Visual Design for a Tree-Oriented Projectional Editor

Tom Beckmann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tom.beckmann@student.hpi.de

Stefan Ramson
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.de

Patrick Rein
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
patrick.rein@hpi.de

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.de

ABSTRACT

Projectional editors show promise for a variety of use cases, for example in language composition and domain specific projections. To allow efficient interactions within a projectional editor, it is necessary for the editor to clearly communicate the structure of the program to the user, such that it is clear what editing operations are supported for a given element. Making the abstract syntax tree visible within the editor may provide this clarity, however, it generally also results in considerably increased space usage, potentially also impacting usability. We present an early prototype of a tree-oriented projectional editor for Squeak/Smalltalk that tries to minimize space usage while retaining a clear visualization of the tree structure, balancing the two problems. We describe and discuss our design prototype and do a preliminary evaluation through individual account of experience working with the editor on various projects.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

projectional editing, visual programming language, Squeak/Smalltalk

ACM Reference Format:
Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3397537.3397560

1 INTRODUCTION

Projectional editors allow composing different languages and choosing different projections for parts of the abstract syntax tree (AST), to for example present them in a domain-specific way. They have also been found to increase developer efficiency if well implemented [1], and are sometimes considered more approachable as the direct composition of elements does not allow for syntax errors. Ensuring usability within projectional editors, however, is a complex problem that requires robust design [12].

One main decision when creating a projectional editor concerns the visual representation of the AST. In traditional text-based representations, the AST is only implicitly visible through the various syntax elements that denote inclusions or delimiters of language constructs. Possibilities within a projectional editor include representations that closely resembles text, for example seen in mbeddr [10]. We call editors that use this type of representation text-oriented. In contrast, Scratch [9] adopts a representation that uses blocks for the elements of the AST and their composition and mainly uses drag-and-drop for editing. Editors that adopt this type of explicit representation of the AST's structure we call tree-oriented. Further, various examples of graph or data flow programming editors exist, especially in designer-oriented applications such as the Unreal Engine. Here, nodes in a graph describe expressions or statements that are connected through edges. Editors may also mix both textual and visual language for their individual strengths [3].

Certain trade-offs have to be considered when choosing a visual representation. Using neighborhood information and a direction of reading to denote structure will likely result in the most space efficient representation. This is the approach used by regular text, as users know to read code from left to right and manually build up inclusion structures in their head as they parse the syntax elements, such as parentheses. While non-text representations alleviate users from having to parse the structure themselves, they may use considerably more space to present the same information. Rather than benefiting from the implicit structure through neighborhood, connections have to be explicitly expressed, for example via edges in a graph, or through blocks that contain one another. Both of these approaches require extensive space between elements to be able to present an unambiguous structure.

Efficiency in layout directly impacts usability, as has for example been studied for the Code Bubbles IDE [2]. Users may have to scroll or navigate considerably more if the part of the program they are editing or have to refer to does not fit on their screen. Instead, they have to retain information in their head that might have been visible with a more concise representation. However, for projectional editors, it has been found that an explicit understanding of the AST's structure or lack thereof will impact users' ability to navigate and modify programs [1]. An implicit representation of the AST, as used in text-based projectional editors, may lead users to perceive arbitrary limitations on what can be edited and what cannot, such as neighboring binary expressions. To address this issue, text-oriented projectional editors may re-introduce parsing steps into the editor. For example, MPS introduced a concept of Grammar Cells [11], which allow users to edit complex expressions as they would in a text-based editor. The input is then parsed and reinserted into the editor's AST.

To find a middle ground between the aspects of space usage and clarity of the tree structure, we present an early prototype of a projectional editor for Squeak/Smalltalk. Our contribution is a set of guidelines that we found to balance the two aspects well, according to an early evaluation of our own usage of the editor in various projects and developing the editor itself in a self-supporting way.

Our design chooses to display the structure of the AST by means of nested blocks. Colors are used to emphasize the blocks' nesting. To support the user's understanding on how to accomplish certain edits, we remove all syntax elements and constrain text editing to identifiers and literals alone. All other operations happen on the tree directly through a separate command mode.

In section 2, we describe different approaches used in projectional editors to visualize ASTs, as well as our host environment. In section 3 we describe our own approach and discuss its limitations and benefits in section 4. Finally, we present areas of future work in section 5 and conclude the paper in section 6.

