Stride in BlueJ – Computing for All in an Educational IDE

Michael Kölling
King's College London
London, UK
michael.kolling@kcl.ac.uk

Neil C. C. Brown
King's College London
London, UK
neil.c.c.brown@kcl.ac.uk

Hamza Hamza
King's College London
London, UK
hamza.hamza@kcl.ac.uk

Davin McCall
King's College London
London, UK
davin.mccall@kcl.ac.uk

ABSTRACT

Block-based programming languages and environments have several benefits for introductory programming courses, compared to more traditional text-based languages. In particular, blocks remove the burden of learning language syntax and dealing with syntax-related errors. Many blocks-based environments are tightly focused on developing graphical games, stories and simulations, while the more general programming environments are typically text-based. In this tool paper, we describe the incorporation of a Stride editor within the BlueJ programming environment. Stride is a frame-based programming language, intended to combine the best of blocks and text programming, usable both as a stepping stone towards text-based languages and as a comprehensive language in its own right. The incorporation of Stride into BlueJ brings some aspects of block programming into a general purpose educational environment.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **General and reference** → *Design*;

KEYWORDS

BlueJ; Stride; Frame-based Editing

ACM Reference Format:
Michael Kölling, Neil C. C. Brown, Hamza Hamza, and Davin McCall. 2019. Stride in BlueJ – Computing for All in an Educational IDE. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287462

1 INTRODUCTION

The teaching of programming, as part of computer science education, was long seen as a specialist skill relevant only to those who were considering a career in a computing-related discipline, or those with a special interest in the subject. As a result, programming was taught in specialist courses, either in higher education or in advanced, usually elective subjects in school.

In the last decade or so, computing has been increasingly viewed as a subject relevant to all learners, as part of a general education. Computer science – and with it, programming – is now often taught early in a school curriculum (in lower secondary school, or in primary school) [11]. This development represents perhaps the single most significant change in the teaching of programming in this century so far, and it has significant ramifications for the methods and tools that should be employed.

The trend towards teaching computing as a discipline for all, as part of a general education, brings with it an expectation that the content learned in this subject cover material that is generally useful in many domains. Consequently, arcane details that are only relevant to professional software developers working with specific systems, which may have been acceptable as part of a specialist programming course, are much harder to justify.

A common challenge in curriculum and tool design is, therefore, the separation of fundamental concepts from accidental complexities. While the former should be exposed as clearly as possible, the latter should be avoided as much as is practical.

In this paper, we discuss the design of a new version of a widely used introductory program development environment, BlueJ, to take account of this development. BlueJ was initially designed to allow development in the Java programming language. The use of Java, a language designed for professional software engineers, is defensible in specialist computer science courses, but much less so in a generalist educational setting with younger learners. Its sometimes arcane and error-prone syntax introduces a significant amount of accidental complexity, mastery of which is hard to justify if the goal is an understanding of fundamental and generic computing concepts.

As a result, we added support for a new, more suitable language to BlueJ: a language called *Stride* [7]. Stride differs from traditional text-based languages mostly in its program manipulation support, provided by its *frame-based editor*. In this paper, we describe and discuss the features of Stride, its integration into BlueJ, and the benefits that result for general early programming education from this design. A link to the full, downloadable system is provided at the end of this paper.

2 A SHORT HISTORY OF BLUEJ

BlueJ was initially designed for use at introductory university level. It is, however, also widely used at schools, usually in elective computer sciences courses in the last years of a school curriculum.

BlueJ differs from mainstream Java development environments in its interface and functionality. The interface emphasises program structure, classes and objects. In its main window, BlueJ shows a simplified UML diagram and a visualisation of objects, which can be interactively created (Figure 1). BlueJ has functionality specifically designed to aid the learning of object-oriented programming, such as the ability to interactively invoke individual methods of arbitrary objects, supply parameters, observe return values, and inspect object state.

