UNIVERSITÄT
BERN

The Moldable Editor

Bachelor Thesis

Aliaksei Syrel
from
Minsk, Belarus

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

1. February 2018

Prof. Dr. Oscar Nierstrasz
Dr. Andrei Chiş, Dr. Tudor Girba

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

It has come to our attention that often ideas are generated in other contexts beyond Java. In this work we show how they can be automatically generated in a statically typed language.

Contents

1 Introduction **1**

2 Related Work **2**

3 The Moldable Editor **3**
3.1 Overview 3
3.2 Text Model 5
3.3 Text data structures 6
3.3.1 Pharo 6
3.3.2 Atom 7
3.3.3 Emacs 7
3.3.4 Rope 8
3.4 Text style 9
3.5 Segments and Rendering 12

4 The Validation **15**
4.1 Overview 15
4.2 Transcript 17
4.3 Connector 20
4.4 Documenter 22

5 Conclusion and Future Work **24**

6 Anleitung zu wissenschaftlichen Arbeiten **25**

1 Introduction

We already know that when it comes to a general user interface and widgets in particular, it is possible to have a single composition tree of visual elements. Because of that, developers are able to implement a wide variety of flexible graphical components. Nevertheless, nowadays most text editors happen to be a leaf in that composition hierarchy, thus playing a role of an end point. Those text editors do not allow developers to easily integrate arbitrary visual components within a text therefore forcing programmers to treat a text, and visual elements differently.

The goal of this work is to show how a text editor can be represented in the same composition tree as the rest of the widgets, so that every tiny graphical bit would be an object - a visual element, hence removing a conceptual gap between text and widgets within the editor. As Alan Kay said, ”objects all the way down”, while in our case, graphical components all the way down.

In the first part, we discuss a few applications of such an editor in order to show how a single composition tree makes that editor extremely flexible, and what it could mean to have it in a live programming environment. In the second part, we explain in more detail how the editor is implemented and describe a rope data structure behind a text model. Additionally, we introduce a novel way of storing text attributes and underlying characters in the same data structure, and we show how it simplifies the way existing attributes are expanded on a new inserted text. In a conclusion we present a few directions in which the editor could evolve. We also discuss possible use-cases and more applications of the editor in a context of a live programming environment such as debugger, inspector or live code snippets.

2
Related Work

An ability to embed pictures or other graphical components is not new and can be found in various text editors. One interesting example is the Jupyter Notebook [1] that combines live code, visualisations and explanatory text to create documents. Highly relevant for this work are approaches combining code and graphical views, common in the area of projectional editors. For example, MPS [2], makes one step forward by displaying domain entities using graphical views (e.g., editing a matrix using a table view, editing a state machine using a graph view); views are selected based structural aspects of the code. Envision[3], proposed a highly scalable visual editor.

(a) (b) (c)

Figure 2.1: Examples of the editors combining code and graphical views: *a)* Jupyter Notebook; *b)* Jetbrains MPS; *c)* Envision.

[1] http://jupyter.org/
[2] https://www.jetbrains.com/mps/
[3] http://dimitar-asenov.github.io/Envision/

3
The Moldable Editor

3.1 Overview

The Moldable Editor is a flexible and scalable editor designed as a single composition tree of visual elements, which makes it so different from other visual editors, unique of its kind.

One of the requirements for the editor was an ability to embed visual components inside of text such that they are not deletable nor selectable. We call those elements *adornments*. In order to unify how text and adornments are being treated, the editor also represents pieces of text as visual elements, thus completely removing a gap between embedded graphical components and text which is what allows us to have a single composition tree.

The basis and foundation of the editor is a text model. One of the interesting aspects of that text model is the fact that it is data structure independent. It also allows us to implement different types of text models, for example `SubText`, that can scope existing text model within a character interval, or `SpanText`, that represents a uniform piece of text where every character has the same attributes. From the text editor's perspective there is no difference between those types of text, the interaction happens through a clear `Text` api. More about text model can be found in the section 3.2.

When talking about text model we should not forget about text style, which is defined by a set of text attributes. Those attributes can be applied on the text manually with the help of corresponding text model api or created automatically by text stylers. Text stylers play an important role in syntax highlighting used by code editor. Even more important

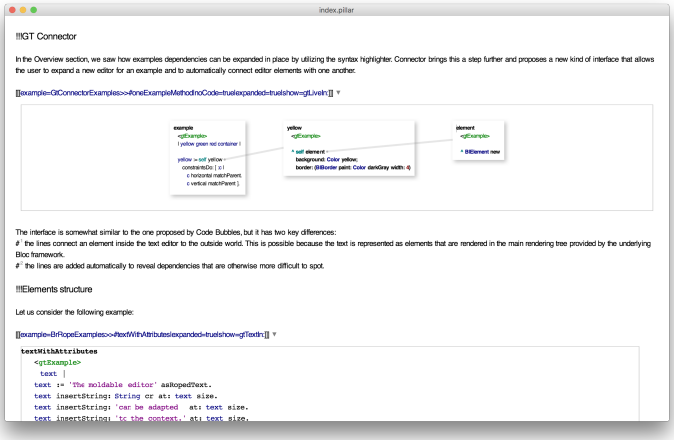


Figure 3.1: The Moldable Editor

role they play in the moldable editor as they are used to add *adornment* attributes and hence provide a nice way to plug-in custom behaviour in the editor. Text stylers take context into account which is essential for creation of context-aware development tools. In order for the editor to be fast and responsive, long and time consuming operations such text parsing and styling should happen in a parallel background thread. While being easy to explain it is in fact a non-trivial task, since users are able to perform text modification operations while styler applies attributes on that text being modified. In the 3.4 section we will talk more about problems and difficulties related to text styling and introduce a solution that is currently implemented in and used by the moldable text editor.

In order for the moldable editor to be scalable, it should split text into logical segments and render only those ones that are currently visible. A segment can be a page, a paragraph or a line. In the current implementation, the text editor creates line segments with the help of a line segment builder. The process of splitting text into segments is trivial, however keeping segments in sync with the text model after modification such as insertions and deletions is a difficult part and very error-prone. Once segments are built they should be rendered as visual elements and displayed within the editor. The way it happens is similar to how modern scrolling lists work, for example *FastTable* in Pharo, *RecyclerView*[1] in Android or *UITableView*[2] in iOS. Segments are hold by a data source object which knows how segments should be represented. It is also responsible for binding a segment model to its visual representation. In most cases a logical segment consists of multiple segment pieces, for example a line segment consists of words - text pieces separated by a white space. White space itself is also a piece within a line segment.

