

u^b

b
UNIVERSITÄT
BERN

The Moldable Editor

Bachelor Thesis

Aliaksei Syrel

from

Minsk, Belarus

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

6. February 2018

Prof. Dr. Oscar Nierstrasz

Dr. Andrei Chiş, Dr. Tudor Girba

Software Composition Group

Institut für Informatik

University of Bern, Switzerland

Abstract

We present a scalable and moldable text editor modelled as a single composition tree of visual elements. It allows us to mix text and graphical components in a transparent way and treat them uniformly. As a result we are able to augment code with views by using special *adornment* attributes that are applied on text by stylers as part of the syntax highlighting process. We validate the model by implementing a code editor capable of manipulating large pieces of text; the Transcript, a logging tool being able to display non-textual messages; the Connector, an example dependencies browser and the Documenter, an interactive and live editor of Pillar markup documents.

Contents

1	Introduction	1
2	Related Work	3
3	The Moldable Editor	5
3.1	Overview	5
3.2	Text Model	7
3.3	Text data structures	7
3.3.1	Pharo	8
3.3.2	Atom	9
3.3.3	Emacs	10
3.3.4	Rope	10
3.4	Text style	14
3.5	Segments and Rendering	16
4	The Validation	19
4.1	Overview	19
4.2	Transcript	21
4.3	Connector	25
4.4	Documenter	26
5	Conclusion and Future Work	29
6	Anleitung zu wissenschaftlichen Arbeiten	31
6.1	Prerequisites	31
6.2	Setup	32
6.3	Usage	33

1

Introduction

Almost all programming languages are purely textual and developers spend most of their time reading and writing code. It makes text editors an important tool in software development. They also happen to be a central IDE tool used by programmers to edit code.

Source code is usually a plain text without any additional formatting, compared to word processor documents or documents written using markup languages. However, modern code editors facilitate syntax highlighting to improve code readability without changing the text meaning. Nevertheless, in most cases, syntax highlighting does not take domain or live objects into account thus still leaving developers with textual representation. This issue can be addressed through code editors allowing developers to seamlessly augment textual code with additional visual information or graphical views for run-time objects. In order for developers to benefit the most, such editors have to be flexible, hence, *moldable* [5] and should allow users to augment code with any graphical components easily going beyond the predefined ones. By any graphical component we mean the whole spectrum of widgets, scalable visualisations or even recursive integration of text editors in themselves.

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.

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 text, and visual elements differently.

During development, it is not unusual to have to manipulate large text files, such as for configurations or logs. To this end, an important requirement for a practical text editor is to be scalable.

In the first part, we explain in more detail how the editor is implemented and describe a rope data structure [2] behind a text model. Additionally, we introduce a way of storing text attributes and their underlying text sequence in the same data structure.

In the second part, we discuss a few applications of such an editor to show how a single composition tree makes that editor flexible, and what it could mean to have it in a live programming environment.

In 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 the context of a live programming environment such as a debugger, an inspector or live code snippets.

2

Related Work

Since one of the first formal descriptions of a text editor was proposed in the early 1980s [17] there exists a plethora of work on text editors.

An ability to embed pictures or other graphical components is not new and can be found in text editors incorporating visual aspects [1, 3, 4, 10, 12–15, 18], in various fields. Highly relevant for this work are approaches combining code and graphical views, common in the area of projectional editors (Figure 2.1 (a)).

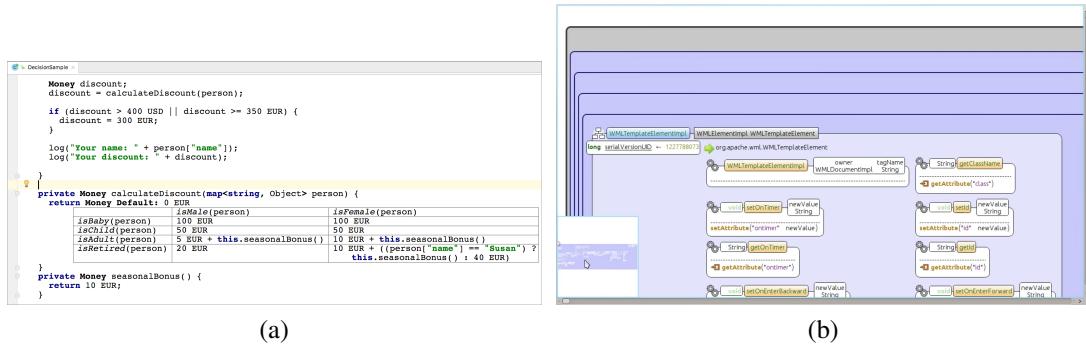


Figure 2.1: Examples of the editors combining code and graphical views: *a*) Jetbrains MPS; *b*) Envision.

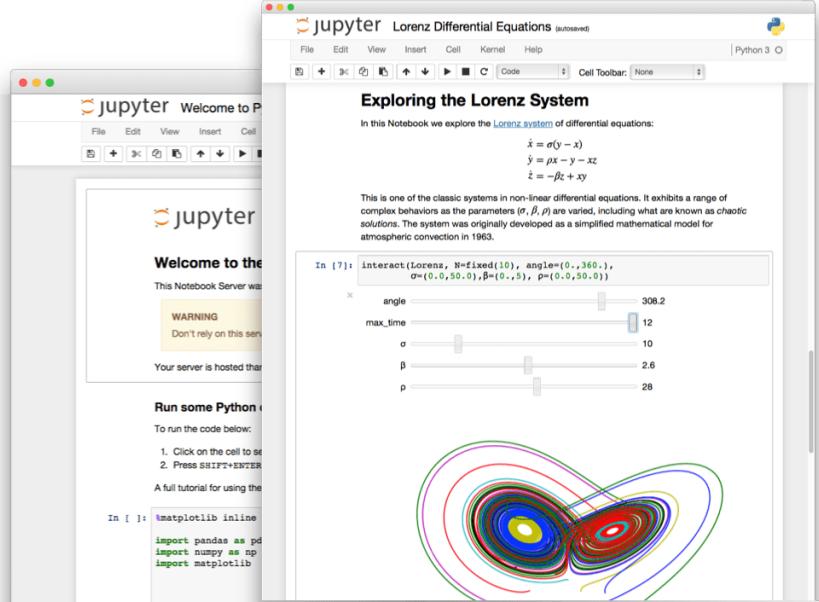


Figure 2.2: Jupyter Notebook

An interesting example is the Jupyter Notebook¹ which combines live code, visualisations and explanatory text to create documents (Figure 2.2).

For example, MPS² takes 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. Envision³, proposed a highly scalable visual editor (Figure 2.1 (b)).

Playgrounds, like Swift⁴, Scala⁵ or Eve⁶, give live and continuous feedback of a computation by adding graphical views next to the code. Many IDEs, when in the debugger, link the code editor with run-time objects (e.g., Eclipse, IntelliJ, VisualStudio), and display views in pop-ups. IDEs like Pharo⁷ or Squeak⁸, embed the code editor into the object inspector. Another category of systems provide complete graphical editors. This includes, for example, modelling frameworks, like EMF⁹ or VPL¹⁰, or visual languages [6].

¹<http://jupyter.org/>

²<https://www.jetbrains.com/mps/>

³<http://dimitar-asenov.github.io/Envision/>

⁴<https://developer.apple.com/swift/playgrounds>

⁵<https://github.com/scala-ide/scala-worksheet/wiki/Getting-Started>

⁶<http://witheve.com>

⁷<http://pharo.org>

⁸<http://squeak.org>

⁹<https://eclipse.org/modeling/emf/>

¹⁰<https://msdn.microsoft.com/en-us/library/bb483088.aspx>

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.

One of the requirements for that type of the editor is an ability to embed visual components inside of text so that they are neither deletable nor selectable. We call such elements *adornments*. To unify how text and adornments are treated, the editor also represents pieces of text as visual elements, thus bridging 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. An interesting aspect of that text model is the fact that it is data structure independent. It allows us to implement different text types, for example `SubText`, which is a subset of existing text within an interval, or `SpanText`, which represents a uniform piece of text where all characters have identical attributes. From the text editor's perspective there is no difference between those types of text, so the interaction happens through a clear `Text API`. More about text model can be found in Section 3.2.

When talking about the text model we should not forget about the text style, which is defined by a set of text attributes. Those attributes can be applied to 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 as part of the code editor. Even more important is the 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



Figure 3.1: The Moldable Editor

in the editor. Text stylers take context into account which is essential for the creation of context-aware development tools. In order for the editor to be fast and responsive, long and time consuming operations such as text parsing and styling should happen in a parallel background thread. It is in fact a non-trivial task, since users are able to perform text modification operations while a styler applies attributes on a text that is being modified. In Section 3.4 we 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 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 difficult 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 held by a data source object which knows how segments should be represented. It is also responsible for binding a segment model

¹<https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html>

²<https://developer.apple.com/documentation/uikit/uitableview>

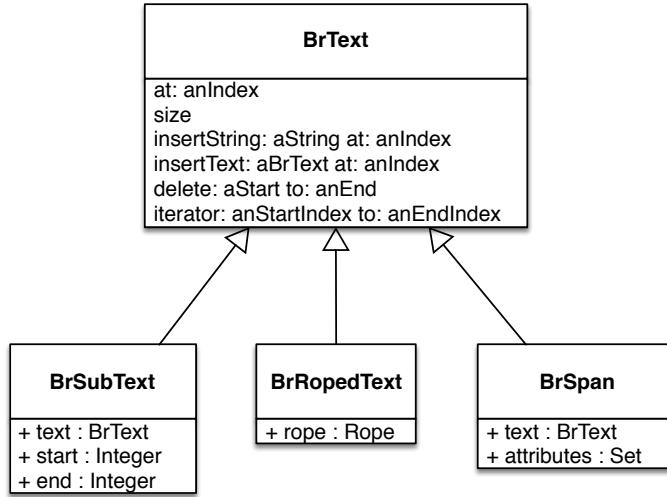


Figure 3.2: The UML class diagram of Text Model

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. A structure of the segment and its visual part is explained in more detail in Section 3.5.