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287462

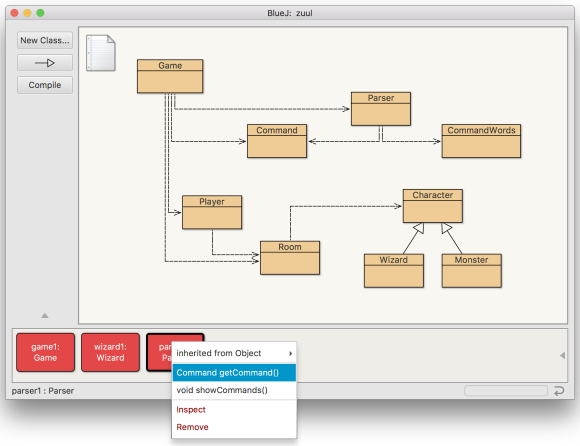

Figure 1: The main window of BlueJ, showing the class diagram and three objects on the object bench. Individual method can be invoked on objects via a context menu.

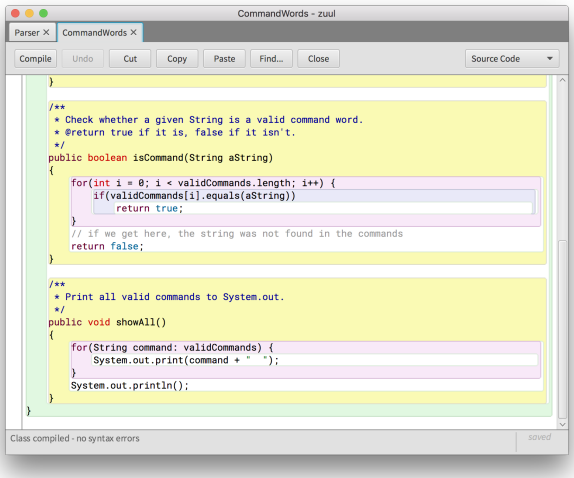

Figure 2: The BlueJ Java editor with scope highlighting, providing dynamic background colouring.

The main interface of the environment is significantly simpler and easier to learn to master than that of professional IDEs. BlueJ is, however, not a subset or cut-down version of a standard IDE: its aim is not to offer fewer features but to offer different features. Many of the educationally important interaction features are not available in other IDEs.

While BlueJ's original functionality provided significant innovation in IDE design and interaction, the programming language supported was standard Java. In the early versions of BlueJ, the editor for program entry and manipulation was a standard text editor, not significantly different from many other editors in use in programming systems.

BlueJ 3.0, released in 2010, introduced a new editor that supported scope highlighting (Figure 2). This functionality addressed one of the most common programming problems for beginners: the management of nested scopes. Beginning programmers often did not maintain consistent indentation and struggled with balancing the scope-defining brackets. Missing or surplus brackets were common and hard to diagnose. Auto-indentation and scope highlighting made these kinds of errors more obvious: The editor provides dynamic, real-time background colouring which makes spotting unbalanced brackets easier. It also aids in general readability of the code. The program manipulation, however, is character-based, and scope highlighting does little to prevent errors being made in the first place.

The overall abstractions provided via the main BlueJ interface worked well to illustrate important concepts, even for generalist programming courses with younger learners. The Java language, however, and particularly the text-based editor, introduced a significant amount of unnecessary complexity that resulted in frequent problems, usually with small-scale syntax issues. Spending time fixing such errors has little benefit for the understanding of fundamental programming principles.

3 THE RISE OF BLOCKS

When programming teaching was considered for younger audiences, including children under ten years old, text-based systems were not feasible. Instead, block-based languages, such as Scratch [8] became popular, which can be used at that age group.

Block-based languages introduced a number of important innovations that provide significant benefits for younger learners: Syntax errors were largely impossible to make, program statements were more readable due to labelling that is closer to natural language, and statements were discoverable by being presented on a block palette for experimentation and discovery.

Most importantly for our discussion: block-based languages removed much of the accidental complexity of dealing with character-based syntax and allowed learners and teachers to concentrate on the underlying computational concepts.

4 FROM BLOCKS TO TEXT

While block-based languages reduce or remove many potential syntax-related stumbling blocks that might obstruct the initial progress of a novice programmer, and are well-suited to younger programmers, they have some drawbacks in relation to the viscosity of the editing process: small code fragments which may be simple in appearance and function require a non-trivial amount of user action to create and place correctly, as each block must be dragged in from the palette. This quickly becomes a restrictive factor in the ability of the learner to create larger programs.