[1] `https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html`

[2] `https://developer.apple.com/documentation/uikit/uitableview`

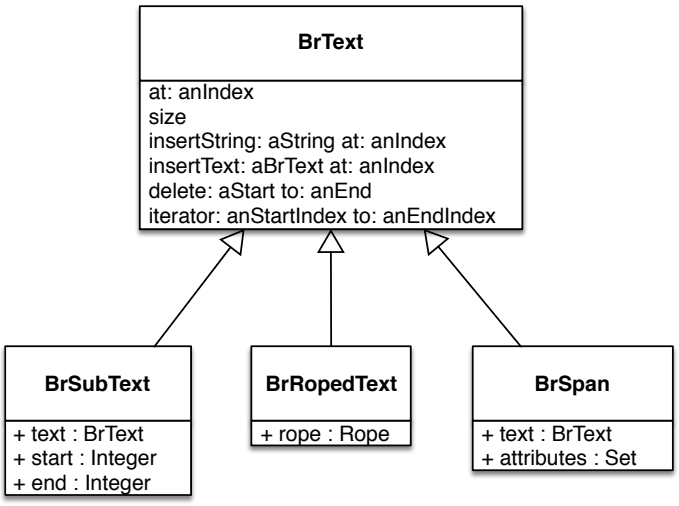


Figure 3.2: The UML class diagram of Text Model

A structure of the segment and its visual part will be explained in more details in the 3.5 section.

The last but not least is a `TextEditor` itself that provides high level text modification api and allows developers to specify shortcuts.

3.2 Text Model

In order to display and edit text, an editor requires a text model that provides text modification and enumeration api. In a context of text editor under *Text* we understand an object that consists of a collection of characters with a set of attributes applied on those characters and a number of api methods to support text modifications such as `insert:` or `delete:`. Additionally, text should play a role of sequenceable and indexable collection, allowing uses to iterate over all characters in a natural ordered way. Being indexable is also an essential property of a text model, since text stylers require text to have characters accessible by index. It is a consequence of a fact that code parsers create an *AST* which consists of nodes bound to original text with the help of integer intervals that are later used by a text styler to apply attributes on a piece of text defined by that integer intervals in a form of a *[from, to]* tuple.

3.3 Text data structures

One of the main requirements for the moldable editor is an ability to manipulate large pieces of text that consist of millions of characters and sometimes even more. It means that choosing an appropriate data structure for storing text is crucial. While looking for a data structure that would fit the needs of the moldable editor we found out that there is no the best data structure that is the only choice. It turned out that depending on a context and the way a text editor will be used it may be important to be able to select one or another data structure. Not to mention is the fact that text data structure should be chosen during early development stages, before the editor becomes usable. That is why the text model of the moldable editor is data structure independent and only defines a public api. In order for it to be used by a text editor developers should create concrete implementations of that api with the data structure of choice as a backend. In the following sub-sections we will look at different data structures being used by text editors and compare them.

3.3.1 Pharo

In Pharo there already exist two text models based on different data structures: one is `Text` which is used by both *Morphic* and *Rubric* text editors, and `TxModel` used by *TxText editor*. In the following sections we will compare and evaluate those text models regarding various properties such as ability to store, access and modify large pieces of text or whether they allow developers to embed non-textual object within text.

Rubric

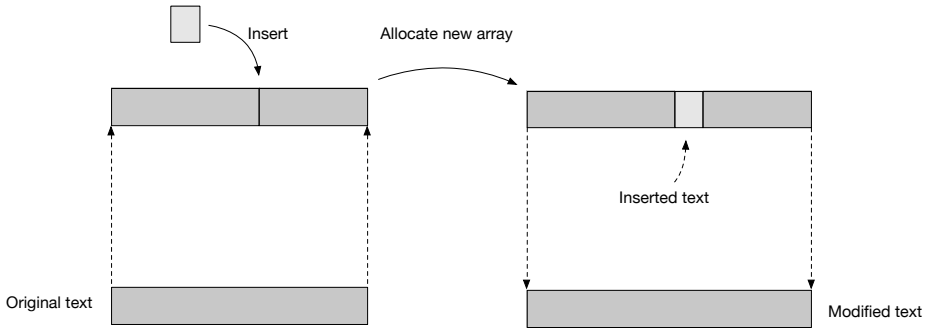


Figure 3.3: The array method

`Text` is a default Pharo text model. It stores a collection of characters and a set of attributes separately. Characters are represented with the help of `ByteString` which is nothing else than an immutable array of characters. It means that every text modification

such as insertion or deletion requires text model to allocate and copy the whole array while replacing a subsequence of characters with a requested one as shown on Figure 3.3 The algorithm of text modifications is in this case linear time and requires massive memory copy operations, which becomes unacceptably slow when text size growths over hundreds of thousands of characters. In fact, array is the worst data structure for text sequences. [1]

TxText

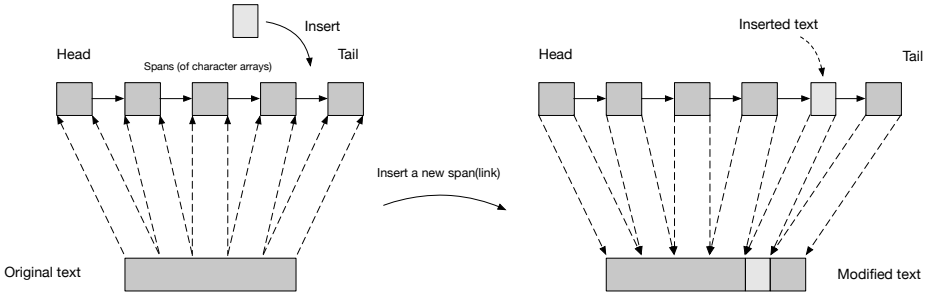


Figure 3.4: The linked list method

`TxModel` is a central class representing a text in the *TxText editor*. Internally is stores character sequences as a double-linked list of spans that consists of an actual text content. A linked list is considered to be an extreme as opposed to an array method of storing text. While insertions and deletions in a linked list are fast and easy, it is only indexable in a linear time which makes styling one of the slowest among all other text data structures. [1]

3.3.2 Atom

Atom text editor uses a memory-efficient data structure similar to a Piece Table.[2] Piece table is a memory efficient data structure based of two buffers, one of which represents original read-only text while the second one stores all modifications done to that text. All essential operations are performed with an adequate performance and some of them can be efficiently improved by using cache. Piece table is considered to be the data structure of choice for a text editor. [1]

