Frame-Based Editing: Combining the Best of Blocks and Text Programming

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

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

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

***Abstract*—Editing program code as text has several major weaknesses: syntax errors (such as mismatched braces) interrupt programmer flow and make automated tool support harder, boilerplate code templates have to be typed out, and programmers are responsible for layout. These issues have been known about for decades, but early attempts to address these issues, in the form of structured editors, produced unwieldy, hard-to-use tools which failed to catch on. Recently, however, block-based editors in education like Scratch and Snap! have demonstrated that modern graphical structured editors can provide great benefits for programming novices, including very young age groups. These editors become cumbersome for more advanced users, due to their unbending focus on mouse input for block creation and manipulation, and poor scaling of navigation and manipulation facilities to larger programs. Thus, after a few years, learners tend to move from Scratch to text-based editing.**

In this paper, we present the design and implementation of a novel way to edit programs: frame-based editing. Frame-based editing improves text-based editing by incorporating techniques from block-based editing, and thus provides a suitable follow-on from tools like Scratch. Frame-based editing retains the easy navigation and clearer display of textual code to support manipulation of complex programs, but fuses this with some of the structured editing capabilities that block programming has shown to be viable. The resulting system combines the advantages of text and structured blocks. Preliminary experiments suggest that frame-based editing enables faster program entry than blocks or text, while resulting in fewer syntax errors. We believe it provides an interesting future direction for program editing for learners at all levels of proficiency.

I. Introduction

Most professional programming is done in languages edited and stored as plain text. This representation causes many needless issues and invites unnecessary errors. Fully typing out statements is unnecessary work for a programmer, simple spelling errors in program keywords or in the syntax of control structures hold up a programmer's flow, writers have to arrange layout as well as program structure, changes in formatting (such as bracket placement, indentation or the formatting of continuation lines) are registered as program changes in version control systems, and many other related productivity issues. These issues are especially problematic for beginners, who must memorise control structure syntax and manage indentation while also trying to learn the semantics of programming.

Most IDEs attempt to help with these problems by offering functionality for abbreviations, shortcuts, auto-completion and auto-indentation. However, the support that text-based IDEs can provide is necessarily limited: Indentation may still be invalidated after the fact, keywords and identifiers altered, single braces deleted leaving partial control structures present, etc. Because of the representation of the program as text, the ability of IDEs to ensure the absence of many types of error is limited. When manipulating code by selecting multi-line chunks, it is easy to accidentally miss or include extra braces or extra lines, as single brace characters are small click targets [1]. Professionals are typically able to fix these errors reasonably quickly, but they still interrupt workflow. When these syntax errors are present, the IDE must use complicated parsing algorithms to try to recover from such errors—but there is no need for these errors to be possible in the first place.

Block-based programming languages, such as Scratch [2], Alice [3], or AppInventor [4], have shown that representing code structurally can be done in a usable manner in modern GUI systems, eliminating many of these needless problems. Scratch has no braces to place, and no indentation to decide upon. It will never include half a control structure. However, block-based programming systems are too cumbersome for professional use: they are designed for small programs. Large pieces of code are hard to read, and the purely mouse-based drag-and-drop interactions are too laborious for creating and editing large programs. This makes these editors unattractive to professional developers, despite their advantages in avoiding many errors.

In this paper, we introduce frame-based editing: a new editing paradigm that combines the best features of block- and text-based programming. Frame-based editing uses the graphical structure of block-based programming, but supports keyboard entry for all editing tasks. It combines the improved display of scope from blocks-based editing with a faster entry of program constructs and avoidance of many syntax errors. However, it is not "blocks all the way down"; expressions are entered textually, while retaining a simple structured representation and display, to allow for faster entry than in block-based editing, but still avoid most of the syntax errors which text-based editing allows in expressions.

In this paper we give a detailed description of frame-based editing (sections III–X) its advantages for usability and in education, before we present promising usability results from early versions of our prototype implementation (section XI).