An additional limitation is inherent in learners' perception of the system: students may perceive a block-based language as a "toy" language [9], not suitable for real-world programming. This may affect their motivation and impact their desire to learn.

To alleviate the eventual frustration with block-based programming, a curriculum must at some point transition to a more traditional language and environment. In schools with a full computer science curriculum, such as those in England, this transition from

blocks to a text-based environment typically occurs between school years 7 and 9 (age 12 to 14). However, this transition is not without its difficulties, as students encounter a raft of potential errors and pitfalls that were not possible in a purely block-based environment. Previous work [6] has identified a number of significant challenges that students face:

(1) Reduced readability due to terser and more punctuation-heavy syntax.
(2) The necessity of memorising commands rather than selecting them from a palette, and of memorising syntactical structure.
(3) The increased need for typing and spelling skills.
(4) An increased number of available commands.
(5) The need to match method call syntax with method definitions residing elsewhere in the code.
(6) Understanding how syntactical elements are used to group compound statements and determine lexical scope.
(7) More complex type systems.
(8) More complex and more technical error messages.
(9) Managing manual layout, in particular indentation.
(10) A potentially different programming paradigm.

The significant set of issues illustrates the challenge of transitioning from a block-based to a text-based programming language. This leads us to pose two questions:

(1) How can we better support learners in the transition from block-based to text-based programming?
(2) Can we apply any aspects of the block-based approach to improve editors for more comprehensive, general programming languages? Although the editing viscosity can make block-based programming unattractive for larger projects, it does also provide some clear advantages. Some of these advantages might also be provided by a hybrid approach that lies somewhere between fully block-based and text-based programming.

In the following section, we introduce *Stride*, a programming language and corresponding editor designed around frame-based editing, a new editing paradigm which incorporates aspects of both block- and text-based programming. Stride both provides an intermediate transitional path for learners, and serves as an example to show that a comprehensive programming language which is suitable for larger and more complex projects can benefit from the ideas of block-based programming editors.

5 STRIDE

Stride is a Java-like programming language manipulated in a *frame-based programming editor*. Statements and other structural elements in the Stride language are represented in its editor as *frames*, coloured nested boxes that can be directly manipulated by the user (Figure 3). Frames are first-class interface elements: they can be inserted and deleted with a single keypress, or dragged and dropped into different location with the mouse. A context menu for common operations is available for all frames.

Frames are always present in their entirety: They are inserted and deleted in atomic operations, either with a single-key keyboard command, or by selection with the mouse from a palette (see below). Structurally incomplete statements, such as if-statements with a missing closing bracket, cannot exist, thus avoiding a large class of common syntax errors.

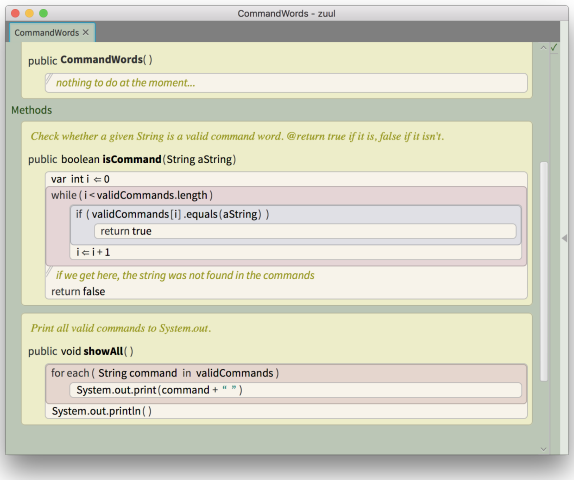

Figure 3: The Stride editor. Statements are represented by frames with a different background colour.

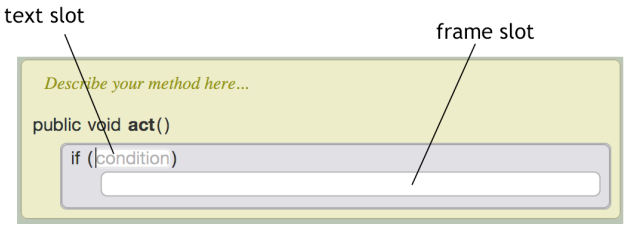

Figure 4: A frame for an if-statement with two empty slots: a text-slot for the condition, and a frame-slot for the body.

