A WYSIWYG Framework

Master's Thesis

Lukas Bombach
538587

26th August 2015

Supervisors:
Prof. Dr. Debora Weber-Wulff
Prof. Dr. Barbara Kleinen

HTW Berlin
International Media and Computing (Master)

Abstract

Browsers do not offer native elements that allow for rich-text editing. There are third-party libraries that emulate these elements by utilizing the `contenteditable`-attribute. However, the API enabled by `contenteditable` is very limited and unstable. Bugs and unwanted behavior make it hard to use and can only be worked around, not fixed. By reviewing the API's history, it can be argued that its design has never been revisited only to ensure compatibility to current browsers. This thesis discusses the API's downsides and demonstrates that rich-text editing can be achieved without requiring the `contenteditable`-attribute with the library ”Type”, thus solving many problems of contemporary third-party rich-text editors.

Acknowledgements

I would like to extend my thanks to my two supervisors, Prof. Dr. Debora Weber-Wulff and Prof. Dr. Barbara Kleinen, for giving me the opportunity to work on a topic I have been passionate about for years.

I would like to thank Marijn Haverbeke for his work on CodeMirror, from which I could learn a lot.

I would like to thank my father for supporting me. Always.

Contents

Chapter 1

Introduction

1.1 Motivation

Written text is the most important cultural tool to pass on knowledge from one generation to another. Designing texts, layouting, adding images and choosing text formattings is an important means to convey a message and to clarify ideas. A newspaper written as a single stream of words would not have served the purpose, that it has for hundreds of years. With the computerization of printing, PCs have been the tool to generate and design texts for decades. There are software solutions for professional and personal use on Desktop PCs. In recent years, many desktop applications have been migrated to browser-based solutions. This has many advantages. Applications can be maintained in a centralized manner and any computer using the software will be updated automatically. Browser-based applications can be accessed from anywhere in the world, without requiring to install further applications. Contents can be shared and edited collaboratively with others.

Still, rich-text editing, i.e. editing text that uses formattings and layouting, cannot be implemented easily in a browser. Browsers offer APIs for rich-text editing, but these APIs are very limited in its functionality, inconsistent across different browsers and known to contain numerous bugs.

This makes it hard for web developers to create rich-text editors. Usually, a third-party editor must be used and customized, which does not necessarily fit the specific needs of a project. The limited features of the browsers' rich-text

APIs only allow for basic editors. A fully featured word-processing application like Google's document editor cannot be implemented with these APIs. For this reason Google omitted these APIs entirely. Unfortunately, there is no library and hardly any editor that implements rich-text editing without these APIs. Google did not publish their solution to the public domain.

The purpose of this thesis is to implement rich-text editing without using the browsers' rich-text editing APIs. This allows more features, a consistent behavior and avoids the bugs of these APIs. The implementation will be distributed as GUI-less software library with a high-level API, to enable web developers to implement rich-text editors specific to their needs, which is currently not possible.

1.2 Terminology

In web development, the term *WYSIWYG* editor is commonly used to describe text-editors that allow formatting. WYSIWYG is an abbreviation for **W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et and describes a text editor's capability to display formatted text as it is being edited. This stands out to plain-text editors that can neither display nor edit formattings. The term *rich-text editor* has often been used for this feature and stands in better contrast to *plain-text editor*. For this reason, the term *rich-text editor* and *rich-text editing* will be used in this thesis.

1.3 Structure

The first part of this thesis explains how rich-text editors are currently being implemented in browsers.

The second part discusses the problems with these approaches, possible alternatives as well as advantages and disadvantages of each approach.

Part three discusses techniques for an implementation of rich-text editing without rich-text editing APIs and part four discusses its implementation.

Part five gives an evaluation of this thesis.

Part I

Theory

Chapter 2

History of markup languages

2.1 Introduction

During the letterpress era, ”marking up” text has been the profession of adding formalized annotations to a text, that described the structure and formattings of the document. The annotated document was given to a typesetter to follow the instructions and use a movable type system to print the document accordingly.

2.2 History of markup languages

With the computerization of typesetting, so-called ”markup languages” have been invented that embedded the annotations in the text that was given to the typesetters. Coombs, Renear, and DeRose describe six types of markup languages: Punctuational, presentational, procedural, descriptive, referential and metamarkup [Coombs et al., 1987]. Punctuational markup solely refers to the use of punctuation to structure text, referential markup describes the ability of a markup language to refer to other documents and a metamarkup language can be used to describe other markup languages. Procedural markup includes commands for a computer program on how to render the text step by step. Presentational markup contains specific descriptions on the formatting of a text, describing particular parts as italicized, bold, indented etc. Descriptive markup describes the elements of a text as types. For instance, a part of a text can be marked to be a quote or a headline, but there would be no definition on how these elements should be displayed. A renderer can parse the document and present it with specific styles. Watson describes the difference between descriptive and presentational markup as generic and specific markup. Generic markup only describes the structure of a document while specific markup explicitly describes its styling [Watson, 1992].

The invention of generic markup is credited to William Tunnicliffe who proposed his ideas at a meeting at the Canadian Government Printing Office in 1967 [Goldfarb, 1990]. Markup that was given to typesetters still needed to be translated for the particular typesetting system that was used. This led to to higher costs and the need for a standard emerged [Watson, 1992]. In the late 60s, Stanley Rice, a book designer, proposed this idea of generic markup to the Graphic Communications Association (GCA), which formed the GCA GenCode committee to work on a standard for generic markup [Goldfarb, 1990]. This work has been authorized by the Organization for Szandardization (ISO) and the American National Standards Institute (ANSI). In 1969 Charles Goldfarb, Edward Mosher, and Raymond Lorie invented the Generalized Markup Language (GML) [Watson, 1992], a generic markup language for IBM. Goldfarb later maintained the cooperation of ISO, ANSI and the GCA GenCode committee and in 1985 drafted the proposal for the ”Standard Generalized Markup Language” (SGML), the first international standard for a generic markup language. SGML was based on the work of the GCA GenCode committee as well as the GML [Goldfarb, 1990] and published in 1986 as ISO 8879:1986 [ISO, 1986].

2.3 HyperText Markup Language

HyperText Markup Language (HTML) is an implementation of SGML [W3C, 1999, SGML and HTML]. One of the primary functions of web browsers is to display HTML formatted sources as visually formatted text. HTML uses tags to specify the contents of a document. Being an instance of SGML it mostly uses generic tags to define headlines, paragraphs or quotations inside a document, but also allows for specific tags, defining parts of the contents as italicized or bold.

Tags are strings inside the document’s text that itself are delimited by the ”<” and ”>” characters. *Listing 2.1* demonstrates the HTML required to render the following text:

In a hole in the ground there lived a **hobbit**

```
In a hole in the ground there lived a <strong>hobbit</strong>
```

Listing 2.1: Text formatted as bold with the ”strong” tag

The word ”hobbit” must be enclosed with a ”strong” start and end tag. Start and end tags are distinguished by adding a solidus to the end tag. HTML defines 127 tags to format document contents as well as to add metadata about the document itself [Mozilla, 2015f].