3.2 Text Model

To display and edit text, an editor requires a text model that provides text modification and enumeration API. In the context of a text editor, by *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 users to iterate over all characters in a natural ordered way. Being indexable is an essential property of a text model, since text stylers require text to have characters accessible by index. Code parsers create an *AST* which consists of nodes bound to original text with the help of integer intervals in a form of a *[from, to]* tuple. Those intervals are later used by a text styler to apply attributes on a piece of text within those intervals.

3.3 Text data structures

To be considered scalable the moldable editor should be able 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. After searching for a data structure to be used by a text model we realised that there is no “silver bullet” data structure that is memory efficient, has the fastest random access and modification operations. Instead, it turned out that depending on the context and the way a text editor will be used it may be important to be able to select one or another data structure based on its properties. 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 backed up by the data structure of choice. In the following sub-sections we look at different data structures being used by other 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*.

Rubric

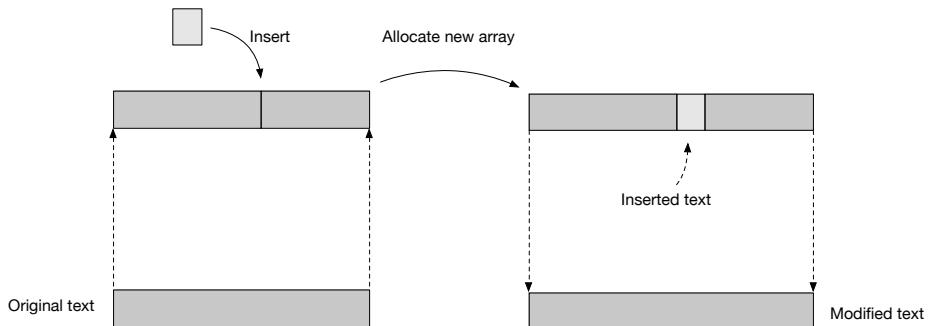


Figure 3.3: The array method

`Text` is the 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 other than an immutable array of characters. It means that every text modification such as insertion or deletion requires a text model to allocate and copy the whole array while replacing a subsequence of characters with a requested one as shown in 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 [7].

TxText

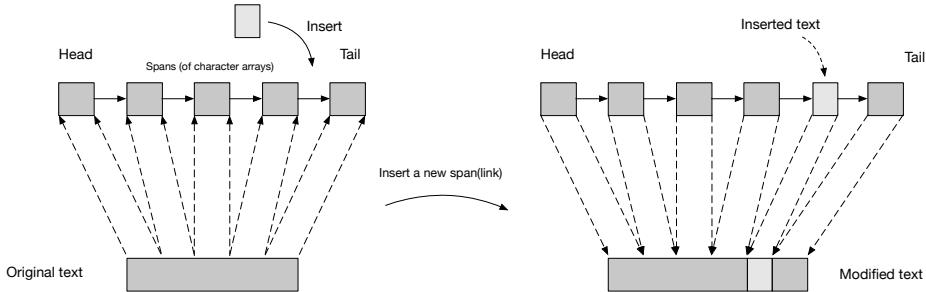


Figure 3.4: The linked list method

TxModel is a central class representing a text in the *TxText editor*. Internally it stores character sequences as a double-linked list of spans that consist of an actual text content. A linked list is considered to be an extreme choice as opposed to an array for 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 choices among all other text data structures [7].

3.3.2 Atom

The Atom³ text editor uses a memory-efficient data structure similar to a Piece Table [9]. A piece table is a data structure based on two buffers, one of which represents original read-only text while the second one stores all modifications to that text. All essential operations are performed with an adequate performance and some of them can be efficiently improved by using cache. A piece table is considered to be the data structure of choice for a text editor [7].

³<https://atom.io/>

3.3.3 Emacs

Emacs⁴ is based on a Gap Buffer, a data structure a little more complex as array but much more efficient [7]. The idea behind the gap buffer is simple, the whole text is stored in a large buffer that contains a gap at a cursor location or at a place where editing operations happen. Of course, this means that the gap must first be moved to the locus of the insertion or deletion. When the gap is correctly positioned editing operations are very efficient since the size of a buffer is small. However, the first editing command in one part of a large buffer, after previously editing in another far-away part, sometimes involves a noticeable delay [8]. A delay happens because to move a gap the whole buffer that stores the text has to be reallocated and memory to be moved.

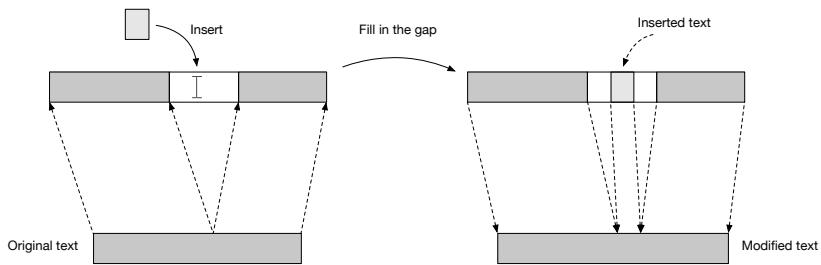


Figure 3.5: The gap buffer method

Figure 3.5 shows the workflow of an insertion operation with assumption that a gap buffer (white block) is already moved to the cursor location.

3.3.4 Rope

An alternative to array or buffer-based data structures is a rope [2]. A rope is a tree of concatenation nodes representing character strings. In addition to concatenation nodes, depending on implementation, a rope may include subset nodes, reverse nodes or other custom types of nodes. A rope allows text editors to manipulate large pieces of text and makes text operations such as random access, insertion and deletion very efficient. However, to maintain high efficiency a rope must be re-balanced otherwise it may lose its binary search tree properties and operations become inefficient.

If all operations are implemented in a non-destructive way, a rope becomes a persistent data structure. To enforce that, any modification should return a new node instance with applied changes. A root node of the returned rope, in this case, does not necessarily have the same type as the one that was modified. From a text editor perspective, persistence makes it easier to implement undo/redo commands and helps to prevent multithreading issues when it comes to text styling.

⁴<https://www.gnu.org/software/emacs/>

One of the main disadvantages of a rope is its implementation complexity and high possibility of bugs as a consequence. Compared to other common text data structures, a rope requires more memory space to store its tree structure. In languages without garbage collection, maintaining node references may be a tedious task and may lead to memory leaks.

Due to its disadvantages, ropes didn't become a traditional and commonly used data structure to be used in text editors. For that reason it was not possible to see ropes in action, in a contrast to gap buffers or piece tables that are nowadays used by modern and popular text editors. Its rare and complex nature, tree structure and possible object oriented implementation as also the existence of a garbage collector in Pharo made ropes a data structure of choice for the Moldable Editor.

Collection node

One of the main building blocks of ropes is a *collection* node (Figure 3.6). It is nothing other than a fixed buffer of characters of a limited length.

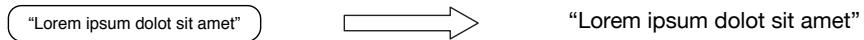


Figure 3.6: Collection node

If a total length of two collection nodes that are being merged exceeds a predefined limit, a result would be a *concatenation* node consisting of those collection nodes.

Concatenation node

A *concatenation* node has a central role in the structure of a rope. Similar to the binary search tree it knows its *left* and *right* child nodes.

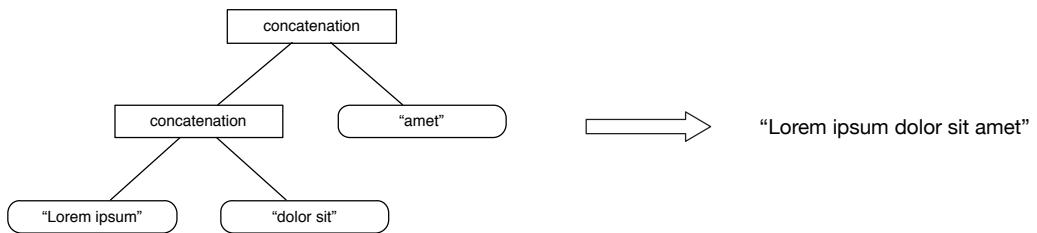


Figure 3.7: Concatenation node

If a total length of two ropes being concatenated is lower than a predefined limit, then instead of creating a concatenation node, they can be merged into a single *collection* node which allows a rope to be more memory-efficient and reduce amount of intermediate nodes.

Subset node

An ability to get a substring defined by an index interval plays an important role for the text editor, for example in case of *copy* or *cut* commands. It can be implemented by introducing a *subset* node that is a wrapper around another rope node with additional `start` and `end` attributes.