2 BACKGROUND AND RELATED WORK

In this section, we present and discuss different approaches to visualize the AST of projectional editors and describe the Squeak/Smalltalk language and environment on which we built our editor.

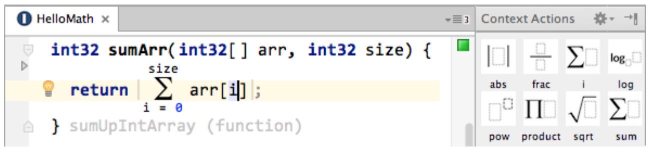
2.1 Visual Design of Projectional Editors

mbeddr [10] is a projectional editor built on the Meta-Programming-System (MPS) that uses a textual representation of the AST. Programs in mbeddr will read very similar, if not identical, to a textual language. Additionally, projections within mbeddr may define visual structures not possible in text editors, such as multi-row matrices that flow within a single line, or tabular state machines that contain other textual code within their cells. Color is used here for highlighting different syntactical elements, as in most text editors.

Lamdu[7] is a projectional editor with a focus on supporting a REPL workflow. Similar to mbeddr, the representation appears very similar to regular text editors, with text generally flowing from left to right and colors used for highlighting syntax elements. Lamdu will use the expression provided by the user in its REPL to evaluate the code the user is currently working on and will insert either value or type annotations under each subexpression.

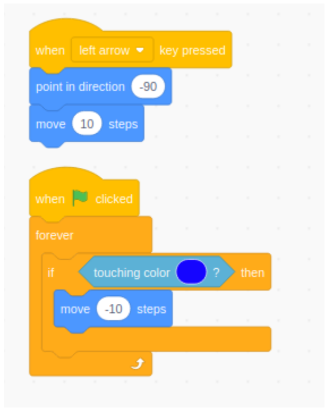
The textual presentation of the code allows reading it in the same way that textual code reads. Editing, however, is typically more complex than working in a text editor, as the structure of the AST has to be adhered to. Incomplete edits, such as inserting an opening parentheses to start restructuring the tree may not be supported by the editor, unless specific care has been taken to support this use case, such as in MPS' GrammarCells [11]. Otherwise, users have to trigger specific commands, in this case for example a tree left rotation. Hiding the AST and only presenting the syntactical, textual representation may, however, make these types of explicit tree transformations more difficult for a user to perform, as boundaries and nesting of nodes are not necessarily apparent [1].

In Scratch [9], elements of the AST are clearly represented in the form of boxes. Text is only used to adjust values and identifiers within the boxes or configure their effect. Most interactions that modify the tree structure happen via drag-and-drop, rather than through placing parentheses or moving text ranges. Colors are used in Scratch to describe different types of domain-specific instructions, such as control flow, events, or motion. The shapes of the blocks resemble jigsaw puzzles where blocks can be attached or placed within another block if they have the correct type. While these interactions feel very intuitive and the structure is very clear, large programs in Scratch take up considerable space. Further, for

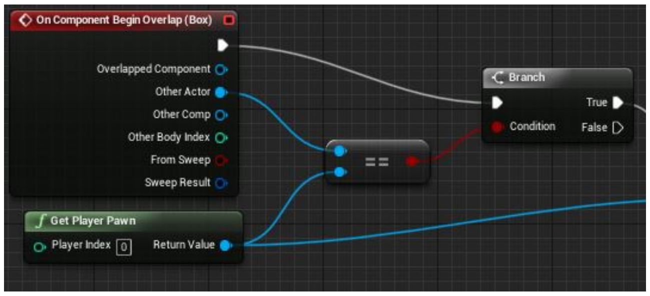


```
int32 sumArr(int32[] arr, int32 size) {
  return  Σ(i = 0 .. size) arr[i];
} sumUpIntArray (function)
```

(a) mbeddr: a projectional editor using a C dialect for embedded applications.



(b) Scratch: an editor specifically aimed for learning to program.



(c) Unreal Blueprints: a graph programming editor for expressing game logic within the Unreal Game Engine.

Figure 1: Three visual programming editors using text-based, block-based, and graph-based interactions.

an editor in a professional setting, keyboard support may be required, as keyboard input could be considered the standard across programming tools, likely due to the increased efficiency obtained from using a keyboard as opposed to a mouse [6].