By the recommendation of the World Wide Web Consortium (W3C), browsers must represent a document marked up with HTML with the Document Object Model (DOM) [W3C, 1998], a tree structure containing every tagged element and its texts as nodes. The specification of the DOM defines an API to manipulate it and change the contents of a website dynamically.

Chapter 3

Text editing in browser environments

3.1 Overview

To develop a text editor in a browser, the DOM API must be used. Development is generally restricted to the components and APIs offered by the HTML5 standard as well as experimental features that are usually implemented in a subset of browsers. The boundaries of these restrictions can be overcome. It is common practice to combine native elements and APIs in ways they have not been designed for to enable features that are not natively offered. These techniques are often referred to as ”hacks” and, despite this terminology, are generally not regarded as a bad practice.

This chapter will discuss the basics of plain-text and rich-text editing in browsers as well as the APIs and techniques that browsers provide.

3.2 Plain-text editing

Text input components for browsers have been introduced with the specification of HTML 2.0 [Berners-Lee, 1995]. The components proposed include inputs for single line (written as `<input type="text" />`) and multiline texts (written as `<textarea></textarea>`). These inputs allow writing plain-text only.

Attribute	Type	Can be set to	Possible values
designMode	IDL attribute	Document	"on", "off"
contentEditable	IDL attribute	Specific HTMLElements	boolean, "true", "false", "inherit"
contenteditable	content attribute	Specific HTMLElements	empty string, "true", "false"

Table 3.1: Editing API attributes

3.3 Rich-text editing

Major browsers, i.e. any browser with a market share above 0.5%[1], do not offer native input fields that allow rich-text editing. Neither the W3C's HTML5 and HTML5.1 specifications nor the WHATWG's "HTML Living Standard"[2] recommend such elements. As discussed in **2.3: HyperText Markup Language**, by being able to display HTML, browsers are rich-text viewers. By the early 2000s, the first JavaScript libraries emerged, that allowed users to interactively change (parts of) a website to enable rich-text editing in the browser. The techniques used will be discussed in section 3.4 through section 3.5.

3.4 HTML Editing APIs

In July 2000, with the release of Internet Explorer 5.5, Microsoft introduced the IDL attributes[3] `contentEditable` and `designMode` along with the content attribute `contenteditable` [Microsoft, 2000a, Microsoft, 2000b]. These attributes were neither part of the W3C's HTML 4.01 specification [W3C, 1999] nor the ISO/IEC 15445:2000 [ISO, 2012], the defining standards of that time. Table 3.1 lists these attributes and possible values.

```
<div contenteditable="true">
  This text can be edited by the user.
</div>
```

Listing 3.1: An element set to editing mode

[1] `http://gs.statcounter.com/#all-browser-ww-monthly-201406-201506-bar`, last checked on 07/25/2015

[2] The Web Hypertext Application Technology Working Group (WHATWG) is a working group that mainly developed the HTML5 standard, which later resulted in the widely acknowledged "HTML Living Standard" see **5.3: Standardization of HTML Editing APIs**

[3] IDL attributes can only be set to DOM objects via JavaScript, whereas content attributes can be set to tags in the HTML source code [W3C, 2012].

Order	Parameter	Description
1	cmdID	The name of the command that will be executed
2	showUI	Determines if the browser will display a dialog if needed
2	value	A parameter that can be passed to the command invoked with the cmdID

Table 3.2: execCommand parameters

By setting `contenteditable` or `contentEditable` to ”true” or `designMode` to ”on”, Internet Explorer 5.5 switches the affected elements and their children to an editing mode. The `designMode`-attribute can only be applied to the entire document and the `contentEditable` and `contenteditable` attributes can be applied to specific HTML elements as described on Microsoft’s Developer Network (MSDN) online documentation [Microsoft, 2000b]. These elements include ”divs”, ”paragraphs” and the document’s ”body” element amongst others. Other than that, there is no difference in these attributes. In editing mode

1. Users can interactively click on and type inside texts
2. An API providing commands for editing text is enabled that can be accessed via JScript and JavaScript

When an element is switched to editing mode, the browser handles setting the caret if a user clicks inside the text, accepting keyboard input and modifying text nodes entirely by itself. No further scripting is necessary.

The API enabled by the editing mode must be called globally on the `document` object, but will only execute when the user’s selection or caret is contained within an element in editing mode. *Table .1* lists the full HTML editing API. To format text, the method `document.execCommand` must be used.

```
document.execCommand('italic', false, null);
```

Listing 3.2: Emphasizing text using the HTML editing API

Listing 3.2 demonstrates an example call of the ”italic” command. Calling this at any time on the `document` object, the browser will wrap the currently selected text (if inside an element in editing mode) with `<i>` tags. The method accepts three parameters.

The first parameter is the ”Command Identifier”, which determines which command to execute. This can be ”italic” to italicize the current selection or ”createLink” to create a link with the currently selected text as label.

```
document.execCommand('createLink', false, 'http://example.com
    /');
```

Listing 3.3: Creating a link using the HTML editing API

The *third* parameter will be passed on to the internal command[4] as a parameter. In the case of a `createLink` command, the third parameter is the URL to be used for the link to create. The *second* parameter determines if executing a command should display a user interface specific to the command. Using the `createLink` command with the second parameter set to `true` while not passing a third parameter, the user will be prompted with a system dialog to enter a URL. Most commands (command identifiers) `execCommand` accepts trigger text formatting. This includes commands to format text as bold, underlined, struck-through or as a headline. A full list of possible command identifiers can be found on MSDN [Microsoft, 2015a]. Apart from executing commands, the API enabled by the editing mode includes the functions `queryCommandEnabled`, `queryCommandIndeterm`, `queryCommandState`, `queryCommandSupported` and `queryCommandValue` which allow reading attributes related to the editing mode.