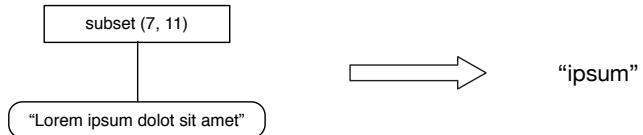


Figure 3.8: Subset node

Attribute node

As mentioned in Section 3.2 *Text* is not only a collection of characters but a set of associated attributes such as font size, text foreground or text style. Traditionally, attributes are stored in a separate data structure along a text sequence. For example *Rubric* text model allocates a dedicated *RunArray* of the same length as text itself, where every item is an array of attributes that corresponds to the character with identical index as shown in Figure 3.9. However, according to *RunArray*'s class comment in *Pharo* its internal implementation is space-efficient:

My (note: RunArray) instances provide space-efficient storage of data which tends to be constant over long runs of the possible indices. Essentially repeated values are stored singly and then associated with a "run" length that denotes the number of consecutive occurrences of the value.

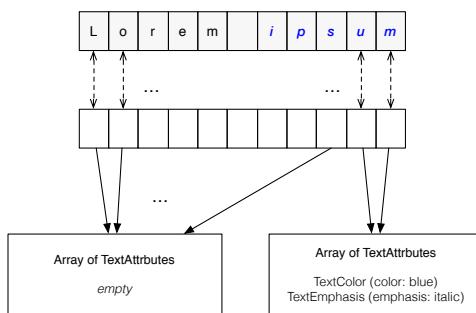


Figure 3.9: Text attributes structure of the Rubric text model

Instead of creating a separate data structure for text attributes, it is possible to incorporate it directly inside of the rope hierarchy by introducing a new *attribute* node. An attribute node is a wrapper around any rope node and it additionally stores a set of attributes that should be applied to that node as shown in Figure 3.10. It makes it possible to apply attributes to the whole rope by only wrapping it in a single object. Interestingly, since an *attribute* node does not operate with text intervals, as opposed to *RunArray*, there is no need to update those intervals when new text is inserted, as they get automatically expanded and applied to a new character sequence.

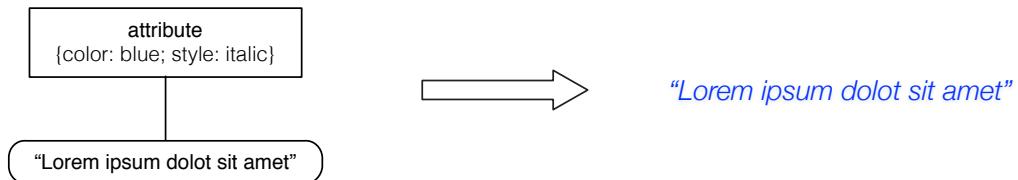


Figure 3.10: Attribute node

Additionally, attribute node makes it easier and more efficient to iterate over text *spans*, pieces of text where all characters have the same attributes. This iteration is necessary during the rendering or text measurement process, where every span has to be rendered or measured separately, since underlying 2D graphics libraries are only capable of rendering text sequences of the same pre-set style [11, 16].

Implementation

Figure 3.11 shows the class hierarchy of the rope nodes as it is implemented in the Moldable Editor.

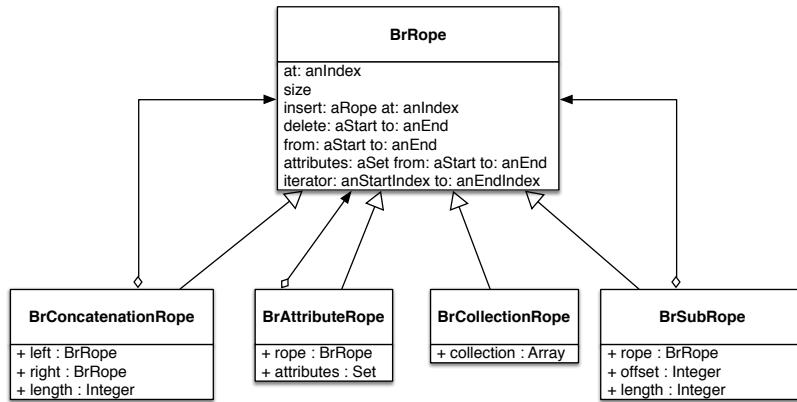


Figure 3.11: UML class diagram of the Rope hierarchy

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 problems is the fact that original text can be changed during the styling process. Assume the code from the Listing 3.1 and imagine we would like to style the `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*. It can be implemented as a visitor of the abstract syntax tree (AST) of that source code. Leaf AST nodes know their `start` and sometimes `stop` positions (Figure 3.12) within original source code. They can be used by a syntax highlighter to apply text attributes on a text within *Interval* of that node. In our example that node interval equals `(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.

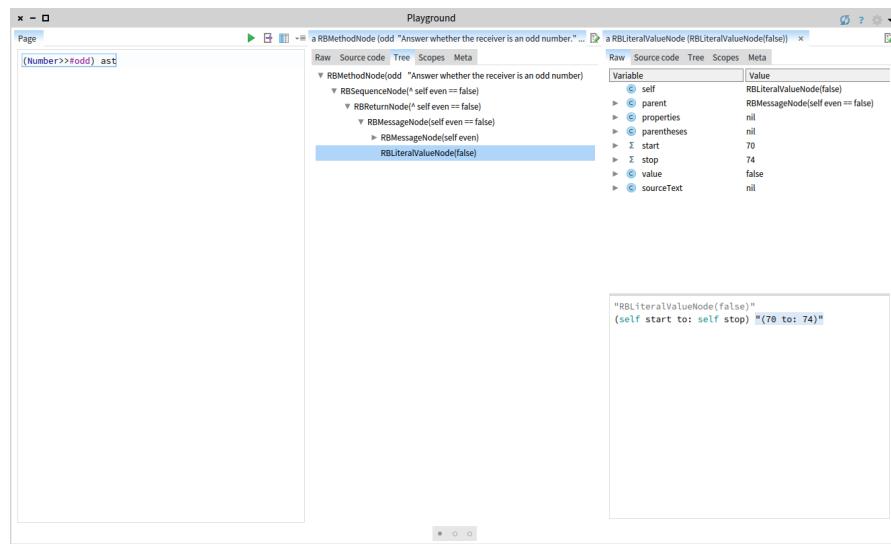


Figure 3.12: AST node of the `false` keyword and its interval in original source code

One possible solution would be to implement a locking mechanism and only allow one thread to modify and access text at a time. However, it would make the overall implementation more complex and affect performance in a negative way. Another solution is to create a copy of a text in the UI thread and let the styler operate on that copy. This way the 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 deferred on the UI thread. However, if an original text was modified during syntax highlighting we cannot replace it with a styled copy and must discard it.

Figure 3.13 gives a high level overview of how the synchronisation problem is solved in the Moldable Editor:

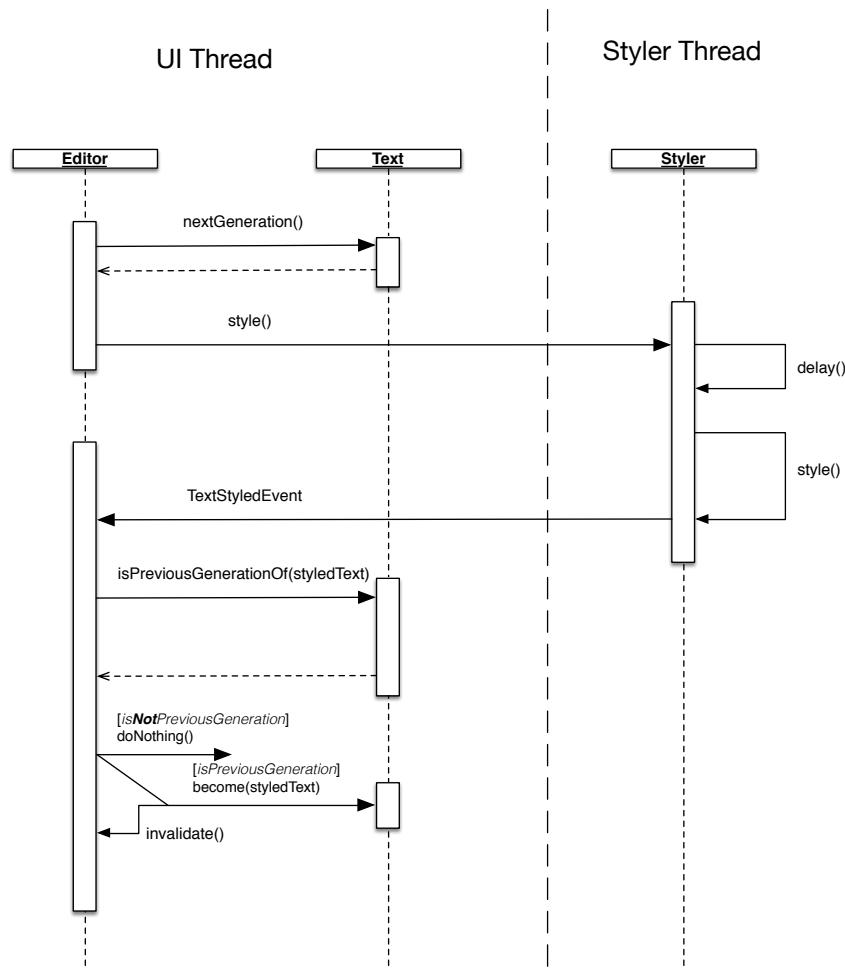


Figure 3.13: UML Sequence diagram of the styling process

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 to check later whether the current editor's text is still a previous generation of the one that was styled. Since a *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 was 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 a 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`, which indicates the fact that the process is finished. That announcement is deferred on the UI thread which allows the editor to handle the event in a synchronous way and perform all necessary invalidation operations without breaking the editor's integrity. Then the 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 the case, the editor asks the current text model to replace its content with the content of the styled one, otherwise the styled version is discarded. As a final step editor performs invalidation 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 of the text size. A common technique to achieve this is to only process the part of the scene that is currently visible to the user. It means that we should not render and lay out text if it is outside 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. Bloc⁵, a 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 3.14 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 the text editor to render its content almost instantly and makes it possible to have smooth scrolling animation. When a viewport of

⁵<https://github.com/pharo-graphics/Bloc>

the text editor is resized only a fraction of the overall text has to be remeasured and laid out.

