Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming

Michael Kölling
University of Kent
School of Computing
mik@kent.ac.uk

Neil C. C. Brown
University of Kent
School of Computing
nccb@kent.ac.uk

Amjad Altadmri
University of Kent
School of Computing
aa803@kent.ac.uk

ABSTRACT

Block-based programming systems, such as Scratch or Alice, are the most popular environments for introducing young children to programming. However, mastery of text-based programming continues to be the educational goal for students who continue to program into their teenage years and beyond. Transitioning across the significant gap between the two editing styles presents a difficult challenge in school-level teaching of programming. We propose a new style of program manipulation to bridge the gap: frame-based editing. Frame-based editing has the resistance to errors and approachability of block-based programming while retaining the flexibility and more conventional programming semantics of text-based programming languages. In this paper, we analyse the issues involved in the transition from blocks to text and argue that they can be overcome by using frame-based editing as an intermediate step. A design and implementation of a frame-based editor is provided.

CCS Concepts

•Applied computing → Text editing; •Software and its engineering → Integrated and visual development environments;

Keywords

Editing, Frame-based editing, Novice programming

1. INTRODUCTION

Plain text as the representation of program text is still the norm for proficient and professional programmers and the accepted educational goal for programming instruction in schools in many countries. This contrasts with direct manipulation programming interfaces, in which users drag and drop "blocks" of program code into place. These block-based interfaces are typically aimed at children learning to program, and are increasingly used in schools with younger age groups. Block-based systems, such as Scratch [16], Snap, AppInventor [24] or Alice [6] now dominate programming in primary and early secondary schools. While these systems provide a very successful and approachable entry into programming for young novices, their widespread use creates a new educational challenge: managing the transition from block-based systems to text-based programming. This transition step presents a significant hurdle which may not be easy to overcome, for learners and teachers alike.

A typical example of the situation described is found in the UK, where programming is now mandatory for all schoolchildren [5]. The UK is one of the leading countries in the more general international trend to strengthen computer science education at school level, and the experience here serves as an early example of a situation that is being or will be faced by other countries around the world.

The recently introduced Computing National Curriculum for England [8] is quite specific about requirements in programming exposure. It states that children aged 7–11 must be taught to "design, write and debug programs that accomplish specific goals... [and] use sequence, selection, and repetition in programs; work with variables and various forms of input and output". At this age level, this requirement is typically met by using block-based programming systems, with Scratch being by far the most popular. This is followed by requirements for 11–14 year-olds who must "use two or more programming languages, at least one of which is textual, to solve a variety of computational problems." For the switch to text-based programming, no single system dominates. Python, often using the IDLE environment, is popular, followed by Java, Visual Basic and various other mainstream programming languages.

These requirements mean that learners face a transition from blocks to text-based programming—typically at an age of about 12 years old. In countries other than England the exact age may vary somewhat, but the challenges this poses in the classroom are similar. Teachers and education researchers have compared these kinds of systems before [22], recognised some of the problems and begun to address the challenge. So far, proposed solutions have been pedagogical, largely based on side by side use of systems and systematic comparison of constructs across this gap [9, 17]. In this paper, we propose an alternative approach: the use of a dedicated programming system that bridges the gap between blocks and text. This system, incorporating a novel interaction style called frame-based editing, combines aspects of blocks and text: it maintains some of the graphical representation advantages, discoverability and error avoidance of blocks while providing the flexibility, keyboard-entry

WiPSCE '15, November 09 - 11, 2015, London, United Kingdom

ISBN 978-1-4503-3753-3/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2818314.2818331

capabilities, and readability of text. It is suitable for development and maintenance of larger programs than would be feasible in block-based systems, while avoiding much of the frustration of struggling with unnecessary syntax errors that text-based environments typically involve.

The frame-based editor presented here supports a new language called Stride, and is integrated in the Greenfoot development environment [14, 15]. Greenfoot is an existing, popular educational programming environment that supports learning and teaching of programming through the development of two-dimensional games and simulations, and previously used Java as the implementation language. A new Greenfoot version now supports Stride in addition to Java. Stride is a Java-like language that differs mostly in the interaction functionality offered for its manipulation.

In this paper, we first discuss the status quo of block- and text-based programming (section 2), followed by an examination of previous work on bridging the gap. In section 3, we identify the key issues involved in managing the transition. Section 4 introduces frame-based editing, discussing some design decisions and presenting examples of frame-based interactions. In section 5, the core contribution of the paper, we provide a detailed examination of how frame-based editing provides an intermediate step between block- and text-based programming.

2. BACKGROUND AND RELATED WORK

Writing programs as plain text is the *de facto* standard for professional programming. All major professional programming languages are written as text: C, Python, Java, etc. The reason for text's historic dominance is a combination of factors:

- History: In the days of text-based computer terminals, there was little displayable alternative.
- Reliable interchange: Text can be edited by all manner of editors (from vim through Visual Studio), so different programmers can edit the same code representation in the environment of their choice.
- Tool compatibility: Plain text makes it easier to work with external tools, such as version control systems. It is also the accepted interchange format to pass program source to third-party compilers.
- Usability: Although text permits many syntax errors, it is more flexible and easier to manipulate than many more rigid representations, such as those found in structure editors.

In the last decade, block-based programming has been widely used to introduce programming to young age groups, and has vibrant online communities [10]. In this isolated educational context, most of the above advantages of text disappear, and blocks' avoidance of syntactic mistakes gives it a compelling advantage over text-based programming. Booth and Stumpf [4] evaluated the relative usability of blocks and text-based programming for the same task set, finding that – for this user group – block-based environments supported easier modification and left users with higher satisfaction.