978-1-5090-2504-6/16 $31.00 © 2016 IEEE
DOI 10.1109/LaTiCE.2016.16

II. Related Work

Prior related work can be broadly split into two categories: older work in the 1980s on structured editing, and more recent work in the 2000s on block-based editing. We will tackle each category in turn.

A. Structured Editing

The insight that plain text is not the optimal representation for source code is not new. The idea of directly editing a program's structure (i.e. its abstract syntax tree, AST) was behind the structured editing movement in the 1980s, with the Cornell Program Synthesiser a notable early example [5]. However, structured editors proved too cumbersome and restrictive in practice and never caught on [6]. It is worth remembering, however, that this work on structured editors was performed at the very beginning of the Graphical User Interface (GUI) era when graphical displays and mice were only just catching on, performance was always a difficult issue, and our knowledge of Human-Computer Interaction (HCI) was very limited.

B. Block-based Editing

Block-based editing clearly shares a similar model with structure editing: blocks are graphical AST elements which can be snapped together into trees. We believe one crucial difference between structure editing and block-based editing is that the former was generally bound to a rigid text-based display, with obtuse keyboard support – in contrast, block-based editing replaced this with a more flexible graphical display and mouse-centric interactions which made it easier to understand easier to access for beginners. This advantage of blocks partly arose from the developments in graphical and interface capabilities that occurred in general computing in the 1990s and 2000s. Thus, when Scratch [2] was developed in the 2000s it was able to make good use of high-resolution full-colour displays, drag-and-drop mouse interactions, and also faster processors. Although not directly descended from structured editors, block-based editors are clearly similar in principle, but have demonstrated that such editors *can* be hugely successful in programming education, if designed well.

C. Frame-based Editing

Like Scratch, frame-based editing is a spiritual descendant of structured editing. Frame-based editing explicitly borrows from blocks-based editing, but steers back closer to structured editors, armed with the similar advantages of developing on modern-day high-powered GUI systems with improved knowledge of HCI and interface design.

A previous paper has been published on a much earlier prototype of this work [1], and another paper has described the issues involved in the transition between blocks- and text-based editing, with a short description of how frame-based editing may aid the transition [7]. In this paper we provide a fuller description of frame-based editing in its own right, and discuss how some of its features are important for programming education.

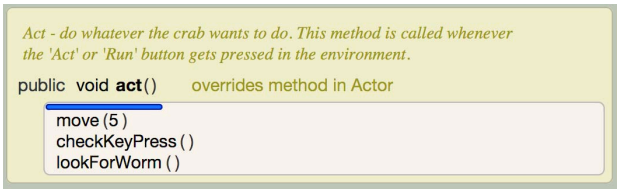

Fig. 1. The frame cursor (horizontal blue bar) in the frame-based editor

III. Frame-Based Editing

Figures 1 and 2 show programs in the frame-based editing interface. The editor supports a language called Stride, a Java-like language used in our educational environment Greenfoot [8]. In this section, we discuss the most important features of our frame-based editor implementation.

Each frame is entered by pressing a single key on the keyboard and—as with blocks—is indivisible. There are no curly braces to mismatch; an if-frame is always closed, and always has a condition present. Syntax errors may be present in the condition or within child frames, but no errors can occur in the *structure* of the if-frame.

Frames are comprised of unmodifiable decoration (such as the "if" caption in an if statement), *frame slots* which contain further frames (e.g. the body of the if statement), and *text slots*, which contain traditional program text (e.g. the condition of the if statement). Many frames omit one or more of these; for example, a method-call frame only has a text slot in which to type the method name, and optional text slots for parameters, but no nested frame slots.

To support keyboard entry and editing of frames, the editor has a *frame cursor*: the horizontal blue bar in Figure 1. This frame cursor is present when focus is in a frame slot, and it behaves in a manner similar to a standard text cursor. It can be moved with the arrow keys and can be used to select frames. Single-key shortcuts are used to enter frames at the position of the frame cursor. When keyboard focus is in a text slot, a traditional text cursor is shown. Whenever the editor window has focus, either a frame cursor or a text cursor is displayed, but never both.