Frames, when inserted, may contain *slots* – holes to be filled to complete the statement. Slots are presented with white background, indicating an incomplete frame (Figure 4). Two types of slots exist: *text slots* for the insertion of expressions, and *frame slots* which hold nested frames.

Stride can be written and manipulated entirely with the keyboard. In contrast to block-based systems, the frame editor always displays a cursor to mark the editing locus. When the cursor is in a frame slot, a frame cursor is displayed that allows the entry of further frames. Focusing a text slot displays a text cursor that allows semi-structured text entry (Figure 5). Thus, editing, while exhibiting some properties similar to block-based systems, is quicker and more efficient than block-based systems which rely entirely on mouse interaction for manipulation.

While frames at the statement level automatically avoid many structural syntax errors, expressions and identifiers are not represented by frames. Here, text entry provides input significantly faster and more flexible than in block systems. Some structure is enforced in these text slots: Entry of parentheses, for example, is always in pairs. Entering or deleting one parenthesis will automatically enter

Figure 5: The two cursors for editing operation: the frame cursor (left) and the text cursor (right).

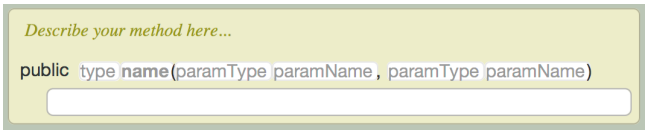

Figure 6: Frame slots are presented with prompts that provide information about the required elements.

or delete its matching counterpart. No unbalanced brackets can ever exist.

Semi-structured text entry represents a compromise between flexibility and error avoidance: It allows some syntax errors to be made in return for more flexible and faster program manipulation. Thus, Stride's syntax error characteristics sit between those of block-based and text-based systems: It avoids some common syntax errors, leading to fewer errors than in text systems, but allows others which are not possible in block-based systems.

Semantically, Stride is very similar to Java, with only minor differences in language constructs and an identical object model and type system. Most importantly, it is interoperable with Java, allowing full use of the standard Java libraries. Thus, the full functionality of the Java ecosystem is available. In fact, Stride and Java classes can be used side-by-side in the same BlueJ project.

A more detailed description of the features of Stride and its frame based editor is presented elsewhere [7].

6 REDUCING COMPLEXITY

Stride's frame editor provides several features that reduce cognitive load for novice programmers, avoid the need to memorise unnecessary detail and aid the transition from block-based programming systems.

6.1 Code structure

The editor provides an overall template for the class definition, with clearly labelled areas for fields, constructors and methods. These program elements, as well as statements, can only be inserted where they are syntactically valid.

Frames, when they are dragged, can only be dropped at syntactically valid places in the source code. Thus, structurally valid code is maintained at all times. The atomic insertion and manipulation of frames also ensures that detailed statement syntax, such as the punctuation required for a for-loop, does not need to be memorised or typed by the novice learner. Both the requirement of memorisation of syntax and the requirement of accurate typing typically present barriers for young learners in text-based systems.

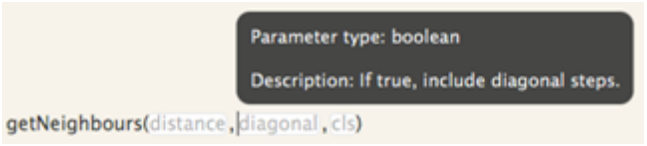

Figure 7: Parameter slots show formal parameter names; additional information is displayed on demand.

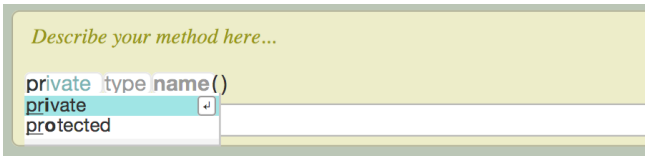

Figure 8: Frame slots are presented with prompts that provide information about the required elements.

6.2 Prompts and help

When compound frames are created, prompt texts in empty slots provide information about the program element expected (Figure 6). These prompts provide valuable help to beginners who may not remember the syntactical structure of a given statement.

