.

Again, the amount of time taken for these edits was added up across all edits and all users, giving a measure of time (both in absolute terms and as a percentage of work time) taken up with unnecessary syntactic work.

| Mistake | Freq. | Total Time to Fix (hours) | Error Type | Freq. Frame |
|---|---|---|---|---|
| C | 793232 | **11700.46** | Syntax | 0 |
| I | 464075 | 12448.11 | Type | = |
| O | 342891 | 8093.39 | Semantic | = |
| A | 173938 | 4790.77 | Syntax | = |
| B* | 121172 | 4474.11 | Semantic | = |
| M | 86625 | 2178.97 | Semantic | = |
| R | 79462 | 2662.71 | Semantic | = |
| P | 52862 | **902.91** | Syntax | 0 |
| E* | 49375 | **1775.27** | Syntax | 0 |
| K | 38001 | **915.11** | Syntax | 0 |
| D* | 29605 | 923.48 | Syntax | = |
| J | 18955 | 388.39 | Syntax | < |
| Q | 16996 | 432.07 | Type | = |
| L | 4214 | 37.70 | Syntax | < |
| F | 2719 | **50.55** | Syntax | 0 |
| H | 1097 | 18.54 | Syntax | = |
| G | 118 | **1.70** | Syntax | 0 |
| | Total time: | 56360.10 | | |
| | Time saved: | **15734.40** | **(28%)** | |

TABLE I

STATISTICS FOR MISTAKES COMMITTED BY BLACKBOX-TRACKED STUDENTS - 1/9/2013 TO 31/8/2014 - IN DESCENDING ORDER OF FREQUENCY. '*' POINTS OUT NON COMPILER-ERROR MISTAKES. '0' INDICATES THE MISTAKE IS NOT POSSIBLE IN THE FRAME EDITOR (AND TIME IS BOLDED), WHILE '=' MEANS IT IS STILL POSSIBLE. '<' SHOWS REDUCED BUT NOT AVOIDED ERRORS IN A FRAME EDITOR.

## V. RESULTS

### A. Mistakes

Our detector looks for the number of instances of mistakes in the data set, as described in Section IV-C. The data set featured 37,158,094 compilation events, of which 19,476,087 were successful and 17,682,007 resulted in error messages.

Each compilation event may include multiple source files – the total number of source files input to compilation was 46,448,212, of which 24,264,603 were compiled successfully and 22,183,609 contained errors.

*1) Mistakes Overview:* The frequencies and total time-to-fix across all observed occurrences of each different type of mistake are shown in Table I, along with our classification of the error message (syntax, semantic, or type error).

The last column in Table I indicates the possible presence of this error in a frame-based editor. If this type of error is not possible in frame editing, it is marked as '0'. Those errors marked with '<' can occur in a frame editor, but their impact or likelihood is reduced. For errors that are unaffected by the change to frame editing, '=' is used. We will examine errors with the first two statuses.

The following errors are eliminated in frame editing:

**C** Mismatched brackets or quotations. This error is prevented by ensuring that they are always paired in the frame editor. In statements, such as method calls or if-statements, brackets are part of the fixed decoration of the frame and cannot be forgotten or removed. In expressions, pairs of brackets, parentheses or quotations can only be added or removed together, avoiding mismatches.

**P** Including a parameter type in a method call. It is avoided by providing a single slot for each parameter; there is no space where a type can be entered. In addition, the formal parameter's name is showed as a prompt, indicating clearly what is expected.

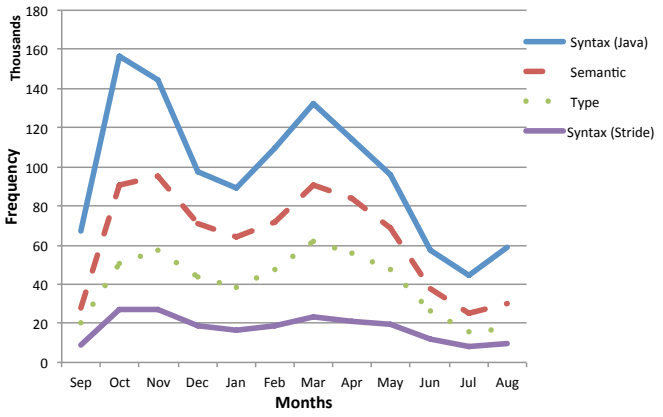


Fig. 4. Mistake frequencies over 12 months (raw numbers).

**E** Incorrect placement of semicolons. Since frames do not need statement delimiters, no semicolons are used.

**K** Incorrect semicolon in method header. Users are not required to type any of the syntactic decoration of frames, including semicolons.

**F** Wrong separators in for-loops. Again, because syntactic decoration does not need to be typed by the user, this mistake is not possible.

**G** Incorrect brackets for if-statements. This error is prevented for the same reason as the two previous ones.

Two additional mistakes are alleviated, but not eliminated entirely. We have *not* included them in the calculated time saved, but note them here:

**J** Missing parentheses after a method call. It is lessened because parentheses are automatically shown in a method call statement frame, and thus cannot be initially forgotten. However, they can be removed, and for method calls used in expressions it is possible to forget to enter them.

**L** Using =< rather than <=. This mistake is alleviated by the latter being recognised as a single compound operator in an expression slot, but the former will be shown as two separate operators with a space between them. Thus, a visual indication exists highlighting the difference, making it likely that users would notice and avoid the mistake.