IV. Frame Slots

Frame slots are typically the body of a program construct: the body of an if-statement, a while loop, or a method declaration. Try/catch frames have a frame slot for the body of the try, and then another for each catch clause.

Frame slots are aware of their context and editing commands are context sensitive. The body of a class, for instance, can hold only methods, constructors and variables (fields). It is not possible to enter frames that are disallowed by the language grammar; command keys for illegal frames are ignored and any attempt to drag or paste an illegal frame has no effect. Thus errors such as entering program code outside a method body—indicating a misconception of a novice or a slip by a proficient programmer—are prevented.

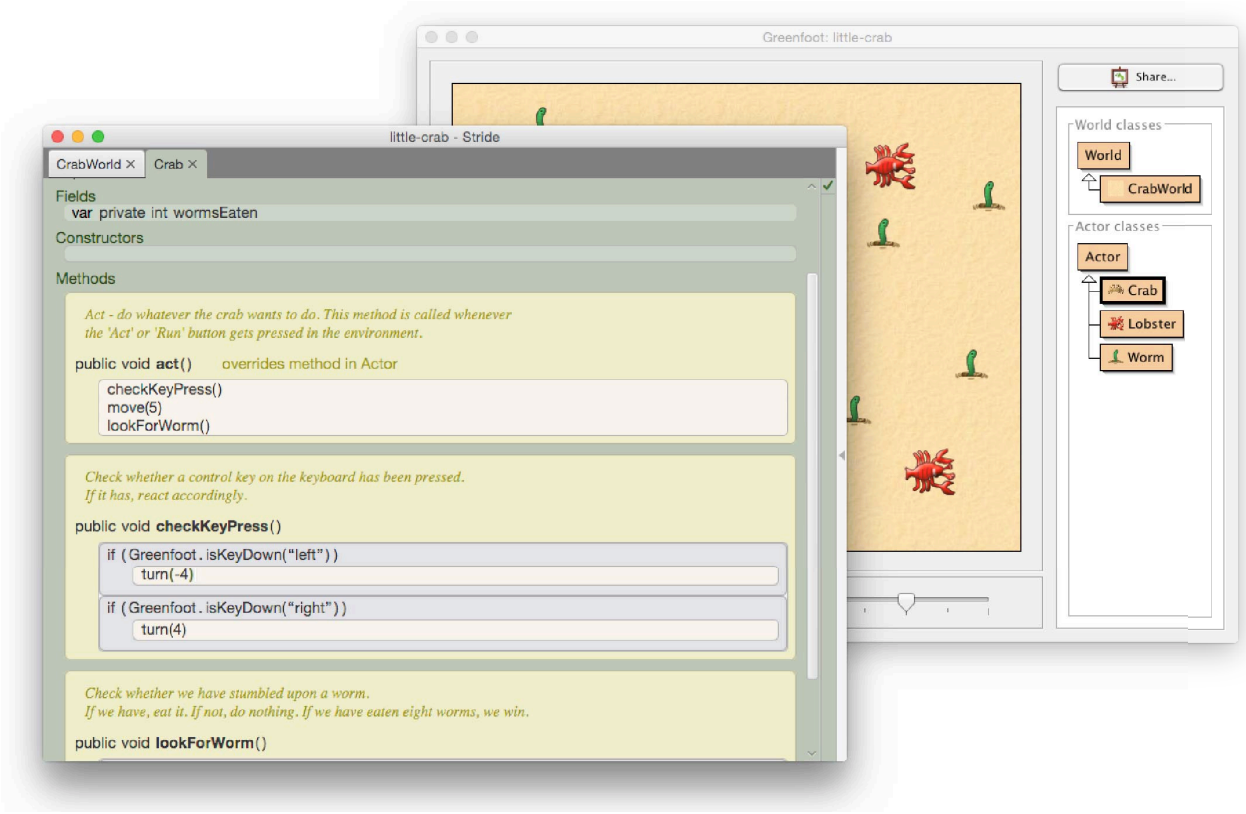

Fig. 2. Frame-based editor interface, integrated into the Greenfoot system

V. The Frame Cursor

The frame cursor (shown in figure 1) is always inside a frame slot. It can be located at the top of the frame slot (including an empty frame slot), or immediately after any frame inside a frame slot. It can be positioned between frames with a mouse click or by using keys to move up or down. By default, up/down arrows move line by line, entering or leaving nested frames in the process. However, using the arrow keys with a modifier key moves the frame cursor at its current scope level, jumping over compound frames in one step. This enables quick navigation. For example, when the cursor is outside a method, this movement navigates at the method level, jumping to each method in turn with a single movement command.

Selection can be performed by dragging the mouse cursor from the current frame cursor to another frame cursor position, by shift-clicking in the same manner, or by using the shift modifier while pressing up/down. Selection is always contained *within a single frame slot*. For example, should a selection start in the body of an if statement, it always ends within the same if-statement. This means that is is not possible to select half a construct; a part of a while loop cannot be selected without selecting its entire extent.