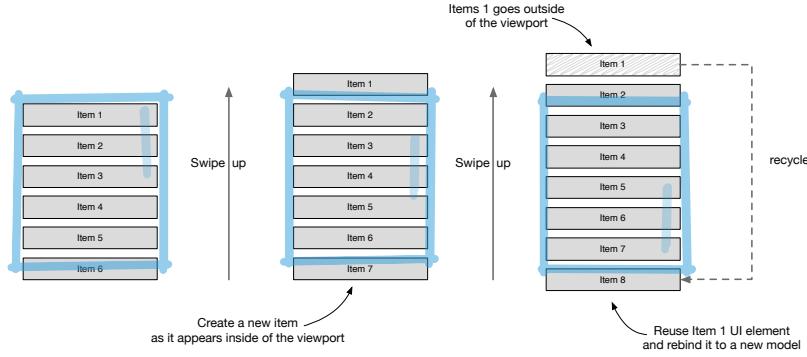


Figure 3.14: Scrolling items in and out of viewport using *InfiniteElement*

However, the described approach has its own limitation. In order for the *InfiniteElement* to create or reuse visual elements on demand, an underlying data source has to be indexable and discrete. It means that the whole text has to be split into 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 it 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 3.15 shows how text segments are mapped to a pre-loaded portion of text and how that portion is related to the original text.

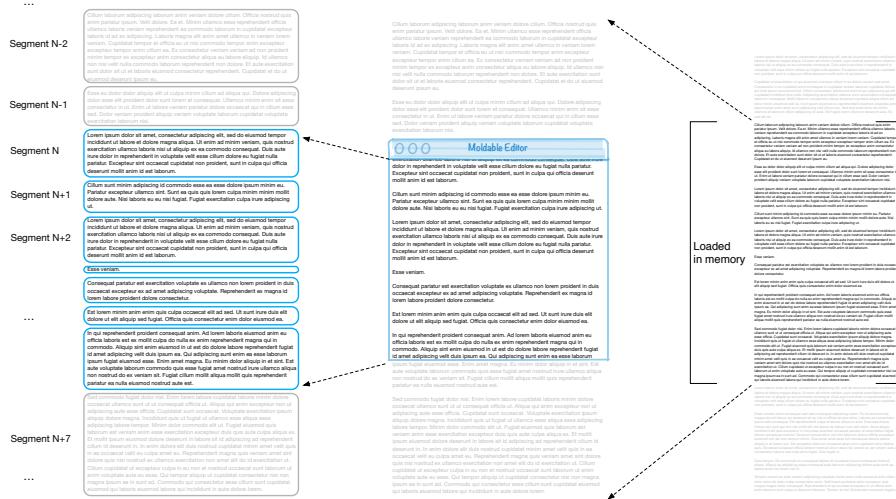


Figure 3.15: The mapping of the segments to the original text

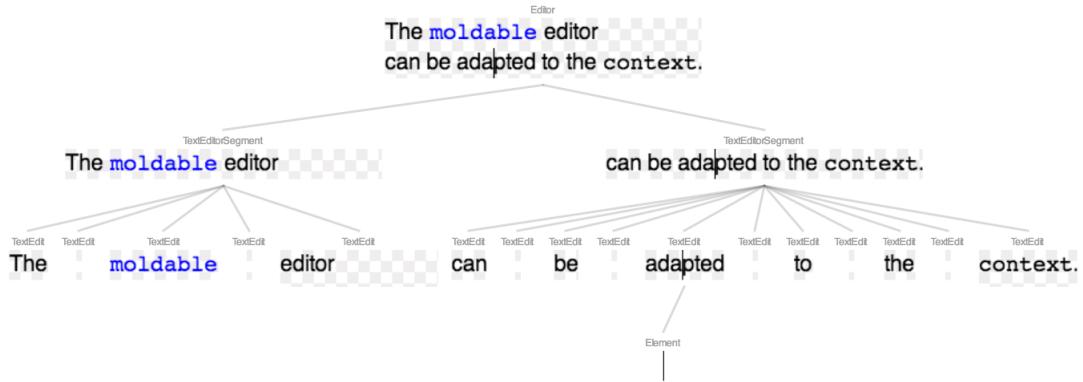


Figure 3.16: A structure of the graphical composition tree

Once segments are constructed, the editor creates visual elements to represent those segments. Figure 3.16 shows how an editor element, opened on a two-line text, is composed. Every line is encapsulated into a segment and is rendered as `TextEditorSegment`. Each segment is then split into pieces, in our case words. Finally, every word is rendered as `RenderingElementsMapping`. A cursor is also a visual element and is contained by a word element that happens to have a focus.

4

The Validation

4.1 Overview

Scalability

The moldable editor is both flexible and scalable. For example, the following piece is a sizeable 100MB of text, and yet it opens smoothly (Figure 4.1). It only takes around 100ms to open an editor window displaying previously mentioned text piece on MacBook Pro 2017.

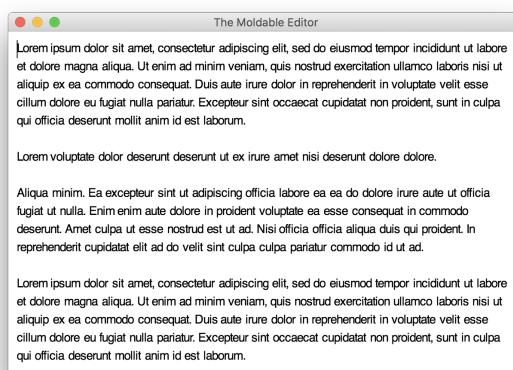
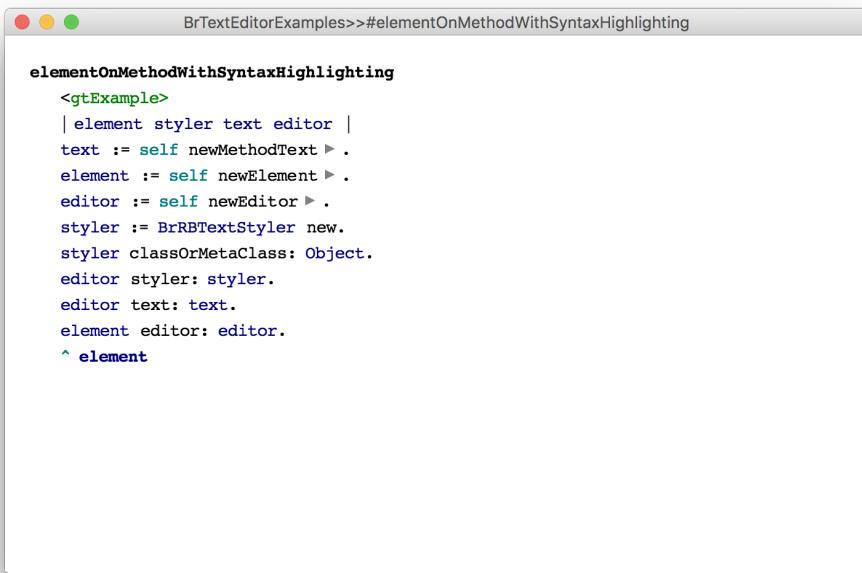


Figure 4.1: The Moldable Editor opened on a 100MB text

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).



```

elementOnMethodWithSyntaxHighlighting
<gtExample>
| element styler text editor |
text := self newMethodText ▶ .
element := self newElement ▶ .
editor := self newEditor ▶ .
styler := BrRBTextStyler new.
styler classOrMetaClass: Object.
editor styler: styler.
editor text: text.
element editor: editor.
^ element

```

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.

```

BrTextEditorExamples>>#elementOnMethodWithSyntaxHighlighting
elementOnMethodWithSyntaxHighlighting
<gtExample>
| element styler text editor |
text := self newMethodText ▶ .
element := self createElement ▽ .
newElement
<gtExample>

^ BrEditorElement new
constraintsDo: [ :c
  c horizontal matchParent.
  c vertical matchParent ];
yourself
editor := self newEditor ▶ .
styler := BrRBTextStyler new.
styler classOrMetaClass: Object.
editor styler: styler.
editor text: text.
element editor: editor.
^ element

```

Figure 4.3: The editor with an 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 the context of 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. The existing *Transcript* in Pharo has an ability to display logged messages in a simple text editor or to write them to a file.

Many languages have support for 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 Firefox provides

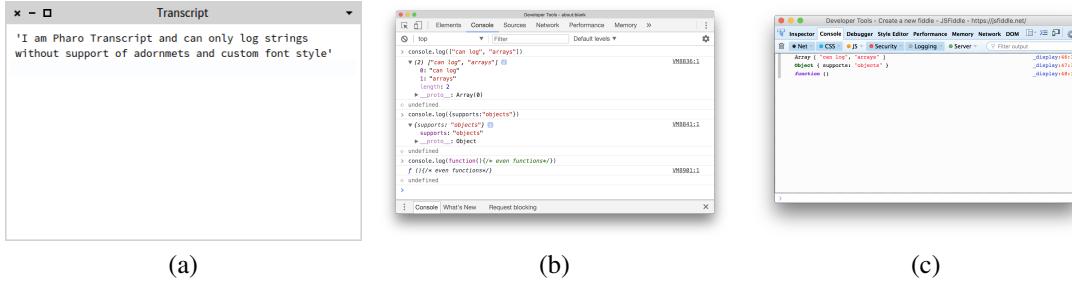


Figure 4.4: Examples of a transcript or console tools: *a*) the standard Pharo Transcript; *b*) Chrome Console; *c*) Firefox WebConsole

*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.