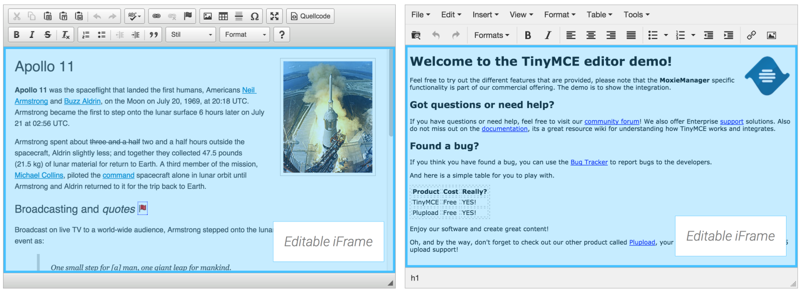
3.5 Usage of HTML Editing APIs for rich-text editors

Most web-based rich-text editors use HTML editing APIs as their basis. The popular editors ”CKEditor”[5] and ”TinyMCE”[6] dynamically create an `iframe` on instantiation and set its `body` to editing mode using the `contenteditable`-attribute. This way, users can type inside the `iframe` which acts as a text input field. Both libraries wrap the `iframe` in a user interface with buttons to format the `iframe`'s contents. Using the interface, the commands of

[4] The command invoked using the command identifier

[5] `http://ckeditor.com/`, last checked on 08/22/2015

[6] `http://www.tinymce.com/`, last checked on 08/22/2015



Apollo 11

Apollo 11 was the spaceflight that landed the first humans, Americans Neil Armstrong and Buzz Aldrin, on the Moon on July 20, 1969, at 20:18 UTC. Armstrong became the first to step onto the lunar surface 6 hours later on July 21 at 02:56 UTC.

Armstrong spent about ~~three and a half~~ two and a half hours outside the spacecraft, Aldrin slightly less; and together they collected 47.5 pounds (21.5 kg) of lunar material for return to Earth. A third member of the mission, Michael Collins, piloted the command spacecraft alone in lunar orbit until Armstrong and Aldrin returned to it for the trip back to Earth.

Broadcasting and *quotes*

Broadcast on live TV to a world-wide audience, Armstrong stepped onto the luna[r] event as:

One small step for [a] man, one giant leap for mankind.

Welcome to the TinyMCE editor demo!

Feel free to try out the different features that are provided, please note that the **MoxieManager** specific functionality is part of our commercial offering. The demo is to show the integration.

Got questions or need help?

If you have questions or need help, feel free to visit our community forum! We also offer Enterprise support solutions. Also do not miss out on the documentation, its a great resource wiki for understanding how TinyMCE works and integrates.

Found a bug?

If you think you have found a bug, you can use the Bug Tracker to report bugs to the developers.

And here is a simple table for you to play with.

Product	Cost	Really?
TinyMCE	Free	YES!
Plupload	Free	YES!

Enjoy our software and create great content!

Oh, and by the way, don't forget to check out our other product called Plupload, your upload support!

Figure 3.1: Usage of HTML editing APIs in CKEditor and TinyMCE

`document.execCommand` will be called on the `iframe`'s `document` and the selected text will be formatted. While using an `iframe` is still in practice, many newer editors use a `div` element instead. The user interfaces vary between different editors.

Usually, rich-text editors implemented this way wrap their editing capabilities (including `document.execCommand`) in an API to enrich functionality and provide higher-level concepts. As discussed in **Part II: Discussion**, using HTML editing APIs requires a lot of workarounds (for example to fix bugs of the APIs) which some editors account for in the implementation of their library. Rich-text editing libraries can be downloaded JavaScript files and included in a web project. To display an editor on a website, it is common to select a `textarea` element on the website, that the library will replace with the rich-text editor. To integrate the editor into web forms, most libraries will mirror their contents to the selected `textarea`, so they can be submitted to a server.

For years, the market of web-based rich-text editors has been dominated by ”CKEditor” and ”TinyMCE”. Both editors remain among the most popular choices. More recently, many new libraries have been published. Popular choices on GitHub, rated by the number of ”stars”, include ”MediumEditor”, ”wysihtml” and ”Summernote”. As Piotrek Koszuliński points out, most editors ”really doesn't[sic] work” [Koszuliński, 2013] for the reasons discussed in **6.3: Disadvantages of HTML Editing APIs**.

Part II

Discussion

Chapter 4

Overview

While HTML editing APIs are the recommended way by the W3C and the WHATWG for implementing rich-text editors on the web, their implementations across major web browsers are inconsistent, known to contain numerous bugs and have a limited and their functionality is limited and imprecise.

Understanding the origins and the history of rich-text editing on the web poses the question if the paradigms it is based on have been thoroughly reviewed and if alternative ways for an implementation, possibly using hacks, should be considered.

Chapter 5 will discuss the history and origins of HTML editing APIs. Chapter 6 will discuss its advantages and disadvantages and chapter 7 will discuss possible alternatives.

Chapter 5

History of HTML editing APIs

5.1 Browser support

As discussed in **3.4: HTML Editing APIs** HTML editing APIs have been introduced in July 2000 with the release of Internet Explorer 5.5 by Microsoft and have not been part of any standard of that time.

With the introduction of editing capabilities, Microsoft released a short documentation [Microsoft, 2000b], containing the attributes' possible values and element restrictions along with two code examples. Although a clear purpose has not been stated, the code examples demonstrated how to implement rich-text input fields with it. Mark Pilgrim, author of the "Dive into" book series and contributor to the the WHATWG, states that the API's first use case has been for rich-text editing[1].

In March 2003, the Mozilla Foundation introduced an implementation of Microsoft's `designMode`—named Midas—for their release of Mozilla 1.3. Mozilla published this as "rich-text editing support" on the Mozilla Developer Network (MDN) [Mozilla, 2003]. In June 2008, Mozilla added support for the `contentEditable` IDL and `contenteditable` content attributes in Firefox 3.

Mozilla's editing API closely resembles the API implemented for Internet Explorer, although, to this present day, there are still differences in the available command identifiers [Mozilla, 2015d, Microsoft, 2015b], as well as the markup generated by invoking commands [Mozilla, 2003].

[1] `https://blog.whatwg.org/the-road-to-html-5-contenteditable`, last checked on 07/10/2015

In June 2006, Opera Software released Opera 9[2], providing full support for `contentEditable` and `designMode`[3], followed by Apple in March 2008[4] providing full support in Safari 3.1[5]. MDN lists full support in Google Chrome since version 4 [Mozilla, 2015a], released in January 2010[6].

5.2 Emergence of HTML editing JavaScript libraries

Around 2003[7] the first JavaScript libraries emerged that made use of Microsoft's and Mozilla's editing mode to offer rich-text editing in the browser. Typically, these libraries were released as user interface components (text fields) with inherent rich-text functionality and were only partly customizable.

In May 2003 and March 2004 versions 1.0 of ”FCKEditor”[8] and ”TinyMCE” have been released as open source projects. These projects are still being maintained and remain among the most used rich-text editors. TinyMCE is the default editor for the content management system (CMS) Wordpress and CKEditor is listed as the most popular rich-text editor for the CMS Drupal[9].

Since the introduction of Microsoft's HTML editing APIs, a large number of editors have been implemented. While many have been abandoned, GitHub lists about 600 JavaScript projects related to rich-text editing[10]. However, it should be noted, that some projects are based on other projects' editors and some projects are stubs.