Other prompt texts remain useful even after statement structures have become familiar. In method-call frames, for example, entering a valid method name automatically creates slots for each expected parameter with prompt texts displaying the parameter name. Hovering over the parameter slots with the mouse displays additional information in a popup, including the expected parameter type and a help text (Figure 7). These mechanisms make the available information much more visible at the point of insertion and reduce cognitive load for the learner.

6.3 Restricted entry

Some type of slots only allow a small number of possible values; in these cases entry is as a selection from a valid value set (Figure 8). Values in these selection slots can be typed, with completions of the typed prefix displayed and offered for selection. Beginners can inspect the full set of available choices.

6.4 Improved error display

Errors in Stride are displayed as red underlines in the program code. Focusing on the error location with the mouse or text cursor displays the error message, and may offer possible fixes (Figure 9). If a fix is offered, it can be selected from the popup to be enacted automatically.

Because of the enforced statement structure of the program text, error messages are generally more accurate than in freeform text editors. The error location can always be correctly attributed to a specific slot. Situations typical in text-based systems, where an error is reported far from its actual cause, cannot arise in Stride.

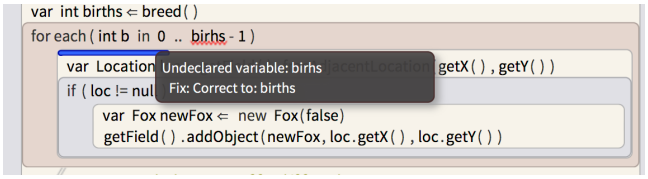

Figure 9: Errors may be followed by suggested fixes, which can be selected to enact the fix.

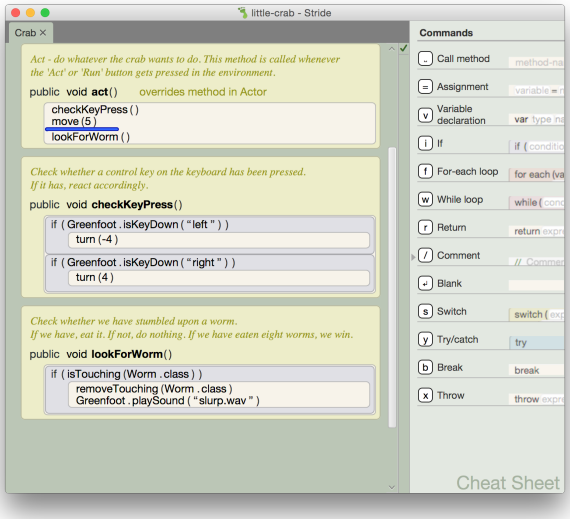

Figure 10: The right-hand side cheat sheet shows the available frames that can be inserted at the current frame cursor position.

6.5 The 'Cheat Sheet'

To aid in the memorisation of available statements and their key commands, a selection palette – named the "Cheat Sheet" – can optionally be displayed at the right-hand side of the editor (Figure 10). This cheat sheet always displays the complete set of available frame options, together with their keyboard shortcuts. A mouse click on a command in the cheat sheet inserts the corresponding frame in the code (as does, of course, activation of its command key). The cheat sheet is context sensitive: it displays only language elements that are syntactically valid at the current cursor position.

The cheat sheet provides an aid for recognition over recall, as the typical block palettes do in block-based systems. In contrast to block systems, not all method calls are listed individually (but rather, "Call method" is listed as a single frame type). As a result, the number of choices at any time is limited, and all available frames can easily be displayed. (The example shown in Figure 10 indeed shows all available program statement types.)

Investigation and choice of individual methods is provided via method call auto-completion functionality.