¹https://developer.mozilla.org/en-US/docs/Tools/Web_Console

²<https://developers.google.com/web/tools/chrome-devtools/console/>

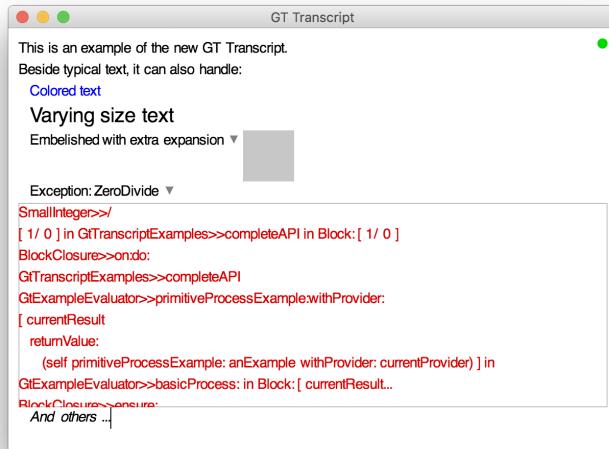


Figure 4.5: Visual output of Listing 4.1

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: [ BIElement 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

When working with animations it is important to be able to debug them. Developers may find themselves in a situation where a debugger is not helpful since it stops an animation process and only lets programmers to browse a frozen state. A different approach would be to observe how an animation progresses over time. Traditionally, developers would insert logging statements and output changing parameters. However, textual representation may be a limited source of information. Instead, we propose to use GT Transcript, for visual logging of animations providing insight far superior to plain text.

To get an idea of how this tool can be useful, take a look at the following example on the Figure 4.6:

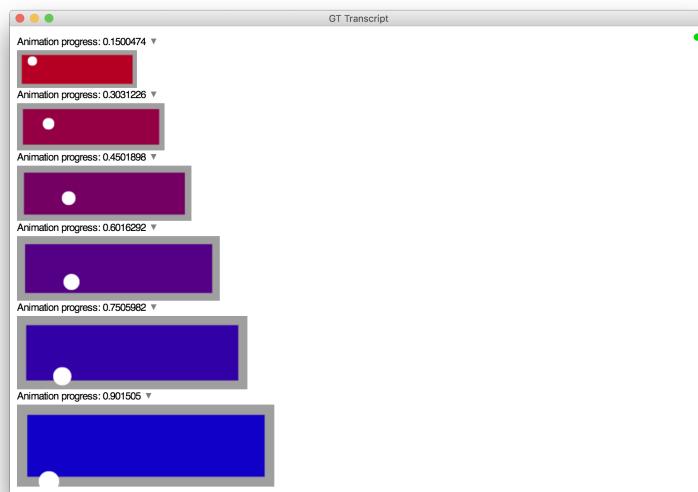


Figure 4.6: Visual logging of the animation with GT Transcript

There we have a rectangle element with a white circle as its direct child. Once an element is constructed we apply a composite Bloc animation that consists of a scale transformation and a color transition. Additionally, we apply a translation animation on the circle to make it move along a parabolic path. During the animation multiple parameters get changed which makes textual logging a tedious task. However, instead of text users could directly log visual elements, which appear as image snapshots in the Transcript.

To implement GT Transcript there were no changes made to the underlying editor model and all additional functionality was easily integrated which indicates the editor's flexibility. In the following sections we continue to validate the Moldable Editor by putting it in more different contexts.

4.3 Connector

In Section 4.1, we saw how example dependencies can be expanded in place by using the syntax highlighter. A Connector brings this a step further and proposes a new kind of interface that allows users to expand a new editor on an example method and to automatically connect editor elements with one another. The interface is somewhat similar to the one proposed by Code Bubbles³ [4].

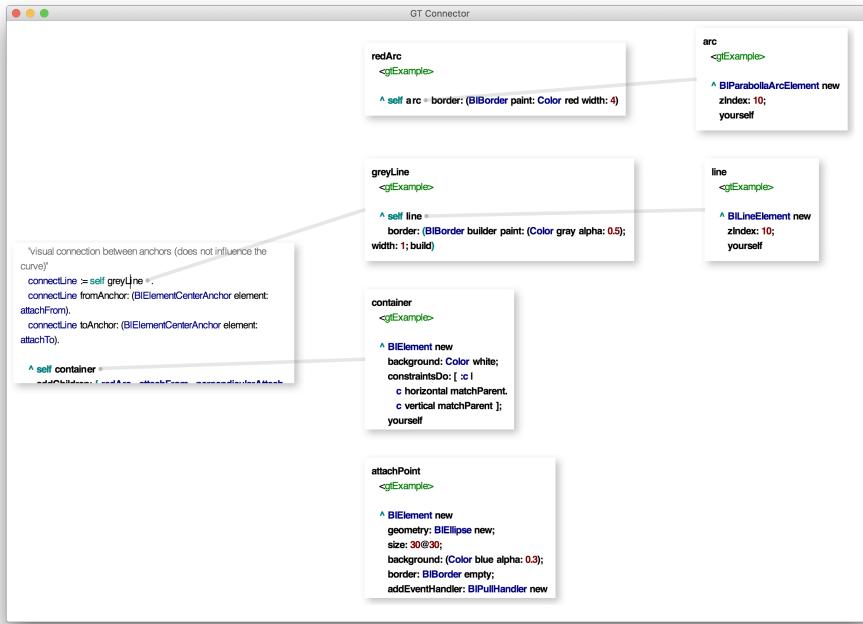


Figure 4.7: GT Connector opened on an example method

However, it has two key differences:

- lines can 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.
- lines are added automatically to reveal dependencies that are otherwise more difficult to spot.

³<http://cs.brown.edu/~spr/codebubbles/>

4.4 Documenter

Documentation plays an important role during development process. Developers spend a sizeable amount of time on writing it, maintaining and evolving it over time. Nowadays there exist techniques and automation processes that allow developers to create *technical documentation* outside of existing source code. Examples of such tools include Javadoc⁴, Doxygen⁵, Visual Expert⁶, to name a few. They help programmers to keep documentation up-to-date with almost no effort (Figure 4.8). However, such documentation is too technical and may not satisfy all the needs of the users. Instead of browsing auto-generated documentation users of a library or a framework could benefit more from *user documentation* such as tutorials, blog posts, thematics, guides or cheatsheets. One of the key points of user documentation is an ability to mix plain text explanations with code snippets and some sort of a preview of the result of those code snippets, sometimes in the form of screenshots. That is where a problem of user documentation comes from; tutorials must be updated due to API changes or UI improvements that may lead to visible differences between screenshots and the actual output of the code snippets if they would be executed in a real environment. Additionally, tutorials are not meant to be testable in any automatic way, for example on CI servers. As a result it increases the cost of maintaining user documentation and may even have a negative impact if new users face issues or discrepancies in their expectations based on tutorial and real results.

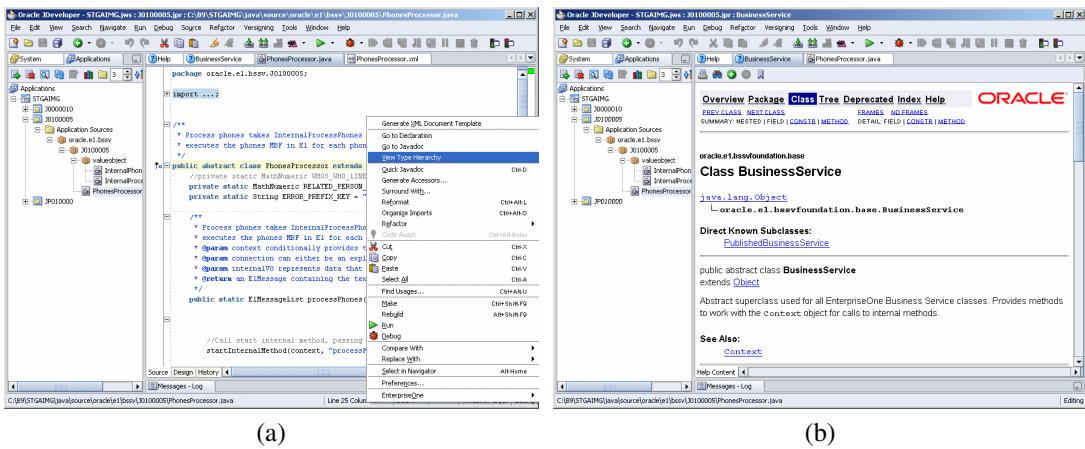


Figure 4.8: Two sides of Javadoc: *a*) source code; *b*) generated documentation

⁴<https://docs.oracle.com/javase/9/javadoc/javadoc.htm>

⁵<http://doxygen.org/>

⁶<http://www.visual-expert.com/>

The goal of the Documenter is to reduce the cost of maintaining user documentation and make it easier for developers to write it.

As a way to achieve it, the Documenter offers live rendering of Pillar⁷ documents. For example, as shown in Figure 4.9 Documenter can embed pictures right in place.

The screenshot shows the Pharo Documenter interface with the title "Inspector on a FileReference (pharo-local/iceberg/feenkcom/gtoolkit/doc/mondrian)". The tab bar includes "Contents", "Raw", "Bloc Pillar", "Pillar" (which is selected), and "Meta". Below the tabs, the content area starts with "!!Mondrian". The text describes Mondrian as an engine for graph-based visualization, mentioning its history and current implementation based on Bloc. It ends with a note about generic concepts for graphs.