[2] `http://www.opera.com/docs/changelogs/windows/`, last checked on 07/10/2015

[3] `http://www.opera.com/docs/changelogs/windows/900/`, last checked on 07/10/2015

[4] `https://www.apple.com/pr/library/2008/03/18Apple-Releases-Safari-3-1.html`, last checked on 07/10/2015

[5] `http://caniuse.com/#feat=contenteditable`, last checked on 07/10/2015

[6] `http://googlechromereleases.blogspot.de/2010/01/stable-channel-update_25.html`, last checked on 07/10/2015

[7] compare figures .1 and .2

[8] Now distributed as ”CKEditor”

[9] `https://www.drupal.org/project/project_module`, last checked on 07/16/2015

[10] `https://github.com/search?o=desc&q=wysiwyg&s=stars&type=Repositories&utf8=%E2%9C%93`, last checked on 07/16/2015

5.3 Standardization of HTML Editing APIs

HTML editing APIs have been the *de facto* standard for implementing rich-text editors on the web, but have only been standardized in October 2014 with HTML5.

HTML5 introduces 13 new types of input fields [W3C, 2014]. It can be imagined that along with these elements, the standard could have introduced a native rich-text input element as well, but none of the elements comprises such capabilities. The WHATWG, the working group that mainly developed the HTML5 standard, discussed this issue publicly. The problems that have been faced with that idea are as follows:

1. Finding a way to tell the browser which language the rich-text input should generate. E.g. should it output BBCode[11], (X)HTML, Textile or something else?

2. How can browser support for a rich-text input be achieved?

Ian Hickson, editor of WHATWG and main author of the HTML5 specification, addresses these main issues in a message from November 2004[12]. He states

> *"Realistically, I just can't see something of this scoped[sic] [the ability to specify a language for a rich-text input and possibly to specify a subset of language elements allowed] getting implemented and shipped in the default install of browsers."*

and agrees with Ryan Johnson, a contributor to the standard, who states

> *"Anyway, I think that it might be quite a jump for manufacturers. I also see that a standard language would need to be decided upon just to describe the structure of the programming languages. Is it worth the time to come up with suggestions and examples of a programming language definition markup, or is my head in the clouds?"*

[11] A then popular markup language for bulletin boards

[12] `https://lists.w3.org/Archives/Public/public-whatwg-archive/2004Nov/0014.html`, last checked on 07/16/2015

Ian Hickson finally concludes

> *"Having considered all the suggestions, the only thing I could really see as being realistic would be to do something similar to (and ideally compatible with) IE's "contentEditable" and "designMode" attributes."*

Mark Pilgrim lists this as a milestone of the decision to integrate Microsoft's HTML editing APIs in the standard of the WHATWG.[13]. In cooperation with the W3C, the work by the WHATWG, including the standardization of the editing APIs, have been incorporated in the HTML5 standard. The cooperation between the WHATWG and the W3C ended in Juli 2012[14], which led the WHATWG to publish and maintain an own standard, the "HTML Living Standard" [WHATWG, 2015] that includes the same specifications on HTML editing APIs as HTML5.

[13] `https://blog.whatwg.org/the-road-to-html-5-contenteditable`, last checked on 07/16/2015

[14] `http://lists.w3.org/Archives/Public/public-whatwg-archive/2012Jul/0119.html`, last checked on 07/16/2015

Chapter 6

Advantages and disadvantages

6.1 Discussion

Understanding the history of the HTML editing APIs, the reasons for their wide browser support and their final standardization are questionable. It can be doubted if they fit their purpose specifically well. In fact, all major browsers mimicked the API as implemented in Internet Explorer 5.5, even though there was no specification for it. The reasons for this have not been publicly discussed. A reason may have been to be able to compete with other browsers. Both, Microsoft's original implementation as well as Mozilla's adoption have been released in the main years of the so-called "browser wars"[1]. Mozilla adopted Microsoft's API applying practically no change to it. It can be argued that this has been part of the struggle for market shares while competing with Microsoft's Internet Explorer. At this time, it was indispensable for any browser to be compatible with as many websites as possible. A great number of websites have only been optimized for a specific browser. To gain market share, it was essential to support methods that other browsers already offered and that have been used by web developers. Being able to display websites just as good as their competitor may have been a key factor for Mozilla's decision to implement Microsoft's HTML editing APIs and not alter them in any way. Creating another standard would have been a disadvantage over the then

[1] The "browser wars" was competition for market shares between Internet Explorer and Netscape Navigator during the late 1990s. Mozilla, Chrome, Safari and Opera participated as they were released in the early 2000s.

stronger Internet Explorer in getting users to choose Mozilla.

As discussed in section **5.1: Browser support**, other now popular browsers, i.e. Chrome, Safari and Opera, implemented these APIs only years later, when JavaScript libraries based on them had already been popular and widely used, which can be seen as a reason for this decision. As described in section **5.3: Standardization of HTML Editing APIs**, it has clearly been stated, that the reason for standardizing these APIs for rich-text editing has been to ensure browser support.

The API itself stems from a time when the usage of the web was different from today. JavaScript has only been standardized 3 years in advance to the publication of the HTML editing APIs. The use cases and products build with this technology are now far more complex and elaborate than of this stage of the internet. The requirements of blogging platforms or products like Google's document editor were yet unknown.

The API itself and especially its implementations across various browsers has been criticized by Google [Harris, 2010], Medium [Santos, 2014], CKSource [Koszuliński, 2013][2] and others. It has led websites to exclude users from editing in certain browsers entirely[3]. Sections 6.2 through 6.4 discuss the advantages and disadvantages, as well as practices for treating the disadvantages of HTML editing APIs.

6.2 Advantages of HTML Editing APIs

Browser support

A fair reason for using HTML editing APIs is their wide browser support. Caniuse.com lists that 92.78% of all web users use a browser that fully supports HTML editing APIs[4].

[2] The creators of CKEditor

[3] `https://medium.com/medium-eng/the-bug-that-blocked-the-browser-e28b64a3c0cc`, last checked on 08/19/2015

[4] `http://caniuse.com/#search=contenteditable`, last checked on 07/17/2015

High-level API

HTML editing APIs offer high-level commands for formatting text. It needs little setup to implement a basic editor, the browser takes care of generating the required markup.

HTML output

HTML editing APIs modify and generate HTML. In the context of web development, user input in this format is likely to be useful for further processing.

Possible third-party solutions for other languages

While HTML editing APIs can be used to generate HTML only, its design offers a way for third-party libraries to build on top of that and implement editors that write BBCode (for instance) and use HTML only for displaying it as rich-text. A dedicated rich-text input might not offer this flexibility.