The movement and selection behaviour allows code to be navigated and selected more easily: movement progresses to relevant locations more quickly (positions within, for example, a language keyword or a control structure's syntax decoration are no longer target locations for a cursor) and code selection extends in meaningful chunks with less cursor movement efforts and larger mouse targets.

VI. Frame Manipulation

Frames, such as if-statements, are inserted by pressing a single key ('`i`' for if-statements) when the frame cursor has focus. The syntactic structure—tedious boilerplate in text-based languages—is thus "free". The single-key insertion is not only preferable to traditional typing of constructs, but is also faster than code completion or abbreviation systems in modern IDEs (which usually require at least Ctrl-Space, a few keypresses, and Enter to be typed). Some frames can be modified after entry: an if-statement, for example, can be extended to obtain an else clause by using a single extension keypress '`e`' when the frame cursor is located within the frame to be extended.

Frames are first class citizens in the GUI structure of the editor's interface. They are mouse targets and can easily be dragged from one position to another, without requiring prior selection, just like blocks in block-based systems. (In fact, the drop targets for blocks in block-based programming correspond to frame cursor locations in frame-based editing.) Again, this provides faster and easier editing gestures for very common operations than those available in traditional text editors. Similarly, multi-frame selections can also be dragged and dropped at new locations. The visual appearance of the

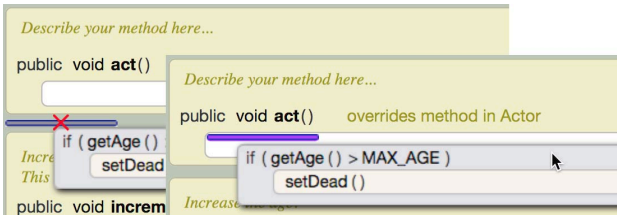

Fig. 3. Dragging a frame, indicating an invalid drop target (left) and a valid drop target (right).

frame cursor during a drag gesture clearly identifies legal and illegal drop targets for frames, as shown in Figure 3. The dragging of custom selections improves on existing block-based systems; previous work [1] found that Scratch's drag and drop manipulation was less effective than desirable due to a lack of expressive precision over which blocks the user wants to manipulate.

