Frame-Based Editing

Michael Kölling, Neil C. C. Brown, Amjad Altadmri
Department of Informatics, King's College London, UK
{michael.kolling, neil.c.c.brown, amjad.altadmri}@kcl.ac.uk

Abstract *In introductory programming teaching, block-based editors have become very popular because they offer a number of strong advantages for beginning programmers: They avoid many syntax errors, can display all available instructions for visual selection and encourage experimentation with little requirement for recall. Among proficient programmers, however, text-based systems are strongly preferred due to several usability and productivity advantages for expert users. In this paper, we provide a comprehensive introduction to a novel editing paradigm, frame-based editing – including design, implementation, experimentation and analysis. We describe how the design of this paradigm combines many advantages of block-based and text-based systems, then we present and discuss an implementation of such a system for a new Java-like language called Stride, including the results of several evaluation studies. The resulting editing system has clear advantages for both novices and expert programmers: It improves program representation and error avoidance for beginners and can speed up program manipulation for experts. Stride can also serve as an ideal stepping stone from block-based to text-based languages in an educational context.*

1. Introduction

Syntax errors are a well-known – and largely unavoidable – problem in text-based programming. The severity of the problem varies widely: They can range from a small nuisance slightly slowing down an expert programmer's workflow to an insurmountable hurdle stopping a novice programmer in her tracks. A significant body of existing published work explores which errors are problematic (for example [1–3]) and how to alleviate these problems via additional tools [4, 5], but it is clear that text-based programming and syntax errors are inseparable.

In this paper, we will introduce a re-thinking of editing interactions in programming environments, which we term *frame-based editing*. A reduction in the number of syntax errors made by a programmer is one advantage, and we will use this goal as one motivation for our new design. We will see, however, that this is not the only benefit. Various other advantages, including improvements in readability, better navigation, and faster program manipulation also follow from our design.

For beginners, syntax errors present a serious and particularly annoying hurdle [4, 6]. Serious, because beginners often lack the skill to remove the error; syntax may still be mysterious, and what later becomes trivial is still the main focus of the programming activity [7]. Annoying, because syntax errors typically do not provide a path to any useful insight or learning experience. While the encounter of a semantic error may expose a misunderstanding and lead to a useful and meaningful learning experience, overcoming a syntax error does not usually teach an important concept of programming; it merely enforces an arbitrary rule to be memorised.

The problem is usually compounded by the dismal quality of error messages in many of our programming systems. Error messages are typically written by compiler writers, and little effort is made to include information useful to beginners. Many errors are reported from the viewpoint of the parser or type checker (such as the well-known "Illegal start of expression" or "Identifier expected" messages in common Java compilers), and in many cases, little useful information is given to a novice [1, 5]. Weinberg [8] summarises this succinctly: "*[H]ow truly sad it is that just at the very moment when the computer has something important to tell us, it starts speaking gibberish.*"

One instinctive goal might be to improve the quality (specificity and correctness) of error messages [9, 10]. However, we can do better: A more worthwhile goal is to avoid syntax errors in the first place, for the benefit of beginners and experts.

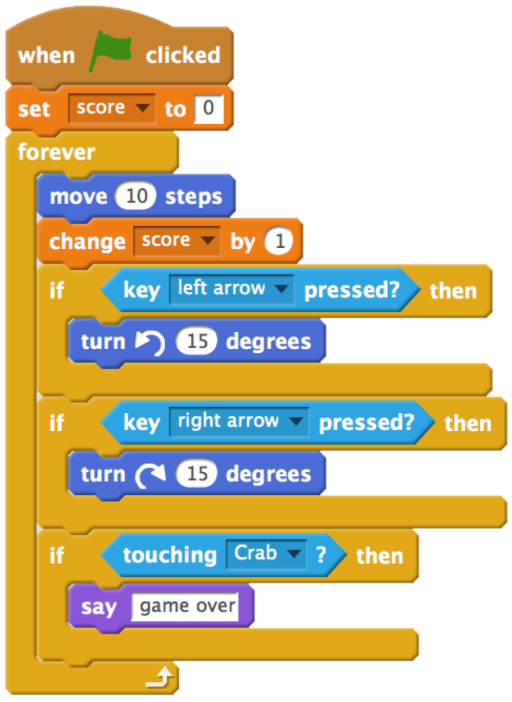
2. Blocks: Avoiding Errors in Programming

When thinking about novice programming, especially for young learners, it is useful to consider other successful areas of learning. When children play with Lego blocks, for example, they typically learn various techniques of construction without ever reading a manual and without any error messages involved in the process. Lego blocks have the inherent quality of allowing experimentation and fitting together only in well-defined ways. It is not possible to connect two Lego bricks erroneously – if they fit together at all, they fit correctly. There are no type errors in Lego bricks.

The equivalent of Lego bricks for programming are block-based languages, such as Scratch [11]. These languages provide statements of the programming language as direct manipulation "blocks", which can be snapped together in syntactically valid constellations.

Direct manipulation programming systems for beginners have become widely popular in the last 10 years. In these systems, language statements are visually represented as user interface entities that can be manipulated: dragged, dropped,

DOI reference number: 10.18293/VLSS2017-012



```
when [flag] clicked
set score to 0
forever
    move 10 steps
    change score by 1
    if key left arrow pressed? then
        turn ↺ 15 degrees
    if key right arrow pressed? then
        turn ↻ 15 degrees
    if touching Crab ? then
        say game over
```

Figure 1. Block-based program notation in Scratch

snapped together, or double-clicked to activate. Due to the block-like appearance of these statements in many systems, they are often referred to as "block-based" languages.

The most popular of these systems in early programming education is Scratch (Figure 1); other notable examples include StarLogo TNG [12], Alice [13] and App Inventor [14]. While these systems differ in many aspects and significant details, they are similar enough for the purpose of our discussion here to be treated as one common class of system.

2.1. Benefits of Block-based Languages

Due to the visual and direct manipulation nature of programming elements, block-based editing achieves a more playful feel of programming, leading young learners to more experimentation and exploration [15]. All possible statements and expressions are represented on screen, supporting recognition rather than requiring recall for selection of statements.

What is more, most common syntax errors found in typical text-based languages are avoided; they simply cannot be made. It is not possible, for example, to forget to close the scope of a conditional statement – the statement is either present in its entirety or not at all. The syntax of statements cannot be mistyped, and statements can only be snapped together in syntactically valid combinations.

Indeed, the error prevention goes further than simple syntax: Where parameters are expected, statements are often created with reasonable default values already inserted. While the default value might be not what the programmer desired, the program is at least syntactically valid and will execute.

Type errors can also be avoided: statements expecting typed expressions can contain slots of specific geometric shapes,

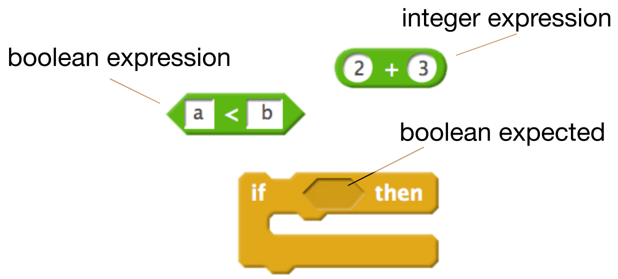


Figure 2. Shapes indicating types of expressions

where the shape denotes the expected type (see Figure 2). Expressions are represented in these shapes; only if the expression type matches the expected type will the shape fit, and the blocks snap together. In some systems, it is not possible to assemble a combination that would represent a type error. (More advanced use of shapes for types in block languages are also possible [16].)

For beginning programmers, these systems offer several tangible benefits: they reduce the rate of errors, allow a better exploration of the available language, make assembly of programs easier, almost always lead to executable code, and increase subjective satisfaction [17]. Many of these benefits are also clearly desirable for experienced developers, so the obvious question is: Why don't we all program like this?

2.2. Limitations of Block-based Languages

Experienced programmers clearly favour text-based languages over direct manipulation systems. The reason lies in several severe limitations of block-based languages: They suffer in readability, viscosity and navigation support.

When programs in block-based languages become large (and "large" in this context is reached very quickly – a few pages of program code already feels "large" in these systems), they become hard to read. The graphical nature – colour, shape and three-dimensional appearance with light and shadow effects – adds visual noise that can overwhelm the program structure and distract from program semantics.

Navigation in these systems is also comparatively poor. Quickly switching focus between declaration and use of an entity is typically not well supported, making reading and exploration of an existing program harder than in typical text-based environments. The ability to position code fragments arbitrarily in Scratch makes it difficult to systematically read a whole program, and little support is provided for organising the code, higher level structuring or obtaining an overview.

Viscosity – the resistance to change [18] – is high in these systems. Making changes to existing programs requires more effort and takes more time than in professional text-based environments. Changing large existing code bases – rather than the development of new, small programs – is the bread and butter work for most experienced programmers, and this is precisely where block-based systems do not offer adequate support.

3. Text-Based Programming

Text-based programming is the current standard in most programming systems for proficient users. Many benefits are obvious: Text is a very expressive, flexible medium that allows fairly clear and concise definitions of programs. Humans are very practised in reading, navigating and understanding text-based representations.

However, text-based systems have a number of limitations. Most of these encumber the programmer with work that could easily be automated by a more sophisticated system. Performing the work impedes productivity, sometimes just by requiring time to perform, and sometimes by adding cognitive load that distracts from the intrinsic complexity of the programming task. Common limitations include:

- Programmers must often type out program statements. Typing out keywords of the language is unnecessary work that slows down program entry – previous work has suggested an association between faster typing speed and increased programming performance [19, 20]. Often, the conceptual space of valid entities to be entered at any point in the program is fairly limited, and more efficient selection interactions could be devised.
- When entering program text, programmers must ensure syntactical correctness of statements (including correct spelling and punctuation). When the statement intended is recognised (e.g. once the programmer has entered the *while* keyword), it is unnecessary to make the human programmer responsible for correct orthography of the remainder of the construct.
- Layout, whitespace and indentation are typically under control of the programmer. Again, this is unnecessary. Many modern programming environments will offer help in automatically indenting correctly, but the indentation can still be "broken" (i.e. changed to contravene coding styles) by the programmer after the fact. This happens easily and often by accident. Since indentation and layout rules exist and follow simple algorithms, there is no reason why this cannot be fully automated, freeing the programmer from one more unproductive task.

The reason that these tasks require more work than would be ideally desirable is rooted in the fact that program representation is based on pure text. A one-dimensional sequence of free-form characters is arranged two-dimensionally on screen, and this serves as the basis for all program elements.

Pure text representation is a technology developed more than half a century ago for early computer terminals, and there is little reason today – other than historical inertia – to restrict program representation to this limited form. (This observation previously led to work on structure editors; later, in section 13, we will examine why these early attempts failed but block-based editors later succeeded.)

Few reasons exist that program statements have to be typed out manually in their entirety, that the programmer should be responsible for correct punctuation, or that characters (tabs and spaces) should be used for the arrangement of program components on a page. Some work has also suggested that the choice of syntax symbols is often arbitrary, and that a randomly chosen syntax is no less usable than existing ones [21].

Scope is represented in many programming languages by using a pair of brackets. This is undesirable for several reasons:

- A pair of brackets is not the best visual representation of the extent of a scope. Considering graphical elements as a possible part of a language, using drawn frames, boxes or colours offers a clearer, continuous representation which is easier to recognise and interpret than two isolated brackets.
- The fact that one can even omit a closing bracket – that it is technically possible to enter *half a statement* – serves no useful purpose. Any modern system should ensure that a statement is either *present* or *absent*, and offer interaction techniques that allow convenient entry and manipulation. Representing program statements as a sequence of characters which can all be edited individually – for example deleting a single character out of the middle of a language keyword – is an archaic accidental artefact that is hard to justify in today's code editors.

Modern IDEs have largely recognised this fact and offer a variety of support mechanisms to address some of these issues. Shortcuts, code completion, auto-indentation and automatic entry of matching brackets and quotes are all designed to alleviate the unnecessary busy work a programmer is tasked with. However, these mechanisms fail to solve the real problem. While they streamline the entry of the program text, the representation is still pure text, with all the resulting problems. Spelling can still be incorrect, parentheses can be deleted after entry to break the balance, indentation can be changed accidentally, and so on. Various possible benefits and improvements cannot be realised due to the reliance on pure text for program representation. In the following parts of the paper, we shall discuss some improvements that become possible when leaving behind pure text as the medium.

4. Blocks versus Text: A Brief Comparison

4.1. Criteria

In the previous sections, we have discussed some beneficial and some problematic aspects of direct manipulation (block-based) and text-based programming environments.

Three aspects emerge as the main areas of consideration:

- *Representation* includes the appearance of the program at various scales, from the visual appearance of a single instruction to the representation of larger structures such as control structures, classes, or modules. Representation is crucial for program comprehension and readability.
- *Manipulation* describes all aspects of program entry and editing, including ease of entry and deletion of program constructs, making changes ranging from small scale edits to large refactorings, and extending existing program source.
- *Error rate* refers to the rate of errors an average programmer makes, or the number of errors that can be made in a system.

As we have discussed, a significant number of syntactical or type errors can be avoided in some systems.

4.2. Comparison

Using these three areas of consideration, which class of system – block-based or text-based – is better? This question cannot be answered without taking the type of user into account, so we will ask this question individually for two relevant distinct user groups: Beginners and proficient programmers.

4.2.1. Novice Programmers

For novice programmers, block-based environments have a lot to offer:

- They provide a clearer, easier to interpret *representation* of individual program statements and their semantics;
- They allow easier *manipulation* of program elements, to a large extent because of the recognition-over-recall characteristic of entering program code; and
- They lead to a significantly lower *error rate*, eliminating many syntax errors outright.

For a typical thirteen-year-old novice, block-based systems win on all counts – a finding confirmed by several recent blocks versus text comparison studies [22, 23].

4.2.2. Proficient Programmers

For proficient programmers, text-based systems have some distinct advantages:

- Once a reader has been trained to read a programming language, text provides a more concise, more readable *representation* than blocks;
- *Manipulation* in standard text editors is faster and more flexible than in block-based systems – viscosity is significantly lower; but
- Text-based environments still allow a higher *error rate*, and even proficient programmers will make some errors which would not be possible in direct manipulation systems. Many of these errors will be slips and typographical errors, which are quickly fixed by experts; however, they still have the potential to interrupt workflow and cognitive processes.

Overall, for proficient programmers, typical text-based environments are clearly preferable.

4.3. Where is the Cut-off?

In the above discussion, we have – rather arbitrarily – distinguished only two groups: "novices" and "proficient programmers". This raises the questions: What about intermediate users? And at what point do programmers become sufficiently "proficient" to warrant a shift to text-based systems?

In fact, programmers reach the point where their proficiency outstrips the usability of typical block-based languages fairly quickly. We believe that a typical sixteen year-old, having programmed for two or three years with Scratch or similar systems, will normally have reached a level of expertise and expectation where she is more efficient and productive with a typical text-based system. For adults, with their higher ability of dealing with abstraction and notation, the time of usefulness of block-based systems is much shorter still (and may be near zero for some novices with good technical and abstraction background).

Despite their clear and distinct advantages in early stages of learning to program, current block-based languages do not manage to support programming activity for a significant length of time beyond the initial learning stages. Programmers outgrow these kinds of system fairly quickly, and their advantages are lost with the switch to traditional text-based systems. (We have discussed elsewhere the issues surrounding the transition from blocks to text programming in more detail [24]).

One could present the view that, at the time of sufficient maturity that a change to a text-based system is advisable, the additional help provided by direct manipulation systems is not needed anymore, and thus there is no problem. This, however, misses an opportunity. We strongly believe that proficient programmers can also profit from the advantages that block-based systems bring to the table: clearer representation, easier manipulation, and lower rates of errors. Programming for *all* users can be improved if the advantages of both kinds of systems can be combined.