2.1 Transition from Blocks to Text

The challenge of supporting students in a transition from block-based programming to text-based programming is well known, and different approaches have been tried. These involve either pedagogical strategies, tool modifications or a mixture of the two.

Armoni et al. [2] looked at the effect of having previously learnt Scratch on the learning of text-based programming. Their results indicated a slight improvement in performance, but also suggested that students with exposure to Scratch were better motivated and learnt more quickly.

Powers et al. [20] found that learners had problems in the transition from block-based programming to text-based programming. They suggest that students did not recognise the importance of precise syntax after switching to text. They also point to another often mentioned problem: that block-based systems, because of their use primarily in education and toy-like appearance, are often viewed by students as not being "real" programming, reducing motivation. Although not using blocks, Hundhausen et al. [12] conducted an experiment analysing whether a direct manipulation interface could provide a successful transfer into text-based programming, and found that there was some successful transfer.

Dorling and White [9] describe a series of teaching strategies for managing the transition, including side-by-side development in the two systems. Like many of the other studies, they used reference aids showing fragments of blocks and text programming side by side with equivalent behaviour.

2.2 Tool Support

Although not directly concerned with the transition from blocks to text, many structure editors contain elements relevant for the work presented here. Possibly the closest work to our system is found in a series of educational editors developed from the early 1980s into the 1990s at Carnegie Mellon University [19], including GNOME, MacGnome and ACSE. While the authors found many of their features successful in improving student performance [19], the systems failed to have a noticeable influence on mainstream environments. The novel features in those systems, while promising, were not adopted elsewhere, and the original systems were not ported and maintained to newer operating systems. Nonetheless, some of the features presented in those systems are early ancestors of functionality found in our work.

Homer and Noble [11] explored an environment with both a block-based and textual view of the same code, in Grace. Code can be edited in text or tiled (block) mode, and there is an animated transition available between the two – features also present in Bau's Droplet editor [3], which supports editing JavaScript code either in text or block form. Droplet even keeps the indentation patterns of the text form visible in the block view.

Tillmann et al. [21] describe TouchDevelop, an environment targeted at programming on touch devices such as smartphones or tablets. TouchDevelop now has "skill levels" which present differing views of the code, from blocks to text-like syntax.

Dann et al. [7] frame the transition in terms of mediated transfer, and examine the features added to Alice 3 to better support the transfer. Alice 3 has two features particularly aimed at supporting the transition. The view of the blocks themselves can be changed to offer a Java-like syntax within the block, and a Java preview can be shown alongside the blocks code.

The designers of all of these environments thus concur that there is a benefit to being able to view the same program

in multiple views to aid transfer. Several of the editors also allow editing in both views – although the switch from text back to blocks is only available if the text has no structural syntax errors.

2.3 Summary

The reason that there is currently a "two-stage" system of program editors is that neither block-based nor text-based editors fit all demographics. Block-based editors are too unwieldy for large programs, both in terms of organising large programs, but also in reading and manipulating them. Text-based editors, on the other hand, require a level of maturity and sophistication that young learners do not possess. In the face of the lack of abstraction and manipulation capabilities of those users, problems with syntax errors dominate all creative endeavour.

The most common approach to support the transition between blocks and text is a preview mode, displaying the familiar blocks in textual form, either by a software tool or in printed format on a reference sheet. This is only possible where the systems align closely enough: if-statements are present in all languages, but more specific blocks tend to rely on having the same API in the text system.

The approach presented in this paper differs: Frame-based editing does not present blocks or text, nor a system offering both, but instead is a novel editing method that genuinely incorporates elements from both block-based and text-based interaction models.

3. SIGNIFICANT TRANSITION ISSUES

The exact challenges encountered when transitioning from a block-based to a text-based system may vary in details, depending on the exact nature of both systems involved in the transition. However, most of the fundamental problems are intrinsic in these types of system and can be identified in the general case. Below, we list the most fundamental issues faced by learners progressing between these environments.

1. **Readability.** Block-based syntax is significantly easier to read for novices than common text-based programming languages. Even after exposure to both kinds of system, some aspects of blocks remain more readable for relative novices [23]. This has multiple aspects: block-based systems tend to use keywords rather than symbols or punctuation in commands, the keywords used tend to be closer to natural language, a more variable operand and operator syntax and sequence is used that partly mimics the grammar of natural languages, and the graphical representation of scopes (using bracket-style blocks) is easier to interpret than traditional text-based scope notation.

2. **Memorisation of commands.** In block-based systems, all available commands are visually represented on screen in a block catalogue. Novice users can browse all available commands, remind themselves of vaguely remembered concepts or gain inspiration from discovering new ones. The visibility of the constructs, coupled with usually more humane naming resembling natural language more closely than many text-based programming languages, allows recognition over recall, greatly helping novices using the tool and encouraging experimentation. In text-based systems, programming constructs must be known, and their representation must be recalled from memory.