Right-clicking on a frame displays a context menu offering options to delete, cut, and copy, as well as several others. Thus, deleting a method requires two clicks (or with the keyboard, pressing the delete/backspace key above/below the method)—an example of a much easier interaction than in text-based editing. In a text editor, selecting exactly one structure is generally more awkward, requires more time due to smaller mouse targets (an example of Fitts' Law [9]), and is sensitive to subtle selection details: whether or not, for example, a trailing newline character or parts of whitespace surrounding the construct is included in the selection will affect its presentation at its drop location, and may require the programmer to adjust layout at the origin of a cut/move. In frame-based programming, the selection will always be entirety of the desired frame(s), and layout is automatically guaranteed to be consistent at both the origin and the target of a moved frame. The programmer does not need to consider line breaks or indentation. While the gain may seem minor at first, it is significant: it not only reduces the need for tedious fine-grain editing, but also removes a cognitive task (ensuring consistent layout) that can easily be automated, freeing the programmer to concentrate on more important issues.

In addition to these interactions, the editor also supports "wrapping" shortcuts: selecting existing frames and then pressing, for example, the '`i`' key surrounds the frames with an if-statement and places keyboard focus in the condition slot for editing. Thus frames can easily be wrapped in a parent frame (e.g. if-statements, loops, try/catch) with just a single key once they have been selected.

Another common edit is to temporarily disable a code segment. In classic text editors, this is achieved by "commenting out" the code in question. This mechanism represents a convenient abuse of the comment construct; the purpose here is not to provide a comment, but something entirely different. The reason a comment symbol (or similarly, a pre-processor directive) is used is merely the lack of a better way. As a result, actual text comments and disabled code segments are visually identical in their presentation in text programming. In frame-based editing, frames can be disabled via an explicit function in the frame's context menu (or by using a keyboard shortcut). The appearance of disabled code is clearly distinct from comments (it is shown blurred, see Figure 4); and as one would expect, it is not compiled and has no effect on the program's behaviour.

VII. Text Slots

Frame-based editing makes use of three different kinds of text slots. Some slots, used when there is a fixed set of options, such as the access specifier for a method, are choice slots. Other slots, such as the type and name slots in a variable declaration, or the name of a method, are identifier slots. The third kind, such as the condition of an if-statement, are structured expression slots. We discuss each of these below.

A. Choice slots

Choice slots behave similarly to combo-boxes in many GUI toolkits: they offer a fixed (usually small) set of values for entry, and one value is always selected. Choice slots in the Stride editor offer selection with the mouse as well as textual entry with optional automatic completion. An example use of choice slots is for the visibility modifiers of fields and methods: only public, protected and private are available and the choice is unique after typing a few characters.

B. Identifier slots

An identifier slot is similar to a standard GUI text field with special behaviour for a set of keys. Users type the contents; a variable or method name slot allows most text to be typed. Invalid characters, such as '!', are ignored. However, some invalid names, such as keywords, are still possible, but a syntax error will be displayed for that slot.

Standard interactions are available for text fields: selection, copy/paste, deletion, etc. The selection cannot extend beyond the text slot. Cursor keys can be used to move the cursor back and forth as usual, but any attempt to go left/right beyond the extent of the slot will move to an adjacent slot. For example, in a text slot for a variable name (e.g. the foo variable in Figure 4), going left from the start will move focus to the last cursor position of the preceding variable type (the int type in the figure), while going right from the end will move focus to the frame cursor immediately after the variable frame.

Text slots are aware of their role and offer code completion accordingly. Type slots offer types (int, String, etc). Name slots in method declarations offer the names of methods from the parent class which could be overridden. Type slots in catch clauses of try/catch frames offer only those types which are subclasses of Throwable (the only valid catchable types in Java). Variable declaration name slots offer no code completion. Thus, frame slots easily offer contextual code completion based on an *a priori* known role. This contrasts with text-based IDEs, which must perform potentially error-prone parsing of partially-written code to offer the same support.