3.3.3 Emacs

TODO

3.3.4 Rope

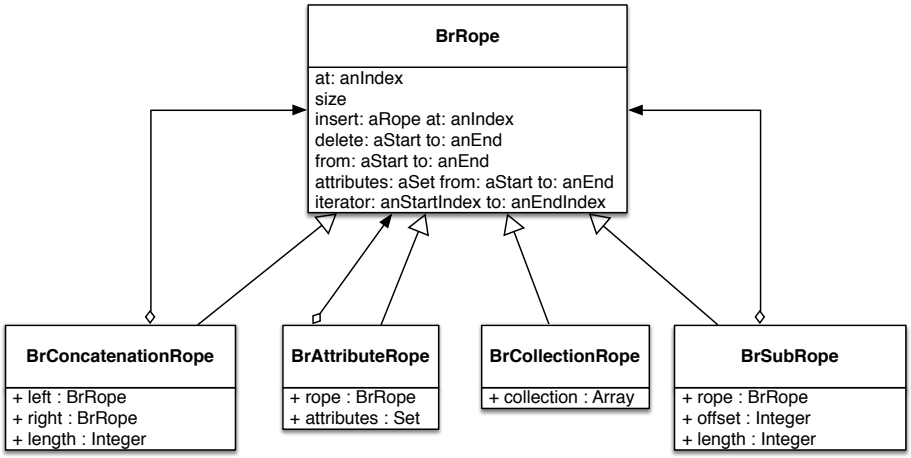


Figure 3.5: UML class diagram of the Rope hierarchy

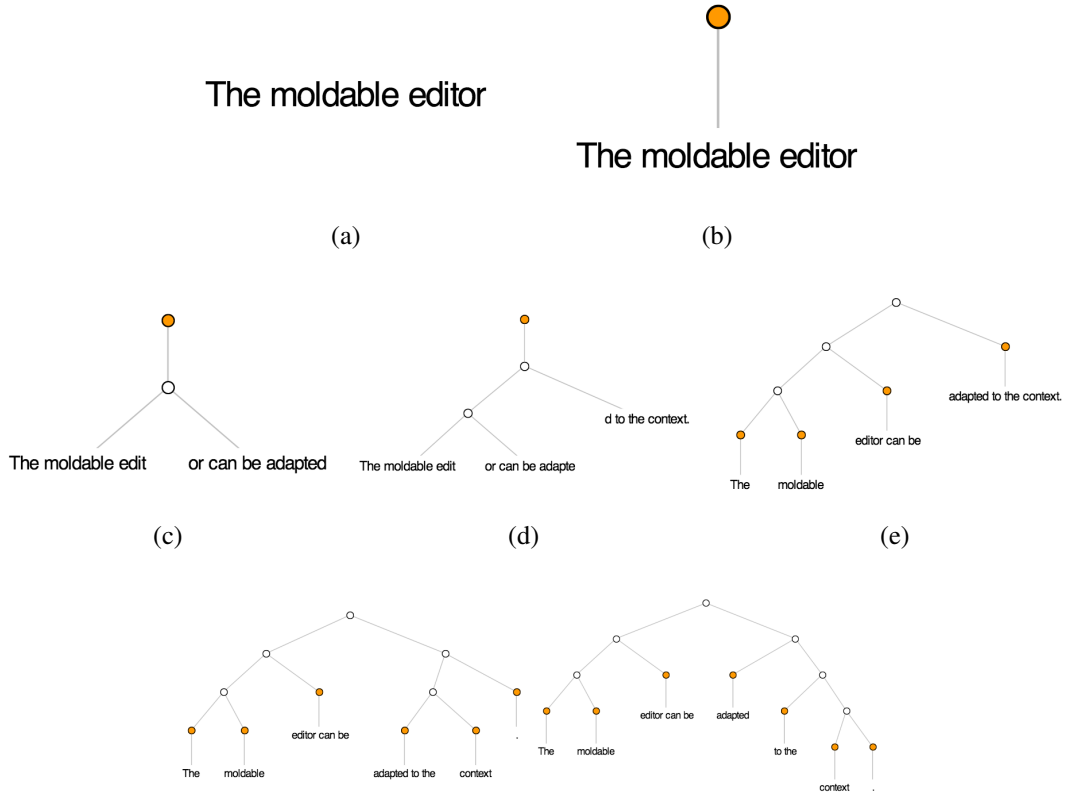


Figure 3.6: Three different ways to display the code of a method: *a)* the standard Pharo Transcript; *b)* replacing the data parameter using two distinct views; *c)* replacing the results of several intermediary computations.

3.4 Text style

Text styling and syntax highlighting play an important role in the editor and should be scalable and responsive. That is why, performing styling or syntax highlighting operations in a parallel thread is the only viable option. Unfortunately, it brings its own problems and difficulties such as thread safety and text synchronisation. One of the main problem is the fact that original text can be changed during styling process. Assume the code from the Listing 3.1 and imagine we would like to style `false` keyword with a style corresponding to Smalltalk pseudo-variables.

```
odd
  "Answer whether the receiver is an odd number."

  ↑self even == false
```

Listing 3.1: Implementation of the *#odd* testing method from the *Number* class

An object responsible for styling of a source code is called *syntax highlighter* and is nothing else than a visitor of the abstract syntax tree (AST) of that source code. Leaf AST nodes know their `start` and sometimes `stop` positions (Figure 3.7) within original source code. They can be used by syntax highlighter in order to apply text attributes on a text within *Interval* of that node. In our example that node interval equals to `(70 to: 74)`. The problem can occur if code gets changed after computation of its AST but right before attributes are applied on the text. For example if a user would delete any character from a source code, an interval `(70 to: 74)` would be no more valid, because an overall length of that source code is 73 which leads to `SubscriptOutOfBounds` exception.