!!A few examples

The first example shows a tree made of nodes and interconnecting edges that are organized as a radial tree. Except that there are also two red edges that do not affect the layout.

```
[[example=GtMondrianBasicExamples>>#passiveEdgesOn:expanded=trueNoCode=trueShow=gtLiveIn:]] ▾
```

A radial tree diagram with nodes numbered from 1 to 100. Most edges are grey, while two specific edges are red, indicating they are passive.

Another one shows a UML-like visualization of a class hierarchy.

```
[[example=GtMondrianDomainExamples>>#umlClassesOn:expanded=trueNoCode=trueShow=gtLiveIn:]] ▾
```

A UML class hierarchy diagram showing inheritance relationships between various classes. A central class, "Object", is at the top, with many other classes like "StringCollection", "ObjectCollection", and "MathematicCollection" inherit from it.

Figure 4.9: Documenter opened on a Mondrian⁸documentation file

⁷<https://ci.inria.fr/pharo-contribution/job/EnterprisePharoBook/lastSuccessfulBuild/artifact/book-result/PillarChap/Pillar.html>

⁸<https://github.com/feenkcom/gtoolkit-visualizer>

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

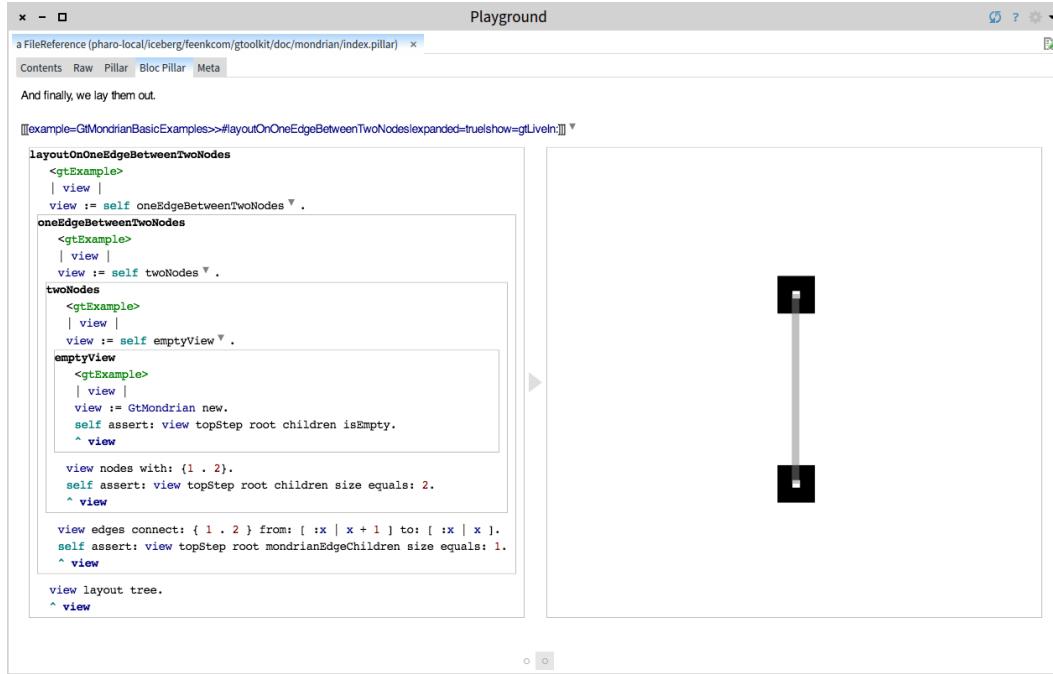


Figure 4.10: Live preview of the example result within the Documenter

As shown in Figure 4.10 the Documenter facilitates an ability of the editor to mix visual components with text, in this case by recursively composing editors with each other in order to expand the example's dependencies. It only becomes possible if we remove any constraints and enforce the editor to treat all contained graphical components uniformly.

5

Conclusion and Future Work

In the introduction we discuss why the text editor is an important and central tool. Being able to mix text and visual elements can enable new sorts of interfaces. For that to be possible a text editor has to be moldable, hence flexible. This thesis shows that to achieve moldability an editor should be represented as a single composition tree of visual elements. However, to be usable and practical it also has to be scalable and performant. In Section 3 we show how a *rope* data structure can be used to allow an editor to manipulate large pieces of text and describe how only visible *segments* of text are created and rendered.

To validate the model we implemented the code editor with an ability to expand methods right in place, within the Transcript, the Connector and the Documenter. These tools are built on top of the same editor infrastructure around the concept of *adornments* and a single composition tree of graphical elements. In Chapter 6 we provide a step-by-step tutorial of how to build an editor for a source code, how to create a live editor for Pillar documents and take a look at a low level use of the *adornments*.

Nonetheless, this work only scratches the surface of what can be done with the Moldable Editor. We already saw that it allows us to implement features and provide new experiences that are simply not possible in other editors. Nevertheless, while the basic infrastructure and underlying model is in place the editor in its current state misses some important features for usability that are normally used during day-to-day activities such as copy-paste, keyboard shortcuts, multiple text selection and smart cursor navigation. The Moldable Editor does not yet provide an auto-completion mechanism or common decorators which include line numbers, clickable links or code critique adornments.

Nevertheless, we proved that it is indeed possible to implement a text editor that can be represented in a single composition tree of visual elements. More importantly, it showed that such an editor not only can exist on paper or as a prototype but can actually be performant enough to be usable. It scales well and is capable of opening large pieces of text. The choice of the underlying Rope data structure turned out to be a good idea. Its persistence property helped us to deal with text synchronisation problems in case of multithreaded use and background text styling.

One more application for this text editor is code snippets¹ which we consider to be future work. The idea is to create an interface that allows users to split a single playground script into multiple independent *snippets* that can be executed and managed independently.

Another application of the Moldable Editor would be a code editor for an inspector and a debugger. There, live objects are already bound to variables. Hence, instead of exploring interesting objects in standalone object inspectors or pop-up views, developers could embed directly in the editor relevant views of those objects (Figure 5.1).

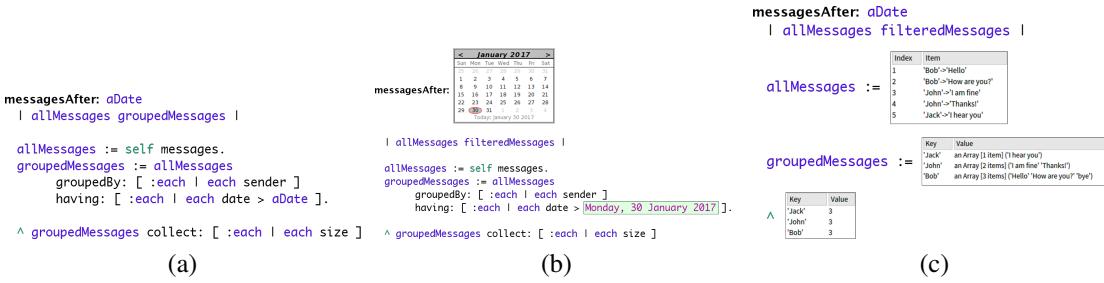


Figure 5.1: Three different ways to display the code of a method: *a*) the standard textual representation; *b*) replacing the data parameter using two distinct views; *c*) replacing the results of several intermediary computations.

¹<http://scg.unibe.ch/wiki/projects/Snippets-GTPlayground>

6

Anleitung zu wissenschaftlichen Arbeiten

In this chapter we go through the installation process of the Moldable Editor and its applications.

6.1 Prerequisites

As of February 2018, the Moldable Editor requires the stable Pharo 6.1¹ that can be downloaded from the official Pharo webpage. Users should choose a Pharo package depending on their operating system as shown in Figure 6.1:

Download Pharo

Version 6.1 for OS X, GNU/Linux, and Windows. The zip files contain all necessary files. Just download and run the executable.

Mac OSX

Windows

GNU/Linux

Figure 6.1: Pharo download page with prepared packages for different operating systems

Once downloaded, users can proceed to the installation and setup procedure as described in Section 6.2.

¹<https://pharo.org/download>

6.2 Setup

To install GToolkit users should execute the following script in Listing 6.1:

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

Listing 6.1: GToolkit installation script

It can be done from the Playground (Figure 6.2), which can be opened from the World Menu. To open the World Menu users should click anywhere on the background within the Pharo window. The installation script can be executed by clicking on the green play button in the top right corner of the Playground.

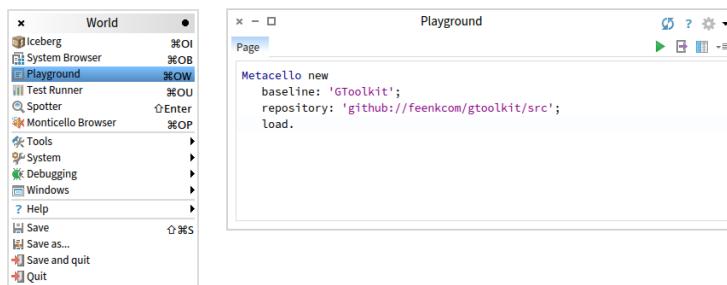


Figure 6.2: World menu and Playground with GToolkit installation script

As soon as the script is executed an installation process should start as shown in Figure 6.3. An image may become unresponsive during the installation.