Overall, this cohort of BlueJ users spent a total of 56,360 hours (approx. six and a half years) on fixing compiler errors. 15,734 hours (28%) of this error-fixing time could have been saved by using a frame-based editor.

*2) Mistakes Over Time:* The rate of syntax errors does not remain constant over the learning progression of a student. Novices make more errors at the beginning, and error frequency reduces somewhat as time progresses and students

become more familiar with some of the syntax. To investigate the effect of this change on the effort that may be saved over time, we analysed the mistake frequency over the course of the year, split into months. Figure 4 outlines the mistake frequencies for the three categories (syntax, type and semantic) over the course of 12 months, and also notes the rate of syntax errors that would remain using Stride's frame-based editor.

The results are not normalised; the graph instead shows raw numbers of error events. The shape of the graph is therefore affected by the number of compilations performed, and shows peaks around the beginning of teaching terms. It is clear, however, that the rate of syntax errors can be significantly reduced by using a frame editor at any stage in this learning. Interestingly, simple syntax errors would become the least frequent error category, instead of the most frequent.

### B. Edits

The edit interaction analysis, as described in Section IV-D, identified 197,091,248 text editing events. For each of these user edit events, the time taken for the edit was calculated as outlined in Section IV-D.

The total editing time of all edits included in this analysis was 625,128 hours (approximately 71 years). Formatting-only edits were identified, that is: those edits affecting only auxiliary text (whitespace, semicolons and positioning of curly brackets). These edits took 152,606 hours – approximately 24% of the total editing time – and would be eliminated in frame-based editing.

## VI. Conclusion

Frame-based editing prevents many syntax errors by design and avoids manual manipulation of formatting symbols. Our exemplar frame-based editor uses the Stride language, which is semantically equivalent to Java. Using the data collected via the Blackbox project, we were able to quantify the amount of possible saved time if a frame-based editor were used rather than a text-based one. Our analysis does not include all errors, and thus provides a conservative estimate.

The first part of our analysis evaluated mistakes in student code. The most common mistake in the Blackbox data set is mismatch of brackets and quotation marks, which is intrinsically prevented in frame-based editing. This is also true for several other common syntax mistakes.

We measured time spent fixing errors by looking at the time between the initial occurrence of the error in an attempted compilation, and the next compilation attempt in which the error did not occur. We calculated that in terms of total time spent fixing errors, 28% of programmer's time would be saved in a frame-based editor.

The second part measured time spent adjusting white spaces in the program code, or on edits only involving curly brackets and semicolons – all of which is unnecessary in frame-based editing. This time is calculated by looking at the time between completing the given edit and the previous user action. These edits made up 24% of total editing time – which would again be saved in a frame-based editor.

Although we have focused on a straightforward measure of saved time, the errors may have consequent effects on levels of student motivation and progress. By saving time, students can more easily progress on to serious issues, rather than dealing with syntax issues which do not advance their understanding of programming. Not being interrupted by syntax issues will allow programmers to concentrate better on the semantics. Perhaps more importantly, and more fundamental than the time saved, the avoidance of syntax errors may improve motivation and make programming more widely accessible.

### A. Limitations

A limitation of this study is inherent in the method of focussing on analysing which text-based mistakes will be avoided in frame-based editing while not accounting for possible extra time that may be required in frame-based editing do to particular aspects of that paradigm. However, initial usability results [17] suggest that frame-editing is in fact faster than text-editing. Therefore we expect time to be saved in general editing, besides the error-fixing time saved described in this paper. Whether frame-based editing leads to any errors or confusions which are harder to fix than with text will be the subject of future research.

## References


[1] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[2] A. Altadmri and N. C. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *SIGCSE '15*, 2015, pp. 522–527.

[3] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *ITiCSE '12*, 2012, pp. 75–80.

[4] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *Frontiers In Edu. Conference*, 2014, pp. 2589–2596.

[5] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *11th Working Conf. on Mining Software Repositories*, 2014, pp. 252–261.

[6] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *WiPSCE '15*, 2015.

[7] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Comp. Science Edu.*, vol. 13/4, pp. 249–268, 2003.

[8] M. Kölling, "The Greenfoot programming environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 14:1–14:21, Nov. 2010.

[9] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *SIGCSE '14*, 2014, pp. 223–228.

[10] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *Trans. Comp. Edu.*, vol. 13/4, pp. 19:1–19:40, 2013.

[11] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *ITiCSE '11*, 2011, pp. 208–212.

[12] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *ICER '15*, 2015, pp. 91–99.

[13] J. Jackson, M. Cobb, and C. Carver, "Identifying top Java errors for novice programmers," in *FIE '05*, 2005.

[14] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *SIGCSE '03*, 2003, pp. 153–156.

[15] R. Koitz and W. Slany, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," in *PLATEAU '14*, 2014, pp. 21–30.

[16] L. R. Neal, "Cognition-sensitive design and user modeling for syntax-directed editors," *SIGCHI Bull.*, vol. 18, no. 4, pp. 99–102, May 1986.

[17] F. McKay and M. Kölling, "Predictive modelling for hci problems in novice program editors," in *BCS-HCI '13*, 2013, pp. 35:1–35:6.