C. Structured expression slots

A significant difference between block-based and frame-based editing is the extent to which the system relies on blocks for the definition of subexpressions. In block-based systems, all expressions are entered as blocks. An addition expression, for example, has its dedicated block which must be dragged

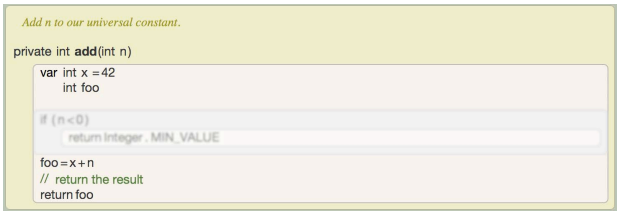

Fig. 4. Disabled frames, such as the if-statement here, are blurred to reflect their status.

into the code structure, and in some systems literals also have to be defined and used as blocks. These interactions are slow (as confirmed by Koitz and Slany [10]), and the interaction overhead results in a little gain.

In contrast, our frame-editor is able to mix frames and structured text. Expressions are entered textually and converted on the fly into structured expressions. Typing a number and a plus symbol into an expression slot, for example, recognises the operator and restructures the expression into an operation with two text fields on either side of the plus operator. The operator itself is not part of either text field.

Typing an opening parenthesis inserts the closing equivalent as well. While this is also common in many existing text editors, the frame editor maintains a stronger link: the bracket pair remains linked in the editor; deleting one parenthesis will automatically remove the other as well, and selecting subexpressions will always include both or neither of the pair. Moreover, explicitly typing the closing parenthesis will overtype the automatically generated one, moving the cursor behind it. This mimics the conceptual model of typing text which professional text-based programmers are used to.

The ability to enter expressions as structured text avoids one of the obvious shortcomings of block-based editors: It combines the avoidance of some errors with the ease and flexibility of text entry, avoiding the mouse-interaction overhead where there is little to gain from using blocks.

An advantage of structured expressions over pure text is the opportunity for better presentation. When the user enters an expression, the editor automatically uses semantically meaningful spacing: higher precedence operators are surrounded with less space, while lower precedence operators are spaced a little more widely. This improves readability of the program text without any additional effort on the programmer's part.

VIII. Error display

We display errors on frame slots using a red wavy underline, recognisable from mis-spellings in word processing systems and error displays in several other professional IDEs. The text of the error is shown in a pop-up when the text slot is focused. The vast majority of errors occur within a *text slot*—unknown types, undeclared variables, type/parameter mismatches, blank slots, and so on. There are very few errors that concern the structure of a frame itself; by design, frames are usually only invalid if one of their contained text slots is invalid or empty. The primary exception to this are errors such as unreachable code, or errors such as invalid overrides or duplicate parameters on methods. Even then, the error location can often be narrowed to a specific text slot (such as displaying method errors on the method name). Wherever possible, we display errors on a text slot as this produces a more localised error; otherwise, in very few cases, an entire frame is highlighted as erroneous.

If an error has an obvious common fix, we display a list of suggested fixes. For example, if a variable is unknown, but there exists a variable with very similar spelling, we offer to correct the use to match the similar declaration. Or if a type is unknown, but there is a type with that name available in a commonly-used package, we offer to add an import for this type.

IX. Easy correction

One problem with many of the old structured editors was not only their usability, but also the difficulty in reversing mistakes while learning the editor. We provide several mechanisms to fix errors. One is a global undo mechanism, allowing users to undo their changes in a standard last-in first-out stack. However, Ko et al. [11] found that programmers don't tend to use global undo, because they want to preserve more recent changes, and cannot do this while undoing older changes. To solve this problem, we provide a local undo mechanism which allows changes to be undone for a specific frame, even if more recent changes were performed since on other frames.

We also provide an easy way to correct accidental keypresses. For example, if a user wants to invoke a method starts with 'e' in an if-block and starts writing the name directly, that will add an 'else' clause containing all the frames that were in the if-block and under the frame cursor. To fix this, the user needs only to press Escape or Backspace to reverse the effect. The same applies when a user presses a wrong command to insert any type of frame: Escape or Backspace reverses the command. Another example is overtyping. As mentioned in section VII-C, to increase usability and maintain the muscle memory that some text-programmers develop, symbols that are automatically inserted to maintain correct syntax, such as closing parentheses, could be overtyped: If the user types a bracket where one already exists, the cursor is moved beyond the bracket, rather than inserting a new bracket.