The Unreal Engine contains a visual programming editor called Blueprints. Statements and expressions are represented through nodes in a graph. Lines connect data outputs to data inputs. Each primitive data type has a distinct color, used to show what data type a line, input, and output carries. Additionally, a white line connects statements that cause side effects to explicitly order them. Colors are also used to show the type of nodes, for example events, side effects, or pure functions. Blueprints defines shortcuts to automatically align a selection of nodes and plugins exist that attempt to auto format entire graphs. For complex expressions, Blueprints may quickly pose challenges to find a suitable layout that still remains readable. Nodes need a minimum distance from one another to allow the eye to follow overlapping lines. Hovering a line will highlight it.

For example in Origami Studio, mathematical expressions, which often have complex graph layouts, can be expressed via text. Using any variable, such as “x” will create an input to the resulting node. We consider this a similar solution to GrammarCells, where an edit that is easy to express in a text-based environment is input as characters and then parsed to integrate with the projectional environment.

2.2 A Projection for Smalltalk

Our editor uses a projection for the Smalltalk language [4], in particular the Squeak/Smalltalk dialect. Smalltalk is a language with comparatively few syntax elements. It defines only six keywords and revolves around the concept of message sends. Creating a projection for Smalltalk is as such easier, when compared to languages where many different types of language constructs need to be disambiguated as the user edits the AST. Further, concepts integral to Squeak/Smalltalk are already only editable via projections, such as classes and method categories. The concept of a source code document does not exist in Squeak/Smalltalk, rather editing happens either on the projections of the organizing elements or on a single method's source code.

Except for method return statements and variable declarations, Smalltalk defines only expressions: assignments, message sends, cascaded message sends, blocks, and literal values such as variables or numbers. An example of Squeak/Smalltalk syntax can be found in figure 2.

3 APPROACH

In this chapter we describe the challenges we identified for a tree-oriented visual editor design and our approach to a solution. The implementation's source code can be found on GitHub[1].

For a projectional editor it is important that addressable units appear clearly or that other means are put in place to ensure that the user can quickly tell how language elements within the editor can be interacted with. What we call a tree-oriented projectional editor relies on explicitly modifying the tree structure of the program, rather than an intermediary textual representations. As such, the first challenge is that each element of the AST needs to be visibly separated and the tree's nesting needs to be apparent. Second, to make editing complex methods feasible, we need to find a way to present as much information as possible in the available space, while still maintaining clarity. The less information about the code's context will fit on the screen, the more users are forced to scroll or navigate. Third, a fast reading flow should be maintained. Important properties of the source code should be immediately apparent to allow users to quickly grasp the code's structure. For example control flow elements could carry stronger emphasis.

[1]https://github.com/tom95/sandblocks

```
exampleWithNumber: x
    " A comment "

    | y |
    (true & false not and: [nil isNil]) ifFalse: [self halt].
    y := self size + super size.
    #($a #a ’a’ 1 1.0) do: [:each | | z |
        Transcript
            show: {each class name. z := 16rFF};
            show: ’ ’].
      x < y
```

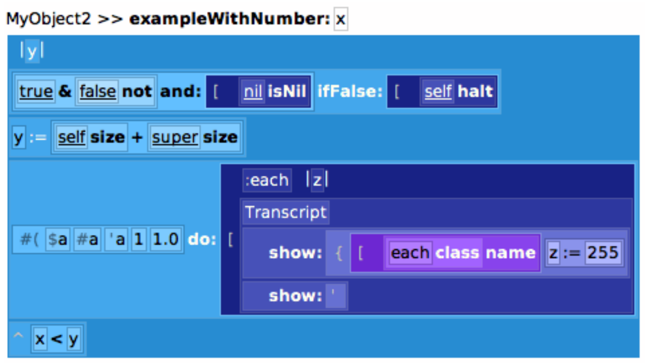
Figure 2: Squeak/Smalltalk's syntax elements in a single method, leaving out pragmas and literal byte arrays.

3.1 Editor Structure

Our editor tries to find a middle ground between fully block-based and text-based appearance, see figure 3a. While we compose the tree via nested blocks, the arrangement of the blocks follows the left-to-right order that English text layout uses, similar to Scratch. For complex expressions, a layout policy ensures that we wrap message sends on multiple lines, such that they typically remain readable without horizontal overflow. This is particularly important as, unlike Scratch where a loop will nest its expressions on the next line, we do not know what type of structure will be nested within a message send and what a suitable alignment might be and can only judge from its size.