5. Frames: A New Editing Paradigm

In the remainder of this paper, we present the concept of *frame-based editing*, a redesign of program editing with the goal of combining the advantages of text-based and direct manipulation editing systems.

Block-based languages provide the following main advantages:

- They make many syntax errors impossible, and thus reduce error rates.
- They make program statements visible and support recognition and experimentation.
- Some selected editing operations are quicker or easier due to the direct manipulation characteristics of the program elements.

Text-based systems have the following strengths:

- The representation is more readable for anyone but early novices.
- Lower viscosity; program manipulation is quicker.
- Navigation and exploration are more flexible.
- Programs can be entered, manipulated and navigated purely via the keyboard.

The goal is to combine the best of both worlds, to create an editing technology that sits in the space between block-based and text-based languages and that combines advantages of both systems.

The principle providing the foundation of this design is to approach the problem from an HCI perspective. Two views are fundamental:

- A programming environment is a user interface for manipulating a program.

- A programming environment is a user interface for understanding a program.

Reading and writing programs is equally important, and therefore both the representation and the manipulation aspects need to be considered equally.

When designing an interface for the representation and manipulation of programs, some elements are better represented graphically. Scope, for example, is a concept of extent in the program text, and can better be presented with graphical elements (such as boxes) than by using characters (such as brackets) in the text flow of the program.

6. Design of a Frame-Based Editor

Figure 3, overleaf, shows the interface of a frame-based editor for a new, Java-like language called Stride[1], integrated into the Greenfoot system since 2015 (and BlueJ since 2017). 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 [25], the system our current implementation was first integrated into, is an introductory development environment aimed at beginning programmers. Previously, when it supported only the Java programming language, it was targeted at those aged from about 14 years old upwards. The new version, also supporting Stride, is aimed at an audience starting younger than that by two or three years. We will use this implementation as the prototype to discuss the concepts of frame-based editing.

While some specific design decisions are influenced by the concrete context (a novice user group likely to transition to Java), most of the aspects described here are independent of this context, and the design advantages would apply equally to professional environments. We will discuss this applicability to professional environments further in section 14.

6.1. Representation

Figure 4 shows the look of a segment of typical program code in Stride. We will discuss several aspects of our code representation, in turn.

6.1.1. Scope

Scopes are represented as frames: graphical boxes, rather than the customary pair of brackets 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 in this representation. Programmers do not need to determine which closing bracket matches which opening bracket, and no additional confusion can be created by misleading indentation.

[1]The exact differences between Java and Stride are detailed in the appendix.

6.1.2. Indentation

In text programming, indentation is created using editable whitespace characters (tabs or spaces; the subject of a long-running debate, which also includes the exact number of spaces to be used), and programmers are responsible for creating and maintaining correct indentation. Both of these are archaic characteristics that have no place in modern editors. We will see that this fact – that all program elements are represented by text characters – forms the basis of problems with many elements of current systems.

Making programmers responsible for maintaining correct indentation – a task that can easily be automated – adds unnecessary work, both manual and cognitive, and may cause distraction from the actual task the user wants to achieve. The fact that almost all modern programming environments provide substantial help with this, in the form of auto-indentation of newly added lines and auto-formatters for whole documents, shows that designers of text editors are aware of the problem. But there is no reason to allow indentation to be later modified to be incorrect, or to require programmers to think about it at all.

Making the editor responsible for indentation also opens up a solution to another problem. Many programmers disagree over the desired depth of indent; anywhere from two to eight spaces is used. Frame-based code does not store the indent depth in the file – it is simply a graphical attribute in the editor[2]. Thus, it becomes possible for each programmer to define their own preferred indent level as a personal display preference without altering the shared code. And because our editor also manages the line-wrapping of code as a visual aspect (without modifying code to add line breaks, etc), the indentation of continuation lines is also automated.

One partial consequence of managing the indentation as an editor attribute is that we are free to use variable-width fonts. The main advantage of traditional monospace coding fonts is that they allow programmers to align code. This is no longer tied to font selection in frame-based editing. Some studies have suggested that variable-width fonts are more readable than fixed-width [26, 27], although these results are often hard to generalise as they are affected by specific choice of font face.

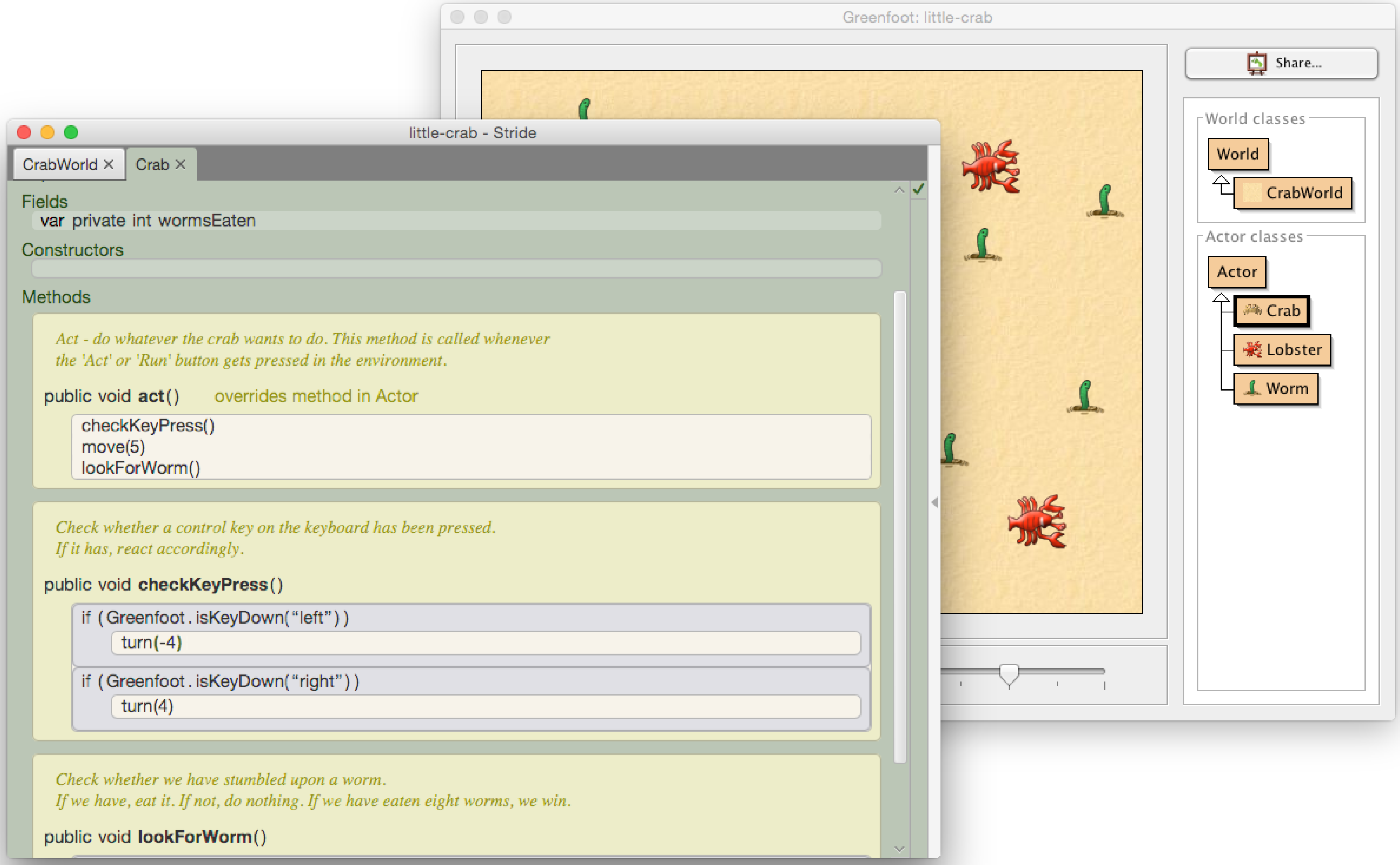
6.1.3. Whitespace

While indentation – leading horizontal whitespace – is maintained automatically in a frame-based editor, vertical whitespace – blank lines in traditional text editors – is partly automatic and partly under control of the programmer.

Spacing between fixed elements of a program (for example, the space between method declarations) is maintained by the system. There is little need for this to vary, so consistency can automatically be maintained.

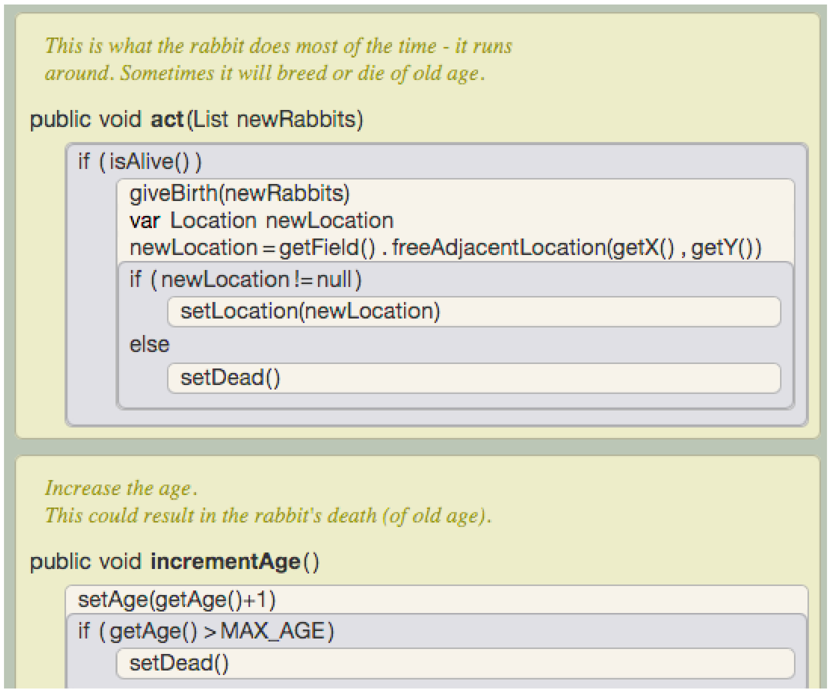
Within a sequence of statements, vertical whitespace is sometimes used to separate logically distinct parts of a method.

[2]A similar argument is made for tabs over spaces, but continuation lines usually present a further problem.



```
Fields
  var private int wormsEaten
Constructors

Methods
  Act - do whatever the crab wants to do. This method is called whenever
  the 'Act' or 'Run' button gets pressed in the environment.

  public void act()      overrides method in Actor
      checkKeyPress()
      move(5)
      lookForWorm()

  Check whether a control key on the keyboard has been pressed.
  If it has, react accordingly.

  public void checkKeyPress()
      if (Greenfoot.isKeyDown("left"))
          turn(-4)
      if (Greenfoot.isKeyDown("right"))
          turn(4)

  Check whether we have stumbled upon a worm.
  If we have, eat it. If not, do nothing. If we have eaten eight worms, we win.

  public void lookForWorm()
```

Figure 3. Frame-based editor interface, integrated into the Greenfoot system



```
This is what the rabbit does most of the time - it runs
around. Sometimes it will breed or die of old age.

public void act(List newRabbits)
    if ( isAlive() )
        giveBirth(newRabbits)
        var Location newLocation
        newLocation = getField() . freeAdjacentLocation(getX() , getY())
        if ( newLocation != null )
            setLocation(newLocation)
        else
            setDead()

Increase the age.
This could result in the rabbit's death (of old age).

public void incrementAge()
    setAge(getAge()+1)
    if ( getAge() > MAX_AGE )
        setDead()
```

Figure 4. Code representation in a frame-based editor

This is a semantic consideration – not a syntactic one – and a programmer, therefore, has the option to enter vertical whitespace between statements, equivalent to inserting a blank line in a text editor.

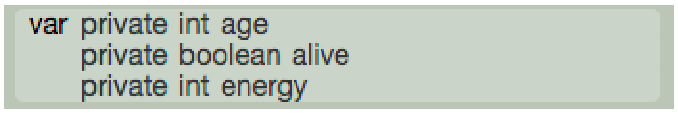
6.1.4. Colour

Background colour is used to identify different frames, which represent different kinds of program elements. In our implementation, the outermost frame – the class – has a green background, methods are yellow, with other types of frames using various colours to distinguish themselves.

Simple statements, such as assignments or method calls, are also represented by frames (although these do not hold nested statements). These simple frames have a greyish-sandy background colour and no border drawn around them. This makes sequences of statements visually less busy than in most block-based editors.

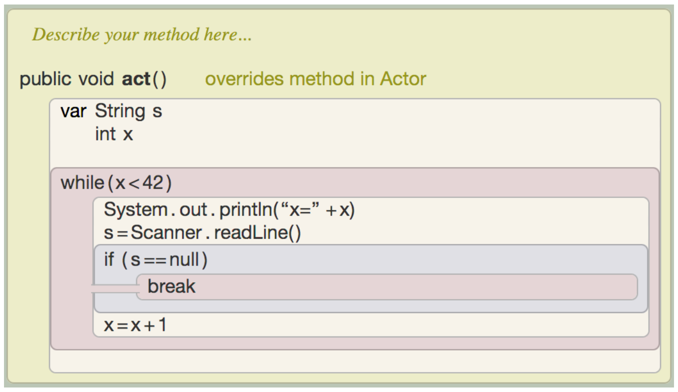
Many frames have "slots" – holes that need to be filled in to complete the statement – such as the condition in an if-statement or the value in an assignment. These slots are white when they are empty. When slots are correctly filled in, they acquire the background colour of the frame, blending into their context. Thus white areas, standing out quite clearly, signify syntactically unfinished code (visible later on in Figure 7 and Figure 8).

Users can get used to these colours quite quickly, and they provide useful cues about program structure that are quicker and easier to recognise than groupings arranged using bracket characters.



```
var private int age
    private boolean alive
    private int energy
```

Figure 5. A group of variable declarations



```
Describe your method here...

public void act()    overrides method in Actor
    var String s
        int x

    while (x < 42)
        System . out . println("x=" + x)
        s = Scanner . readLine()
        if ( s == null )
            break
        x = x + 1
```

Figure 6. Representation of a break statement

6.1.5. Context Sensitive Display

The visual representation of statements can be context sensitive. For example, a variable declaration starts with the keyword "var" (Figure 5). However, if a variable is declared directly below another variable declaration, the keyword is not repeated; for visual simplicity, the keyword is shown only once for a variable group. The indentation of the rest of the frame is kept constant, to indicate a grouping of variable declarations.

Another context-adaptive example is the presentation of

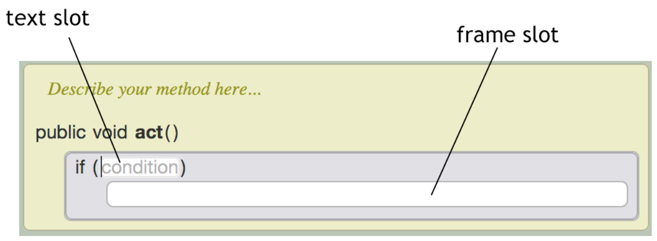


Figure 7. An if-statement with empty slots

break statements, which exit the innermost loop or switch statement in which they are contained. The background colour of the break statement automatically matches the innermost enclosing loop or switch statement, representing its context, and a solid band of colour is drawn from that container's indent to the break statement (Figure 6).

This reflects again the underlying principle that the programmer is responsible for creating the structure of the program, but not for creating or maintaining its visual representation.

6.2. Manipulation

Manipulation of programs takes place mostly at the frame level. Users enter, remove or manipulate frames as a whole. Since frames represent statements (and other program elements) the main unit of manipulation is complete statements, not single characters. Character level editing exists only in text slots within frames (see section 6.2.2).

6.2.1. Insertion of Statements

Statements are inserted by inserting a frame. Every kind of statement has its corresponding frame, which can be inserted using a single command key when a frame cursor has focus (see section 6.2.3 and section 7.5).