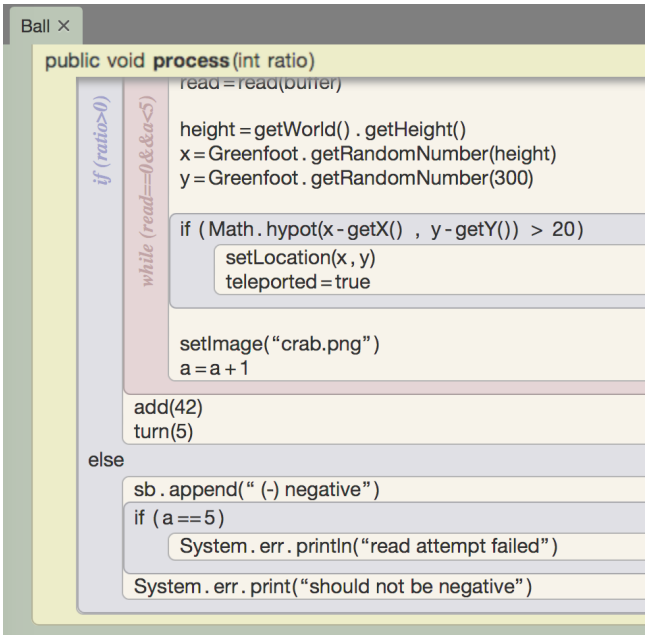

Figure 11: When code scrolls out of view, method headers are pinned to the top of the editor window. Header text of scopes that are not fully visible is displayed in the side bar of the scope frame.

6.6 Visual aids

The move away from characters as the basic unit of representation of program text opens the door for many options of visual program presentation that adds additional useful information for programmers. Figure 11 shows two examples of this: When longer methods scroll partially out of view, it is generally useful to still be able to tell which method is currently displayed. Consequently, the Stride editor pins the method header to the top of the window and scrolls the method body underneath.

In similar fashion, when the header of long scope frames scrolls out of view, the header text is displayed in the left indent area of the frame.

Both these examples demonstrate how the move away from plain text editors opens opportunities for better program display that increases readability.

7 STRIDE IN BLUEJ

Stride was originally implemented in the Greenfoot IDE [5], which – like many block-based systems – is a development environment with a restricted application domain: It is designed exclusively for the development of two-dimensional graphical applications, typically simple games or simulations. With the release of BlueJ 4.0, Stride is now available in BlueJ, a general-purpose IDE with a generic, non-restricted application domain – alongside the pre-existing Java editor.

The availability of Stride in BlueJ brings frame-based editing to a more general application domain that was previously covered only by Java, and not typically addressed by block-based environments.

Typical BlueJ-based teaching examples include GUI programs using Swing or JavaFX, programs implementing or making use of sophisticated data structures, applications connecting to databases or networks, and many more. All of these examples can easily be rewritten in Stride. In fact, BlueJ includes an automatic Stride-to-Java and Java-to-Stride conversion feature, which makes the translation of existing teaching examples easy. The translation feature also helps students in the progression from Stride to writing Java programs.

The availability of Stride in BlueJ brings the advantages of frame-based editing to general programming courses, and makes it feasible to use BlueJ in generalist courses aimed at all students, rather than specialist electives. While the target user group of BlueJ previously was learners from age 16 upwards, we now recommend use with students from about 13 years old.

We believe Stride serves well as a first programming experience at that age group, or as a follow-on system after gaining experience with block-based systems. When transitioning from block systems, the problems listed in section 4 are reduced. Some of the challenges remain and lead to new learning and insights (such as an extended command set, a more powerful object model, a more generic type system), while many of the incidental, purely syntactic complications are avoided [6].

When Stride is used in general programming courses for all students, its use serves two different purposes for two distinct user groups. For learners who do not want to progress further in software development, Stride can be the endpoint of programming education. All meaningful programming concepts can be illustrated with Stride, and the subsequent learning of specific text-based syntax or professional development tools serves little purpose for a generalist education.

For programmers who want to progress to more powerful systems, on the other hand, Stride serves as a well-working stepping stone into fully text-based Java. The concepts learned in Stride translate directly into traditional languages and environments, and a Java-preview feature in the Stride editor helps in investigating syntactic differences that remain.

This combination ensures that interested students can progress to other systems smoothly, while those wanting to get an insight into the concepts of programming in general are not distracted by unnecessary syntactic problems.

8 EVALUATION

Prior to the addition of Stride, BlueJ already contained support for the Blackbox project [1], which tracks the programming activity of opted-in BlueJ users. Blackbox has support for global data collection, and for supporting local studies by tagging users of interest, with access available to interested researchers. Support for tracking Stride users has been added to Blackbox, which can thus be used as an evaluation platform.