3. **Memorisation of syntax.** In addition to recalling the existence and function of a programming construct in text-based systems, users also have to remember its exact syntax. It is not enough to know that a for-loop exists and what it is used for, the programmer must also know its keyword and where exactly to put the commas, semicolons or brackets. For beginners, this is an additional cognitive challenge beyond recalling the command itself. An added challenge for non-English speakers in many cases is remembering the keyword in a foreign language, as text-based systems are almost always used with English language keywords. For many learners, these keywords are not initially understood, losing the advantage of help from colloquial understanding of the terms. In these cases, keywords become meaningless strings to be memorised. Some block-based systems are available with localised block names, removing the language hurdle. But even when used in a foreign language, we speculate that the recognition characteristic (rather than necessary recall) reduces the problem introduced by lack of language understanding.

4. **Typing/spelling.** In addition to *knowing* the syntax (the previous point), programmers must also *create* it. The pure mechanical act of typing the program text poses an additional hurdle. For young learners, this can be significant, both in cognitive as well as in motor terms. While Scratch, for example, is very successfully used with 10-year-olds, this target group often does not possess sufficient typing skills to be able to produce textual artefacts of the required length in reasonable time. Even for older learners who have the ability to type, the necessity to do so adds an extra load (often cognitive load because typing skills are often rudimentary), and cognitive distractions when inevitable typing errors have to be corrected.

5. **Number of commands.** The available commands in typical text-based systems are not only invisible, they are also much larger in number (and potentially infinite, if we consider possible inclusion of third party libraries). The standard Java library, for example, includes tens of thousands of methods. Strategies that work for exploring block-based systems, such as memorisation and investigative browsing, fail for these larger systems. Novice users have to learn to explore and navigate extensive documentation.

6. **Prototype versus definition.** The format of available method calls listed in block-based languages are prototypes of invocations of these methods. When a user drags a command into the program, they have a syntactically correct call in place – only possible parameters need to be filled in or adjusted. On the other hand, the list of available methods in text-based languages is typically provided in the form of method definitions (commented signatures). These differ in syntax from the method invocation. Programmers have to construct the syntax of the method invocation on their own, deriving the details from the method definitions. For novices in these languages, this is a non-trivial

step and requires understanding of a number of non-trivial concepts (including return values, parameters and types).

7. **Matching identifiers.** A specific subgroup of the spelling challenge is found in dealing with user-defined identifiers. When identifiers, such as variable names, are referenced, most block-based systems offer a selection of known identifiers via user interface elements, such as drop-down menus. In text-based systems, users must not only distinguish between the definition of and the reference to an identifier, they must also understand that correct spelling is important and master its input. In many systems, this includes dealing with case sensitive syntax – particularly difficult for those whose native language has no concept of case.

8. **Grouping.** Another subgroup of spelling that deserves explicit mention as a separate challenge is the treatment of compound statements and definition of scope. Missing or superfluous brackets in languages that define scope with explicit symbols, or incorrect indentation in languages where scope is defined through indentation, are among the most common and persistent syntax errors holding up novice programmers [1]. It seems that *knowing* about the mechanism for scope is not enough; the mechanics of arranging and maintaining scope are still challenging in these languages, even when the principle is understood. This problem does not exist in block-based languages.

9. **Writing expressions.** The writing of expressions, such as multi-operator arithmetic expressions, is significantly more difficult in text-based systems where the format of the complete expression is not apparent through its use. In block-based systems, blocks representing expressions clearly indicate number and position of parameters and in many systems also its type.

10. **Understanding types.** In many block-based systems, only few types are used (often just distinguishing booleans from other data values) and users can successfully create operational programs without gaining a meaningful understanding of the concept of data types. In text-based systems (even in dynamically typed languages) an understanding of data types is typically necessary for the construction of even simple meaningful programs.

11. **Interpreting error messages.** Many block-based systems manage to avoid almost all syntax errors. In systems where syntax errors occur, these are typically narrowly localised, and a message is displayed pointing accurately to the origin of the error in the source code. In text-based systems, on the other hand, where structure is derived from typed representation, error messages are notoriously vague, misplaced from their causal location and wrong in their wording. Interpreting these error messages is a non-trivial skill that takes novices considerable time to master.

12. **Managing layout.** Layout, such as indentation and spacing, is done automatically in block-based systems and must be done manually in text-based programming languages. Even with editors that auto-indent

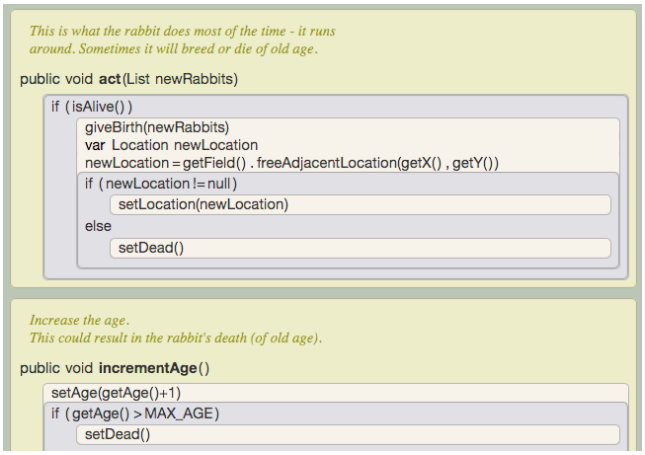

Figure 1: Program representation in a frame-based editor

lines, it is possible – and, in fact, very common – for beginners to create incorrect indentation. In languages where indentation is semantically meaningful (usually to indicate scope), incorrect indentation leads to syntactic or semantic errors. In languages that ignore whitespace, incorrect indentation "just" affects readability. This, however, also creates problems as lack of readability often leads to errors in the program.