Command keys are simple character keys on the keyboard – they do not need to be combined with a modifier key. Thus pressing the 'i' key when the frame cursor is focused will enter an if-statement (not the character 'i'). This is different from auto-completion of statements as supported in many traditional environments. There is no need to trigger any code completion system; one keypress is all that is needed. The command keys are not necessarily the first characters of a keyword. (Assignment, for example, can be inserted by typing an equals symbol.)

6.2.2. Slots

Some frames are complete just by inserting the frame itself (such as a *break* statement). However, most frames require additional information to be filled in to be complete; this information is provided in *slots*.

Frames can contain two different kinds of slots: *text slots* and *frame slots*. Text slots accept (almost) free-form text entry, whereas frame slots contain nested frames. A frame for an if-statement, for example, has two initially empty slots: a text slot to specify the condition and a frame slot to hold the body of the statement (Figure 7).

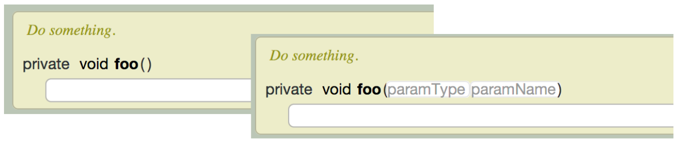


Figure 8. Optional text slots: invisible without focus (left) and visible when holding keyboard focus (right)

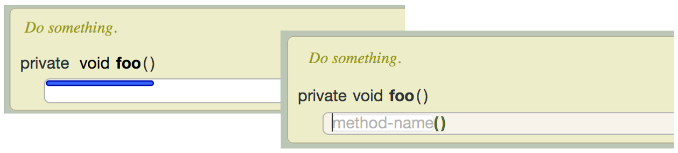


Figure 9. Cursors: a frame cursor (left) and a text cursor (right)

Text slots have a white background when they are empty, expecting text entry. Two varieties of text slot exist: *compulsory* and *optional* text slots.

Compulsory text slots are always visible, and content must be supplied to create a syntactically valid program. The condition of an if-statement is an example.

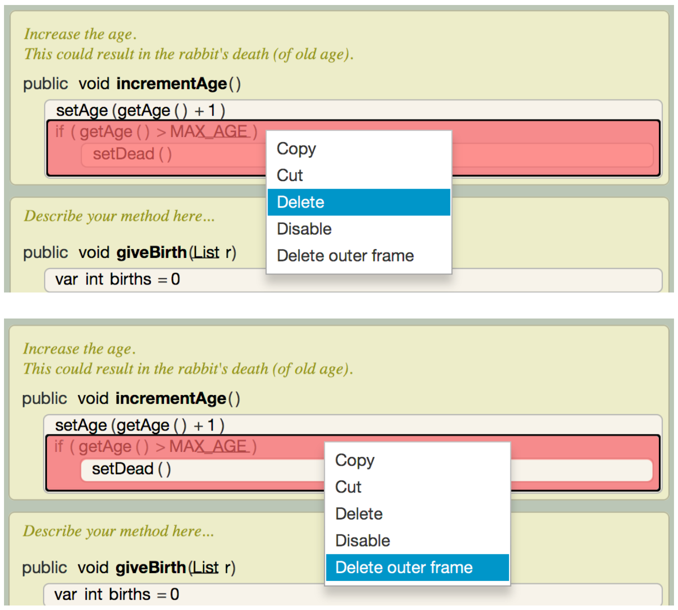
Optional slots are only visible when the cursor navigates to their potential location. An example is a formal parameter in a method declaration. The parameter list always has optional slots at the end so that additional parameters may be entered. When the cursor is not at the location of the optional slot, it is invisible (Figure 8, left); however, when the cursor is moved to the location of the optional slot it becomes visible, gains focus and text can be entered (Figure 8, right). In the case of the formal parameter, two optional slots are present, one for the type and one for the parameter name. When one is filled in, the other slot becomes compulsory.

6.2.3. Frame Cursor versus Text Cursor

A focused 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 9, left); inside a text slot, the cursor changes to a text cursor (Figure 9, 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, key 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.

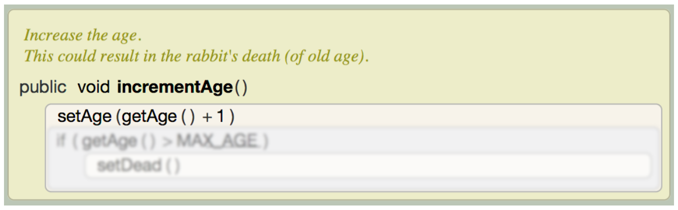
Technically, this introduces two separate modes: a frame editing mode and a text-editing mode. These modes are entered by cursor movement, and visually distinguished by a different cursor representation. Whether this causes confusion to users was one of the important early questions in this design, and is discussed further below.



```
Increase the age.
This could result in the rabbit's death (of old age).
public void incrementAge()
    setAge (getAge () + 1)
    if ( getAge () > MAX_AGE )
        setDead ()

Describe your method here...
public void giveBirth(List r)
    var int births = 0
```

```
Increase the age.
This could result in the rabbit's death (of old age).
public void incrementAge()
    setAge (getAge () + 1)
    if ( getAge () > MAX_AGE )
        setDead ()

Describe your method here...
public void giveBirth(List r)
    var int births = 0
```

Figure 10. Preview of delete operations



```
Increase the age.
This could result in the rabbit's death (of old age).
public void incrementAge()
    setAge (getAge () + 1)
    if ( getAge () > MAX_AGE )
        setDead ()
```

Figure 11. A disabled frame

Statements are frames, and entered in frame slots, while expressions are not frames, and are entered in text slots.

6.2.4. Deletion of Statements

As with insertion, deletion of a frame deletes the whole statement. Deletion can be achieved using the delete or backspace keys when the frame cursor is before or after the frame, respectively.

Another option to delete a frame is to right-click the frame with the mouse, and selecting a delete option from the frame's pop-up menu. Different delete options may be available: If, for example, the frame is an if-statement, the statement can be deleted while leaving the statements contained in its body present, or the statements in the body may be deleted with it. While a function is selected in the menu, a preview annotation in the source code hints at the effect of the selected function (Figure 10).

6.2.5. Disabling Frames

The example above, selecting 'Delete' from a frame's context menu, shows another advantage of elevating program elements to first class citizens in the interface. Since declarations and statements have become interface entities in their own right, they can have associated properties and functionality. Most obviously, they can have a context menu which offers operations on the frame. One of these operations is *disabling* a frame.

Disabling a frame (Figure 11) temporarily treats the frame as if it were deleted. In traditional systems, this is typically done by "commenting out" a block of text. (One study [28] found that 63% of all comment usages were disabling code, not providing an actual text comment.) Again, we see the richer possibilities of an interface not relying solely on characters for functionality: "Commenting out" a sequence of statements to temporarily disable them is technically a misuse of the comment symbol – unused code is not a comment. The purpose here is not to comment on other code, and a comment symbol is used merely because of the absence of other mechanisms.

If a comment symbol is entered at each line, entry and removal can be tedious, and individual lines can be missed. If a block comment is used, they typically cannot nest and so consideration must be given to whether the commented section already contains a block comment.

Existing text-based environments can alleviate the issue of the display by giving comments a different appearance (although there is no distinction between actual prose comments and disabled code). Our frame editor gives disabled frames a lightly blurred appearance (Blockly has similar functionality and changes the background appearance). Unlike in traditional editors, it is not possible to comment out part of a statement, for instance missing a closing bracket of a scope, and thus breaking program structure.

The explicit disable function in the frame editor is both visually clearer and less prone to syntax errors. A disabled frame can be re-enabled by the user when required. All frames inside a disabled frame are always disabled; you cannot re-enable a frame inside a disabled block (we believe this would make the program flow too hard to follow). However, inner frames can be easily dragged out and re-enabled.

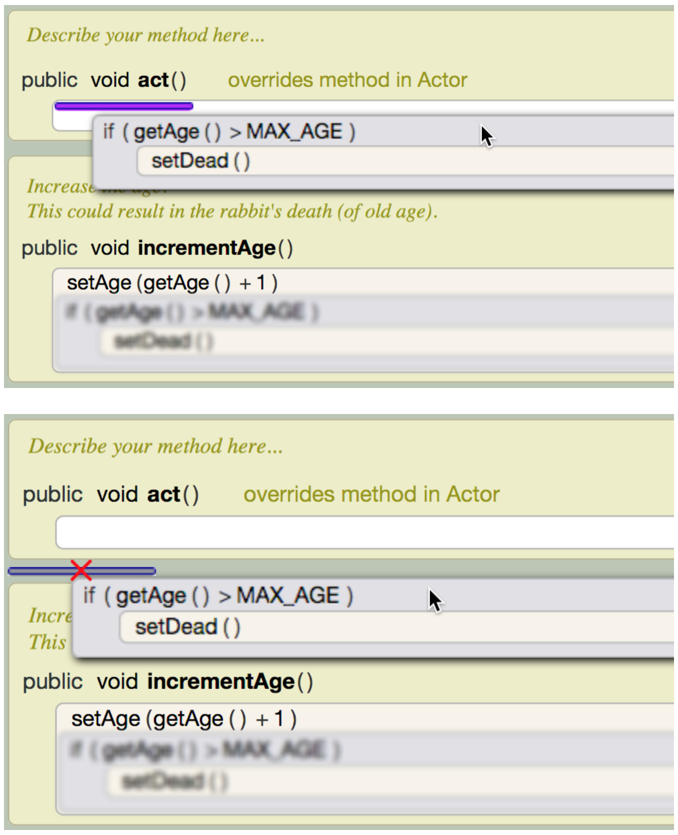
6.2.6. Selection and Clipboard Operations

One theme in frame-based editing is the idea that most operations should act on whole frames (structures), and not parts of frames. This also applies to selection.

Selection of frames always selects a whole frame, or multiple adjacent frames inside the same enclosing frame slot. If the user selects a loop frame, for example, all of its content is automatically included in the selection. As a consequence, a frame selection must always begin and end within the same frame slot (scope).

Multiple selected frames can be dragged simultaneously as described in the next section, or the usual cut and copy operations can be performed. Frames may be pasted at a frame cursor location. So cut/copy/paste may be performed on single frames or multiple consecutive frames (at the same scope), and is also possible on the text content of individual slots.

When one or more frames are selected, inserting a control structure frame wraps the selection in that frame; the selection becomes the body of the inserted structure. This provides an



```
Describe your method here...
public  void  act ( )      overrides method in Actor
      if ( getAge ( ) > MAX_AGE )
          setDead ( )
Increase
This could result in the rabbit's death (of old age).
public  void  incrementAge ( )
    setAge (getAge ( ) + 1 )
    if ( getAge ( ) > MAX_AGE )
        setDead ( )
```

```
Describe your method here...
public  void  act ( )      overrides method in Actor

    if ( getAge ( ) > MAX_AGE )
Incre   setDead ( )
This
public  void  incrementAge ( )
    setAge (getAge ( ) + 1 )
    if ( getAge ( ) > MAX_AGE )
        setDead ( )
```

Figure 12. Cursor indicates valid (top) and invalid (bottom) drop targets. The drag source (at bottom of each) is blurred during drag.

easy way to add a loop or if-guard to existing code, as would be done in text by adding a header at the beginning, then adding a closing curly bracket at the end, then correcting the indent.

6.2.7. Drag and Drop

Treating statements and declarations as interface entities also naturally leads to the expectation of being able to perform drag-and-drop operations on them. Frames can be dragged with the mouse and dropped at alternative locations.

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

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

- 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 are required in a frame editor; the frame is a large target, and does not require selection before dragging.
- In text editors, dragged text may be dropped anywhere, again potentially breaking program structure. The vast majority of potential drop locations 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. While frames are being dragged, the cursor indicates whether a potential target is valid or not (Figure 12).

It makes little sense to allow, for example, placing statements outside of a method body or method declarations at locations where they are invalid. Since many more potential edits invalidate legal structure than maintain it, disallowing invalid manipulation severely cuts down the space of possible user actions. Fewer possible actions lead to simpler and more expressive interfaces and fewer mistakes.

Dragging and dropping in frame-based editing also has advantages over block-based editing. In most block-based editors, it is awkward to drag out individual blocks or two adjacent blocks from a larger body. Generally, you must either drag the blocks separately, or (in the case of Scratch and many others), drag out the body, pick it apart with a further drag or two, then drag back the pieces you wanted to keep in place. Allowing a frame selection followed by a drag provides a much easier and clearer way to perform such a manipulation.

6.2.8. Changing Frame Type

One manipulation that is often highlighted as difficult in block-based or structure editors is that of changing the type of frame. A common example is changing while loops into if statements, and vice versa. There is also the issue of changing, for example, method calls into assignments if you wish to store the result (either because it was a mistake to discard it, or because you now need the result[3]), or vice versa if you decide not to. We have different mechanisms for these manipulations, which we will consider in turn.

We resolve the issue of changing method calls into assignments by direct text editing. If the user types an assignment symbol (the '=' character) in a method call frame then the frame is automatically converted into an assignment frame. Similarly, deleting the assignment symbol from an assignment frame (by placing the cursor before and pressing delete, or after and pressing backspace) converts in the opposite direction. This means existing text-manipulation patterns transfer directly. This does present a question as to whether it is worth conceiving of method call and assignment frames as different

[3]Many Java methods have a side-effect and return a value which may or may not be of interest. For example, the remove method on lists in Java removes the element from the list if present, and returns a boolean indicating whether the item was present or not. Depending on your code, you may or may not care whether it was present or not beforehand.

at all. We believe it is pedagogically useful to consider assignment different from a plain method call, even if technically in the editor, there is minimal difference.

The issue of transforming while to if statements is handled differently, because we do not allow editing of keywords, and thus text-manipulation cannot be used to make the change. Initially in the Stride editor, we were uncertain whether it was worth providing this transformation. Our intuition was that this transformation was rarely needed. However, we have access to a dataset to verify this. The Blackbox data set [29] records the editing behaviour of several million sessions of BlueJ, our beginners' Java IDE.

We looked in the Blackbox data set for the number of edits (and session counts) which introduced a new line of code containing an if statement, a new line of code containing a while statement, or changing if to while (and vice versa) without changing the accompanying condition[4]. We found that (to 3 significant figures):

- In total, there were 732,000,000 edits across 9,160,000 sessions.
- 2,030,000 edits (across 1,030,000 sessions) introduced a new line with an if statement header
- 46,900 edits (across 35,200 sessions) changed an if statement header to a while
- 376,000 edits (across 277,000 sessions) introduced a newline with a while loop header
- 40,600 edits (across 29,400 sessions) changed a while loop header to an if

This means that over 10% of the while loops written later get changed to if statements (minus those which are changed back and forth between while and if, something which is difficult to track). This was a much larger proportion that we had anticipated, and on this basis we introduced a context menu option to change an if statement to a while loop, and vice versa.

6.2.9. Localised History

Our frame-based editor has a standard, class-wide undo system: The standard Ctrl-Z shortcut undoes the last edit, whether it altered the value of a text slot or moved, added or deleted a frame.

A previous study of programmer behaviour, however, found that programmers rarely used the undo feature [28]. Programmers observed in this study preferred backspace for correcting recent typing mistakes. They were often unable to use undo for their other corrections because they had already made subsequent correct edits elsewhere before noticing the error; undo would have removed these first.

To solve this problem, we offer localised history. Each slot keeps a history of the three most recent content values. A submenu in the context menu offers these values for selection to restore an earlier state. This mechanism provides independent undo functions for logical elements of the code, even if other edits have been made elsewhere. Apart from providing more flexible undo, this function also supports easy experimentation: Values may be changed temporarily with the ability to restore previous values easily. Like several features in our frame-based editor, this feature is difficult to implement well in text-based editors [30], but is straightforward in frame-based editing.

[4]This stipulation avoids falsely counting cases where the user has happened to paste completely different line(s) of code over the previous code.