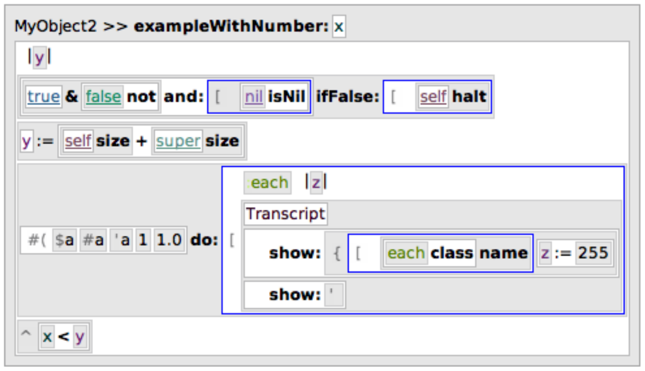
In general, we observed a strong tendency of the discussed projectional editors to use their domain's properties, such as types of data or control flow elements, to make the code's visualization clearer. In Squeak/Smalltalk, this could be considered a downside of having only few language constructs, as each has to serve a variety of purposes and tooling needs to execute the code to get any deeper insights.

Parent-child-relationships between nodes are displayed by nesting blocks inside one another with a small gap to the edges, to ensure that a block's nesting level is discernible. This means a row's height is increasing as the nesting gets deeper, an effect that text-based editors do not have to deal with. While we found that increasing the gaps further helped in terms of clarity, the gaps are also the most considerable reason why our layouts take more space than text layouts. See figure 3c for a configuration that uses no vertical inset inside its nodes. In particular the first row's nesting



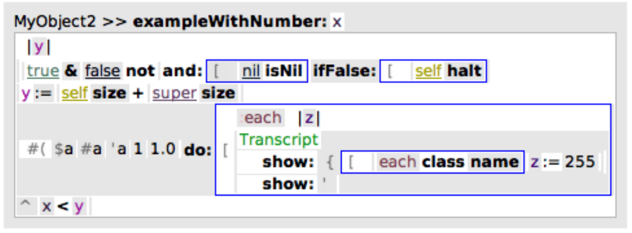
```
MyObject2 >> exampleWithNumber: x
|y|
true & false not and: [ nil isNil ifFalse: [ self halt
y := self size + super size
#( $a #a 'a 1 1.0 do: [ :each |z|
    Transcript
    show: { [ each class name z := 255
    show: '
^ x < y
```

(a) The default configuration discussed in this paper. Colors increase in brightness as the nesting gets deeper.



```
MyObject2 >> exampleWithNumber: x
|y|
true & false not and: [ nil isNil ifFalse: [ self halt
y := self size + super size
#( $a #a 'a 1 1.0 do: [ each |z|
    Transcript
    show: { [ each class name z := 255
    show: '
^ x < y
```

(b) Colors are assigned per unique identifier. Nodes have alternating background colors.



```
MyObject2 >> exampleWithNumber: x
|y|
true & false not and: [ nil isNil ifFalse: [ self halt
y := self size + super size
#( $a #a 'a 1 1.0 do: [ each |z|
    Transcript
    show: { [ each class name z := 255
    show: '
^ x < y
```

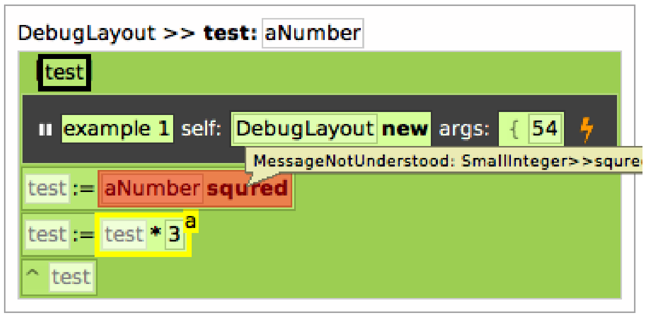
(c) Alternating colors with vertical insets for each node removed.

Figure 3: Three configuration options for our editor. Figure 3a is the default and primarily described configuration.

gets hard to recognize, but space usage is now roughly equivalent to that of the text layout in figure 2.