13. **Changing programming paradigm.** Often the progression from blocks to text also involves a change of programming paradigm. Scratch, for example, is object-based (program scripts specify behaviour of individual objects), while the most common text-based languages are either procedural or class-based (programs define classes of objects, and instantiation is required).

Not all of these issues are encountered to the same degree for any particular pairing of block-based and text-based system, but any transition typically involves most of them. What is worse, many of these problems amplify each other when encountered in conjunction. Together, these issues present a significant challenge to novices, even if they were aiming at creating programs of no higher complexity than in the previous system.

4. FRAME-BASED EDITING

Frame-based editing aims to combine many of the beneficial aspects of block-based and text-based systems into a single interface. The aim is to achieve the small-scale readability and some of the error avoidance and discoverability of blocks, while retaining the flexibility, manipulation efficiency, keyboard control and large-scale readability of text.

Figure 1 shows a segment of a program in a frame-based editor for a new, Java-like language called Stride, integrated into the Greenfoot system since 2015. The editor uses some graphical elements (shapes and colours) to present aspects where graphics have advantages over characters. Overall, however, the presentation maintains the look of a program as essentially a textual, if coloured, document.

Greenfoot, the system our current implementation is integrated in, is an introductory development environment

Figure 2: A frame with empty slots

aimed at beginning programmers aged from about 14 years old upwards. We will use this implementation as the prototype to discuss the concepts of frame-based editing.

4.1 Frames

All statements in Stride are represented by frames. This is true for compound statements, whose frames are painted with distinct background colour and a visible border, as well as simpler, one-line statements that appear in a default colour and without a visible border.

Frames are inserted with a single keypress and are either entirely present or absent; it is not possible to have half a statement in the code (such as, for example, an if-statement with a missing closing bracket in a traditional text editor).

This means that scope is represented as a frame: a graphical box, 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.

The advantages are fairly obvious: Recognising the extent – beginning and end – of a scope is much easier and quicker than in textual representation. Programmers do not need to determine which closing bracket matches which opening one, and no additional confusion can be created by misleading indentation. The overall visual overhead, however, is much less than for typical block-based systems with their strong colours, 3D and shadow effects, and heavily nested blocks for expressions.

4.2 Slots

Frames may contain *slots* for further entry of code. Our frames distinguish between *text slots* and *frame slots*, for the insertion of text or nested frames, respectively (Figure 2).

At the statement level, where maintaining structure is helpful, frames are assembled in a manner similar to block-based languages (albeit largely keyboard driven; see below). However, at the expression level the interaction mechanics differ. In block-based editors the condition of an if-statement, for example, is made up of more blocks: expression blocks. Multi-operator expressions must be built by dragging and dropping multiple expression blocks and assembling them in nested configurations. The calculation of the hypotenuse of a triangle, for example, can be seen in Figure 3: Although not very long, it had to be assembled by dragging eight blocks together. Koitz and Slany [13] noted this as a problem with block-based systems and proposed a different editor for expressions/formulae on mobile devices.

In contrast, the condition in an if-statement's frame is an *expression slot*: it is written as text, in a manner very similar to that used in a plain text editor, although some structure is still maintained (brackets and quotes, for example, can never be mismatched and are always inserted and deleted in pairs).

Figure 3: Example of a nested expression in Scratch: the above expression involves eight blocks.

Figure 4: A frame cursor and a text cursor

4.3 The Frame Cursor

A frame editor always displays one cursor, and the cursor is always in a slot. Two different types of cursor exist, depending on what kind of slot has focus: When the cursor is in a frame slot, a frame cursor is shown (Figure 4, left). Inside a text slot, the cursor changes to a text cursor (Figure 4, right). It is not possible to have a frame cursor and a text cursor at the same time.

Interpretation of input differs with the two different cursors: when the frame cursor is visible, keyboard input is interpreted as commands, and corresponding frames are inserted. When the text cursor is visible, keys insert their own character literally, as in a traditional text editor.

The existence of the frame cursor is a significant difference of frame-based editing to both block- and text-based systems. Block-based editors usually have little or no keyboard support. Blocks cannot be inserted with keys; instead, the destination for new blocks is selected with a mouse drag. The frame cursor improves this by providing a focal point for navigation, selection and insertion, in a similar way as a text cursor does for text entry.

The frame cursor can be moved using the cursor keys and in combination with the shift modifier can create frame selections. Selections can be cut, copied, pasted, deleted or dragged to a new position. This allows for more flexible manipulation of nested blocks, which was shown to have a high viscosity overhead using the mouse-based block manipulation in Scratch [18].

4.4 Insertion

In text-based programming, code is inserted by typing the syntactic representation, sometimes with assistance of code completion in IDEs. In block-based systems, blocks are dragged into the program from a prototype palette. In our frame editor, there are two primary ways to insert new frames.

The first has already been mentioned: Single-key commands insert new frames at the position of the frame cursor. If a selection is present, inserting a frame surrounds the selection with the new frame. For example, selecting a group of three statements and using the 'i'-command (inserting an if-frame) surrounds the selected statements with the if-statement, placing the selection into the body.