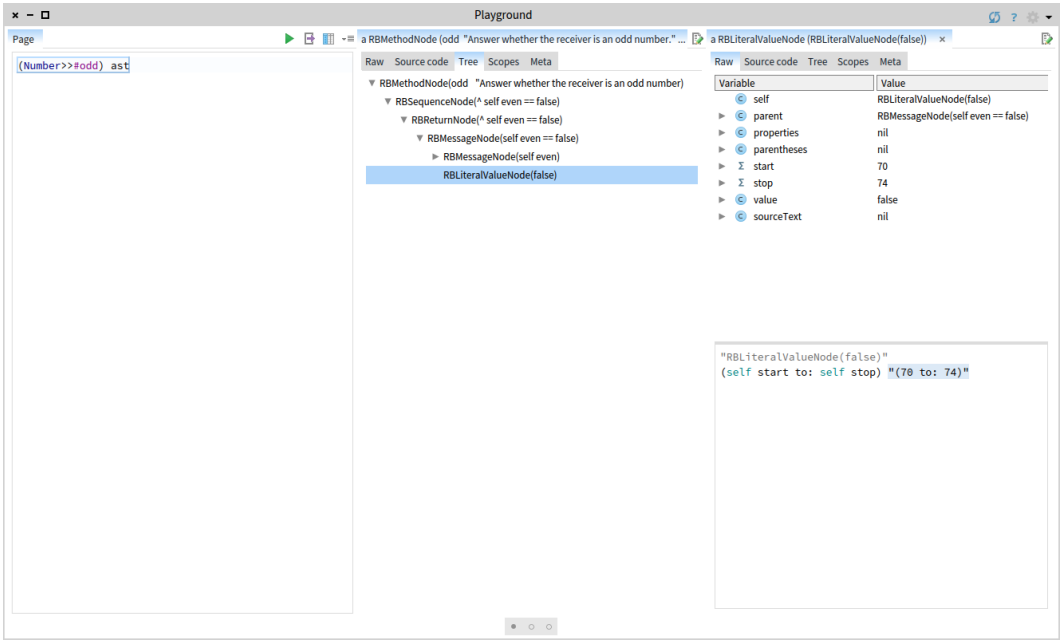


Figure 3.7: AST node of the *false* keyword and its interval in original source code

One possible solution would be to implement a locking mechanism and allow only one thread to modify and access text at a time. However, it would make the overall implementation more complex and will affect performance in a negative way. Another solution is to create a copy of a text in the UI thread and let styler operate on that copy. This way text editor and styler threads do not share a text instance and can operate

independently. The downside of that method is a need to create a copy of a text while blocking a UI thread during the copying process. Once styling is complete an original text should be replaced with a styled one on UI thread. However, if original text was modified during syntax highlighting we can not replace it with a styled copy and must discard it. Figure 3.8 gives a high level overview of how synchronisation problem is solved in the Moldable Editor.

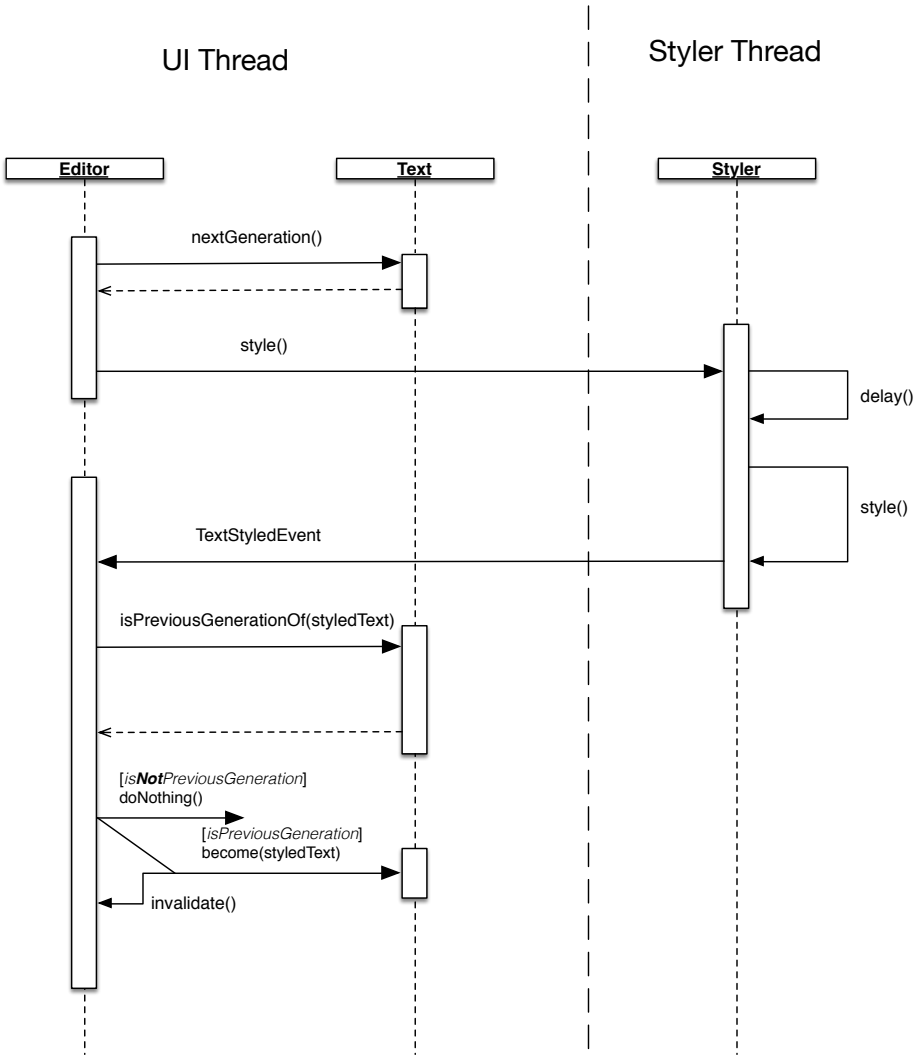


Figure 3.8: UML Sequence diagram of the styling process

As a first step the original text is asked to create a copy of itself and to mark it as a next generation: `nextGeneration()` . In this case a special immutable identifier object is used in order to check later whether the current editor's text is still a previous generation

of the one that was styled. Since *Rope* is a persistent data structure it can be directly used as a generation identifier. Moreover, its immutability allows us to create new text copies without any additional overhead, because a copy simply refers to the same rope instance as a text that has been copied. As soon as the editor receives a next generation back from the text it asks a styler object to perform necessary operations on that copy. On the other side, a styler responds to the `style` message by terminating any existing styling process (in Pharo a green thread is called a *Process*) and creating a new one that starts with a `delay()` as its first operation. Delay allows styler to save computation resources by waiting until user stops typing. It improves an overall editor responsiveness as perceived by the user. Exact delay time is configurable and may vary.

As soon as text is styled a styler announces `TextStyledEvent` that indicates the fact that the process is finished. That announcement is deferred on the UI thread which allows editor to handle the event in a synchronous way and perform all necessary invalidation operations without breaking editor's integrity. Then editor checks whether the original text was changed since the beginning of a styling process by making sure that a styled text is a next generation. If it is a case, the editor asks current text model to replace its content with the content of the styled one, otherwise a styled version is discarded. As a final step editor performs invalidation in order to update the on screen rendering to correspond a new text state.