6.2.10. Extending Frames

Many frames, such as while loops, have a fixed structure. Other frames, however, can be extended: An if-statement frame, for example, begins without an "else" clause, but can be extended to add it. Another example is the extension of a constructor definition to add a "super" or "this" call to invoke another constructor. Frame extension is triggered by command keys, just as frame insertion. For example, to add an "else" clause to an if-frame, the user should place the frame cursor inside an if-frame, and press the 'e' key; any frames after the cursor position will be used as the body of the new else clause. Adding "else if" clauses is done in a similar way, with the 'l' key. Deleting such extensions is done by placing the frame cursor at the top of the following frame slot and hitting backspace.

6.3. Navigation

The frame cursor allows navigation and manipulation to be performed with the keyboard, which is the preferred input method for navigation for practised programmers [31].

The frame cursor is always positioned between frames. Up and down navigation moves in steps of single lines by default (mimicking traditional editors). However, combining the use of cursor keys with a modifier key moves the cursor at the current scope level, jumping over compound frames in a single step. This adds a generic method of quick movement; if the cursor is outside of a method, for example, this command moves in increments of whole method definitions and provides quick navigation through a class.

The left and right cursor keys enter a slot in the neighbouring frame, whether it is a text slot or a frame slot, positioning the cursor at the beginning or end of the slot: The left key goes to the end of the last slot in the previous frame, while the right key goes to the beginning of the first slot in the next frame. The TAB and Shift-TAB keys can also be used to navigate slots.

Overall, this key binding largely mimics movement in traditional text editors, but adds generic navigation options for additional structure-based movement.

6.4. Overtyping

Most syntactic elements, such as parentheses, commas and spaces, are automatically displayed annotations and decorations; they do not need to be typed and cannot be edited. A new frame is created with these decorations, and only slots need to be filled in (Figure 13). Using a TAB character or right arrow advances to the next slot.

While this is a common interaction sequence for form fill-in, it goes against the habits of programmers, who are used to

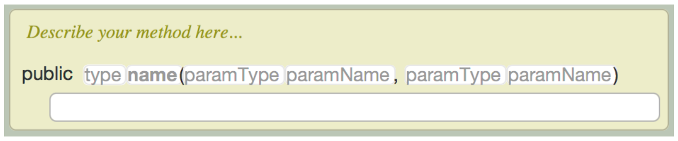

Describe your method here...
public type name(paramType paramName, paramType paramName)

Figure 13. A method frame with empty slots

typing the spaces and other syntactic elements. In the Stride editor, typing the syntax elements is permitted and advances the cursor to the next slot. The programmer can effectively "type over" a syntax element. For example, when entering the method return type in Figure 13, typing a space moves the cursor to the method name field. From there, typing an open parenthesis advances the cursor to the parameter list. Pressing space again moves from parameter type to parameter name. A comma in the parameter name slot creates the additional parameter slots and places the cursor into them.

As a result, an input key sequence that would type the method signature in a traditional text editor also works identically in the Stride editor. This supports the muscle memory of experienced programmers, who do not need to relearn all habits of program entry.

6.5. Editing in Text Slots

When filled in correctly, text slots appear as normal text on the background colour of their enclosing frame (see, for example, Figure 4). This ensures that lines of code can be read as flowing text without unnecessary visual overhead. Text slots that are either empty or have keyboard focus have a white background.

The Stride editor has three types of text slots: *identifier slots* that support entry of an identifier of the supported language, *choice slots* that allow entry of a limited set of fixed values and *expression slots* for the entry of expressions. Using specialised slots for different kinds of token in the syntax tree allows support for more efficient content entry as well as better avoidance and reporting of errors. We will give more detail on each type of text slot in turn.

6.5.1. Identifier Slots

Identifier slots allow the textual entry of program identifiers, but inhibit entry of characters that are syntactically invalid in this context, such as punctuation characters. Typing characters that follow the identifier slot, such as a space or the opening parenthesis after a method name, causes slot advance by overtyping as described above. Special characters can also cause the insertion of optional slots; typing, for example, a comma in a formal parameter name inserts an additional pair of slots for entry of an additional parameter.

6.5.2. Choice Slots

Choice slots allow selection of one of a small number of possible values. They are used, for example, for the access modifiers of method declarations (Figure 14) which have three possible choices.

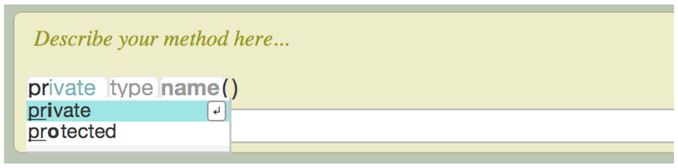


Figure 14. Text entry in a choice slot

Choice slots behave in ways similar to combo boxes in many interface toolkits: One value is always selected (they are never empty), and choices can be made using the mouse or cursor keys. Textual entry, however, is also possible. When the slot gains keyboard focus, a menu of choices is displayed, with the top choice selected. Typing narrows the choice list to those values starting with the characters entered (invalid characters are ignored), and using TAB, return, space, or a right arrow at any point confirms the current selection and advances focus to the next slot.

This method again ensures syntactically valid program structure while allowing overtyping of the whole text if desired, with the added possibility of much faster entry.

6.5.3. Expression Slots

Expression slots allow the entry of expressions, including arithmetic expressions, variables and function calls.

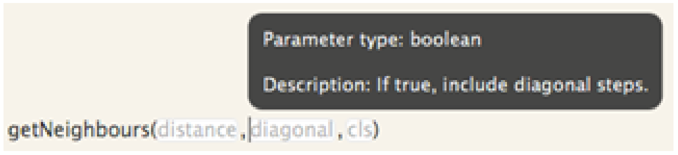
Expressions are structured. When an operator is entered in an expression slot (such as a plus symbol), the expression acquires a new structure, with an operator in the middle between two text fields for each of the operands. The operator itself is not part of the text fields; it can be deleted, merging the text fields again, but cannot be edited.

Multi-character operators are easily inserted. For example, typing "<" (the less than symbol) splits the current text field into two, with the less than operator between them. Typing "=" (the equals symbol) at the beginning of the second field will automatically merge with the less-than operator to make less-than-or-equal-to, just as would happen when typing text.

Entities that appear as pairs of symbols (such as parentheses, brackets and quotes) always appear in full: entering one half enters the other half automatically. While traditional text editors typically also do this for entry (e.g., typing an opening parenthesis automatically inserts the closing one), the link in the Stride editor is stronger: The pair of symbols remain linked and are processed during editing as a single operator. Deleting one also deletes the other, selection and drag-and-drop operations always operate on the complete subexpression, processing both or none of the bracket symbols. Existing expressions can be enclosed in parentheses by selecting the expression in question and pressing the opening parenthesis. This will enter both surrounding symbols.

Again, as before, various advantages flow from the fact that the user edits the structure of the code, not the representation:

- Some syntax errors become impossible to make, cutting the overall error rate.


Parameter type: boolean

Description: If true, include diagonal steps.

getNeighbours(distance, diagonal, cls)

Figure 15. Prompt text and tooltip for parameters

- Edit operations, such as selection and drag-and-drop can guarantee to maintain syntactically valid structure.
- Errors that do occur can be associated more precisely with a particular token in the program source, because there are no structural syntax errors such as missing brackets.
- While structure is under the control of the programmer, representation can be automated. For example, spacing between symbols and arranging line breaks and indentation in long lines can be automated to be consistent and meaningful which can enhance readability.

Automatic adaptive spacing is used in the Stride editor to clarify precedence of operators (higher precedence operators use less adjacent space than lower precedence operators). This technique has previously been used in mathematical editors [32] and in print-outs of code [33] but not usually in WYSIWYG programing editors. It can lead to respacing of an expression as it is entered, which is not ideal but we believe is outweighed by the ensuing readability advantage.

All Java expressions can be entered in Stride's structured expression editor using exactly the same keypresses (although almost all the space characters are redundant due to the automatic spacing), meaning that those used to text will notice no difference on entering text, just in the display.

Again, through the use of overtyping and flexibility in navigation, text entry feels natural to seasoned programmers, with opportunities for faster entry when becoming familiar with the frame editor.

Our expressions are displayed and entered infix. We believe it is important that the entry and display align. While we did not choose to use prefix notation in Stride, in prefix-expression languages like Lisp at least the entry and display are also aligned. Many of the early structured editors displayed infix expressions but required prefix entry. This required that the user understand the abstract syntax tree behind the expressions being created – but novices do not yet understand that an expression is a structured tree. Therefore we felt it was important to be able to enter expressions infix, to match how they are displayed in the editor.

6.5.4. Prompts and Hints

One advantage of using interface elements other than pure text for the representation is the ability to display prompts to provide guidance for expected entry. We have seen examples in the condition of an if-statement (Figure 7) and the definition of a method signature (Figure 13). While these examples are mainly useful for beginners who have not yet memorised the syntax of those statements, there are other situations where these prompts remain helpful for experienced programmers. One such example is found in the actual parameter(s) of method calls (Figure 15). In this example, the prompt text shows the name of the formal parameter, and a tooltip may provide additional information, showing the type and parameter comment.

Many existing program editors also provide helpful content for actual parameters when entering method calls via an auto-complete mechanism. Some enter the formal parameter name (as in our prompts), while others guess at a possible intended value and enter the name of a nearby variable of matching type. None of these is ideal. In the first case, the resulting program text is almost certainly wrong, but appears to have been completed. In the second case, the program may compile and run, without leaving a hint that the programmer may not have considered and confirmed the default choice, potentially introducing semantic errors.

The problem, again, stems from the fact that the interface elements are just plain text: traditional text editors cannot display text to the user embedded in the program source that is not also part of the program, and therefore interpreted by the compiler. (Modern IDEs are starting to develop the use of prompts and pop-up windows for this purpose.)

In the Stride editor, using richer interface elements, we can display the prompt text to the user while still recognising the slot as unfilled, delivering helpful information and more accurate errors at the same time. For example, if you write a method in Java and forget the type or name of a parameter (e.g. "`public void setX(x) { }`"), you will get an unhelpful error "`<identifier> expected`". In Stride, either the type or name slot will be empty, showing a more helpful message such as "name cannot be empty".

7. Interface Elements

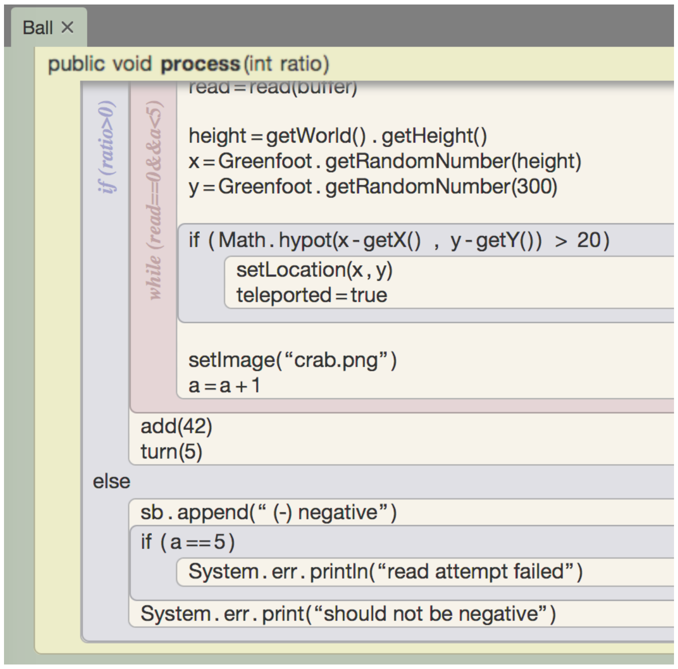
The Stride editor uses various additional interface elements to improve readability and provide additional information to a programmer.

7.1. Method Header Display

The signature of a method contains important information: its name, parameters, and the return type. This information is frequently useful while reading or editing any given method. However, if the method is longer than a few lines, the information often scrolls out of view. In the Stride editor, scrolling up leaves the header visible, sticking to the top of the window, and the body of the method appears to slide underneath it (Figure 16). Thus, the most important contextual information remains visible.

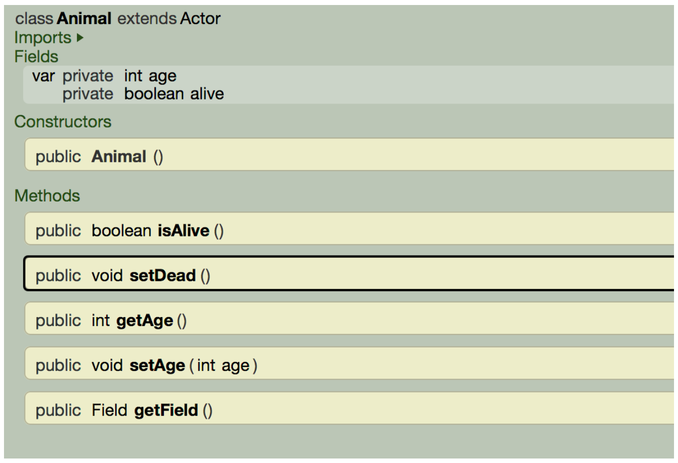
7.2. Long Scope Annotation

Another example where header information is useful is in the use of frames which contain conditionals in their definition, such as if-statements and loops. Figure 16 shows examples of these where the frame header has scrolled out of view; the header information is then displayed in the left margin of the frame, leaving it visible for the programmer.



```
Ball ×
public void process(int ratio)
    if (ratio>0)
        while (read==0&&a<5)
            read = read(buffer)

            height = getWorld() . getHeight()
            x = Greenfoot . getRandomNumber(height)
            y = Greenfoot . getRandomNumber(300)

            if ( Math . hypot(x - getX() , y - getY()) > 20)
                setLocation(x , y)
                teleported = true

            setImage("crab.png")
            a = a + 1
        add(42)
        turn(5)
    else
        sb . append(" (-) negative")
        if ( a == 5 )
            System . err . println("read attempt failed")
        System . err . print("should not be negative")
```

Figure 16. Method signature pinned to top of screen

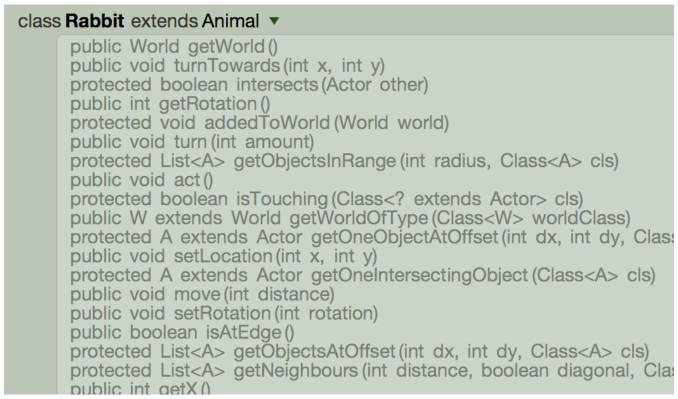


```
class Animal extends Actor
Imports ▸
Fields
  var private  int age
      private  boolean alive
Constructors
  public  Animal ()
Methods
  public  boolean isAlive ()
  public  void setDead ()
  public  int getAge ()
  public  void setAge ( int age )
  public  Field getField ()
```

Figure 17. The bird's eye view

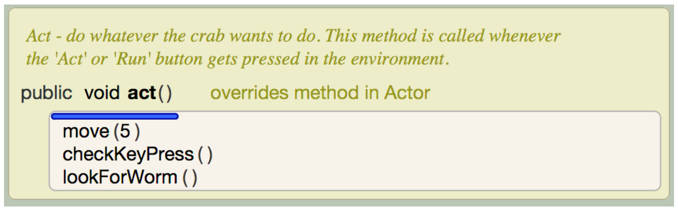
An alternative would have been to treat these frames identically to method frames and pin the header to the top of the display (or, indeed, to also use the left margin for method header displays). Pinning all open frame headers to the top was considered undesirable as it would have resulted in a potential stack of multiple headers, consuming more space and reducing readability.