6.3 Disadvantages of HTML Editing APIs

No specification on the generated output

The specifications on the HTML editing APIs do not state what markup should be generated by specific commands. There are vast differences in the implementations of all major browsers. Calling the `italic` command Internet Explorer, Firefox and Chrome all generate different markup.

```
<i>Lorem ipsum</i>
```

Listing 6.1: Markup of italic command in Internet Explorer

```
<span style="font-style: italic;">Lorem ipsum</span>
```

Listing 6.2: Markup of italic command in Firefox

```
<em>Lorem ipsum</em>
```

Listing 6.3: Markup of italic command in Chrome

This is a *major* problem for web development, because it makes processing input very difficult. For a content management system or a blogging platform, it can be very hard to handle the input of users only because different browsers are being used. Given the number of possible edge cases, it is very intricate to normalize the input.

Apart from that, Internet Explorer's output is semantically incorrect for most use cases[5], while Firefox's output is breaking semantics entirely and is considered a bad style in terms of the separation of concerns of HTML and CSS[6].

Furthermore, different browsers will not only generate different markup when executing commands: when a user enters a line break (by pressing enter), Firefox will insert a `<br>` tag, Chrome and Safari will insert a `<div>` tag and Internet Explorer will insert a `<p>` tag. Most features of the HTML editing APIs that generate markup show different implementations across different browsers.

Flawed API

The original and mostly unaltered API is limited and not very effective. MDN lists 44 commands available for their `execCommand` implementation [Mozilla, 2015c]. While other browsers do not match these commands precisely, their command lists are largely similar. 23 of these commands format the text (i.e. to italicize or make text bold) by enclosing the current selection with tags like `<em>` or `<strong>`. The only difference between these commands is which tag will be used. At the same time there is no command to wrap the selected text in an arbitrary tag, for example to apply a custom class to a text[7]. All 23 commands could be summarized by a single command, that allows to pass custom tags or markup, that the selected text will be wrapped with. This applies to inserting elements as well. 7 commands insert different kinds of

[5] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/i#Notes`, last checked on 07/17/2015

[6] `https://en.wikipedia.org/wiki/Separation_of_concerns#HTML.2C_CSS.2C_JavaScript`, last checked on 07/17/2015

[7] For example to apply the class ”highlight” in the following manner: `<span class="highlight">Lorem ipsum</span>`

HTML elements, this could be simplified and extended by allowing to insert any kind of (valid) markup with a single command.

Both alternatives would also give developers more control of what to insert. As previously discussed, browsers handle formatting differently. Allowing to format with specific HTML would generate consistent markup (in the scope of a website) and would allow developers generate the markup that fits their needs.

Restrictions

Google points out that implementing an editor using HTML editing APIs comes with the restriction that such an editor can only offer the least common denominator of functions supported by all browsers. They argue, if one browser does not support a specific feature or its implementation is buggy, it cannot be supported by the editor[8]. This is mostly true, although it is to be noted, that editors like CKEditor show, that some bugs can be worked around as well as some functionality be added through JavaScript. These workarounds still have limitations and not everything can be fixed. In particular there can be cases where the editing mode is not able to handle content inserted or altered by workarounds, thus limiting the features of an editor. Google names layouting the editor's contents with tab stops as one example.

Clipboard

When dealing with user input, usually some sort of filtering is required. It is possibly harmful to accept any kind of input. This must be checked on the server side since attackers can send any data, regardless of the front end a system offers. However, in a cleanly designed system, the designated front end should not accept and send "bad" data to the back end. This applies to harmful content as well as to content that is simply *unwanted*. For example, for aesthetic reasons, a comment form can be designed to allow bold and italic font formatting, but not headlines or colored text.

[8] `http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html`, last checked on 07/18/2015

Implementing a rich-text editor with HTML editing APIs, unwanted formatting can be prevented simply by not offering input controls for these formattings (assuming no malicious behavior by the user). However contents can be pasted from the clipboard that contain any kind of formatting into elements in editing mode. HTML editing APIs provide no way to define or apply filtering to the formattings of pasted contents.

Recent versions of major browsers allow observing paste events. Chrome, Safari, Firefox and Opera grant full read access to the clipboard contents from paste events. In these browsers, the event can be stopped and its contents can be processed. Internet Explorer grants access to plain-text and URL contents only. Android Browser, Chrome for Android and IOS Safari allow reading the clipboard contents on paste events as well. Other browsers and some older versions of desktop and mobile browsers do not support clipboard access or listening to paste events. Overall, 82.78% of internet users support listening to and reading from clipboard events[9].

When dealing with the clipboard, especially older browsers show an unexpected behavior. Older WebKit-based browsers insert so-called ”Apple style spans”[10] on copy and paste commands. ”Apple style spans” are pieces of markup that have no visible representation, but clutter up the underlying contents of an editor. When pasting formatted text from Microsoft Word, Internet Explorer inserts underlying XML, that Word uses to control its document flow, into the contents of the editor.

Bugs

HTML editing APIs are prone to numerous bugs. Especially older browser versions are problematic. Piotrek Koszuliński states:

> *”Don’t write wysiwyg editor[sic] - use one that exists. It’s going to consume all your time and still your editor will be buggy. We [...] are working on this for years and we still have full bugs lists [Koszuliński, 2012]”*

[9] `http://caniuse.com/#feat=clipboard`, last checked on 07/18/2015

[10] `https://www.webkit.org/blog/1737/apple-style-span-is-gone/`, last checked on 07/18/2015

Mozilla lists 1060 active issues related to its ”Editor” component[11]. Google lists 420 active issues related to ”Cr-Blink-Editing”[12]. The WebKit project lists 641 active issues related to ”HTML Editing”[13]. Microsoft and Opera Software do not allow public access to their bug trackers. As quoted above, some rich-text editors like CKEditor have been developed for over 10 years and still need to fix bugs related to the editing API[14] [Koszuliński, 2012]. Some bugs have caused big websites to block particular browsers entirely[15].

Given the argument that editing APIs provide easy to use and high-level methods to format text, in practice, the number of bugs and workarounds required, renders a ”quick and easy” implementation impossible. Most importantly, browser bugs cannot be fixed by web developers. At best they can be worked around, enforcing particular software design on developers, possibly spawning more bugs and making the development dependent of the development of browsers and user adoption.

6.4 Treating HTML editing API related issues

Since the issues arising with HTML editing APIs are part of the browser's implementation, they cannot be fixed by JavaScript developers. The common approach for most rich-text editors is to use HTML editing APIs and wrap it in a library while using workarounds for its issues and bugs internally. It is to be noted, as Piotrek Koszuliński points out, that the majority of rich-text editors ”really do not work” [Koszuliński, 2013]. This is usually the case when the problems discussed in **6.3: Disadvantages of HTML Editing APIs** have not been addressed and the library solely consists of a user interface wrapping an element in editing mode.