X. Educational Advantages

In the previous sections we have explained the general design of frame-based programming. Our frame-based editor is deliberately designed for education, and brings with it the advantages of blocks, along with several further advantages:

- **Reduced syntax:** The design of frame-based editing prevents swathes of syntax errors. It is impossible to mismatch brackets or to leave scopes unterminated. Constructs cannot appear in invalid locations: statements cannot appear outside methods, methods cannot be put within statements, and so on.
- **Recognition over recall:** Programming in text requires memorising the syntax for each construct: a switch statement is the keyword switch, followed by round brackets, then curly brackets, in which you write the case keyword then the value then a colon, and so on. Our cheat sheet, like palettes in block-based

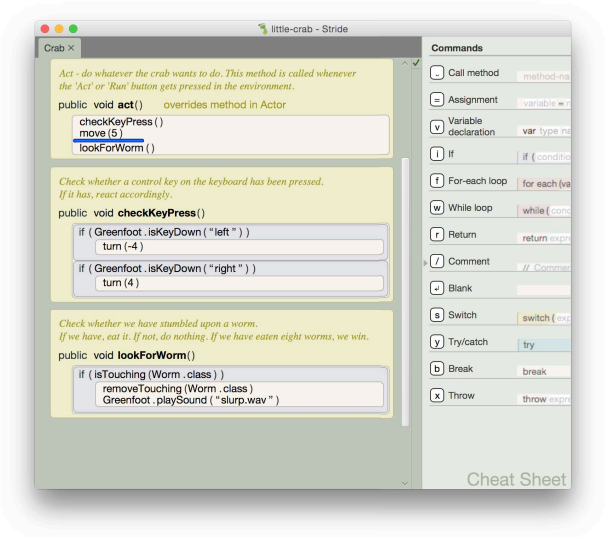

Fig. 5. Cheat Sheet.

editing systems, supports recognition of program statements; Figure 5. The command key 'S' produces a switch statement, as shown in the cheat sheet. Less memorisation will be required by learners, freeing up cognitive load capacity to concentrate on more important issues.

- **Same keystrokes:** Those who continue to study programming will likely transition from frame-based editing into text-based languages. While our expression slots are structured, they can be entered with exactly the same keystrokes as in a text editor. This will ease any later transition into text-based languages.
- **Inheritance display:** Students have several common struggles with object-orientation. One is understanding the difference between classes and objects, which is already dealt with in Greenfoot's interface (classes are the editable code on the right; objects are the items with images which have active behaviour on the left). Another is understanding the mechanics of inheritance. Students program their own Actor subclasses, calling Actor's methods such as move and turn, but they often struggle to understand where these methods "come from". In Stride, the editor has a fold-out display of methods inherited from each superclass, in order to help clarify this; Figure 6. Unlike other editors which only display the code written in that subclass, Stride displays accessible code from superclasses, to aid learning about inheritance.
- **Prompts:** When learning to program, students can often feel lost when writing program code. We try to structure the student's code entry as much as is reasonable. When the frame cursor is focused, the cheat sheet shows available options. When a blank frame is inserted, its empty slots are shown with a hint (e.g. "method-name" for method call frames). When a method call is specified, hints are shown for the

Fig. 6. List of inherited methods.

Task	Scratch	Alice	NetBeans	Frames
Insertion	4.9	6.6	5.1	**1.6**
Modification	5.6	7.1	5.5	**5.0**
Deletion	5.4	2.6	7.8	**2.4**
Movement	5.5	**3.1**	6.0	4.8
Replacement	9.8	8.9	5.1	**2.3**

TABLE I. MEAN TIMES IN SECONDS (1 D.P.) FOR VARIOUS PROGRAM MANIPULATION TASK TYPES [1]. LOWER TIMES ARE BETTER, BEST IN EACH ROW IS HIGHLIGHTED IN BOLD.

names of the parameters. In this way, we try to guide students with what needs to be written next, without ever constraining what they can enter.