The other option to insert frames is to select them from

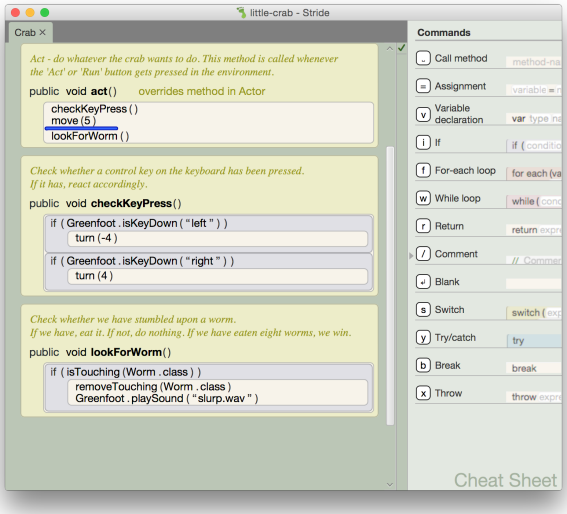

Figure 5: The editor's cheat sheet

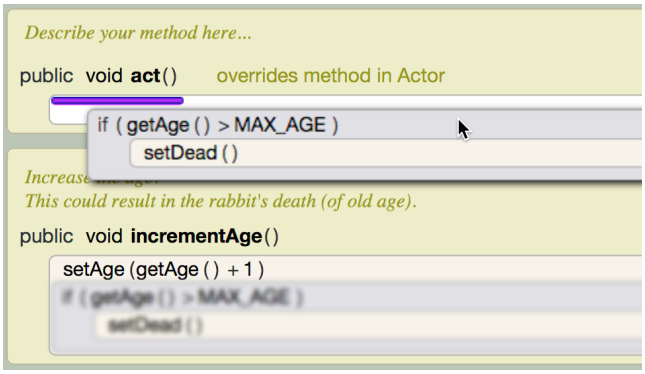

Figure 6: Dragging a frame

the "cheat sheet". The cheat sheet is an optionally displayed sidebar (Figure 5) that shows all frames available for insertion. Frame prototypes can be clicked, and are then inserted at the current frame cursor position.

In addition to allowing frame entry via mouse clicks, the cheat sheet also serves to make available frames discoverable and to inform users of their associated keyboard commands. This allows recognition of constructs combined with fast keyboard entry.

4.5 Manipulation

Frames are first class entities in the editor's user interface. They have associated context menus that can be triggered via a right mouse click, offering functions to, for instance, delete or temporarily disable a frame.

Frames can also be dragged and dropped using the mouse, in a manner similar to blocks in block-based systems. When frames are being dragged, a temporary frame cursor indicates valid drop locations (Figure 6); frames can only be dropped where they are syntactically valid.

In traditional text editors, similar functionality is usually available: Text can be selected, and the selected text can be dragged and dropped at a different location.

Again, the different unit of manipulation (editing frames instead of editing characters) leads to various advantages in our editor:

- 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. In the frame editor, only complete frames can be dragged. (This includes simple one-line statements – these are also frames.)
- Selecting a complete multi-line statement in a text editor typically requires careful targeting with the mouse, making this a high-overhead operation. It usually requires careful consideration of including whitespace, indentation or trailing return characters in the selection, resulting in different formatting for subtly different choices. No such consideration and fine targeting is required in a frame editor.
- In text editors, dragged text may be dropped anywhere, again potentially breaking program structure. The majority of potential drop locations by far are syntactically invalid, yet no help is provided by the editor in identifying the few valid ones. In the frame editor, frames may be dropped only at locations where they maintain a syntactically correct structure.

In summary, the frame-based editor for the Stride language combines aspects from block- and text-based systems, while adding some novel concepts not found in either of those systems. In doing so, it positions itself between blocks and text, offering advantages from both not previously available simultaneously, and providing a stepping stone between these two modes of manipulation.

5. FRAMES AS A STEPPING STONE

In section 3, we listed a number of fundamental issues that pose challenges for many learners transitioning from block-based to text-based systems. We propose to use a frame-based system as a stepping stone between these two traditional types of system. This reduces the incidental complexities of the step from one system to the other.

When introducing a frame-based system as an interim step, learners progress through two transitions instead of one. Figure 7 illustrates this, showing which of the issues identified above are encountered at which part of the transition. In this figure, we maintain the numbering of issues introduced in section 3.

Two distinct kinds of advantage result from splitting the transition in two:

- Simply reducing the *number* of issues to be mastered at each step is a considerable advantage (even though the total number across both transitions remains the same). Many of the most difficult problems for beginner stem not from a single issue, but a combination and concurrence of multiple characteristics. The need to memorise and type syntax in text-based systems, for example, combined with the poor quality of error messages, poses a significant challenge. The combination of these characteristics amplifies the problem.
- The design of the frame editor includes numerous aspects that significantly reduce the difficulty of many