3.5 Segments and Rendering

In order for the editor to be scalable it should be implemented in such a way that the overall performance is independent from the text size. A common technique to achieve it is to only process a part of the scene that is currently visible to the user. It means that we should not render and lay text out if it is outsize of the current viewport of the editor. Similar behaviour can be found in various graphical frameworks. A set of widgets that work only with visible elements includes for example *FastTable* in Pharo, *RecyclerView* in Android and others. Bloc [3] graphical framework that is used as an underlying layer for the Moldable Editor contains such a widget, called *InfiniteElement*, where *infinite* stands for *practically infinite* as it allows developers to create scrolling lists that are able to display large datasets. Figure 3.9 shows a high level overview of the *InfiniteElement's*scrolling behaviour. At any time only visible graphical elements are added to the composition tree. It allows text editor to render its content almost instantly and makes it possible to have smooth scrolling animation. When a viewport of the text editor is resized only a fraction of the overall text has to be remeasured and layered out.

However, the described approach has its own limitation. In order for the *InfiniteElement* to create or reuse visual elements on demand, underlying data source has to be

[3] `https://github.com/pharo-graphics/Bloc`

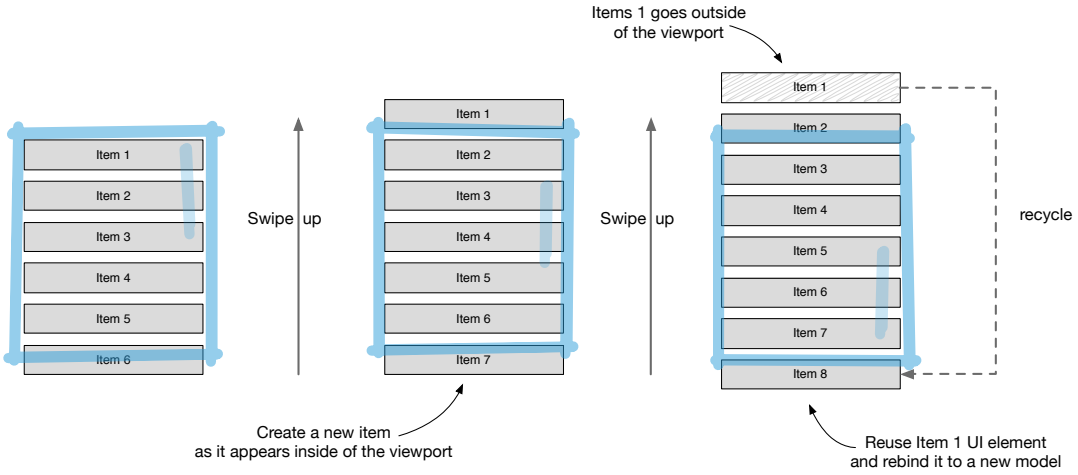


Figure 3.9: Scrolling items in and out of viewport using *InfiniteElement*

indexable and discrete. It means that the whole text has to be split in so called *Segments*. In its current implementation, a *Segment* represents a line of text, however, it can be a whole page or a collection of paragraphs. Nevertheless, if a text file is large (Gigabytes of data), reading it and splitting into segments becomes slow and inefficient. To solve this problem the editor pre-loads only a portion of the text and splits it into segments. Figure 3.10 shows how text segments are mapped on a pre-loaded portion of text and how that portion is related to the original text.

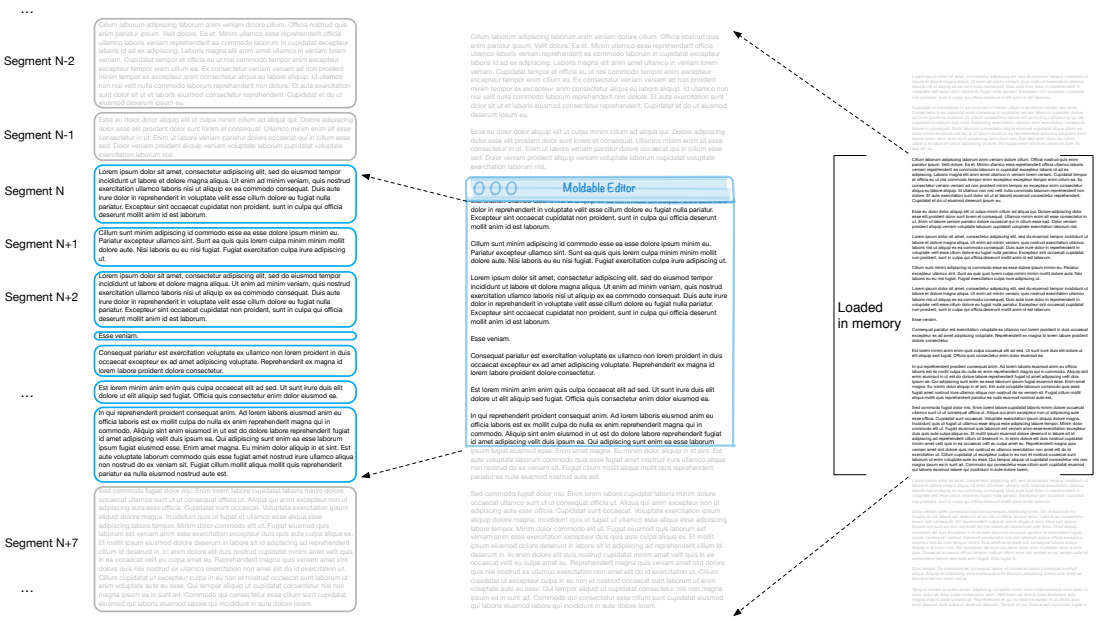


Figure 3.10: Segments

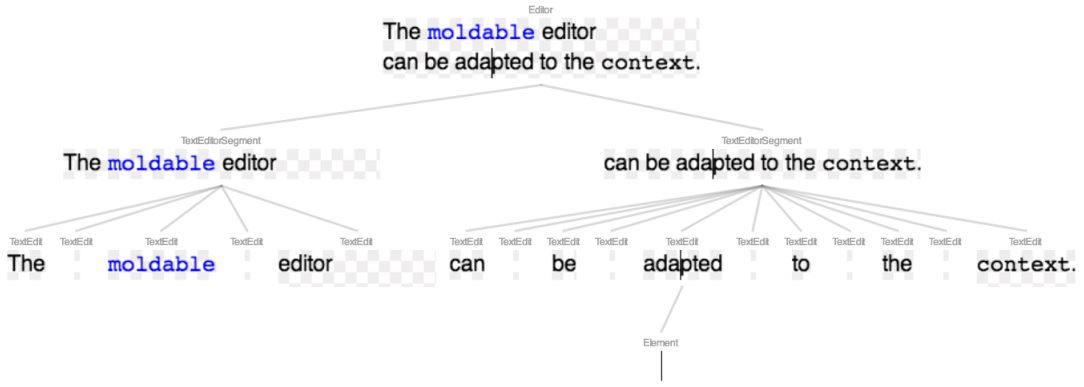


Figure 3.11: A structure of the graphical composition tree

4 The Validation

4.1 Overview

Scalability

The moldable editor is both flexible and scalable. For example, the following piece is a sizeable 2MB of text, and yet it opens smoothly (Figure 4.1).