The more significant context of the enclosing method, and the fact that there is only ever one enclosing method in Stride (which does not support Java's inner classes), justifies different treatment with more prominent presentation.



```
class Rabbit extends Animal ▾
    public World getWorld ()
    public void turnTowards (int x, int y)
    protected boolean intersects (Actor other)
    public int getRotation ()
    protected void addedToWorld (World world)
    public void turn (int amount)
    protected List<A> getObjectsInRange (int radius, Class<A> cls)
    public void act ()
    protected boolean isTouching (Class<? extends Actor> cls)
    public W extends World getWorldOfType (Class<W> worldClass)
    protected A extends Actor getOneObjectAtOffset (int dx, int dy, Class
    public void setLocation (int x, int y)
    protected A extends Actor getOneIntersectingObject (Class<A> cls)
    public void move (int distance)
    public void setRotation (int rotation)
    public boolean isAtEdge ()
    protected List<A> getObjectsAtOffset (int dx, int dy, Class<A> cls)
    protected List<A> getNeighbours (int distance, boolean diagonal, Cla
    public int getX ()
```

Figure 18. List of inherited methods



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

Figure 19. The override annotation

7.3. Bird's Eye View

The bird's eye view is an alternative (temporary) view of the class for quick orientation and navigation. Bound to a command key, it can easily be activated and displays only instance field, constructor and method frames, reduced to their signatures (Figure 17). A second press toggles the display or hiding of documentation (useful to see more details about methods, but reduces the conciseness of the method list) and a third press exits bird's eye view.

As a result, users can get an easy and quick overview over available methods, and easily navigate the class. Up- and down-arrows select method frames, and pressing the Return key or clicking with the mouse returns to the standard display with the selected method in view. (A similar view-only operation is available in Blockly to collapse all visible blocks, but without the functionality to easily navigate between the collapsed blocks.)

7.4. Inherited Methods

Inherited methods are available for invocation in subclasses. Traditionally, the available inherited methods are not easily visible without referring to documentation outside of the class under construction. This causes problems for learners who would use methods from the superclass without understanding where the methods were declared.

In the Stride editor, a small arrow next to the superclass name in the 'extends' declaration allows a list of inherited methods to be unfolded (Figure 18).

This list serves two purposes: It provides easily accessible in-editor documentation of the methods available, and it allows

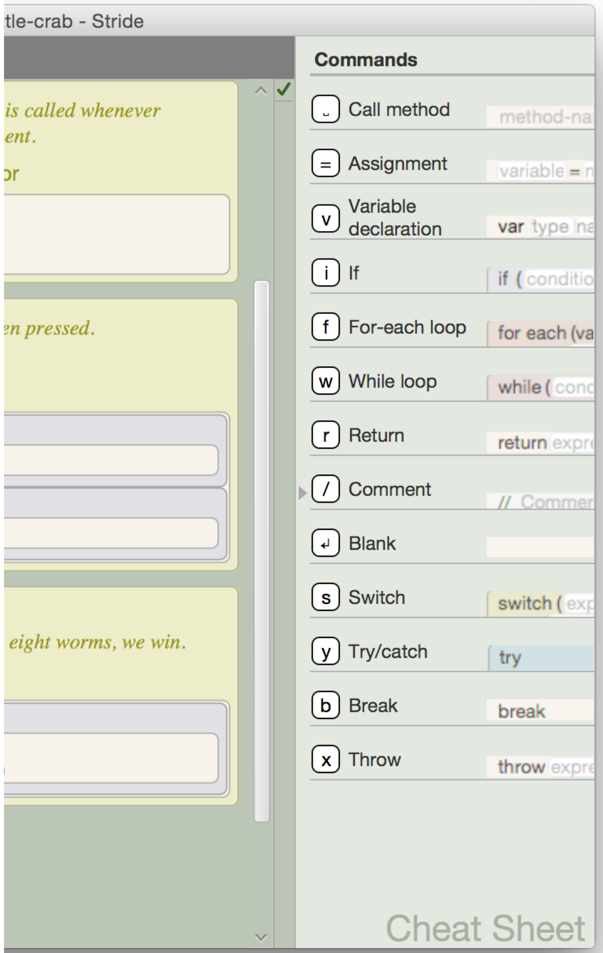


Figure 20. The “Cheat Sheet”, which appears on the right of the editor

the user, via a command in a contextual menu, to override an inherited method. Selecting the override function on an inherited method inserts the appropriate method definition into the current class. It should stop the availability of methods seeming ‘magic’ to beginners, by allowing all available method calls to be seen in a class, either directly declared or shown in the inherited list.

Methods can also be overridden in the traditional way: by simply defining a method with a matching signature. In this case an override annotation is automatically added to the method definition display (Figure 19), serving as information to the reader. This contrasts with override annotations in Java, which the programmer must enter and manage themselves.

7.5. The Cheat Sheet

A small arrow on the right-hand side of the editor allows a separate window pane to be displayed, showing the “Cheat Sheet” (Figure 20). Viewing or hiding this pane is also possible via a shortcut key, and is shown by default for new users.

The Cheat Sheet lists all frames that can be inserted at the current cursor position, together with their command keys. Using the command key or clicking on the frame in the Cheat Sheet inserts the frame into the program text.

The Cheat Sheet serves a similar purpose as the block catalogue in block-based languages: It supports recognition over recall and encourages experimentation for users who are not familiar with the whole range of options. One significant difference is that block catalogues in most block-based languages show all available operations (method calls) separately, while Stride offers only different types of frames. Thus, a method call is shown as a single option, encompassing all possible method calls. Selecting a specific method from those available is left to the code completion mechanism, described below.

As a result of this choice, the number of available options at any point is limited and a complete set of options can be displayed in a relatively small amount of space. (Figure 20 shows the complete list of all available statement frame types that can be inserted at the a typical frame cursor position inside a method.)

The Cheat Sheet is context sensitive: At any time, only valid options are shown. If, for example, the cursor is between method definitions, only commands to insert methods or comments are offered. One potential disadvantage is that if a student wants to insert a frame not available at the current context (e.g. a method, but their cursor is already inside a method) then they will not find what they are looking for. However, we expect that this will only be an issue for beginners, and students will soon learn which elements are available in which context.

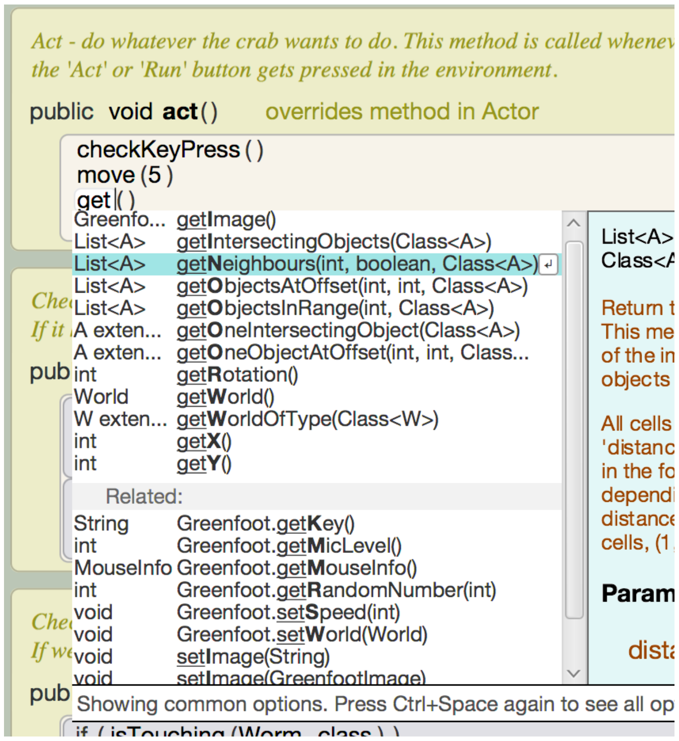
8. Context-Aware Editing

In a traditional text editor, the entire text area is one interface element. If the editor aims to offer context-sensitive support, the editor must infer the type of structure currently being edited from the representation. In a frame-based editor, each part of the code (i.e. each node in the syntax tree) is represented by a separate interface element. The programmer edits the structure, not the representation, so the context of the element edited is always known. As a result, frame-based editors can much more easily offer context-aware editing: we describe several examples of this here.

8.1. Contextual Code Completion

Code completion allows a programmer to insert code more quickly than typing it in full, by selecting “completions” from a generated list, usually chosen by matching the already typed prefix to possible suggestions (e.g. typing “get” would offer the completion “getNeighbours”, Figure 21). Text-based editors often offer completion of variable and method names.

The Stride editor also offers completion of variable and method names – but only in expression slots. If the user is editing a type slot (e.g. a method return type or variable



```
Act - do whatever the crab wants to do. This method is called whenev
the 'Act' or 'Run' button gets pressed in the environment.
public  void  act()      overrides method in Actor
   checkKeyPress ( )
   move (5 )
   get( )
   Greenfo...  getImage()                              List<A>
   List<A>     getIntersectingObjects(Class<A>)        Class<A
   List<A>     getNeighbours(int, boolean, Class<A>)
Che List<A>    getObjectsAtOffset(int, int, Class<A>)  Return t
If it List<A>  getObjectsInRange(int, Class<A>)        This me
   A exten...  getOneIntersectingObject(Class<A>)      of the in
   A exten...  getOneObjectAtOffset(int, int, Class...  objects
pub int        getRotation()
   World       getWorld()                              All cells
   W exten...  getWorldOfType(Class<W>)                'distanc
   int         getX()                                  in the fo
   int         getY()                                  dependi
      Related:                                         distance
   String      Greenfoot.getKey()                      cells, (1
   int         Greenfoot.getMicLevel()
   MouseInfo   Greenfoot.getMouseInfo()                Param
   int         Greenfoot.getRandomNumber(int)
Che void       Greenfoot.setSpeed(int)                 dista
If we void     Greenfoot.setWorld(World)
   void        setImage(String)
   void        setImage(GreenfootImage)
pub Showing common options. Press Ctrl+Space again to see all op
   if ( isTouching (Worm . class ) )
```

Figure 21. Code completion in Stride

declaration type), code completion offers type completions instead (e.g. “St” will offer “String” as a completion). If the cursor is within a string literal, code completion offers a list of frequently used string literals (in Greenfoot’s case: image and sound filenames in the current Greenfoot scenario) as choices. If the user is editing a method name slot and requests code completion, names of methods in parent classes are offered for overriding. In a “throws” clause, only subtypes of type Throwable are offered, and so on.

In short: The fact that the Stride editor maintains code structure more explicitly enables better contextual support for code entry.

8.2. Error Messages

In a text-based editor, a single character (present or missing) can cause a parse error that affects the parsing of the rest of the file. Too many closing curly brackets or an unterminated string literal can affect the parsing of the remaining code. Thus, relevant syntax error messages can be hard to determine and can be affected by distant mistakes.

In a frame-based editor, the location of the source of a syntax error can be much more precisely determined. There is no possibility of unclosed scope, unterminated string literals or missing semicolons. Each syntax error can be attributed to a single slot in a frame.

For example, entering the text “wait(,,);” into Eclipse shows the error “wait cannot be resolved to a type”, as the editor struggles to decide if this is intended to be a method call or a variable declaration. In a frame-based editor, this choice is made explicitly upfront, and entering “wait(,,)” into a method call frame will display two errors stating “parameter cannot be empty”.

In fact, because so many invalid inputs are disallowed in frame-based editing, the majority of remaining *syntax errors* are simply “X cannot be blank”, positioned exactly at the offending text slot.

8.3. Suggested Fixes

Some IDEs offer suggested fixes to a program when they detect an error. Our Stride editor does the same, and usually with less technical effort. In order for a text-based IDE to offer the fix “You are attempting to assign to undeclared variable x; fix by declaring here?”, an IDE must parse the source, determine that this is an assignment statement, check whether the left-hand side is declared, offer this fix, and then manipulate the text to change to a variable declaration. In our frame-based editor, we implicitly know that an assignment frame is an assignment; we must still check if the left-hand side is undeclared, but then implementing the fix simply requires swapping the assignment frame for a variable declaration frame.

These examples demonstrate that offering improved help and support for programmers, while not impossible in text-based editors, is technically easier to implement in a frame-based editor. As a result, we are able, in some areas, to offer better functionality in the first release of a new editor than existing professional IDEs currently offer after many years of development.

8.4. Context-Awareness vs Error Tolerance

There is sometimes a tension between context awareness and error tolerance. As a simple example, when the user is entering content in a type slot, we could only allow entry of existing known types. However, it may be that the user intends to introduce a new type (and thus precluding its entry would be frustrating) or they want it to be incomplete (e.g. they have a List but need to look up its inner type). So we allow the erroneous type to be entered even though we know it to be wrong.

Another example is the context awareness of the frame cursor, where we make different choices in different contexts. We do not allow arbitrary code to be inserted outside methods: we know this to be wrong, and if the user wants to enter an arbitrary code fragment, they can always add it inside a nearby method. More subtle is the case of break frames: we know that they are only valid inside a loop or switch frame. But this means the context relies not only on the immediate parent, but the grandparent or further. Additionally, the user may write the break ahead of enclosing the current code in a loop. We thus allow break frames to be entered anywhere a statement can, and an error will be issued if this is an invalid placement.

9. Implementation

Frame-based editors must store program code and attribute information, such as the enabled/disabled state of frames. It would be possible to store the frame structure as standard text (e.g. as Java code). An advantage then would be the interoperability on the same source files with other editors. We do not do this for several reasons.

Firstly, it requires extra technical effort to read and write the code from/to Java compared to storing it in a structured format. Secondly, if the file was externally edited it could be in an invalid state that is unrepresentable in our frame editor. Thirdly, any details such as white space in the Java code would be difficult or impossible to preserve during frame-based editing. Fourthly, Stride only supports a subset of Java, which makes it awkward if the Java code contains features unsupported by Stride (see appendix).

In short, storing the code as Java text would expose many of the problems that frame-based editing eliminates. Thus, the Stride editor stores the code in a simple structured form, using XML.

One historical advantage of text is that it was portable between tools. But this is not an inherent advantage of text-based programming: the advantage of being able to edit code in multiple editors, and for code written in Emacs to be analysable in Eclipse is not to do with text, it is because there is a standard format for Java code (defined by the Java language standard). There is no reason that Stride code stored as XML could not also be edited, analysed or compiled by other tools (including existing text editors, although this would be slightly unwieldy). Alternatively, Droplet [34] has shown that frame-based or block-based editors can use text as the canonical representation if desired.

We transform the program to Java source code for compilation, and then use a standard Java compiler. Any semantic errors are passed back into the editor for display to the user. (There should be no syntax errors generated from frame-based code, as syntax errors are detected during the Java generation phase and short-circuit the process.)

To support the transition of learners from Stride to Java, we also allow to preview Stride code as Java (using an animated transformation) directly in the Stride editor, and to convert the Stride code to Java for further editing in standard Java form in a traditional text editor. Greenfoot supports development in either Java or Stride, and classes in the two languages are interoperable. This transition is also behind the syntax choices of Stride: with a frame-based editor, the choices of keywords or syntax (e.g. brackets around the if-condition) are a choice solely based on visual design and comprehension rather than ease of entry. We retain similarity to Java to allow an easier transition from Stride to Java.

10. Evaluation A: CogTool

To evaluate the effectiveness of the design features of the Stride editor, we have carried out a study comparing task times of common editing tasks performed in a frame based editor and seven other commonly available and popular programming systems.

The editors used for this study were an earlier prototype of our Stride editor to represent frame-based editing, Scratch, Alice, and StarLogo TNG as examples of traditional block-based systems, the Lego Mindstorms NXT [35] editor as an example of a visual editor with an alternative design, and NetBeans (Java), Greenfoot (Java), and IDLE (Python) as representatives of common text-based editing.