3.2 Color

We use colors to support the hierarchy. A method base color is chosen per class, such that methods of the same class will have the same base color. As the nesting increases, we decrease saturation and increase brightness of the base color. This will quickly clamp at white if the nesting gets too deep, which in general is problematic, but as a side effect may encourage users to write less complex code. The foreground color adapts to different backgrounds to ensure legibility.



```
DebugLayout >> test: aNumber
test
example 1 self: DebugLayout new args: { 54
MessageNotUnderstood: SmallInteger>>squre
test := aNumber squred
test := test * 3 a
^ test
```

Figure 4: All types of markers currently implemented in our editor: a black outline showing the user's cursor; a red marker for an error that occurred during example execution; a yellow outline for the jump mark "a"; white highlights for usages of the currently selected identifier.

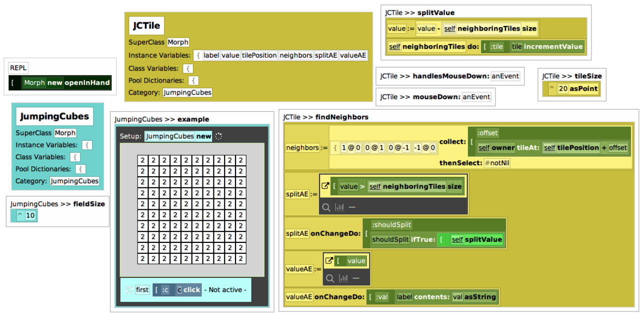
Each Smalltalk block starts a new base color to help the user quickly discern control flow. We calculate this color by rotating the base color's hue value around the hue wheel to arrive at a distinct, but related color. We also adjust the color's brightness up or down, depending on whether the color is already rather bright or dark. This results in a loosely alternating color scheme as blocks nest deeper, without creating jarring differences.

We explored various alternatives, one can be seen in figure 3b. Renditions that do not use color to support the structure allow using color for other purposes, such as syntax or identifiers, as seen in figure 3b. This evokes a stronger resemblance to textual code editors, but the distinction of nesting and blocks gets harder. Alternating colors solve the issue of colors getting too bright, however, the intuition of bright colors meaning deep nesting gets lost.

3.3 Language Specific Support

Textual Smalltalk syntax elements appear in different places, for example the caret in front of a return expression. These elements are, however, not editable and act as icons or signifiers. They are kept in a lighter text color to show their symbolic nature. Text that is full white or black is editable and is exclusively used for named identifiers or literal values, such as numbers. All structuring syntax elements have been removed from the AST and replaced with explicit node nesting. As such, parentheses are no longer necessary, as users see the evaluation order through the AST structure instead. This was an important step during interaction design, as we found that including textual syntax elements led to false assumptions on the editor's abilities to deal with incomplete syntax. For example, a user may assume that deleting the statement separator would be the fastest way to join two statements during a refactor, while the editor will generally not be able to join two expressions without first knowing the message call in which the two expressions should reappear. As such, we would need to ask the user to explicitly disambiguate their intent, in this case by requiring input of a message selector or a similar structure that can contain two expressions, or support incomplete tree transformations.

The editor augments the AST with different markers and borders as seen in figure 4. When users move to select an identifier, we



```
REPL
[ Morph new openInHand

JCTile
SuperClass Morph
Instance Variables: { label value tilePosition neighbors splitAE valueAE
Class Variables: {
Pool Dictionaries: {
Category: JumpingCubes

JCTile >> splitValue
value := value - self neighboringTiles size
self neighboringTiles do: [ :tile tile incrementValue

JCTile >> handlesMouseDown: anEvent

JCTile >> mouseDown: anEvent

JCTile >> tileSize
^ 20 asPoint

JumpingCubes
SuperClass Morph
Instance Variables: {
Class Variables: {
Pool Dictionaries: {
Category: JumpingCubes

JumpingCubes >> fieldSize
^ 10

JumpingCubes >> example
Setup: JumpingCubes new
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
first [ :c c click - Not active -

JCTile >> findNeighbors
neighbors := { 1@0 0@1 0@-1 -1@0 collect: [ :offset
                                              self owner tileAt: self tilePosition + offset
                                  thenSelect: #notNil
splitAE := [ value > self neighboringTiles size
           ---
splitAE onChangeDo: [ :shouldSplit
                      shouldSplit ifTrue: [ self splitValue
valueAE := [ value
           ---
valueAE onChangeDo: [ :val label contents: val asString
```

Figure 5: Classes, methods, an embedded example, and a REPL open on a workspace. Artifacts push each other to get a non-overlapping layout and can be collapsed to only show a header.

highlight all occurrences of identifiers that resolve to the same declaration currently visible in the editor. If the user provides an executable definition, such as an example or a test case, we run this code and display errors by adding red highlights. The user's current selection is displayed as a strong black or white border, depending on the surrounding colors. Users can mark jump locations similar to Vim's jump registers. Marking a block in this way will place a border around it and display the character this jump corresponds to.