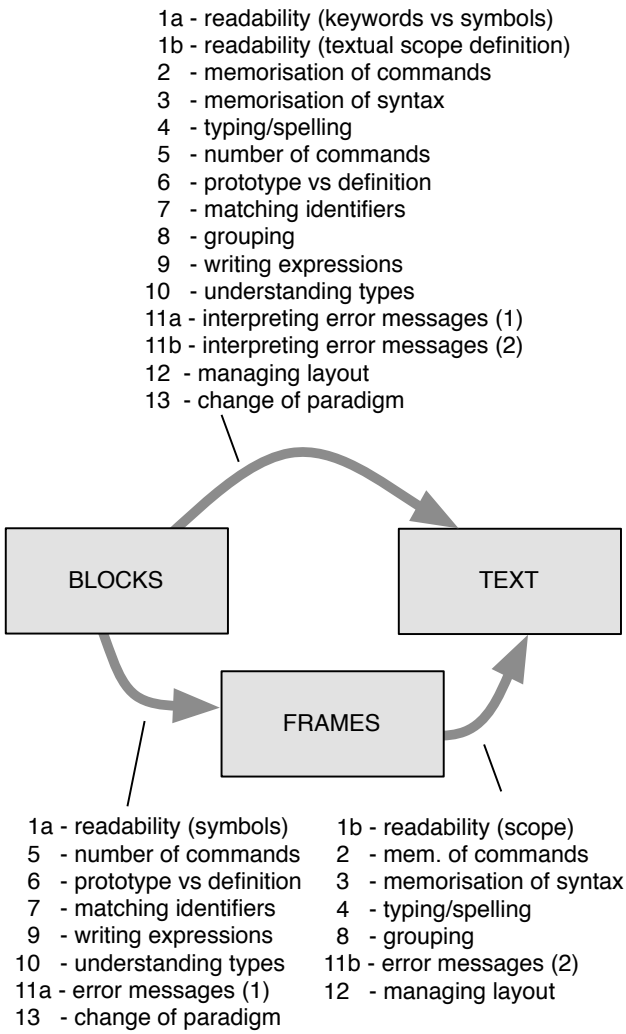

Figure 7: Transitioning in one step versus two steps

individual transition issues. Thus, the issues at each step are not only fewer in number, but also easier to master. This is discussed in more detail below.

For the following discussion, we again assume our Stride editor as the example of a frame-based editor. Details of the discussion may be different with frame-based editors for other languages, but the principles are general enough for this discussion to be useful in the general case.

5.1 Transition from Blocks to Frames

Figure 7 shows, in the bottom left, which of the issues identified will be encountered in the transition from a block-based to a frame-based system.

5.1.1 Readability

Issue of small scale readability (the readability of single statements, *Issue 1*) are partly encountered in the transition to frames and partly in the transition from frames to text. The aspect that is encountered in the former is the use of symbols (rather than keywords) for some functions and the more formal format for statements (being further removed from natural language). However, the impact of

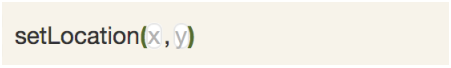

Figure 8: A method call frame providing prompts for parameters

this aspect is less severe that in the block-to-text transition, because much of the syntax does not need to be memorised and typed, but is provided implicitly as decoration of the frames. The readability aspect that is delayed to the transition to text is the representation of scopes using textual symbols.

Large scale readability (the readability of large portions of code or whole programs) is significantly improved, since the visual overhead of the Stride editor (colouring, number and prominence of blocks) is greatly reduced, methods are arranged sequentially (as in text-based systems) and better navigation is provided.

5.1.2 Number of Commands

The issues of the greatly increased number of available commands (*Issue 5*) is encountered here, since all methods from the Java library are available for invocation in Stride. However, we can distinguish two independent aspects here: The number of different *kinds* of command is actually reduced (since all possible method calls are presented as individual commands in block-based systems, whereas a method call is a single generic command in Stride) while the number of available *method calls* is greatly increased.

Because of this, the number of different available commands in Stride is actually quite small – the Cheat Sheet in Figure 5 does indeed show a complete list of all commands that can be used here.

The distinction between kinds of command and different methods eases the transition considerably, since it separates a combined challenge (*"Which command should I use?"*) into two separate, logically distinct parts (*"Which type of command should I use?"* and *"If it is a method call, what is the method?"*).

In addition, well designed contextual code completion in the Stride editor helps in identifying and selecting methods for invocation. On the first invocation of code completion, only a small handpicked list of commonly used methods is shown. This provides novices with assistance without overwhelming them. They can then choose to show a full list if required.

5.1.3 Prototype Versus Definition

The fact that available methods are presented in form of a method definition in the available documentation rather than as invocation prototypes (*Issue 6*) is encountered here. Again, this issue is made easier to master, since the frame for method invocation automatically offers the correct invocation syntax. This does not need to be memorised. Once a method name is typed for invocation (or selected from a code completion menu), the Stride editor automatically creates text slots for the correct number of parameters and adds annotations showing the parameter names to aid in providing parameter values (Figure 8).

5.1.4 Matching Identifiers

The typing of identifiers (*Issue 7*) needs to be mastered at

this step. This can be eased in frame-based editors through contextual code completion. Since frame based editors know the context of an identifier to be typed (they know, for example, whether the cursor is in the name slot of a method call or the type slot of a variable declaration), code completion can be implemented offering identifiers of the right kind for insertion (method names and type names, respectively).

5.1.5 Writing Expressions

Expressions are written as text in frame-based editors, rather than assembled as blocks (*Issue 9*). This change has to be mastered at this stage. Again, the frame editor provides more help than text editors in maintaining structures that are written as pairs of symbols (parentheses and quotes), making this step a little easier. While this transition itself presents a challenge (because the mechanism is new), it can also be seen as an improvement over blocks because the efficiency gain from textual expression entry can outweigh the advantages in recognition and error avoidance of using expression blocks.

5.1.6 Understanding Types

The challenge to understand and deal with data types (*Issue 10*) is encountered here. The Stride editor provides some additional help with this by offering contextual code completion on types: If the cursor is in a slot that expects a data type, known type names are offered for completion. The offered list can be restricted if more context is known. In a frame to catch an exception, for instance, only *Exception* and its subtypes are offered for insertion.

