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

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”, but 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

3
The Moldable Editor

3.1 Overview

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 the 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 some character interval, or `SpanText`, that represents a uniform piece of text with every character having 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 "Text model" chapter.

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 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 ”Text styler” chapter 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. A segments update procedure will be discussed in the ”Segments” chapter.

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* in Android or *UITableView* 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. A structure of the visual part of the editor will be explained in more details in the ”Rendering” chapter.

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.

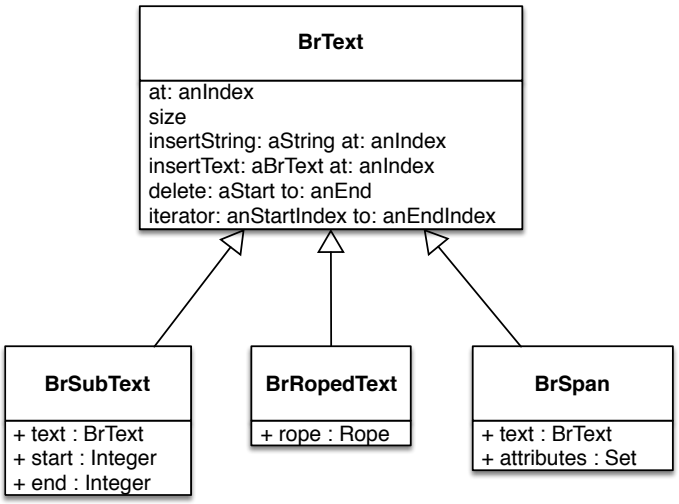

Figure 3.1: The UML class diagram of Text Model

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

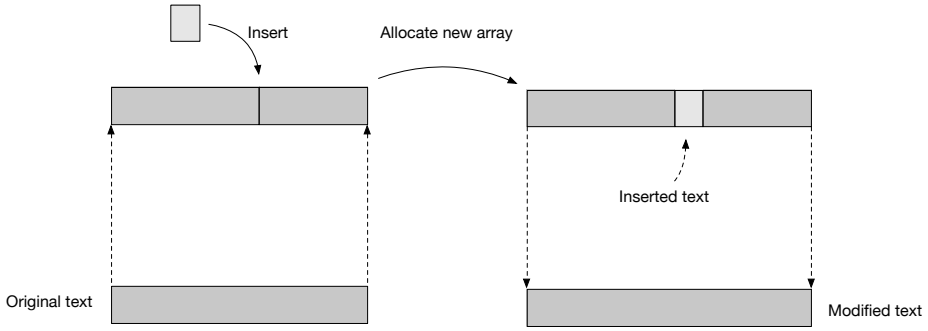

Figure 3.2: The array method

`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 3.2 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. [2]

TxText

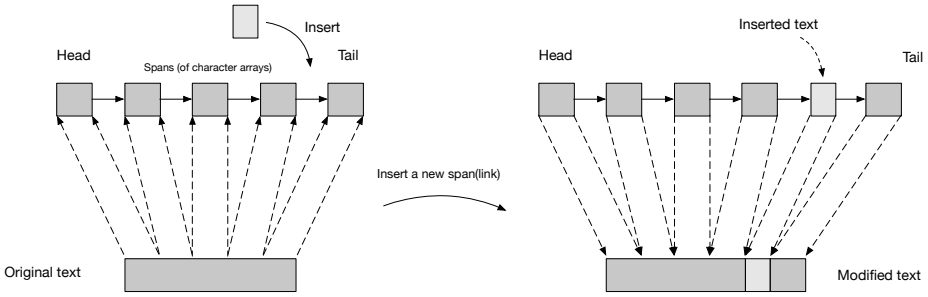

Figure 3.3: The linked list method

`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. [2]

3.3.2 Atom

Atom text editor uses a memory-efficient data structure similar to a Piece Table.[4] 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. [2]

3.3.3 Emacs

TODO

3.3.4 Rope

TODO

4
The Validation

In which you show how well the solution works.

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

This consists of additional documentation, e.g. a tutorial, user guide etc. Required by the Informatik regulation.

Bibliography

[1] Alexandre Bergel, Felipe Banados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Computer Languages, Systems & Structures*, 38(1):16–28, 2012.

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

[3] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

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

[5] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495. IEEE Press, 2012.