Currently, Stride use is low in BlueJ due to it being a recent addition. We can examine whether users are using the keyboard controls – a distinctive feature of Stride – or whether they are using the mouse as they would in other blocks editors. Looking at the data up to 2018-08-06 (all numbers to nearest 100), we find that 13,500 frame insertions were performed by clicking on the cheat sheet, whereas 30,300 insertions were done using the keyboard commands. Thus it seems that most users are taking advantage of the keyboard commands to insert frames. Looking at the statistics for frame manipulation, there were 800 paste operations and 1300 drag operations, suggesting that the ability to drag frames by mouse remains a useful and popular way to manipulate code, and is used more than the clipboard.

9 RELATED WORK

There are many existing purely text-based and purely block-based programming languages, that we will not list here, as we are interested in hybrid approaches like Stride – in which there has been increasing interest. Currently the closest system to Stride is GP, a block-based system for general purpose programming [10]. GP shares some aspects with Stride, such as the ability to edit block-like syntax with the keyboard. The main difference is that GP retains Scratch's pattern of having a different customised block for each environment capability (e.g. move, turn), which limits its use to strongly restricted application domains, whereas Stride has a single method-call frame which can be used to call any Java method.

Looking to the past, structure editors, such as Boxer [2] and Barista [4], and older editors like GNOME [3], share various features with our frame-based editor. Structure editors failed to gain widespread adoption. We believe one reason for this was the poor usability of older editors. For example, navigating a structured expression often used abstract syntax tree-based navigation, so that to move in the expression "1 + 2" from the 1 to the plus, the user had to press the up cursor key. This is logical in terms of internal syntax trees, but to a novice programmer, it is counter-intuitive. (In Stride, this is achieved by pressing the right cursor key). Another issue is that many structure editors became structure editor generators, as many programmers realised that it was possible to generate an editor automatically from language grammar descriptions. Our experience with Stride suggests that many design decisions in editors need to consider carefully the language grammar, rather than being language-agnostic. Finally, previous structure editors enforced fixed structure all the way down, as Scratch does. We believe that this is too cumbersome for large program manipulation; our approach of having structure at high-levels and free-form text at lower levels is a more promising practical compromise. A fuller discussion of prior related work is presented elsewhere [7].

10 CONCLUSIONS

A programming environment for beginners should minimise accidental complexity. The main interface of BlueJ and the design of its interaction facilities for easy compilation, execution and testing of code have followed this principle for a long time.

BlueJ's reliance on standard text-based Java programming, however, still introduced a notable hurdle for beginners, especially younger beginners. The text editor presented complexity for little educational gain. With the addition of Stride, BlueJ removes another layer of complexity that enables novices to make fewer errors, progress more smoothly and better concentrate on fundamental programming concepts.

Download: BlueJ 4.*x* can be downloaded at www.bluej.org. BlueJ is free, open source and supports both Java and Stride.

REFERENCES

[1] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[2] Andrea A diSessa. 1997. Twenty reasons why you should use Boxer (instead of Logo). In *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference*. 7–27.

[3] David B. Garlan and Philip L. Miller. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, USA, 65–72. https://doi.org/10.1145/800020.808250

[4] Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[5] Michael Kölling. 2010. The Greenfoot Programming Environment. *Trans. Comput. Educ.* 10, 4, Article 14 (Nov. 2010), 21 pages. https://doi.org/10.1145/1868358.1868361

[6] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331

[7] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2017. Frame-Based Editing. *Visual Languages and Sentient Systems* 3 (2017), 40–67.

[8] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[9] José Alfredo Martínez-Valdés, J. Ángel Velázquez-Iturbide, and Raquel Hijón-Neira. 2017. A (Relatively) Unsatisfactory Experience of Use of Scratch in CS1. In *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM 2017)*. ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/3144826.3145356

[10] Jens Mönig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) (BLOCKS AND BEYOND '15)*. IEEE Computer Society, Washington, DC, USA, 51–53. https://doi.org/10.1109/BLOCKS.2015.7369001

[11] Informatics Europe & ACM Europe Working Group on Informatics Education. 2013. *Informatics education: Europe cannot afford to miss the boat.* Technical Report. ACM Europe.