[11] https://bugzilla.mozilla.org/buglist.cgi?bug_status=__open__&component=Editor&product=Core&query_format=advanced&order=bug_status%2Cpriority%2Cassigned_to%2Cbug_id&limit=0, last checked on 07/18/2015

[12] https://code.google.com/p/chromium/issues/list?q=label:Cr-Blink-Editing, last checked on 07/18/2015

[13] https://bugs.webkit.org/buglist.cgi?query_format=advanced&bug_status=UNCONFIRMED&bug_status=NEW&bug_status=ASSIGNED&bug_status=REOPENED&component=HTML%20Editing, last checked on 07/18/2015

[14] http://dev.ckeditor.com/report/2, last checked on 07/18/2015

[15] https://medium.com/medium-eng/the-bug-that-blocked-the-browser-e28b64a3c0cc, last checked on 07/18/2015

Having to account for multiple browser implementations, working around bugs can result in a big file size and a complex architecture. Most edge cases can only be learned from experience, not be foreseen or analyzed by debugging source code.

In practice, there are a few attempts to implement pure wrappers that will take care of the beforementioned issues, to support other developers with a working api. This approach is generally not well-adopted though. In general, most libraries are distributed as independent editors implementing their own solutions for addressing these issues—or are forks of other editors implemented with a different user interface.

Subsections **6.4: HTML output** through **6.4: Restrictions** will discuss some approaches to treat the beforementioned issues.

HTML output

Editors like CKEditor offer some configuration on the generated HTML output[16], but in the case of CKEditor this is very limited. The underlying issue is that HTML editing APIs cannot be configured. The only way to work around this issue is to implement custom methods to apply formatting in JavaScript and not using the `execCommand` interface[17]. The proprietary ”Redactor Text Editor” demonstrates such an implementation.

Medium.com takes a different approach and implements an extensive framework that will compare the markup of the editor with a model of the visual representation that the markup generates and corrects the DOM on each change[18] to conform a defined norm.

Flawed API

HTML editing APIs are usually wrapped in the API of an editor, that offers more functionality than the original API. `execCommand` offers the `insertHTML`

[16] http://docs.ckeditor.com/#!/guide/dev_output_format-section-adjusting-output-formatting-through-configuration, last checked on 08/19/2015

[17] HTML editing APIs can still be used for text input and other functionality

[18] `https://medium.com/medium-eng/why-contenteditable-is-terrible-122d8a40e480`, last checked 08/19/2015

command that allows inserting custom elements. As discussed in the previous paragraph, extending the formatting capabilities requires a JavaScript implementation.

Clipboard

For browsers, that do not offer native support to control and process the contents pasted from the clipboard, workarounds must be used. There are two approaches to this

1. Sanitize the editor's contents after a paste event.

2. Proxy a paste event to insert its contents into another element and read the contents from it.

The "Redactor Text Editor" uses the first approach. While reading contents from the a paste event is not fully supported, the event itself will be triggered by all major browsers, even most older versions[19]. Once the event has finished and the contents have been inserted to the editor, a "cleaned up" procedure can remove unwanted contents.

CKEditor and TinyMCE have been developed before most major browsers supported clipboard events. Both editors implement a technique to permit pasting formatted text, that has been the standard for many years. CKEditor and TinyMCE create a hidden `textarea` element and listen for common "paste" keyboard shortcuts (ctrl v and shift ins). When a user presses these keys, the hidden `textarea` will be focused and thereby be the target in which the browser will paste the clipboard's contents. After a short delay, the editors can read the `textarea`'s contents. Since `textarea` elements allow plain text only, the contents will be removed of any formatting and can then be inserted to the editor. However, this does not account for pasting from the context menu. For this CKEditor overrides the native context menu with a custom menu containing a custom "paste" menu item, that will open a modal instructing the user to paste his or her contents using the keyboard shortcuts. TinyMCE

[19] That have a market share BETTER CHECK THIS AGAIN

overrides the native context menu too, but does not display a paste option. Up to the current versions CKEditor[20] and TinyMCE [21], this is still the case.

CodeMirror[22], a web-based source code editor enhances this approach by moving the textarea to the cursor's position when the user presses his or her right mouse button. This way a native context menu can be displayed while the paste option would insert the clipboard's contents to a designated `textarea`, that can be read from.

On the downside, the paste event cannot be proxied to the `textarea` if the user uses the browser's menu bar to paste contents.

Bugs

Generally, bugs cannot be fixed. The only way to treat bugs in browsers is by avoiding them, shimming them with JavaScript methods or "cleaning up" after they have occurred.

Restrictions

The restrictions the HTML editing API imposed on the contents of the editor is an even bigger problem. Taking the example of layouting with tab stops[23], the only solution is not making the entire contents of the editor editable, but implementing a layouting engine in JavaScript and enabling the editing mode only on parts of the layout.

[20] CKEditor 4.5.1

[21] TinyMCE 4.2.3

[22] `https://codemirror.net/`, last checked on 08/22/2015

[23] `http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html`, last checked 08/19/2015

Chapter 7

Rich-text editing without editing APIs

7.1 Alternatives to HTML editing APIs

Overview

HTML editing APIs are the recommended way for implementing a web-based rich-text editor. This section will discuss possible alternatives to editing rich-text.

Native input elements

Native text inputs are hard-wired to plain-text editing. No major browser offers an API for formatting. There is also no option to write HTML to an input and have it display it as rich-text. `input` fields and `textarea` elements will simply display the HTML as source code. Rich-text can only be implemented as an editable part of the website.

Image elements

In February 2015, Flipboard Inc. demonstrated an unprecedented technique to achieve fluid full-screen animations with 60 frames per second on their mobile website[1]. Instead of using the DOM to display their contents, the entire website was rendered to a `canvas` element. When a user swiped over the website the canvas element was re-rendered, essentially imitating the browser's rendering engine. `canvas` elements allow rendering rich-text too. A rich-text editor can be implemented using this technique. This however has two major downsides. On the one hand it would require implementing a text-layouting engine. The `canvas` API is not capable of layouting text. On the other hand, making the editor accessible to other developers would be much more complex since the text only exists in an internal representation inside the editor and would not be exposed as DOM component to other developers.

An approach related to rendering the text on a `canvas` element is to render the text inside a Scalable Vector Graphic (SVG). In contrast to `canvas` elements, SVGs contain DOM nodes that can be accessed from the outside. However this has no benefit over using HTML DOM nodes with the downside that SVG too has no native implementation for controlling the text layout.

Furthermore, while both alternatives can display rich-text, neither provides an dedicated API to manipulate rich-text, which gives neither alternative an advantage over using regular DOM structures to display rich-text.

Third-party plugins

Another way to display and edit rich-text inside a browser is through third-party plugins like Adobe Flash or Microsoft Silverlight. Flash and Silverlight lack mobile adoptions and have been subject to critique since the introduction of smartphones and HTML5. Other third-party plugins are even less well adopted. This makes Flash, Silverlight and other third-party browser-plugins a worse choice as compared to displaying and manipulating rich-text though the DOM.