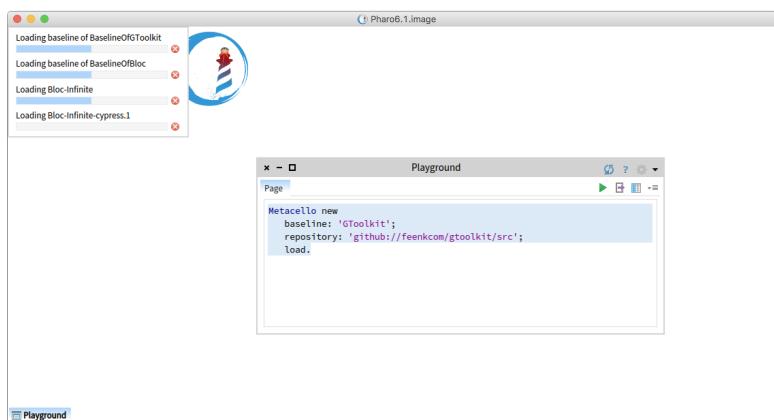


Figure 6.3: Installation process of GToolkit

6.3 Usage

This section shows how to create an instance of the editor for the source code, aGT Example and Pillar document. We also explain how adornments can be used manually.

Source code

In this part we build an editor for the `#fullDrawOn:` method implemented in the `Morph` class.

As a first step we create a text model with a *Rope* as a backend data structure. To do that we send the `#asRopedText` message to the source code of the chosen compiled method:

```
text := (Morph>>#fullDrawOn:) sourceCode asRopedText.
```

Since we build an editor for a Smalltalk code we should use an appropriate code styler, in this case `BrRBTextStyler`. To perform a correct syntax highlighting the styler needs to know the text context, in this case it is the `Morph` class in which the `#fullDrawOn:` method is implemented:

```
styler := BrRBTextStyler new.  
styler classOrMetaClass: Morph.
```

Once we have a text and a styler we can proceed to the instance creation of the editor model. We configure that model with the styler and the text that we instantiated during previous steps (note, the configuration order does not matter):

```
editor := BrTextEditor new.  
editor text: text.  
editor styler: styler.
```

After finishing with the model we can move to the UI part and create an editor element. We configure the layout constraints of the editor element to match a parent in both the horizontal and the vertical directions. It means that the editor will try to fill all available space of its parent element:

```
element := BrEditorElement new.  
element constraintsDo: [ :layoutConstraints |  
    layoutConstraints horizontal matchParent.  
    layoutConstraints vertical matchParent ].
```

However, sometimes users may want the editor to be large enough to fit all the text. To achieve this behaviour we should ask the layout constraints of the editor to fit its content:

```
element := BrEditorElement new.
element constraintsDo: [ :layoutConstraints |
    layoutConstraints horizontal fitContent.
    layoutConstraints vertical fitContent ].
```

Another option would be to create an editor of the exact size, for example 500pt width and 300pt height:

```
element := BrEditorElement new.
element constraintsDo: [ :layoutConstraints |
    layoutConstraints horizontal exact: 500.
    layoutConstraints vertical exact: 300 ].
```

Layout constraints can be configured independently, for example it is possible to match a parent horizontally and fit content vertically, or the other way around.

Last but not least, we should attach our editor model to the editor element:

```
element editor: editor.
```

When we put all snippets together and inspect the result in the Playground we should get an editor with syntax highlighting as shown in Figure 6.4.

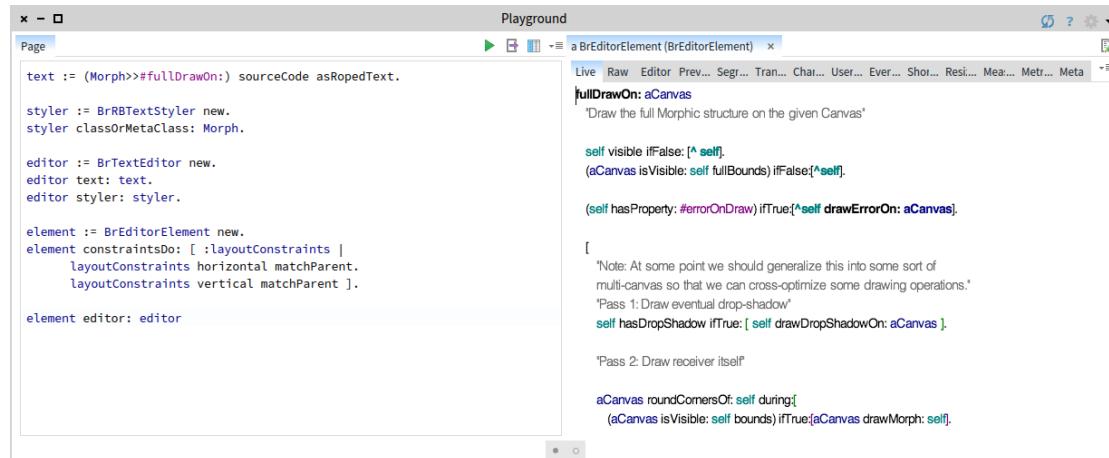


Figure 6.4: The Moldable Editor (on the right) opened on the source code of the `fullDrawOn:` method

GT-Example

Let's create an editor for a GT-Example method with an ability to expand its dependencies.

Most of the code will be the same, except for the *text* and the *styler*. Since we want to build an editor for a GT example, we should first create a text model from the source code of that example. We also choose to have a monospace font, therefore apply a font family attribute of generic *monospace* type and make it not overwritable by the styler:

```
text := (BrTextEditorExamples>>#elementOnMethodWithSyntaxHighlighting) sourceCode asRopedText.
text attributes: { BrFontGenericFamilyAttribute monospace beNotOverwritableByStyler }.
```

To make the editor know and understand dependencies of the example method we use a slightly modified syntax highlighter built for GT-Examples:

```
styler := GtExamplesStyler new.
styler classOrMetaClass: BrTextEditorExamples.
```

When composed and executed, users should get an editor capable of expanding dependencies of GT-Examples. Compared to the editor for source code, there are additional grey triangles along the text. They are visual elements with attached click event handlers that expand the source code of the dependent method as shown in Figure 6.5:

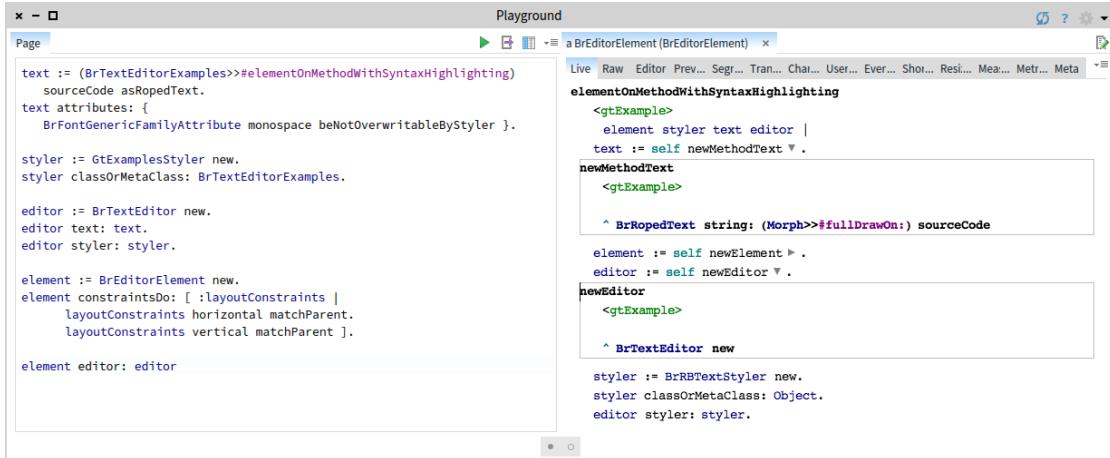


Figure 6.5: The Moldable Editor (on the right) opened on the source code of the GT-Example method with expanded dependencies

Pillar

In this section we build a live editor for Pillar documents.

Similar to the editor for GT-Examples we only make changes to the *text* and *styler*. Imagine we have a pillar file named `connector.pillar` in the same folder as a Pharo image. When inspected it appears to be a plain text file as shown in Figure 6.6. One can notice a reference to the example on the last line. In a default text editor that reference is displayed as a plain text. The goal is to build an editor that would render a result of the execution of the method referenced by our Pillar file right in-place.