5.1.7 Error Messages

The challenge of dealing with error messages (*Issue 11*) is split between the two transition steps. In Stride, learners will encounter more errors than in common block-based systems, however, the location and accuracy of the error is much better than in text-based systems (thus, dealing with incorrectly worded or positioned messages is deferred to the next transition). Also, the number of errors made is significantly lower than in text-based systems, since a considerable number of syntax errors cannot be made.

This separation leads users more gently towards reading and interpreting error messages.

5.1.8 Change of Paradigm

In many block-to-frame combinations, a change of paradigm will be encountered (*Issue 13*). For example, if learners transition from Scratch or Snap to Stride, they will move from an object-based to a class-based paradigm. (Other combinations, for example Alice-to-Stride, would not encounter this issue, since Alice is a block-based and class-based system.)

While this necessitates the learning of a potentially new programming model, it also provides the advantage of the greater power and flexibility that comes with class-based systems. The restrictions in most block-based systems, both in expressiveness and performance, do not exist in Stride, and the scope and scale of programs that can be attempted in Stride is the same as in Java and other professional languages.

5.2 Transition from Frames to Text

The remainder of the issues listed in section 3 are encountered when transitioning from frames to text. These are the reading and writing of scopes defined by textual symbols (*Issues 1 and 8*), required memorisation of commands and their syntax (*Issues 2 and 3*), the need to type and spell the syntax explicitly character by character (*Issue 4*), including whitespace and layout (*Issue 12*), and the need to deal with unhelpful error messages (*Issue 11*, see Figure 7).

These transition steps are significantly easier than they were in the block-to-text transition, especially in the transition from Stride to Java, because of a number of characteristics of Stride, its frame-based editor, and its integration into the Greenfoot system:

- Stride syntax is very close to Java syntax. While it is a significant difference to type the syntax out in full rather than using frames, students have seen much of the syntax for a long time before being required to type it themselves. Thus, a level of familiarity exists before syntax has to be reproduced exactly. (Stride differs from Java mainly through the absence of semicolons and curly brackets.) Frame editors could be created to resemble syntax of other languages, either approximately or exactly.

- Greenfoot supports both Stride and Java, and classes written in each can be used in the same project. Both languages are programmed against the same API (the Greenfoot API and the Standard Java Library). Thus, when transitioning into Java, users are already familiar with the API, and the semantic problems (choosing the right methods to call) can be separated from the syntactic ones.

- Because of the similarity of syntax and semantics, frame-based programs can be compared to Java statement by statement; the change is not so much a *translation*, but rather a *transliteration*: near-identical code with a slightly altered syntax, entered using a different interaction model.

- To further facilitate this transition, Greenfoot provides a *Java preview* feature in its Stride editor which shows an animated transition from Stride to Java. This animation can be run forwards and backwards, and doing so repeatedly gives students the opportunity to directly compare the two syntaxes.

In Greenfoot, users can start developing projects entirely in Stride, and later transition to writing very similar code in Java. The change to Java can happen through the creation of separate Java projects, or on a class-by-class basis, mixing Stride and Java classes in the same project.

5.3 Logic First, Syntax Later

We have seen that the use of a frame editor as a stepping stone separates the transition issues into two groups, to be addressed at different times. The issues identified also fall into two categories separated by their nature: some are caused by intrinsic complexities of the programming model, others are caused by accidental complexities.

Intrinsic complexities are those that are inherent in the nature of the programming model. These cannot be avoided if one wants to learn to program in that model, and mastering these issues is a useful learning experience. Examples of these are the number of available commands, reading standard format documentation (class and method definitions),

understanding a type system, and mastering a class-based paradigm. Learning each of these concepts is a useful step.

Accidental complexities are those caused by poor syntax or tools used for creating the artefact, but not intrinsically related to useful aspects of the challenge at hand (understanding object-oriented programming). Examples from our list include poor readability, having to memorise syntax, having to type commands, and dealing with poor error messages.

It is important to note that all challenges representing intrinsic complexities (those that are useful to master) are included in the first transition step, from blocks to frames. Thus, learning to program in Stride provides all the educational benefits of the complete transition, while avoiding a number of the accidental (syntactic) complexities. Stride programs are equivalent to programs in professional text-based languages in their programming model, expressiveness and performance, removing the ceiling imposed by most block-based systems. The transition to Stride addresses all challenges related to logic, reaping all their benefits, while relegating pure syntactic and mechanical challenges that offer little reward to a later date.

The transition to pure text systems at a later date is then motivated purely by the desire to learn to use other popular systems, potentially using other paradigms or platforms.

6. LIMITATIONS AND DRAWBACKS

One of the most obvious downsides of the proposed approach is that it requires two transitions to new editing models rather than just one, and the mastering of three systems rather than two. There is, of course, always a possibility (or even likelihood) that using a third system adds new, additional transition issues to the ones already present. Inserting a frame-based editing model adds at least the mastery of the associated editing interactions as an additional challenge. Thus, the overall amount of material to be mastered increases.

While we speculate that the benefits of easing the transitions using a frame editor offset this growth in material to be mastered, we currently have no evidence to support such an assertion. This aspect remains speculation at this stage.

In practice, programming education now often starts at an age below 10 years old and continues until late teenage years and beyond. In these cases, the system progression employed often already includes three or more different systems. In these cases, the overhead of using a frame-based system may not be any more than what is already in place.