The results of this evaluation have been presented in an earlier paper [36] and are summarised here.

10.1. Cognitive Modelling

To compare the editing tasks performed in each editor, we used CogTool [37], a software tool that automates the creation of cognitive models for the purpose of evaluating and comparing interactions in competing systems. Cognitive models extend on keystroke-level interaction models by including "mental" operators, such as eye movement, reading time and thinking time in addition to explicit interaction events (such as key presses, mouse movements and mouse clicks). It should be noted that CogTool is primarily intended for measuring expert use, and aspects such as novice learning processes are not addressed.

Cognitive models more accurately reflect interaction complexity and time than keystroke-level models, but are more difficult to correctly construct by hand [38]. It can be difficult to judge which mental operators to include at what time, and errors are easily introduced. CogTool automates the creation of the cognitive models, improving accuracy considerably [38]. CogTool records the interaction sequence and uses the "Adaptive Control of Thought – Rational" (ACT-R) architecture, a computer model of human cognition [39] to generate a model of the task.

10.2. The Experiment

Green and Blackwell [40] define six "cognitive activities" in relation to programming activity: "incrementation" (adding new code), transcription (copying a design into code, or copying code from somewhere else), modification, exploratory design, searching, and exploratory understanding. For our experiment, we primarily covered the incrementation and modification categories. These cover common editing tasks, including inserting and modifying statements, deletion and restructuring of code.

Forty-six common editing tasks were defined and carried out in each of the target systems. CogTool was used to record and analyse the resulting interactions. For simplicity of presentation, the tasks were grouped into five categories: adding new statements (n=6), modifying part of a statement (n=8), deletion (n=12), moving code to another location in the program (n=13) and replacing code with another statement (n=7).

10.3. Results

The recording of the sessions allowed us to observe the number of steps involved in each task, and the cognitive models generate simulated tasks times. The mean task times

Table 1. Mean editing times in seconds for various editing tasks. Lower times are better. Best time highlighted.

	Scratch	Alice	Mindstorms	StarLogo	Python	NetBeans	Greenfoot	Frame editor
Insert	4.87	6.56	15.95	12.50	3.95	5.09	3.80	**1.64**
Modify	5.61	7.05	9.10	8.29	5.44	5.53	5.84	**5.01**
Delete	5.44	2.56	6.51	5.59	5.53	7.82	6.53	**2.42**
Move	5.48	**3.09**	3.82	4.97	5.18	6.01	12.20	4.84
Replace	9.80	8.90	18.55	11.55	5.16	5.10	4.69	**2.29**

produced by the model for each system, grouped by task type, are presented in Table 1.

An analysis of variance (ANOVA) shows that the differences between systems are significant in all groups except "Modify". Mean task times are lower for the frame editor in all groups except "Move". These results support our hypothesis that frame-based editing can improve on the efficiency of common editing tasks compared to existing editors.

11. Evaluation B: Frames vs Text in Middle School

We aided in an evaluation of frame-based editing against text-based editing among middle-school students. The results have previously been published elsewhere [41] but are briefly summarised here.

18 middle-school students were set a task in Java in Greenfoot, while 14 other students were set the same task in Stride in Greenfoot. This allowed the task and IDE to be kept constant between the conditions, with the only difference being the editor paradigm: text-based Java or frame-based Stride. Students were given a 25-minute introduction followed by the 60-minute task.

The students in both conditions rated the activity as low frustration and high satisfaction. No differences in satisfaction were found between Java and Stride; there exists, however, a potential of a ceiling effect. Students in the Stride condition advanced through the task instructions faster than the Java side and completed more objectives with less idle time than Java. Less time was spent making syntactic edits in Stride than in Java, and less time was spent in Stride with non-compilable code. Stride users encountered issues with remembering to press the command key to insert a frame rather than just typing immediately.

12. Evaluation C: Experienced User Study

We conducted a small user study with participants who had programmed before to collect their opinions on using the new editing paradigm. The results of this study have not previously been published.

12.1. Method

We recruited 23 participants by advertising on the postgraduate student mailing lists of the Computing, and Engineering and Digital Arts departments at the University of Kent. The study was approved under the ethics procedure at the University of Kent.

- What, if anything, did you find more difficult in a frame-based editor compared to a text-based editor which you might usually use?
- What, if anything, did you find easier in a frame-based editor compared to a text-based editor which you might usually use?
- Did you complete task 1? Did you have any particular difficulties doing so?
- Did you complete task 2? Did you have any particular difficulties doing so?
- Did you complete task 3? Did you have any particular difficulties doing so?
- If the frame-based editor had all the advanced features of a professional IDE (e.g. easy project organisation and navigation, refactoring support, version control, etc), would you consider using a frame-based editor over a text-based editor? Whether yes or no, please explain your reasons.
- Briefly describe your previous experience in programming: what language(s) you commonly use, how many years you have been programming, and what editor(s) you usually use.
- Do you have any comments you want to make about the frame-based editor which did not fit into the previous questions?

Figure 22. The free-text questions asked in the electronic survey.

In one parallel session, the participants were seated at computers and given a 5-minute verbal introduction and demonstration of the Stride editor on a projected screen. Participants then worked through three sets of tasks in order. Task 1 involved detailed steps to carry out various editor interactions (e.g. inserting frames, deleting frames, editing frames). Task 2 asked participants to enter large pieces of provided code into the editor, and task 3 (for those few who finished all of task 2) was to program extensions to the Greenfoot project they had entered in task 2. Participants worked on the tasks for around 50 minutes, with most near the end of task 2 at the end of the allotted time. Subsequently, they were asked to fill in an electronic survey containing some open questions with free-form text responses (shown in Figure 22) and some questions to be answered using a 7-point Likert scale (shown in Figure 23).

12.2. Results

Participants were asked (as free text) to provide details on their programming experience. 21 of the 23 participants indicated how long they had been programming: the median was 7 years. The most popular programming languages were C++ and Java (17 of 23 participants in each case) and the most popular editors mentioned were Visual Studio (7 participants) and Eclipse (5 participants). We did not collect data on na-

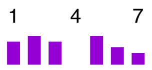
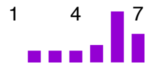
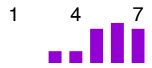
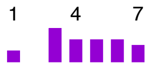
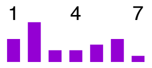
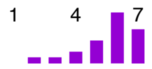
Which editor...	Mean (1 d.p.)	Distribution
...is faster to program in	3.6	
...requires fewer keypresses	4.7	
...makes code easier to read	5.6	
...makes it easier to insert new code	4.4	
...makes it easier to edit existing code	3.4	
...makes it easier to delete existing code	5.6	

Figure 23. The results of comparative questions on a 7-point Likert scale, where 1 meant text-based editor, and 7 meant frame-based editor. Thus, 4 is neutral, numbers lower than 4 favoured the text-based editor, above 4 favoured the frame-based editor.

tionality but we observed that many participants did not have English as a first language. We do not believe this affected their ability to follow the instructions (they were all studying at an English-speaking university), but it did lead to some slightly broken English in the responses.

The results for the Likert scale questions are shown in Figure 23 with details of the distribution of the responses on the 7-point scale.

12.2.1. Negative Comments

To analyse the open text responses, we created categories for the responses and tagged the answers given by the participants, which we will detail now, beginning with the negative issues.

Learning Curve: The issue of the learning curve was raised by 9 participants. By far the most frequently cited issue was remembering to press a command key to insert a frame before typing the content: "The shortcuts to start a new frame take a bit of getting used to (in the beginning I'd often open a frame by mistake by starting typing my line of code)." One participant noted: "especially with the assignation [i.e. assignment frame] because we are use to write the left side first before telling it's an assignation".

Navigation: Four participants mentioned frame cursor navigation being difficult, and another four mentioned some specific cases where they found it difficult to navigate between certain slots using the keyboard. We plan to address these issues in the software; it is our belief that they are solvable usability issues rather than inherent aspects of the design (as, for example, the command keys are).

Expression Editing: Seven participants mentioned that editing text at the slot level could be more awkward than in a text editor. Two highlighted the difficulties of editing incorrect brackets when the brackets must remain paired (which prevents deletion of a single bracket to move it elsewhere). This is a cost of the advantage of never having mismatched brackets. We plan to investigate whether there are usability additions (e.g. allowing the drag of an opening or closing bracket to move it) that can mitigate this problem.

Software Issues: Some responses mentioned issues with the software, such as the code completion interface being slow, one or two small bugs, or the lack of a "saved" indicator when the work is auto-saved. Although we intend to address any problems with the specific software, they do not relate to the editing paradigm design, so we will not discuss them further here.

Two participants also offered the viewpoint that frame-based editing was not as advantageous as it perhaps would have been when text was first used with primitive text editors, because many IDEs now offer what they viewed as similar functionality: auto-indentation, forms of scope highlighting, text selection and mouse-dragging.

12.2.2. Positive Comments

The positive comments were categorised as follows.

Easier to read: As shown in the Likert scales in Figure 23, most of the participants found the frame-based editor easier to read, and this was confirmed by several mentions in the free-text responses (e.g. "It's more structured and easier to follow your code."), although none went into great detail as to why (two mentioned the colouring and scope highlighting).

Easier to insert new code: The Likert scales for easier to insert new code show a slight favouring of frames, and the fewer keypresses most likely relates to inserting new frames too (since the command-keys is one of the major differences to text). This is confirmed by many text responses, e.g. "I like the speed with which you can get code created. I like not having to worry about saving, semi colons, braces and parentheses."

Easier to delete new code: This was one of the Likert items that most favoured frame-based editing, and was backed up by several text responses, e.g. "It is nice being able to treat a whole frame as one unit for the purposes of moving / deleting etc. Since it is broken up into frames you won't get errors with missing / extra closing braces after such tidying up of code."

12.3. Study Conclusions

The main problem participants highlighted was that of the learning curve, and in particular, remembering to press a command key to insert a new frame, rather than just beginning to type. It is impossible to tell from a 1-hour session how long this problem persists, but many participants seemed to view the issue as a temporary initial hurdle rather than a permanent problem.

The participants were almost equally split as to their preference between frames and text. We view this as a very positive result once you consider the context: after only one hour with a new paradigm which has a difficult learning curve, many participants (all of whom were quite experienced in programming) indicated that they would consider using frames over text.

To avoid overloading participants, we did not make mention of all the frame-based editor features listed in this paper. Several of the more advanced features – such as bird's eye view, the long scope annotation, the inherited methods pane, tooltips, and overtyping – were not explicitly mentioned. Some participants may have discovered a few of them, but we focused primarily on the core interactions and features (the frame cursor, frame insertion and deletion, etc). It may be that participants would have been more positive about the possibilities for long-term use if they had also seen the more advanced features.

Participants praised the readability of frames, and the speed/ease with which you can create new code, manipulate existing frames and delete whole frames. The main sticking point appears to be the editing and navigation of slots and their content. Given that this hybrid approach of text and frames (rather than pure-text of mainstream programming, or only-blocks of Scratch et al.) is novel, there may still be room to improve the text aspect so that any usability issues are minimised or removed.

12.4. Threats to Validity

Participants knew that the experimenters had created the Stride editor software being tested, which created a risk of response bias. To mitigate this, we assured all the participants that their responses were anonymous, and we avoided collecting detailed demographic data (age, sex, etc), and used an online form, to emphasise the anonymity of the responses.

It is possible that being non-native speakers may have hampered the performance of some students both in terms of the software (two students mentioned the QWERTY keyboard layout of our PCs as being a problem for them) and the instructions. However, all students achieved a similar amount of progress through the tasks (all completed task 1 successfully, and most were near the end of task 2) and since we are not using task progress as a measure for our analysis, this does not seem to present a serious issue.

13. Related Work

There are two strands of previous work which relate to frame-based editing: the work on structure editors from the 1980s and early 90s, and work on block-based editing in the past ten years.

These strands of work suffered different fates. Structure editing had a period of limited popularity but – apart from a few Lisp editors – is barely used today. Block-based editing has been a resounding success and is now the predominant form of programming for younger age groups, via systems such as Scratch, Snap!, Blockly editors and many others. Given that our work relates to both, it is instructive to investigate the fate of structure editors, and the relation between the two strands of work.

13.1. Structure Editing: First Attempts

Structure editing started to receive attention in the late 1970s, beginning with systems such as the Cornell Program Synthesizer [42]. Work on structure editors continued throughout the 1980s, producing systems such as GENIE/GNOME [43] and Boxer [44] that saw some success, as well as many other systems. There is a noticeable pattern in the literature, with excited descriptions of new structure editors in the early 1980s giving way to critical retrospectives of structure editors' failure in the late 1980s and early 1990s: "Despite [structure editors'] potential, they have not been successful, as evidenced by limited use. In general, they are perceived as being too difficult to use and the benefits of their use are outweighed by the difficulties." [45].

So why did structure editors fail? We believe there are four main reasons. Firstly, we should consider the computing environments in the early 1980s. Text-mode displays were the norm, mice would only become popular later in the 1980s, and high resolution full-colour graphical displays would only become widespread in the 1990s. Also crucial was that the discipline of human-computer interaction (HCI) was in its infancy in the 1980s. Interface design has improved immensely in the 30 years since. All of these contextual issues meant that the interfaces constructed for structure editors were hampered in what they could achieve.

Secondly, there was a lack of flexibility in the structured approach. Over time, the creators of structure editors noted that structure editing could be awkward and overly restrictive at the lowest levels in the syntax tree. Expressions were difficult to manipulate, and were entered differently to their display in many cases. They were entered prefix, with the operator specified before the operands because the operator is higher in the syntax tree than the operands, but displayed infix as is standard in text editors (outside Lisp). Expressions also caused issues with cursor movement: the cursor was generally moved around the syntax tree which was not visible at the expression level. For example, moving from 1 to 4 in the expression (1+2)*(3+4) required moving "up" two levels (once to the addition operator, once more to encompass the whole bracket), then across one level (to the other bracket), then "down", and right. But these concepts of "up" and "down" are abstract and not visualised; the user must understand syntax trees before being able to navigate their program. As Miller et al. [46] pointed out, structure editors were already quite different from text editors in that the selection in a structure editor is almost always a node and thus a range of text, rather than a point (between characters) as users were used to in text editors. These problems led to many systems providing hybrid editors, where code could be entered either in text mode or in structure mode. As per Minör [47], this could create difficulties for users in obtaining a consistent mental model of the lexical structure of their program code. Welsh and Toleman [48] suggested that users do not think of their program as structured all the way down, which relates to our decision to provide different editing at different levels of the syntax tree.

Thirdly, we believe a further issue was that programming instruction in the 1980s was primarily university-based. First-year university students learning structure editing would be required to switch to text-based editing (ready for exit into industry) within the next year or two. Thus structure editing had only a small window available: it had to provide sufficient benefits that it was worth using for only a year or two, before students had to make a switch which would be unnecessary if text-based tools were simply used throughout. As we will discuss in section 13.3, this is a different environment than the one in which block-based editing would achieve great success.

Fourthly, there were elements of over-design in the editors. Many, if not most, of the structure editors were not a single editor, but rather editor generators. Given a formal language grammar, the generator would automatically produce an editor. As later work in this area pointed out [45, 49], this led to awkward editing interactions. With present-day interface design sensibilities, for example, we would not expect that given a description of a database table, we could automatically generate a perfectly usable web form/user workflow to populate the table. In our own work we have made several decisions specific to the Stride language which would not generalise to other languages: We believe an automated editor generator is highly unlikely to produce very usable editors in each case. (We note that similarly, Scratch and Snap! are editors for specific languages, although Blockly provides a counter-example as it is a framework for creating block languages.)

Crucially, none of these reasons for the failure of structure editors strike at their core argument: that text is an inappropriate way to model structured program code. (Note: we use structured here to refer to the lexical structure of code, not the semantic structuring that "structured programming" referred to in the 1960s). Instead, the arguments were that the structure editors were too unusable. The open question going into the 1990s was: did structure editors fail because no-one had yet worked out a way to produce a more usable editor than a text editor, or would this never be possible?