Manipulation via the DOM APIs

The only way to natively display rich-text on a website is through the Document Object Model (DOM). Editors based on HTML editing APIs utilize the

[1] `http://engineering.flipboard.com/2015/02/mobile-web/`, last checked on 07/24/2015

DOM to display their rich-text contents too. Only the editing (of the DOM), commonly phrased ”DOM manipulation”, is implemented with HTML editing APIs.

Manipulating the DOM has been possible since the first implementations of JavaScript and JScript. It has been standardized in 1998 with the W3C’s ”Document Object Model (Core) Level 1” specification as part of the ”Document Object Model (DOM) Level 1 Specification” [W3C, 1998].

Other than for rich-text editing, the DOM and its API is the recommended[2] way to change a website’s contents and—apart from HTML editing APIs—the only option *natively* implemented in any major browser. Popular libraries like jQuery, React or AngularJS are based on it. The API has been developed for 17 years and proven to be stable across browsers.

MDN lists 44 commands for the `execCommand` interface [Mozilla, 2015c].

- 23 commands apply text formatting.
- 6 commands insert HTML elements.
- 2 commands remove contents.
- 2 commands remove formatting.
- The other commands enable control over the clipboard, implement undo/redo commands, set settings for the editing mode and one command can select all text of the editable element.

Algorithms 1 through 4 demonstrate alternatives to commands of the `execCommand` interface related to text formatting, insertion and deletion implemented with methods of the ”Document Object Model (Core) Level 1” specification.

Algorithm 1 demonstrates a simplified procedure to wrap a text selection in a tag. To implement the `bold` command of `execCommand`, this procedure can be implemented using the `strong` tag. The text selection can be read with the browser’s selection API [Mozilla, 2015g][3].

[2] recommended by the W3C and WHATWG

[3] Internet Explorer prior version 9 uses a non-standard API [Microsoft, 2015c]

Algorithm 1 Simplified text formatting pseudocode

```
procedure FORMAT
    $s \leftarrow$ split text node at beginning of text
    $e \leftarrow$ split text node at end of text
    $t \leftarrow$ new tag before $s$
    for all $n$ in selection do                ▷ n is a node in the selection
        Move $n$ to $t$
    end for
end procedure
```

Algorithm 2 Simplified element insertion pseudocode

```
procedure INSERT
    if Selection is not collapsed then
        $s \leftarrow$ split text node at beginning of text
        $e \leftarrow$ split text node at end of text
        for all $n$ in selection do            ▷ n is a node in the selection
            Remove $n$
        end for
        Collapse selection
    end if
    Insert new tag at beginning of selection
end procedure
```

Algorithm 2 demonstrates a simplified procedure to insert a new tag and possibly overwrite the current text selection and thereby mimicking `execCommand`'s insertion commands.

Algorithm 3 Simplified text removal pseudocode

```
procedure REMOVE
    if Selection is not collapsed then
        $s \leftarrow$ split text node at beginning of text
        $e \leftarrow$ split text node at end of text
        for all $n$ in selection do            ▷ n is a node in the selection
            Remove $n$
        end for
        Collapse selection
    else
        Remove one character left of the beginning of the selection
    end if
end procedure
```

Algorithm 3 demonstrates a procedure to mimic the deletion commands of `execCommand`.

Algorithm 4 Simplified element unwrapping pseudocode

```
procedure UNWRAP($e$)                         ▷ e is an element
    for all $n$ in $e$ do                     ▷ n is a node in e
        Move $n$ before $e$
    end for
end procedure
Remove element
```

Algorithm 4 demonstrates a procedure to unwrap an element, mimicking the commands of `execCommand` to remove formatting.

With formatting and removing text as well as inserting and unwrapping elements, we can find equivalents for all commands of the HTML editing APIs related to manipulating rich-text using only methods specified by the ”Document Object Model (Core) Level 1”. This shows, that HTML editing APIs are not a necessity for rich-text editing. Chapter **13: Implementation** demonstrates ways to implement clipboard, undo/redo and selection capabilities.

7.2 Rich-text without HTML editing APIs in practice

Google completely rewrote their document editor in 2010 abandoning HTML editing APIs entirely. In a blog post[4], they stated some of the reasons discussed in section **6.3: Disadvantages of HTML Editing APIs**. They state, using the editing mode, if a browser has a bug in a particular function, Google won't be able to fix it. In the end, they could only implement ”least common denominator of features”. Furthermore, abandoning HTML editing APIs enables features otherwise impossible, for example tab stops for layouting [Harris, 2010]. With the Google document editor, Google demonstrates it is possible to implement a fully featured rich-text editor using only JavaScript without HTML editing APIs.

[4] `http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html`, last checked on 07/18/2015

Google's document editor is proprietary software and its implementation has not been documented publicly. Most rich-text editors still rely on HTML editing APIs. The editor "Firepad"[5] is another exception. It is based on "CodeMirror"[6] and extends it with rich-text formatting. The major disadvantage of Firepad is its origin as a source code editor. It generates "messy" (non-semantic) markup with lots of control tags. It has a sparse API that is not designed for rich-text editing and has no public methods to format the text. It is to be noted that Google's document editor generates lots of control tags as well, but it is only used within Google's portfolio of office apps where it may not be necessary to create *well-formatted*, semantic markup. A list of rich-text editors using and not using HTML editing APIs can be found in **Figure .1** and **Figure .2**.

7.3 Advantages of rich-text editing without editing APIs

With a pure JavaScript implementation, many of the problems that HTML editing APIs have, can be solved. The issues discussed in **6.3: Disadvantages of HTML Editing APIs** will be addressed hereinafter.

Generated output and flawed API

The generated markup, if implemented through JavaScript and DOM Level 1 methods, can be chosen with the implementation of the editor. Furthermore, the decision of the generated output can be given to the developers working with the editor. Section **6.3: Disadvantages of HTML Editing APIs** describes the inconsistent output across various browsers as well as the restrictions of the API design of `execCommand`. Both issues can be addressed by offering a method to wrap the current selection in arbitrary markup. jQuery's `htmlString` implementation[7] demonstrates a simple and stable way to define markup as a string and pass it as an argument to JavaScript methods. A sample call could read as follows.

[5] `http://www.firepad.io/`, last checked 07/23/2015

[6] A web-based source code editor

[7] `http://api.jquery.com/Types/#htmlString`, last checked on 07/19/2015

```
// Mimicking document.execCommand('italic', false, null);
editor.format('<em />');

// Added functionality
editor.format('<span class="highlight" />');
```

Listing 7.1: Example calls to format text

This will allow developers to choose which markup should be generated for italicizing text. The markup will be consistent in the scope of their project. Since the DOM manipulation is implemented in JavaScript and not by high-level browser methods, this will also ensure the same output across all systems and solve cross-browser issues. The second example function call in listing 7.1 demonstrates that custom formatting, fitting the needs of a specific project, can be achieved with the same API, giving developers a wider functionality.