For navigating multiple methods, or other artifacts like class definitions, we adapted a layout similar to Code Bubbles [2], see figure 5. Artifacts do not overlap, they can be toggled to only a declaration or header, and a set of currently opened artifacts can be persisted into a workspace.

We currently have three categories of projections that substitute Smalltalk message sends with specific user interface elements. Tooling projections, such as watches, breakpoints, or examples are kept in a dark gray. Any nested expressions will be treated either as defined within a new Smalltalk block or as part of the current one. Substituted expressions, for example domain specific queries or regular expressions, will appear in the regular color scheme. They may contain arbitrary user interface elements, such as color pickers or checkboxes. Lastly, comments appear in white, similar to commented-out code, which uses a white base color, such that all child nodes also appear in white. As such, disabled code is distinct from used code, but remains editable and selectable in the same way as enabled code.

3.4 Explored Alternatives

Before settling on the presented visual design, we explored alternative options.

A projection in a graph as seen in figure 6 did not lend itself very well to the Smalltalk language, in general. The main issues we encountered were a strong focus on imperative statements and an inversion of the reading flow. Similar to Unreal's Blueprints, we also had to introduce an explicit ordering mechanism, here through a red line that flows from top to bottom. In contrast to the Blueprints, we only connect top level statements, where an order cannot otherwise be derived. For all other expressions we

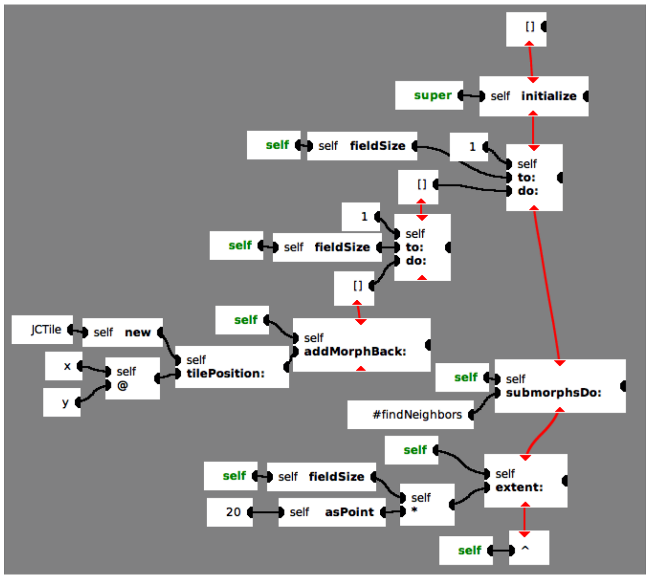


Figure 6: A projection of a Smalltalk method into a graph. A red line denotes explicit control flow. Further red lines are started by nodes representing Smalltalk blocks.

rely on the Smalltalk calling conventions of evaluating from inner expressions to outer expressions, meaning expressions that appear more to the top and more to the left than its siblings will evaluate first.

We decided to place the receiver and arguments of a message send on the left hand side of the node to be consistent with other graph tools, where inputs appear left and outputs appear right. As a consequence, a conditional statement will have its branches extend to the left of the main flow, resulting in left leaning trees, unlike the structure of most textual languages, where text aligns to the left and extends to the right. If breaking the convention of placing inputs to the left would lead to clearer programs is unclear and would need to be investigated.

In general, we found that with this projection typical Smalltalk methods will spread very far apart and present a very unfamiliar and impractical structure that requires the eyes to jump around frequently. Since Smalltalk also has a functional programming subset, we believe that this projection may still be well suited for Smalltalk code that embraces data flow more strongly than imperative concepts. It could then serve as an alternative projection, similar to for example Luna [8], where text and graphs present the same program side-by-side.

3.5 Interactions

The editor uses mode-based interactions, reusing the letter keys in the same way Vim does while in the command mode. Some commands will put the user into input mode, where letter keys translate to actual characters again. Hitting escape will return to command mode.