A fascinating glimpse of the potential of structure editors was provided near the end of the first wave of interest, in a 1992 special journal issue on structure editors. Minör [47] describes there a possible direct manipulation structure editor, and shows a prototype interface (reproduced here in Figure 24, overleaf). We would now identify this as looking like an early prototype of the today's block-based editors. So the structured programming era ended by pointing towards block-based editing, just as interest in it died. Over a decade later, these ideas would be independently reinvented by later work such as Agentsheets [50] and Scratch to great success, as discussed in section 13.3.

13.2. Structure editing: Recent Work

There were isolated pieces of further work on structure editors after the 1990s, followed by several recent examples of work in the area. In 2006, Ko and Myers [31] described Barista, a structured code editor generator (see earlier comment) which allowed editing in text or structured mode as discussed earlier. Barista was a prototype, with a mouse-focused interface for adding new blocks and little support for keyboard navigation or manipulation.

A recent effort at structure editing was presented by Osenkov for C# [51]. The indentation level is under editor control, as in our frame-based editor, and keywords are entered through a classic code completion menu. At times (when the cursor is at the beginning of what we call a frame slot) the cursor shares similarities with a frame cursor. The type of syntactic elements, however, is not explicitly declared by the programmer on entry (as it is for frames through the use of a command key), but rather is deduced from the textual representation once the user has written enough text to resolve the ambiguity between different possible constructs. Thus the C# structure editor does not automate the creation of boilerplate to the same extent, nor does it prevent subsequent editing of keywords.

JetBrains MPS [52] is a framework/generator for creating structure editors. MPS does not have the concept of frames as first-class entities which can be picked up and manipulated, and is closer to classic structure editors than to block-based languages. With no frame cursor, navigation and selection revolve around what we call text slots in our editor, meaning that direct manipulation of existing code is not as easy, especially with the mouse. MPS also lacks the visual presentation of frames.

The Envision editor [53] is structural, but moves much further away from the traditional text-based presentation with methods arbitrarily placed on a two-dimensional grid and the use of icons instead of keywords, and provides alternative visualisation displays such as a flowchart view. It is designed to be possible to visualise large code-bases on a single large two-dimensional display. Envision focuses on keyboard-based interactions, in contrast to our editor, which supports keyboard- and mouse-based interactions.

An alternative approach for touchscreen devices has been created by Almusaly and Metoyer that uses specialised buttons to generate common patterns in code [54]. Like our work, this greatly reduces the number of keypresses required to enter core program code. The keyboard is primarily focused on the entry of code, and does not have the structured frame manipulation that our editor provides.

Other editors support structure editing modes. One example is the ParEdit mode in Emacs, which prevents unbalanced parentheses and curly brackets. Gomolka and Humm [55] described a structure editor for Lisp which also prevented unbalanced parentheses. These tools match frame-based editing's advantage of avoiding unbalanced parentheses/scopes, but do not address issues such as easy automatic keyword insertion, illegal edits of syntactic structure or typing of syntactic symbols. Lisp advocates may argue that this is because control structures should not be special cases in the language; we would argue that by making them known special cases, we gain significant advantages in editing speed and readability.

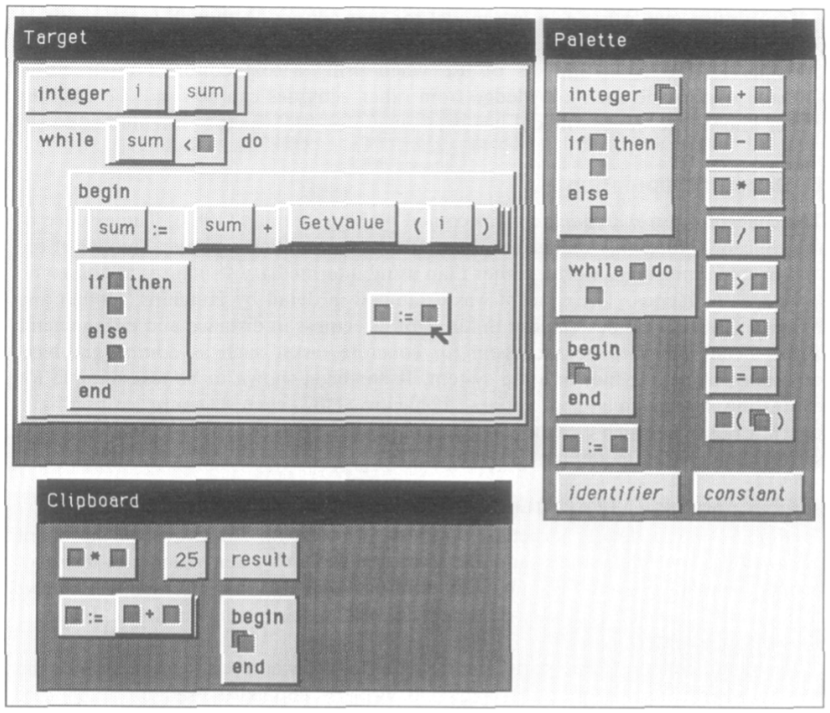


Figure 24. Editor interface proposed by Minör, 1992.

13.3. Block-based Editing

Block-based editing was in a sense a reinvention or re-imagining of many of the ideas of structure editing. It is, therefore, instructive to consider why it succeeded where structure editing failed. We believe there are two main reasons.

Firstly, block-based editing had a more usable interface, informed by fifteen years further development in HCI, and also the ability to use high-resolution, full-colour displays with a fast graphical system. This eliminated many previous constraints on designers of program editors.

Secondly, some of the concerns about structure editors do not apply in blocks' target audience of young children. There are few worries about the transition into text, or compatibility with other tools or many of the other issues which dogged structure editing in contexts such as universities or industrial use. Thus, all blocks had to do was to provide an easier interface for that age group – and the elimination of syntax was an even bigger advantage for young children (who tend to struggle with the precision required in text-based syntax) than older learners.

Meerbaum-Salant et al. [56] examined some of the downsides of block-based editing, highlighting an extremely bottom-up approach, and an issue with "extremely fine-grained programming", where users split their programs into particularly small fragments, making them hard to read. This arises from the way that Scratch and most other block-based editors allow execution of single blocks, and also allow programs to be split into very small event handlers. Neither of these characteristics are present in our frame-based editor: code must be contained in one single coherent class, thus not allowing either of these issues to occur. (There may be other bad habits which frame-based editing permits – something to investigate in future.)

Many block-based program editors have been designed, most quite similar in design to Scratch, the system we have used as our primary example in this paper. We will focus here on work performed to extend block-based programming to be more powerful, or closer to text-based programming.

Alice 3 [57] is a block-based environment that is closer to text programming than many others. It uses method call syntax similar to Java rather than phrases resembling natural sentences, and code is organised into methods. However, like many other block editors, it lacks support for keyboard navigation and entry and does not include many of the other possible representation improvements presented here.

Alice 3 has a text (Java) preview mode, as do other editors such as Tiled Grace [58] and Droplet [34]. The latter two editors allow for a two-way translation between blocks and text representation of the exact same code, with Droplet even preserving the indentation pattern of the original text in its blocks view. This idea of a hybrid editor (editing code either as blocks or as text) has similarities to the hybrid structure editors described earlier, which allowed editing as structure or as text. It is our belief that a successful editing paradigm should not need such a hybrid mode; we seek to merge blocks and text in one representation rather than requiring two distinct modes.

A significant recent effort in block-based editing to add keyboard support is GP [59]. GP is close to Scratch and Snap!, but seeks to improve accessibility, add keyboard support and generally improve the design of block-based programming languages. The designers have added a block cursor very similar to our frame-based editing cursor, which allows navigation along the same lines. They have chosen a more search-focused paradigm to entering new blocks (since GP has many more blocks available than Stride has frames) rather than our command keys, but GP and Stride are evolving along similar lines (thanks in part to discussions between the two teams).

There have been attempts to apply block-based programming to mainstream programming languages, such as C [60]. This acts as a proof of concept for the transferability of block programming to other languages, but does not add any new editing features to mainstream block editors.

TouchDevelop [61] has several modes, from a classic block-based view through a hybrid view to a mostly text-based representation. The hybrid view includes elements from classic structure editing and frame-based editing; some fields are free text entry like our text slots, while others are similar to the choice slots or code completion, with a grid-based touch-oriented restricted list of selections. It employs a classic text cursor instead of a frame cursor, but syntactic constructs (such as if-statements) can be selected by typing, in a way that is slightly faster than IDE template code completion.

TouchDevelop allows manipulation by drag-and-drop similar to that for frames and similar schemes for frame selection and surrounding a selection with compound statements. Being touch-oriented, the keyboard navigation and manipulation are less flexible than in frame-based editing, resembling classic structure editing. For example, it is not possible to enter a non-existent variable name in an expression, intending to declare it later – an intermediate error state that we explicitly choose to allow in frame-based editing. (This is a significant difference in programmer flexibility: A study of programmer behaviour [28] found that around one-fifth of all name edits resulted in the use of an as-yet undeclared name.) TouchDevelop does not contain the structured expression slots of frame-based editing, nor many of the informational display elements described here, but its hybrid mode, and GP, are probably the most similar of the existing systems to frame-based editing.

13.4. Contribution and Novelty of Frame-Based Editing

The related work described here has a complex correspondence to frame-based editing. Several individual features of frame-based editing are present in isolation in other editors, while we believe others are entirely novel. The contributions of frame-based editing encompass these new features and new combinations of existing features into a robust editor publicly released and available for serious use:

- The horizontal frame cursor, as originally developed by McKay and Kölling, is a new feature for block-like programming (as a target for keyboard-based insertion and selection rather than just as a drop target), which enables or simplifies a variety of keyboard based editing interactions, and adds new interactions such as wrapping a selection into a control structure.
- The combination of keyboard overtyping, bracket balancing and automatic spacing in expression slots (and prompt text for parameters), while partially available in existing IDEs, is a novel way to enter expressions.
- Keyboard-enabled frame extensions are a new way to manipulate blocks that can be arbitrarily extended with additional body elements.
- The display of inherited methods as a foldout display within the body of the code itself (rather than as a pop-up display) is not present in other editors.
- The contextual code completion is not available outside professional text-based IDEs, and here goes beyond them, e.g. restricting the types available for catch clauses.
- The Cheat Sheet provides help for textual entry not previously available for keyboard-driven programming.
- The localised per-frame/per-slot history is a novel way to support returning to earlier versions of individual elements of program code.

14. Discussion

Our frame editor aims to fuse benefits of blocks and text editors to increase usability, whether for writing new code, reading or manipulating existing code. We believe there are several advantages for novice programmers compared to block-based and text-based programming:

- Easier to enter new code: Frames save typing much text-based boilerplate, while keyboard commands allow faster entry than block-based programming. Expressions can be entered via a keyboard rather than through mouse-controlled drag-and-drop operations.
- Fewer errors: Many syntax errors, including those related to balancing brackets, punctuation and semicolons, are eliminated.
- Better error messages: Those errors that do remain have clearer messages and more accurate error locations than in text-based programming.
- Readability: Frames are more easily readable than text at the statement level (for example by depicting scope more clearly). At the same time, its look is designed to have lower visual overhead and offer better visual text flow than current block-based editors. This makes programs more readable than in block systems for larger segments of code.

We believe that expert programmers also benefit, particularly in contrast to text-based programming:

- Faster entry: The number of keystrokes required to enter code is significantly lower than in traditional text editors. In addition, there is no responsibility on the programmer to spend time maintaining the layout of code.
- Readability aids: The highlighting of scopes, improved display of method headers and annotations of long scopes all serve to aid readability of standard programs.
- Better navigation: The Stride editor offers quicker navigation for some use cases than many traditional text editors.

This includes easy navigation at the method level and the availability of the bird's eye view.

Many of these beliefs have been backed up in our user studies. Middle-school students spent less time with non-compilable code (in effect, had fewer errors) in Stride and were able to proceed faster than students using Java on the more advanced parts of a task. Expert programmers who tried the system agreed that it made program code more readable, and that inserting new code and deleting existing code was faster in Stride than in text-based systems.

14.1. Separation of Code and Representation

In a frame-based editor, the structure and content of the code and its representation on screen are decoupled.

This removes long-standing problems in programming teams which include members with different visual preferences. Arguments about the preferred depth of indentation or the best placement of curly brackets disappear.

In traditional systems, if one team member changes the number of spaces used for indentation, version control systems would register changes for all lines of code (creating possible conflicts with other edits) even though the code structure is unchanged. Layout changes are indistinguishable from code changes. Thus, the same layout preferences must be imposed on all team members.

In a frame editor, the visual appearance (including depth of indentation and other layout details, including colours) can be a personal preference of any team member. The code structure is shared with the team; the visual representation can be separately customised by any individual programmer without affecting others.

14.2. Edit Flexibility and Tolerance of Error States

The Stride editor allows entry of many erroneous statements and expressions, even where they could feasibly be prevented. When entering a method call, for example, it would have been possible to provide an interface mechanism that only allows entry of existing method names, thus avoiding possible mistyping of identifiers. We chose not to implement this restriction.

While such a mechanism has the potential to prevent some errors, it would also prevent some valid use cases and potential work patterns. A programmer might, for example, conceive of a new method while implementing an algorithm, write a call to this new method first, with the plan to add the definition of that method later. An editor that only allows calls to existing methods would not allow this sequence; our Stride editor does.

Similarly, during the editing of an expression containing multiple operators, the code might be temporarily in a syntactically incorrect state. An editor could force a user to fix a recognised problem before allowing entry of additional code, ensuring that any sub-expression is syntactically correct before allowing the cursor to move away. (The Visual Basic editor used to have this restriction: An attempt to move the cursor away from an incorrect line caused a modal error dialog to appear; this was perceived as deeply frustrating by many users.)

We consider these error prevention possibilities too restrictive. Users must be allowed the flexibility to choose their order of work, including the ability to leave parts of work half finished and incorrect to work on or consult other parts of the system.

We believe that the over-restrictive nature of the systems is a further significant reason that previous attempts at popularising structure editors failed [45]. Structure editors managed to prevent many errors, but most locked users into fixed workflows, resulting in frustrated programmers reverting to free form editors.

In the Stride editor, we do not prevent entry of many incorrect code segments, choosing instead to subtly indicate erroneous code (via a red underline) without blocking the programmer from additional manipulations.

Deciding the exact line to draw – how much help (and restriction) to provide, and how much freedom (and errors) to allow – provides an interesting and complex design space. The concrete decisions made in each individual editor implementation will strongly colour users' views and opinions, determine acceptance or rejection of the tool, and ultimately contribute to success or failure of achieving user acceptance.

Whether we have achieved an acceptable balance in the implementation of the Stride editor remains to be seen. We considered many cases and attempted to make reasonable choices, but whether we have succeeded will only be known after a longer period of use by a larger user base.

14.3. Modal Entry and Learnability

Text editing in a frame editor is modal. Pressing a key in a text slot has a different effect from pressing the same key when the cursor is in a frame slot. The former enters the key verbatim, while the latter is interpreted as a command key that may enter a frame.

Modal interfaces are known for their potential to introduce confusion and misinterpretation in interface operations. Users may misread the mode and their expectations of command entry effects may be at variance with the actual system state and behaviour. Thus, in many situations, it may be beneficial to avoid modal interfaces if possible.

Some examples exist, however, of very successful and accepted modal interfaces. Paint programs, for instance, often offer a variety of paint tools, and the current choice of tool represents a mode. This mode is typically indicated by the shape of the mouse cursor, which may take the form of a pencil, an eraser, or a paint bucket.

Modal interfaces can be successful if the existence of the modes mirrors a user's mental model and the current mode is sufficiently clearly indicated for users to be aware of it at all relevant times.

In the Stride editor, the mode is indicated by the shape of the input cursor. This mirrors the successful indicators in paint programs: Because the indicator is, by definition, at the point of the user's focus, it is implicitly noticed and hard to overlook.