XI. PRELIMINARY USABILITY RESULTS

A study by McKay and Kölling [1] investigated editing performance of an early prototype of this work in comparison with a variety of other programming systems. The study compared cognitive models of different program modifications (insertion, modification, deletion, movement and replacement) in a prototype frame-based editor with various other systems, including Scratch, Alice, and NetBeans. Cognitive modelling computes a measure of task time by recording and analysing keystroke level interactions (such as key presses and mouse clicks) as well as "mental" operators, such as eye movement and reading time. The study was performed using CogTool [12], a software system that automates the recording and analysis of interaction sessions. The prototype frame-based editor was found to be the fastest in four out of five categories. Selected results are reproduced here in Table I.

XII. CONCLUSION

In this paper, we have given an overview of frame-based editing, a new interaction paradigm intended to combine the best features of text-based programming and block-based programming. We achieve the easy manipulation of code structures and error avoidance of block-based programming, but use a textual entry for the contents of frames and introduce a new frame cursor mechanism to enable easy and efficient keyboard-driven navigation and manipulation of code.

Frame-based editing operates on the syntax tree of the underlying program and edit operations preserve its structural integrity. Frames are indivisible (no deleting one curly bracket) and expression brackets are always paired. This support, however, does not come at the cost of usability or flexibility. Manipulation in a frame editor requires fewer keypresses and can be performed faster than editing text programs.

Another advantage of frame-based editors is that much of the contextual knowledge used for tool support is known *a priori* and explicitly, rather than deduced from parsing partially-written code, or guessing. This enables better support in IDEs, including better code entry support and error reporting.

Our frame-based editor supports Stride, a new language that is very similar to Java. This does not mean, however, that a language must be designed specifically to be amenable to frame-based editing. Creating a frame-based editor for any existing programming language should be possible, as well as for other structured text formats, such as HTML or XML.

An exemplar frame-based editor for Stride is available (free, and open-source) in the Greenfoot programming environment[1]. Although the authors' primary motivation was improving educational programming environments, we believe that the principles of frame-based editing could also offer improvements for proficient programmers in professional IDEs.

ACKNOWLEDGMENT

The work presented here builds on an earlier prototype by Fraser McKay and Michael Kölling.

REFERENCES

[1] 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*, ser. BCS-HCI '13. BCS, 2013, pp. 35:1–35:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2578048.2578092

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/1868358.1868363

[3] S. Cooper, "The design of Alice," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 15:1–15:16, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1868358.1868362

[4] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor - Create Your Own Android Apps.* O'Reilly, 2011.

[5] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Commun. ACM*, vol. 24, no. 9, pp. 563–573, Sep. 1981. [Online]. Available: http://doi.acm.org/10.1145/358746.358755

[6] L. R. Neal, "Cognition-sensitive design and user modeling for syntax-directed editors," *SIGCHI Bull.*, vol. 18, no. 4, pp. 99–102, May 1986. [Online]. Available: http://doi.acm.org/10.1145/1165387.30866

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

[8] M. Kölling, "The Greenfoot programming environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 14:1–14:21, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1868358.1868361

[9] P. M. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement." *Journal of Experimental Psychology*, vol. 47, no. 6, pp. 381–391, 1954.

[10] R. Koitz and W. Slany, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '14. New York, NY, USA: ACM, 2014, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2688204.2688209

[11] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," in *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '05. New York, NY, USA: ACM, 2005, pp. 1557–1560. [Online]. Available: http://doi.acm.org/10.1145/1056808.1056965

[12] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, "Predictive human performance modeling made easy," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04. New York, NY, USA: ACM, 2004, pp. 455–462. [Online]. Available: http://doi.acm.org/10.1145/985692.985750

[1] Available at http://www.greenfoot.org