3.5.1 Context-Aware Input. Users can enter input mode on any block that contains editable text. The editor will then forward all keyboard events to that element, letting it decide how to handle the events. For example, a block for a character literal replaces its one character by any new input, while a string literal will insert any typed character into its contents.

Entering the character space or a character that is a valid binary operator in Smalltalk while editing a variable or number will wrap this block in a message send and let the user continue typing the message send. We can perform this action without requiring disambiguation, since these characters are otherwise not valid in the context of the current block. This allows users to type the number 2, and, while still in input mode, press the plus sign, which then wraps the number in a binary message send and keeps the new message's selector selected. If the user then presses a character that is invalid for a binary message selector, for example "3", it is forwarded to the hole after the operator, resulting in the expression "2 + 3", which was typed with the same number of keystrokes as in a text-based editor. For a language with more language constructs than Smalltalk, a declarative concept like MPS' grammar cells may be desirable [11].

3.5.2 Reusable Command Set. We define a limited number of commands with the hope that they will generalize to all required editing operations. In practice, users appear to only be issuing few different high-level commands frequently while editing source code [5]. For less common operations, it should be possible to use a combination of lower-level commands to achieve the intended transformation. Alternatively, some blocks define commands for less frequently used operations via a command menu, for example turning the usage of a variable node into a message send to "self", a typical refactoring in Smalltalk. This operation would otherwise require copying the text from the variable, replacing the variable with a new message send, and pasting the text.

All commands can generally be put into one of three categories. One set of commands modifies tree structures in general. This includes swapping siblings, replacing a child with a parent, or performing a tree rotations. Further, common commands for language semantics exist, such as "extract variable", or "send message to expression". The last set of commands modifies the state of the user interface, such as folding methods, moving the selection, or opening other methods.

We currently define 25 unique, essential editing commands and a total of 90 commands across all modes. We tried to make it easier for users to memorize the commands by reusing syntax elements of the Smalltalk language as command keys. For example, "^" will wrap the selection in a return statement. Additionally, a subset of commands has a corresponding uppercase version, which will typically invert the action, so turn an undo into a redo. Where appropriate, we reused Vim's mappings, so "hjkl" will move the selection, "i" and "a" will enter input mode, and "x" and "d" will delete nodes. Lastly, some commands have different effects with similar semantics, depending on the block they are invoked on. For example, the plus sign adds an element to an array, adds another part to a keyword message send, or adds another message send to a Smalltalk cascade. These abstract ideas of concepts, such as adding, that map to the same key, further reduce the number of different commands a user has to memorize.

4 DISCUSSION

We are developing the editor in a self-supporting manner and used it in various other Squeak/Smalltalk projects. In our own experience, the layout is inconvenient for reading large methods, but the refactoring tools of the editor made reducing method complexity a lot easier than the built-in Squeak refactoring tooling. Having auto-layout while writing new methods or editing complex expressions provided a great editing experience, even though the layout did not always choose the best possible configuration. We iteratively improved the layout as we found edge cases that seemed not satisfactory. For us, the goal of communicating the tree structure sufficiently clear to allow finding the right editing commands, was achieved. After some initial training, we were able to use the editor efficiently and did not encounter effects of commands that surprised us.

In practice, we found that colors getting too bright was rarely an issue. Typically, it did indeed indicate an expression that could be simplified or was already simple enough to easily be understood even without the aid of the color scheme, such as a mathematical expression involving mostly binary operators. Since we compute unique colors per class, we have run into cases where a method's base color was not very suitable for our approach of increasing the brightness. In general, however, we found the colorscheme with increasing brightness values to work best, in particular as it conveyed both an intuition of nesting depth and control flow elements at a glance. We explored restarting with the base color when the changes in color become indistinguishable, but we found that this led to confusion with our visualization of Smalltalk blocks: a bright color starting would no longer be an indicator of a new block, but could instead be an expression with deep nesting.

Table 1: The mean, maximum, and summed space usage of 8162 methods in the Morphic package (in a Squeak/Smalltalk 5.3 image), using our custom layout and the methods' text layout in square pixels.

	Mean	Max	Sum
Text Layout	64,777	979,200	523,728,000
Our Layout	101,915	4,144,608	831,837,993

As apparent from figure 3c, a layout that takes just as much space as a text layout does not visualize the AST structure very well. Working with this condensed layout caused uncertainty about the exact nesting of expressions, as parentheses were omitted. To get a sense of the impact of the extra spacing, we calculated the mean, max, and sum of all methods' area in the Morphic package in square pixels with both our layout and a pretty-printed layout text with comments stripped in both layouts, see table 1. Mean space usage turned out to be around 1.6 times higher in our layout compared to text layout. Maximum space usage, however, was around 4.2 times higher.