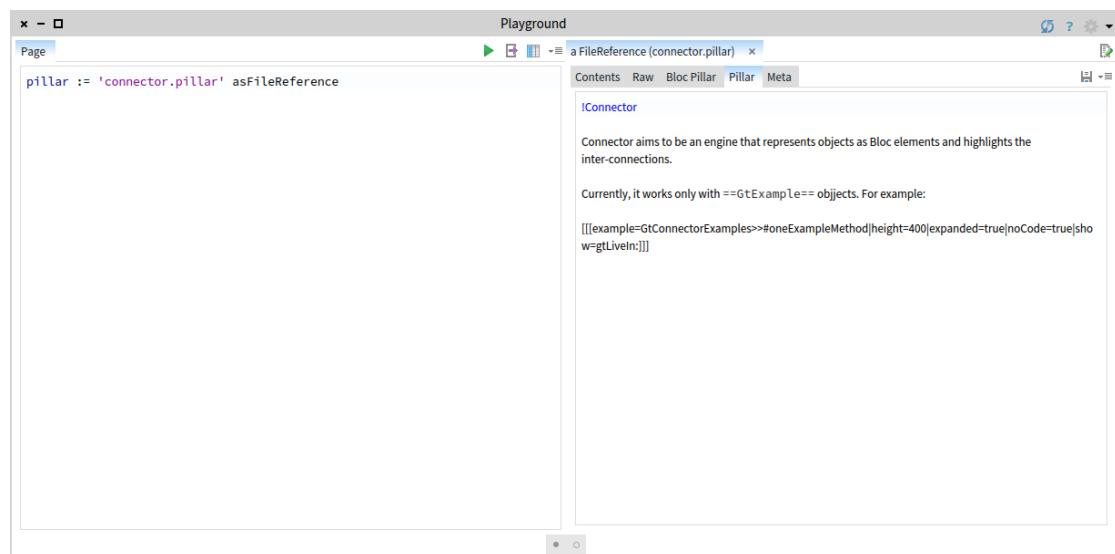


Figure 6.6: The content of a Pillar file opened in a traditional text editor

First of all we should create a file reference to our Pillar file:

```
pillar := 'connector.pillar' asFileReference.
```

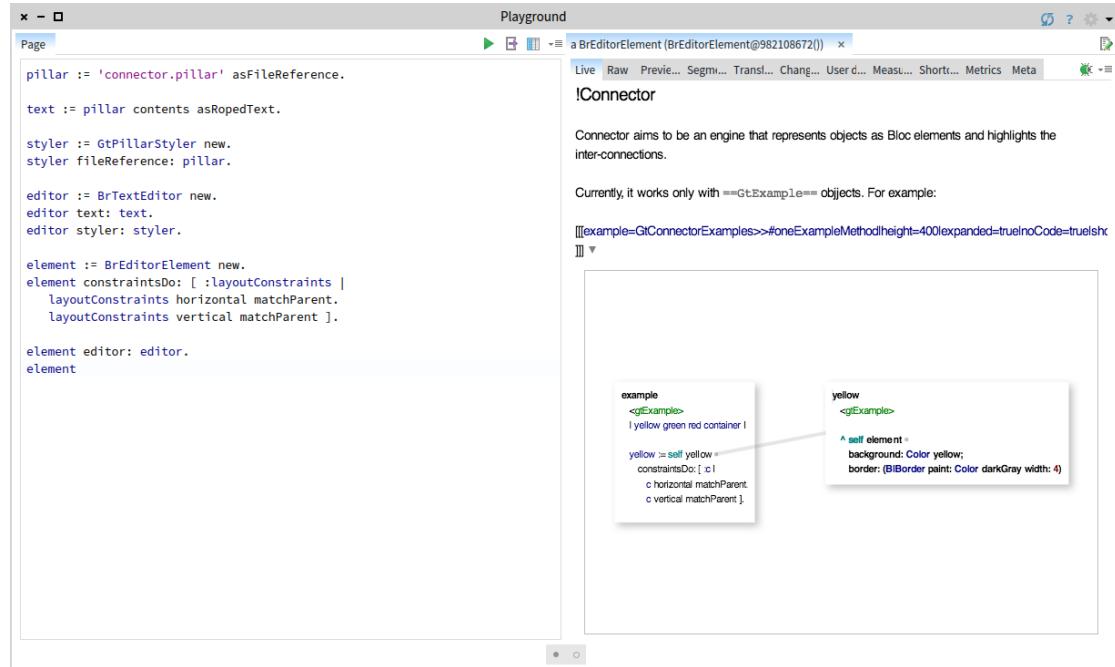
Next, we should read its content and create a roped text:

```
text := pillar contents asRopedText.
```

Finally, we have to instantiate a special styler that knows how to parse a Pillar file. We pass a reference to the original pillar file to help styler resolve possible image file references relative to the folder in which the Pillar file is located:

```
styler := GtPillarStyler new.  
styler fileReference: pillar.
```

The rest of the code is exactly the same as in the *Source code* section. Figure 6.7 shows the resulting editor opened on the `connector.pillar` :



```

Playground
Page a BrEditorElement (BrEditorElement@982108672())
Live Raw Previe... Segmu... Transf... Chang... User d... Measu... Short... Metrics Meta

!Connector
Connector aims to be an engine that represents objects as Bloc elements and highlights the inter-connections.

Currently, it works only with ==GtExample== objects. For example:
[[example=GtConnectorExamples>>#oneExampleMethodHeight=400expanded=truenoCode=trueish
]] ▾

example
<gtExample>
| yellow green red container |
yellow > self yellow +
constraintsDo: [ c |
c horizontal matchParent.
c vertical matchParent ].

yellow
<gtExample>
^ self element =
background: Color yellow;
border: (BIBorder paint: Color darkGray width: 4)

```

Figure 6.7: The content of a Pillar file opened in the Moldable Editor

Adornments

In this section we take a look at how *adornment* text attributes can be used to replace text with visual elements. The same attributes can also be used to append visual elements to a piece of text leaving it untouched.

`BrTextAdornmentDynamicAttribute` can be used to mix visual elements in the text. The following snippet creates an instance of a dynamic adornment that replaces a piece of text on which it is applied with a grey circle:

```
BrTextAdornmentDynamicAttribute new
beReplace;
elementBlock: [ (BIEllipse radius: 20) asElement ]
```

When applied on a `Hello world` piece of text of font size 30 on a character at index 8 (character `o`) we get a text as shown in Figure 6.8:

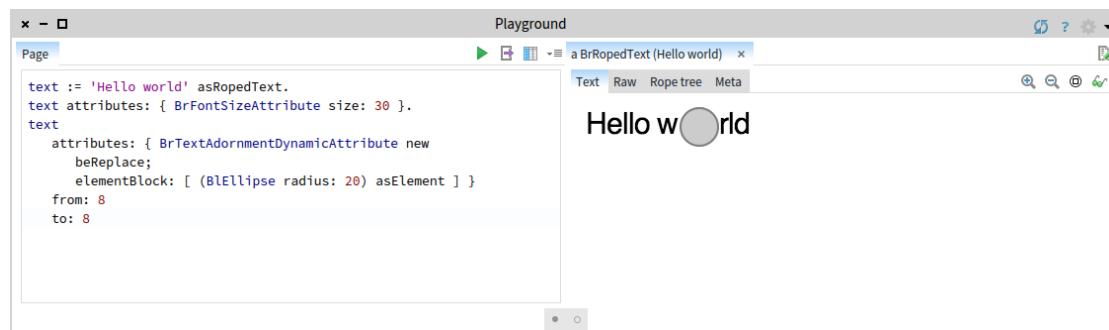


Figure 6.8: An example of adornment that replaces a piece of text with an ellipse element

The same attribute can be used to append a circle after a character at index 8 by sending `beAppend` instead of `beReplace` to the attribute. The difference can be clearly seen on the Figure 6.9:

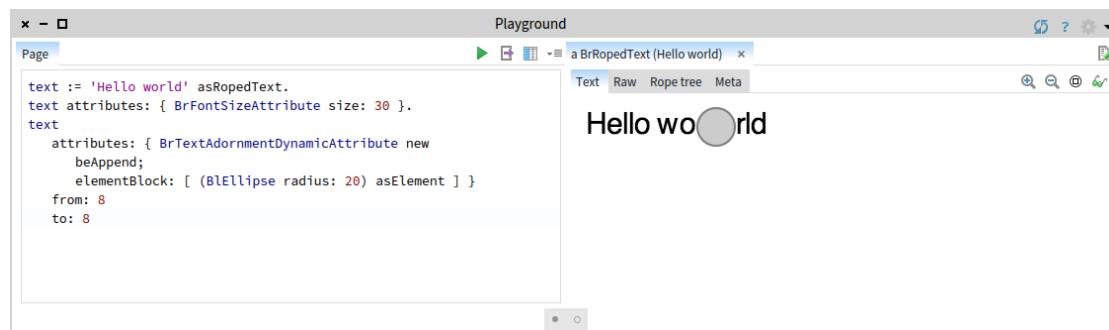


Figure 6.9: An example of adornment that appends an ellipse element after a character

Bibliography

- [1] Dimitar Asenov and Peter 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, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 9–12, 2014.
- [2] Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
- [3] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. Technical Report Report No. UCB/CSD-04-1368, University of California, Berkeley, December 1997.
- [4] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.
- [5] Andrei Chiş. *Moldable Tools*. PhD thesis, University of Bern, September 2016.
- [6] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Visual object-oriented programming. pages 67–93. Manning Publications Co., Greenwich, CT, USA, 1995.
- [7] Charles Crowley. Data structures for text sequences. *Computer Science Department, University of New Mexico, Date*, pages 1–29, 1998.
- [8] Inc. Free Software Foundation. The Buffer Gap — GNU Emacs Lisp Reference Manual. https://www.gnu.org/software/emacs/manual/html_node/elisp/Buffer-Gap.html#Buffer-Gap, April 2017.
- [9] GitHub Inc. The state of Atom’s performance. <http://blog.atom.io/2018/01/10/the-state-of-atoms-performance.html>, January 2018.

- [10] Dan Ingalls. Fabrik: A visual programming environment. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 176–190, November 1988.
- [11] Behdad Esfahbod Keith Packard, Carl Worth. Cairo Documentation — showGlyphs. <https://cairographics.org/manual/cairo-text.html#cairo-show-glyphs>.
- [12] Andrew J. Ko and Brad 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, CHI '06*, pages 387–396, New York, NY, USA, 2006. ACM.
- [13] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.
- [14] Sean McDermid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 53–62, New York, NY, USA, 2013. ACM.
- [15] F. Perez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering*, 9(3):21–29, May 2007.
- [16] Google Skia Inc. Skia Documentation — drawText. https://skia.org/user/api/SkCanvas_Reference#SkCanvas_drawText.
- [17] Bernard Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1(3):157 – 202, 1982.
- [18] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016*, pages 28–40, New York, NY, USA, 2016. ACM.