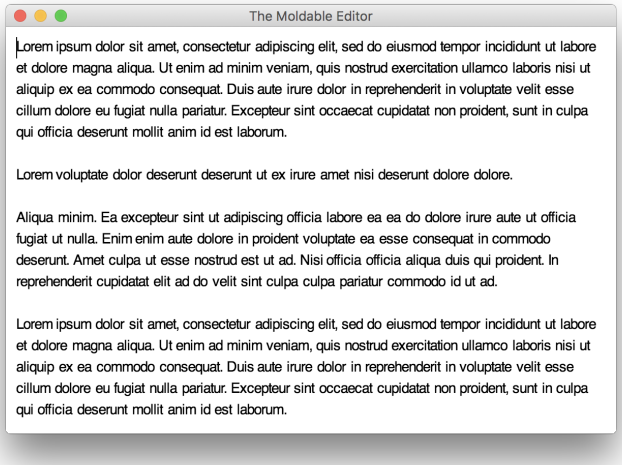


Figure 4.1: The Moldable Editor opened on a 2MB text

Syntax highlighting and adornments

An obvious application for the text editor is a code editor with syntax highlighting. Like in any other text editor that supports syntax highlighting, the syntax highlighter works in a separate process and changes the existing text. Typically, the syntax highlighting affects text attributes such as color or font weight. However, the moldable editor brings this concept further and allows us to add arbitrary visual elements to a text scene.

For example, in the snippet below (Figure 4.2), we see the code associated with the example that produces the editor element with syntax highlighting. In addition to the typical Pharo syntax highlighting, we can also notice small triangles inserted in the code. These triangles denote a dependency to another example method, and clicking on one expands the code in place showing another editor (Figure 4.3).

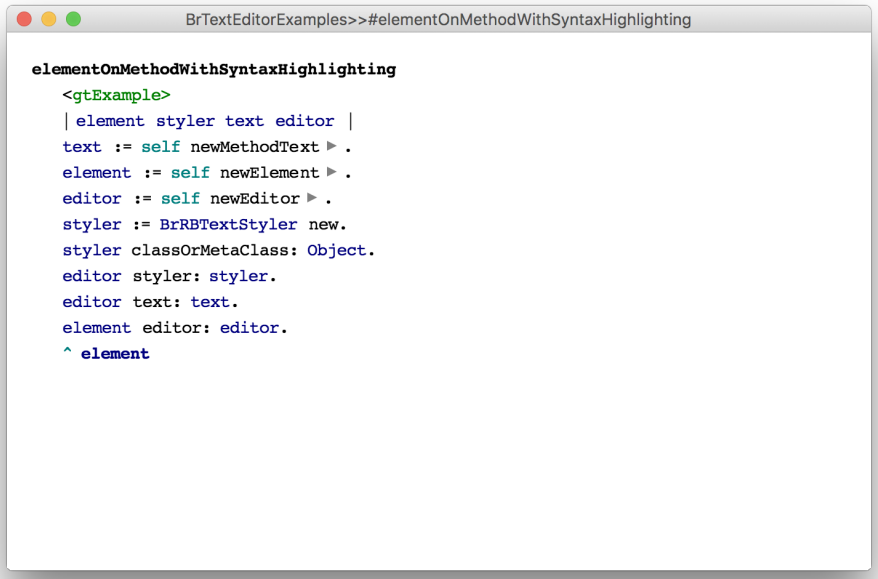


Figure 4.2: The editor with syntax highlighting

This functionality is obtained through a dedicated syntax highlighter that extends the default Pharo highlighting with an extra logic that adds the triangles as adornments. Clicking on such an adornment adds another adornment with another editor element. Interestingly, this all happens live, which means that if you change the code from the root editor element to no longer refer to an example, the corresponding triangle and embedded editor elements will disappear.

Figure 4.3: The editor with expanded example

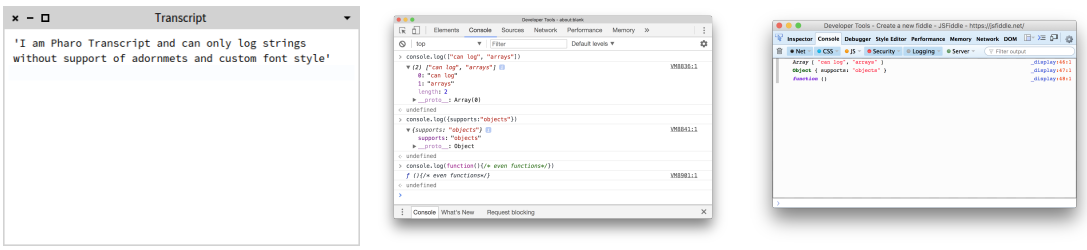
The example above also reveals the way to initialise an editor element.

4.2 Transcript

Due to its rich abilities, the moldable editor has the potential of changing all tools that rely on textual representations. One such a tool is the Transcript.

In a context Pharo the *Transcript* is a tool that allows users to log stream messages. Additionally, Pharo provides a user interface to show those messages. It means that when it comes to logging tools we should distinguish an API used to output to a stream and a user interface, which is one of the multiple ways to display the output. Existing *Transcript* in Pharo has an ability to display logged messages in a simple text editor or to write them into a file.

Many languages have support of the message logging and their IDEs provide dedicated tools allowing users to browse and read that log. For example in *JavaScript*, the *Console* is an object with an API that can be used to log runtime artefacts such as strings, arrays, functions or object. By default, most modern web browsers are shipped with developer tools and one of them is an interactive console. For example. Mozilla



(a) (b) (c)

Figure 4.4: Examples of a transcript or console tools: *a)* the standard Pharo Transcript; *b)* Chrome Console; *c)* Firefox WebConsole

Firefox provides *Web Console* [1] which is different from Google Chrome's *Console* [2] while providing very similar functionality.

The goal of GT Transcript is to offer a rich and interactive interface for displaying live information coming from a system.