In limited tests, code appeared readable even to people not familiar with the editor, but familiar with Smalltalk. Removal of syntax elements did not appear to be an issue, but our current usage of signifiers caused confusion at first, as we only display the opening elements of nesting groups, for example of Smalltalk blocks. The color choices were generally perceived as pleasant and helpful, in particular when compared to the alternating color scheme in figure 3b.

5 FUTURE WORK

The most important next step is a formal evaluation within a study. For this, both the stability and explorability of the editor needs improvement. Its current design is not well suited for novice users, as it relies almost exclusively on shortcuts that need to be memorized. An advantage, however, is that it seamlessly integrates into the existing Smalltalk environment. As such, testing if a group of users will use the editor throughout their workday or rather keep switching back to their known tooling may provide first insights.

An area of improvement for layout could be the definition of a common baseline for long message sends that split vertically. Currently, the message part, that is the "ifTrue:" of an "ifTrue:ifFalse:" message, will center vertically with respects to its argument, rather than appear aligned to the first statement of the argument. In particular for argument blocks with a large height, this will distribute the message parts over a large distance. Aligning to the text baseline of the first statement may look more intuitive and keep message parts closer together.

Further, a more well distributed, well balanced computation of base colors could be investigated. At the moment, some computed base colors may not work well in that they turn white after just two adjustments of their brightness. This could be improved by constraining colors to a certain part of the Hue-Saturation-Value cone. The chosen base colors also appear to not currently exhaust the full range of possible colors, our current method of using the class object's identity hash as a base for computation may thus not lead to a well distributed color spectrum.

More approaches of condensing the representation's space usage could be considered. We investigated reverting parts of the program further away from the cursor back to their textual representation and only expanding again when the user navigated to that part of the program. The transition, however, appeared jarring and requires the user to work with two different representations at the same time. Experimenting with an animated transition between block and text display could lessen the impact of this issue. Having well developed interactions for zooming within the editor may also help for navigation, even though this may only be considered a workaround.

6 CONCLUSION

We presented our approach to the visual design of a projectional editor that clearly displays the AST's structure. Through this, we hope to allow users to efficiently and confidently navigate and edit the program on an AST-node level, rather than falling back to parsing text expressions back into the AST.

Achieving this clear representation comes at the expense of screen space: our layout will on average take 1.6 times as much space as an equivalent textual representations. We further support the representation by using colors to provide an intuition of nesting depth and control flow elements.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of HPI's Research School[2] and the Hasso Plattner Design Thinking Research Program[3].

REFERENCES

[1] Thorsten Berger, Markus Völter, Hans Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of projectional editing: a controlled experiment. 763–774. https://doi.org/10.1145/2950290.2950315

[2] Andrew Bragdon, Robert Zeleznik, Steven Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph Jr. 2010. Code bubbles: A working set-based interface for code understanding and maintenance, Vol. 4. 2503–2512. https://doi.org/10.1145/1753326.1753706

[3] M. Erwig and B. Meyer. 1995. Heterogeneous visual languages-integrating visual and textual programming. In *Proceedings of Symposium on Visual Languages*. 318–325. https://doi.org/10.1109/VL.1995.520825

[4] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Amy Ko, Htet Aung, and Brad Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. 1557–1560. https://doi.org/10.1145/1056808.1056965

[6] David Lane, H. Napier, S. Peres, and Aniko Sandor. 2005. Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts. *Int. J. Hum. Comput. Interaction* 18 (05 2005), 133–144. https://doi.org/10.1207/s15327590ijhc1802_1

[7] Eyal Lotem and Yair Chuchem. 2016. *Lamdu*. https://web.archive.org/web/20191002233046/http://www.lamdu.org/

[8] Piotr Moczurad and Maciej Malawski. 2018. Visual-Textual Framework for Serverless Computation: A Luna Language Approach. 169–174. https://doi.org/10.1109/UCC-Companion.2018.00052

[9] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and et al. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[10] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

[11] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

[12] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

[2]https://hpi.de/en/research/research-school.html
[3]https://hpi.de/en/dtrp/