The text cursor takes a very traditional shape, indicating text entry as in common editors. The frame cursor has a very different, distinct and prominent shape and colour, signalling different behaviour.

Whether the cursors as visual mode indicators are sufficient will remain to be seen with increased use of the Stride editor. Initial observations in user trials are promising: Users seem to adapt to the two-mode editing model easily and quickly, without the need for formal introduction or explanation.

One misinterpretation remains longer than others after a short time of use of Stride: When entering method calls, novice Stride users often forget to issue the method call command (space) before starting to type the method name. Just starting to type the name seems a habit that is harder to break than with other commands, perhaps because no modern language uses a keyword before method calls, whereas other constructs (if, while, etc) do begin with a keyword. Whether this habit disappears after some time remains to be seen; currently, there are not enough longtime Stride users to assert this either way.

It is our intention to conduct further detailed user studies to assess learnability and usability issues. One of our existing studies (evaluation B) already turned up one example issue: the ability to delete an entire frame by placing the frame cursor before/after it and hitting delete/backspace respectively was used accidentally by students who then removed a large piece of their program without meaning to. The students did not generally think to undo, preferring instead to re-enter the code. We have now added an overlay after deleting a large piece of code which offers to undo it. We believe that other issues will be on this scale: minor to medium usability issues which can be designed against, rather than serious flaws in the interaction model.

14.4. Code Folding

Some text-based program editors include support for code folding: Selected segments of code (usually constructs which in our editor are represented by frames) can be temporarily "folded in" to reduce them to their header, similar to the bird's eye view described earlier. Frame-based editing and code folding clearly can work well together – we could easily provide a clickable control (and a shortcut key) which reduces any given frame to a single line. Blockly already does this for block-based editing, for example.

In Stride, we have chosen not to do so. The main reason for this is our personal belief that code folding is not very useful when good navigation functionality is provided. However, it is clear that code folding can easily be implemented in a frame-based editor.

14.5. Accessibility

Making programming accessible, for example to blind programmers, is a difficult challenge [62]. Furthermore, it has been noted as a particular issue for block-based languages [63] (and was a motivation for keyboard support in GP [59]), due to their focus on a visual display and mouse-based manipulation – items less amenable to supporting screen-reader technology than text-based display and keyboard-based manipulation.

It is a shame that the visual aspect and mouse-focus has prevented accessibility support, as the fundamental model of block-based (and frame-based) editors is more suited to accessibility. The reduction in syntax means that blind users do not have to worry about entering or correcting the syntax; with a screen reader, the users must either hear "semi-colon" read out all the time, or if it is not read out, will find it harder to correct when it is missing. The inherent chunking of the code into semantic units should make it easier to provide screen readers with relevant information, compared to the line-based approach that existing screen readers take.

Frame-based editing has several features that should allow it to be more accessible than existing block-based editors. The ability to perform all edit operations (entry and navigation) using the keyboard instead of the mouse allows use of the editor even if the user cannot operate the mouse (physically, or due to poor sight). The way that expressions are entered by typing them rather than entering a series of blocks should allow for faster use by blind users than block-based entry, even with keyboard support for blocks (as in GP). The decision to have all program code in a single long sequence of named methods allows easier navigation and organisation for a blind user than multiple scattered unnamed fragments as in many block-based languages.

14.6. Frame-Based Editing of Other Languages

In this paper, we have presented a frame-based editor for a language very similar to Java (for compatibility, both technical and pedagogical, with our Greenfoot system). However, the principles of frame-based editing are independent of this language. It is possible to edit any typical programming language (or structured tree format, such as XML or HTML) with a frame-based editor.

We note a few features that may make a language more or less amenable to frames-based editing, starting with issues we encountered while aiming for Java compatibility:

- Java's overloading of less-than/greater-than symbols as angle brackets in generic types is not ideal, especially when combined with the casting syntax. In a type slot, this does not pose a problem because here these characters always denote a generic type. But parsing expressions proves more problematic: Because type casts are not identified by a keyword, types can appear in expressions. Thus we must allow having mismatched angle brackets because we must treat these characters as less-than/greater-than symbols, the more flexible form.
- Java generally makes little use of whitespace to separate lexical tokens, which allows us to not require the user to enter spaces in expressions. We can let the user enter 1+2 or x/3 and let the editor space the expression. Because most operators have a different alphabet to operands, we can easily distinguish one from the other. For keyword operators,

such as “new” and “instanceof”, spaces are still required between the operator and its arguments. To alleviate this problem, the instanceof keyword has been replaced with a symbolic operator in Stride (“<:”). The “new” operator is treated as a special case. In general, keyword operators are more awkward than symbols for frame-based editors.

- Java’s lambda expressions remain an open issue for us. A lambda expression can appear in any expression and contain arbitrary segments of code. This means that an arbitrarily complex frame structure could appear in an expression slot. This may become complicated to display and manipulate. (For the same reason we also do not support anonymous inner classes). Devising a manipulation system for this is not impossible, but complicates the interface. A stricter model of programming, where statements contain expressions and no further nesting is possible (avoiding arbitrary levels of nesting of statements inside expressions) is easier to implement in frame-based editors.
- Frame-based editing works well where there is an expression/statement divide, or more generally, a setting where there is a high-level structure with program units, and a lower-level expression editing. Although in theory everything in Haskell is an expression and there are no statements, it would actually be a good fit as you could have frame slots for monadic ‘do’ blocks, ‘case’ statements, ‘where’ clauses, individual functions and so on. In Lisp-like languages where every construct is an S-expression, frame-based editing would be almost non-applicable because there would be no apparent way to distinguish what is a frame vs what is a structured slot.
- Frame-based editing can also be used for non-programming languages. On similar lines to the previous point, it would work well for HTML and XML where each element could be a frame, attributes could be slots, and text content and sub-elements could go in frame slots. Because each element has a name (and potentially type information in the DTD) you could have context-sensitive input assistance. In contrast, JSON is closer to Lisp S-expressions and would work much less well. Languages with markup which have tags rather than begin/end hierarchal structure (e.g. $\LaTeX$’s section commands) would need to be converted to nested frames to work well.

15. Conclusion

In this project, we set out to build a system that combines advantages of block-based and text-based systems.

We have designed and implemented an editor for a new, Java-like language called Stride that incorporates this goal. The Stride editor demonstrates that such a system can be designed and that it presents a number of advantages over each of the competitor systems.

Our initial user studies, on middle-school students and experienced programmers, suggest that Stride has several advantages over text-based programming. Users reported that they found it more readable, and easier to insert new code and delete existing code. When given the same task, students using Stride progressed further than students using Java within the same environment. We continue to iterate and improve the software in order to address possible weaknesses, such as the navigation between slots and editing of structured expressions.

The Greenfoot system including the Stride editor is available for free download at http://www.greenfoot.org/. The same Stride editor has also recently been incorporated into BlueJ, which is also available for free download (http://www.bluej.org/).

15.1. Future Work

There are several avenues for future work. The first is to perform further user studies and evaluation with the editor. Computer science education unfortunately produces many new tools, but little evidence of their effectiveness. We have begun to address this already with the evaluations reported in this paper, but we would like further evaluations, carried out either by ourselves or by others.

A second avenue for future work is to create frame-based editors for other languages, and/or integrate frame-based editors into other tools (such as professional IDEs). (We have frequently been asked if there is a frame-based editor for Python, or whether the editor is available as an Eclipse plugin, etc.) We do not have the resources available to perform this extra technical work, but we would welcome such work by others.

A third avenue for future work is to build on the new editor model. We have so far implemented editor interactions in frame-based editing, but editors in modern IDEs support several other modes of use, such as debugging and version control. We believe there is potential to integrate these other modes of use into a frame-based editor in a better way than they are supported in current plain-text-focused editors.

Appendix

Appendix: Java vs Stride

In one sense it is difficult to precisely define the difference between Java and Stride. The characters which form necessary keywords and syntax in Java are often mere decoration in Stride, which we have kept looking similar to Java. For example, whether Stride shows round brackets surrounding the condition in a while loop is a cosmetic decision, whereas those characters are essential for Java code to be parsed correctly. On similar lines, Stride has no curly brackets around its scopes, whereas Java requires them, Stride displays a var keyword which Java does not use.

Broadly, however, the languages are near-identical. Stride currently lacks several Java constructs: inner classes, anonymous inner classes, synchronised blocks, several advanced modifiers (such as volatile), enums, and interfaces, although we are likely to add the latter two. Stride uses `<:` instead of `instanceof` due to the difficulty of distinguishing variable names from letter-based symbol names, and also disallows floating-point literals of the form `.5`, requiring `0.5` instead. Semantically, Stride is an unaltered subset of Java: what Stride includes from Java is identical to Java.

Acknowledgments

Some of the initial design work of the frame editor, and the CogTool work presented in section 10 were carried out by Fraser McKay as part of his PhD work. The study described in section 11 was primarily carried out by Thomas Price, in conjunction with Dragan Lipovac and Tiffany Barnes. We are grateful for their contributions.

References

[1] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 75–80.

[2] 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*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 522–527.

[3] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *2014 Frontiers In Education Conference*, October 2014.

[4] S. K. Kummerfeld and J. Kay, "The neglected battle fields of syntax errors," in *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ser. ACE '03. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 105–111.

[5] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 252–261.

[6] M. C. Jadud, "An exploration of novice compilation behaviour in bluej," Ph.D. dissertation, University of Kent, 2006.

[7] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 208–212.

[8] G. M. Weinberg, *The Psychology of Computer Programming (2nd edition)*. Dorset House Publishing, 1998.

[9] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 499–504.

[10] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: ACM, 2014, pp. 273–278.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[12] K. Wang, C. McCaffrey, D. Wendel, and E. Klopfer, "3D game design with programming blocks in StarLogo TNG," in *Proceedings of the 7th International Conference on Learning Sciences*, ser. ICLS '06. International Society of the Learning Sciences, 2006, pp. 1008–1009.

[13] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D tool for introductory programming concepts," *Journal of Computing Sciences in Colleges*, vol. 15, no. 5, pp. 107–116, Apr. 2000.

[14] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media, Inc., 2014.

[15] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: Urban youth learning programming with Scratch," *SIGCSE Bull.*, vol. 40, no. 1, pp. 367–371, Mar. 2008.

[16] M. Vasek, "Representing Expressive Types in Blocks Programming Languages," Undergraduate thesis, Wellesley College, 2012.

[17] 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 Transactions on Computer-Human Interaction*, vol. 16, no. 3, pp. 13:1–13:40, Sep. 2009.

[18] T. R. G. Green, "Cognitive dimensions of notations," *People and Computers V*, pp. 443–460, 1989.

[19] R. C. Thomas, A. Karahasanovic, and G. E. Kennedy, "An investigation into keystroke latency metrics as an indicator of programming performance," in *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ser. ACE '05. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2005, pp. 127–134.

[20] J. Leinonen, K. Longi, A. Klami, and A. Vihavainen, "Automatic inference of programming performance and experience from typing patterns," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 132–137.

[21] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[22] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 91–99.

[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*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 101–110.

[24] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '15. New York, NY, USA: ACM, 2015, pp. 29–38.

[25] M. Kölling, "The Greenfoot programming environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 14:1–14:21, Nov. 2010.

[26] J. S. Mansfield, G. E. Legge, and M. C. Bane, "Psychophysics of reading. XV: Font effects in normal and low vision." *Investigative Ophthalmology & Visual Science*, vol. 37, no. 8, pp. 1492–1501, 1996.

[27] M. Bernard, B. Lida, S. Riley, T. Hackler, and K. Janzen, "A comparison of popular online fonts: Which size and type is best," *Usability News*, vol. 4, no. 1, 2002.

[28] 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.

[29] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 223–228.

[30] Y. S. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 223–233.

[31] A. J. Ko and B. A. Myers, "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*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 387–396.

[32] P. H. F. M. Verhoeven, "The design of the mathspad editor," Ph.D. dissertation, Technische Universiteit Eindhoven, 2000.

[33] R. Baecker and A. Marcus, "Design principles for the enhanced presentation of computer program source text," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '86. New York, NY, USA: ACM, 1986, pp. 51–58.

[34] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, Jun. 2015.

[35] "Lego mindstorms," Retrieved on 17 Dec 2015 from http://www.lego.com/en-us/mindstorms, 2015.

[36] 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. Swinton, UK, UK: British Computer Society, 2013, pp. 35:1–35:6.

[37] 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.

[38] B. E. John, "Reducing the variability between novice modelers: Results of a tool for human performance modeling produced through human-centered design," in *Proceedings of the 19th Annual Conference on Behavior Representation in Modeling and Simulation (BRIMS)*, 2010, pp. 22–25.

[39] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind." *Psychological review*, vol. 111, no. 4, p. 1036, 2004.

[40] T. R. G. Green and A. Blackwell, "Design for usability using cognitive dimensions," 1998, tutorial session at British Computer Society conference on Human Computer Interaction HCI98.

[41] T. W. Price, N. C. C. Brown, D. Lipovac, T. Barnes, and M. Kölling, "Evaluation of a frame-based programming editor," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16. New York, NY, USA: ACM, 2016, pp. 33–42.

[42] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563–573, Sep. 1981.

[43] D. B. Garlan and P. L. Miller, "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*, ser. SDE 1. New York, NY, USA: ACM, 1984, pp. 65–72.

[44] A. diSessa, "Twenty reasons why you should use Boxer (instead of Logo)," in *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference, Budapest, Hungary*, 1997, pp. 7–27.

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

[46] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors of Carnegie Mellon University," *Interactive Learning Environments*, vol. 4, no. 2, pp. 140–158, 1994.

[47] S. Minör, "Interacting with structure-oriented editors," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 399–418, Oct. 1992.

[48] J. Welsh and M. Toleman, "Conceptual issues in language-based editor design," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 419–430, Oct. 1992.

[49] B. S. Lerner, "Automated customization of structure editors," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 529–563, Oct. 1992.

[50] A. Repenning and J. Ambach, "Tactile programming: a unified manipulation paradigm supporting program comprehension, composition and sharing," in *Proceedings 1996 IEEE Symposium on Visual Languages*, Sep 1996, pp. 102–109.

[51] K. Osenkov, "Designing, implementing and integrating a structured C# code editor," *Brandenburg University of Technology, Cottbus*, 2007.

[52] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 41–61.

[53] D. Asenov and P. Müller, "Envision: A fast and flexible visual code editor with fluid interactions (overview)," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 9–12.

[54] I. Almusaly and R. Metoyer, "A syntax-directed keyboard extension for writing source code on touchscreen devices," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 195–202.

[55] A. Gomolka and B. Humm, "Structure editors: Old hat or future vision?" in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science, L. Maciaszek and K. Zhang, Eds. Springer Berlin Heidelberg, 2013, vol. 275, pp. 82–97. [Online]. http://dx.doi.org/10.1007/978-3-642-32341-6_6

[56] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in Scratch," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 168–172.

[57] 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*, ser. SIGCSE '12. New York, NY, USA: ACM, 2012, pp. 141–146.

[58] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Proceedings of the 2014 Second IEEE Working Conference on Software Visualization*, ser. VISSOFT '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–10.

[59] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015.

[60] S. Federici, "A minimal, extensible, drag-and-drop implementation of the C programming language," in *Proceedings of the 2011 Conference on Information Technology Education*, ser. SIGITE '11. New York, NY, USA: ACM, 2011, pp. 191–196.

[61] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt, "Touchdevelop: App development on mobile devices," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 39:1–39:2.

[62] A. M. Stefik, C. Hundhausen, and D. Smith, "On the design of an educational infrastructure for the blind and visually impaired in computer science," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 571–576.

[63] A. Wagner, J. Gray, D. Marghitu, and A. Stefik, "Raising the awareness of accessibility needs in block languages (abstract only)," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 497–497.