The API

The API is backward compatible with the existing transcript. To enable the new features, we introduced a builder. For example, `transcript nextPutAll: 'something'` becomes `transcript next putAll: 'something'`. Between *next* and *putAll:*, we can add multiple attributes to the text output.

[1] https://developer.mozilla.org/en-US/docs/Tools/Web_Console
[2] https://developers.google.com/web/tools/chrome-devtools/console/

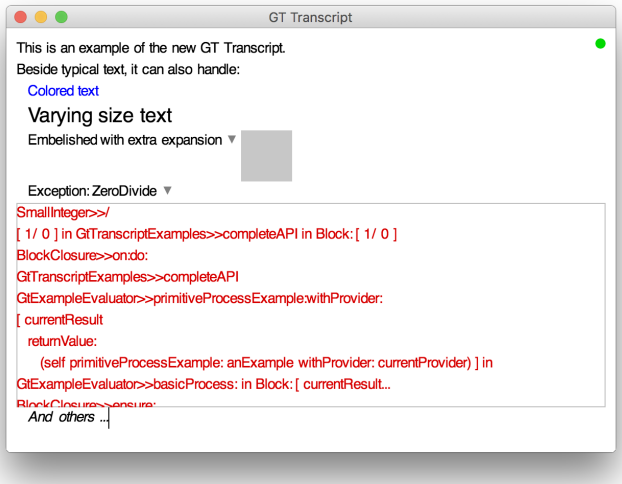


Figure 4.5: Visual output of Listing 4.1

The following example shows the complete API:

```
| transcript |
transcript := GtTranscript new.
transcript
  nextPutAll: 'This is an example of';
  space;
  nextPutAll: 'the new GT Transcript';
  nextPut: '.';
  cr.
transcript next
  putAll: 'Beside typical text, it can also handle:';
  cr.
transcript next
  tab;
  color: Color blue;
  putAll: 'Coloured text';
  cr.
transcript tab.
transcript next
  fontSize: 20;
  putAll: 'Varying size text';
  cr.
transcript next
  tab;
  expanding: [ BlElement new background: Color gray ];
  putAll: 'Embellished with extra expansion';
  cr.
[ 1/0 ] on: Error do: [ :err |
  transcript next
    tab;
    putAll: 'Exception: ';
    showException: err;
    cr ].
transcript next
  tab;
  italic;
  streamAll: [ transcript next putAll: 'And others ...' ].
```

Listing 4.1: The complete api of the GT-Transcript

Logging an animation

To get an idea of how this tool could be useful, take a look at the following example on the Figure 4.6. A Bloc animation is logged in a visual, domain centric way, providing insight far superior to plain text.

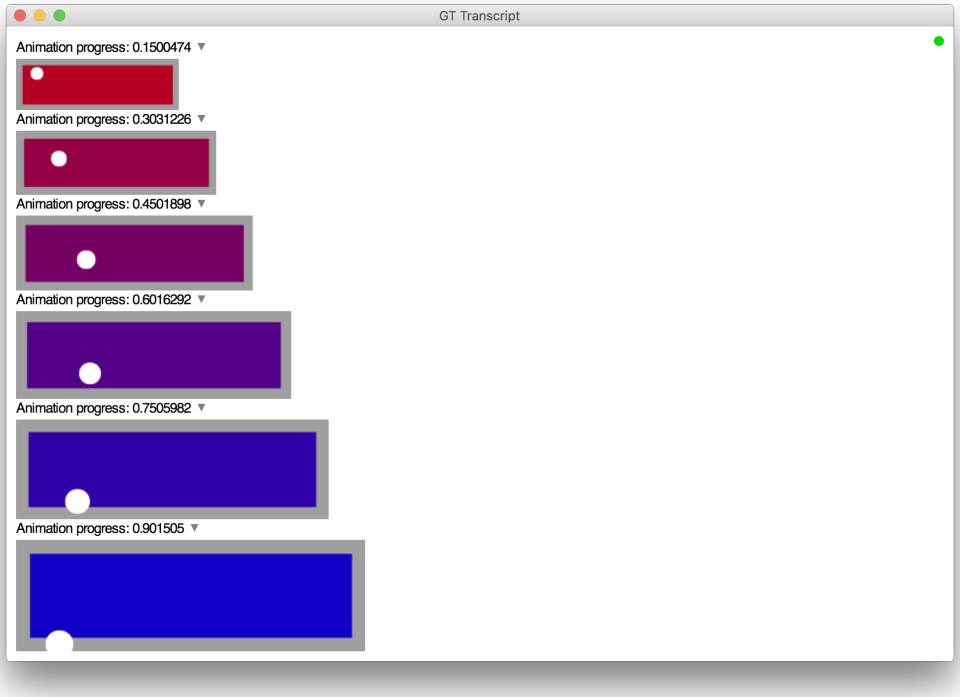


Figure 4.6: Visual logging of Animation with GT Transcript

4.3 Connector

In the 4.1 section, we saw how examples dependencies can be expanded in place by utilising the syntax highlighter. Connector brings this a step further and proposes a new kind of interface that allows the user to expand a new editor for an example and to automatically connect editor elements with one another. The interface is somewhat similar to the one proposed by Code Bubbles [3].

[3] http://cs.brown.edu/~spr/codebubbles/

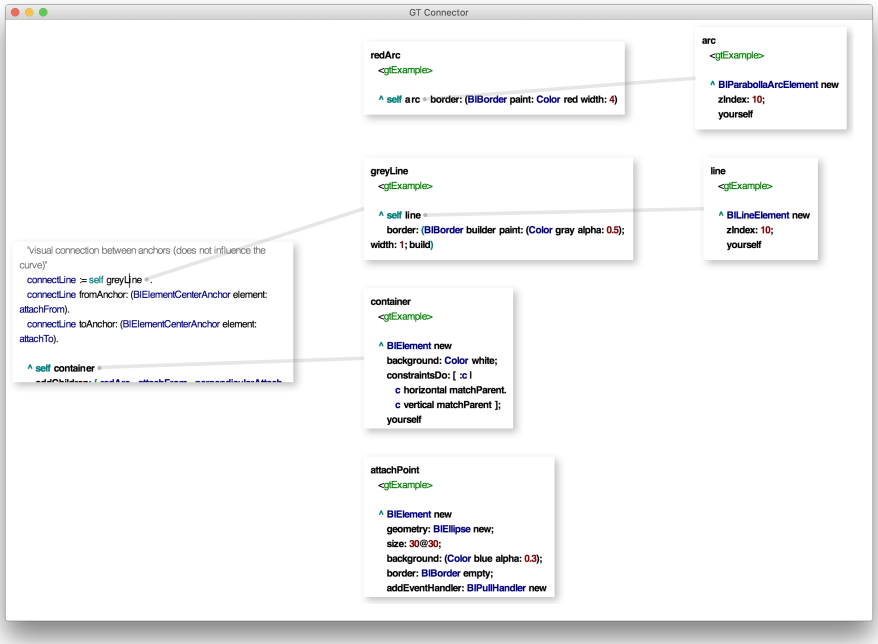


Figure 4.7: GT Connector opened on an example method

However, it has two key differences:

- the lines connect an element inside the text editor to the outside world. This is possible because the text is represented as elements that are rendered in the main rendering tree provided by the underlying Bloc framework.
- the lines are added automatically to reveal dependencies that are otherwise more difficult to spot.