Another potential problem is the limited availability of frame editors. Currently, Greenfoot's Stride editor is the only system closely implementing this model. Stride uses an object-oriented programming paradigm and static types. If a desired progression involved, for example, Scratch as the starting point and a procedural use of Python at its end, then Stride would impose additional concepts unnecessary to support the progression. Both the block-based and text-based systems in this example do not make use of classes or static types, so Stride's use of them would pose a potential distraction from a straight progression path. If, on the other hand, the end point is Java, then the progression works very well, as Stride is aligned with this language.

It is our hope that designers of other systems will apply the ideas of frame-based editing to create frame editors for other languages, such as for a Python-like language, to support the transition across more varied systems. There is no specific feature that ties frame-based editing to a particular programming paradigm; the reason that the Stride editor is integrated into Greenfoot and targets Java as its successor language is coincidental, and driven only by the authors' involvement with this particular environment.

7. CONCLUSIONS

Frame-based editing is a new hybrid editing mode, combining features of block-based and text-based programming. As such, it is suitable as a transitional step between blocks and text in a sequence of programming systems for novice programmers.

We have discussed the design of frame-based editing in general and provided an implementation supporting Stride, a new frame-based language similar to Java. In this paper, we have discussed in detail the benefits arising from using a frame-based editing system, not only in separating the transition to text into two steps, but also in easing the learning of new concepts significantly in many cases by providing better support.

While we have discussed the transition as a three-system sequence, in practice variations of this may often be employed. If, for example, learners are beginning to program at an age of 12 or older, block-based systems could be skipped altogether, in favour of using frame-based programming from the start. Since frame-based editors remove some of the most difficult mechanics of typing and syntax, they can be used from a younger age than pure text-based systems.

If, on the other hand, the educational goal is to simply provide an insight into programming as a discipline, without the intention to progress to more professional or extensive programming, then a text-based system might not be used at all. Frame-based systems are capable of illustrating all important concepts of programming, and text-based editors may not offer an important enough learning experience in their own right for generalist programming education.

There is, in fact, no clear reason why text-based programming would have any advantage at all over frame editors if good implementations of these existed for popular languages and the tool chain were adapted to operate on the necessary storage formats. Frame-based programming may eventually replace text-based programming as the dominant editing model even for professional programmers, entirely removing the need for a second transition.

Until this ambitious goal is achieved, frame-based editing instead offers an ideal stepping stone between block-based programming and text-based programming.

8. REFERENCES

[1] A. Altadmri and N. C. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527. ACM, 2015.

[2] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. From scratch to "real" programming. *Trans. Comput. Educ.*, 14(4):25:1–25:15, Feb. 2015.

[3] D. Bau. Droplet, a blocks-based editor for text code. *J. Comput. Sci. Coll.*, 30(6):138–144, June 2015.

[4] T. Booth and S. Stumpf. End-user experiences of visual and textual programming environments for

arduino. In Y. Dittrich, M. Burnett, A. Mørch, and D. Redmiles, editors, *End-User Development*, volume 7897 of *Lecture Notes in Computer Science*, pages 25–39. Springer Berlin Heidelberg, 2013.

[5] N. C. C. Brown, S. Sentance, T. Crick, and S. Humphreys. Restart: The resurgence of computer science in uk schools. *Trans. Comput. Educ.*, 14(2):9:1–9:22, June 2014.

[6] S. Cooper. The design of Alice. *Trans. Comput. Educ.*, 10(4):15:1–15:16, Nov. 2010.

[7] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 141–146, New York, NY, USA, 2012. ACM.

[8] Department for Education. *National Curriculum from September 2014*. 2013.

[9] M. Dorling and D. White. Scratch: A way to logo and python. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 191–196, New York, NY, USA, 2015. ACM.

[10] D. A. Fields, M. Giang, and Y. Kafai. Programming in the wild: Trends in youth computational participation in the online scratch community. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, WiPSCE '14, pages 2–11, New York, NY, USA, 2014. ACM.

[11] M. Homer and J. Noble. Combining tiled and textual views of code. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 1–10, Sept 2014.

[12] C. D. Hundhausen, S. F. Farley, and J. L. Brown. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Trans. Comput.-Hum. Interact.*, 16(3):13:1–13:40, Sept. 2009.

[13] 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*, PLATEAU '14, pages 21–30, New York, NY, USA, 2014. ACM.

[14] M. Kölling. Greenfoot: A highly graphical ide for learning object-oriented programming. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 327–327, New York, NY, USA, 2008. ACM.

[15] M. Kölling. The Greenfoot programming environment. *Trans. Comput. Educ.*, 10(4):14:1–14:21, Nov. 2010.

[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[17] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai. Language migration in non-cs introductory programming through mutual language translation environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 185–190, New York, NY, USA, 2015. ACM.

[18] 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*, BCS-HCI '13, pages 35:1–35:6, Swinton, UK, 2013. British Computer Society.

[19] P. Miller, J. Pane, G. Meter, and S. Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[20] K. Powers, S. Ecott, and L. M. Hirshfield. Through the looking glass: Teaching cs0 with alice. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 213–217, New York, NY, USA, 2007. ACM.

[21] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie. The future of teaching programming is on mobile devices. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 156–161, New York, NY, USA, 2012. ACM.

[22] I. Utting, S. Cooper, M. Kölling, J. Maloney, and M. Resnick. Alice, greenfoot, and scratch – a discussion. *Trans. Comput. Educ.*, 10(4):17:1–17:11, Nov. 2010.

[23] D. Weintrop and U. Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 101–110, New York, NY, USA, 2015. ACM.

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