4.4 Documenter

The Documenter offers live rendering of Pillar[4] documents. For example, Documenter can embed pictures right in place:

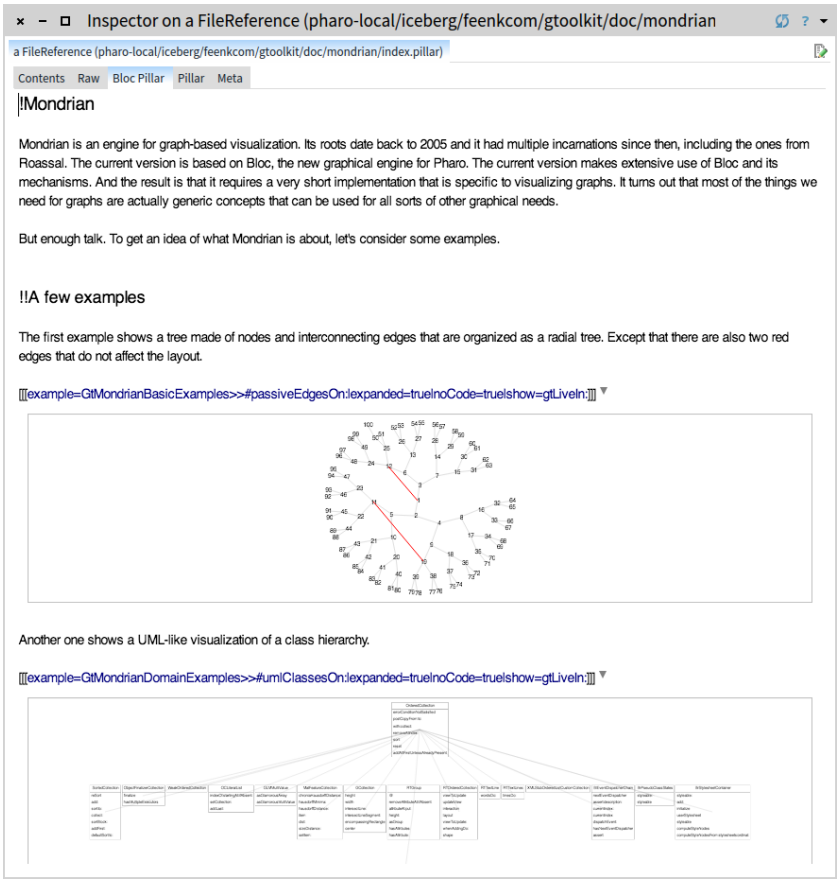


Figure 4.8: Documenter opened on a Mondrian[5]documentation file

[4] https://ci.inria.fr/pharo-contribution/job/EnterprisePharoBook/lastSuccessfulBuild/artifact/book-result/PillarChap/Pillar.html

[5] https://github.com/feenkcom/gtoolkit-visualizer

And it can even embed live code that can be previewed in place:

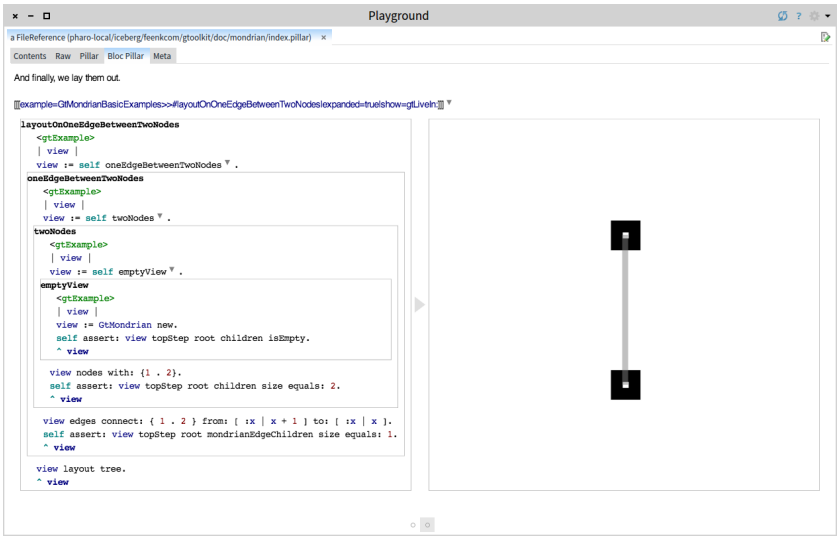


Figure 4.9: Live preview of the example result within the Documenter

5
Conclusion and Future Work

In which we step back, have a critical look at the entire work, then conclude, and learn what lays beyond this thesis.

6

Anleitung zu wissenschaftlichen Arbeiten

The Moldable Editor is a part of Brick[1], a widget library on top of *Bloc*, new low level vector graphical framework for *Pharo*.

It can be installed in Pharo 6 or later by executing the following script in *Playground*:

```
Metacello new
    baseline: 'Brick';
    repository: 'github://pharo-graphics/Brick/src';
    load: #core
```

Listing 6.1: Brick installation script

The applications of the Moldable Editor (Transcript, Connector and Documenter) are part of the Glamorous Toolkit (GT)[2], which is the second generation of GT[3] that is based on the Bloc project.

It can be installed with the help of the following script:

```
Metacello new
    baseline: 'GToolkit';
    repository: 'github://feenkcom/gtoolkit/src';
    load.
```

Listing 6.2: GToolkit installation script

[1] https://github.com/pharo-graphics/Brick
[2] https://github.com/feenkcom/gtoolkit
[3] http://gtoolkit.org/

Bibliography

[1] Charles Crowley. Data structures for text sequences. *Computer Science Department, University of New Mexico, Date*, pages 1–29, 1998.

[2] GitHub Inc. The state of atom's performance, 01 2018.