7.4: Native components discusses the disadvantage, that when not using HTML editing APIs, native components like the caret or the text input must be implemented with JavaScript as they are not provided without using HTML editing APIs. On the flip side, this allows full control over these components that can be exposed via an API to other developers.

Restrictions

When implementing an editor in pure JavaScript, the limitations imposed by the HTML editing APIs, do not apply. Anything that can be implemented in a browser environment can also be implemented as part of a rich-text editor. The Google document editor demonstrates rich functionality that would not be possible with an implementation based on HTML editing APIs.

Clipboard

Without a native text input or an element switched to editing mode with HTML editing APIs, clipboard functionality is not available. Users cannot paste contents from the clipboard unless one of these elements is focused. However chapter **13: Implementation** demonstrates a way that not only allows clipboard support, but also grants full control over the pasted contents.

Bugs

By refraining from using HTML editing APIs, all of its numerous bugs will be avoided. An implementation can be aimed to minimize interaction with browser APIs, especially unstable or experimental interfaces. DOM manipulation APIs have been standardized for more than 15 years and tend to be well-proven and stable. Bugs that occur will mostly be part of the library and can be fixed and not only worked around. Bug fixes can be rolled out to users when they are fixed. This will free development from being dependent on browser development, update cycles and user adoption.

7.4 Disadvantages of rich-text editing without editing APIs

Formatting

The HTML editing APIs' formatting methods take away a crucial part of rich-text editing. Especially on the web, where a text may come from various sources, formatting must account for many edge cases. Nick Santos, author of Medium's rich-text editor states:

> *"Our editor should be a good citizen in [the ecosystem of rich-text editors]. That means we ought to produce HTML that's easy to read and understand. And on the flip side, we need to be aware that our editor has to deal with pasted content that can't possibly be created in our editor. [Santos, 2014]"*

An editor implemented *without* HTML editing APIs does not only need to account for content (HTML) that will be pasted into the editor[8] (in fact, content can be sanitized before it gets inserted in the editor, see **13.8: Pasting**), but also for content that will be loaded on instantiation. It cannot be assumed that the content that the editor will be loaded with (for example integrated in a CMS), is *well-formatted* markup or even valid markup. "Well-formatted" means, the markup of a text is *simple* in the sense that it expresses semantics

[8] Medium uses HTML editing APIs

with as few tags as possible (and it conforms the standards of the W3C). In HTML, the same visual representation of a text, can have many different—and valid—underlying DOM representations. Nick Santos gives the example of the following text [Santos, 2014]:

> The hobbit was a very well-to-do hobbit, and his name was ***Baggins***.

The word "Baggins" can be written in any of the following forms:

```
<strong><em>Baggins</em></strong>
<em><strong>Baggins</strong></em>
<em><strong>Bagg</strong><strong>ins</strong></em>
<em><strong>Bagg</strong></em><strong><em>ins</em></strong>
```

Listing 7.2: Different DOM representations of an equally formatted text

A rich-text editor must be able to edit any of these representations (and more). Furthermore, the same edit operation, performed on any of these representations must provide the same *expected* behavior, i.e. generate the same visual representation and produce predictive markup. Above that, being a "good citizen" it should produce simple and semantically appropriate HTML even in cases when the given markup does not conform this rule.

Native components

As discussed in section **3.4: HTML Editing APIs**, when an element is switched to editing mode using the HTML editing APIs, users can click inside the text and will be presented with a caret. They can move the caret with the arrow keys and enter text that will be inserted at the appropriate offset. They can use keyboard shortcuts and use the mouse's context menu to paste text. Behavior that is common for rich-text input, for instance that a new list item will be created when users press "enter" inside a list, is implemented by the browser. None of this is available when not using HTML editing APIs. All of this must be accounted for and implemented in JavaScript. Elements like the caret must be mimicked with DOM elements like the `div` element. The users' input must be read with JavaScript and either move the caret or modify the text.

Possible performance disadvantages

Modifying the text on a website means manipulating the DOM. DOM operations can be costly in terms of performance as they can trigger a browser reflow[9]. While it should be a goal to keep browser interactions to a minimum, there is no way to avoid DOM interaction with any visual text change.

File size

While bandwidth capacities have vastly improved, there may still be situations where a JavaScript libraries' file sizes matter. This may be for mobile applications or for parts of the world with less developed connections. When not using HTML editing APIs, a lot of code must be written and transmitted just to enable basic text editing, which would not be needed otherwise.

[9] `https://developers.google.com/speed/articles/reflow`, last checked on 07/19/2015

Chapter 8

Conclusion

8.1 Conclusion

HTML editing APIs will generate different markup on most browsers, their functionality is limited and restricts web developers from extending it. As for the current state, their implementations contain plenty bugs on almost every system, which cannot be fixed by web developers. The ”DOM Level 1” APIs required to perform the same tasks as HTML editing APIs have been developed and tested for more than 15 years and tend to be stable. Google's document editor demonstrates it is possible to implement a fully featured editor without using editing APIs. Doing so will avoid any restriction and limitation of these APIs and give web developers full control of all components, the generated markup and possible bugs.

Part III

Concept

Chapter 9

Approaches for enabling rich-text editing

9.1 Overview

This section will discuss the options to implement rich-text editing without relying (entirely) on HTML editing APIs and approaches to avoid their disadvantages and bugs.

9.2 Native inputs, images and third-party plugins

As discussed in **7.1: Alternatives to HTML editing APIs**, native text inputs cannot be used for rich-text editing, using image elements has no benefits and many disadvantages and third-party plugins lack user adoption. For these reasons, none of these approaches will be considered.

9.3 Enabling editing mode without using its API

One way to enable editing but avoid many bugs and browser inconsistencies, is to enable the editing mode on an element, but refrain from using `execCommand` to format the text. The latter could be implemented using the DOM core APIs. This would provide the user with all basic editing functions, i.e. a caret, text input, mouse interaction and clipboard capabilities—all of this would be taken care of by the browser.

This approach would solve the problem of buggy and inconsistent `execCommand` implementations but not the problems that arise with different browser behavior on the user's text input—for instance when entering a line break. If the markup is customly generated with JavaScript but the input would be handled by the browser's editing mode, the browser may not be able to work on the structures generated by JavaScript and break elements or simply get stuck. This was one of the reasons why Google decided to abandon editing APIs entirely[1]. It could be the source to many bugs and ultimately restrict the editors capabilities.

9.4 Native text input imitation

The only other option to allow the user to change the text on a website is by manually fetching the user's input and manipulating the DOM with JavaScript and DOM Level 1 APIs. However, this does not suffice to provide the experience of a text input. The following components, common to text editing, must also be accounted for:

Caret

The caret is an essential part to text editing. Even if a user types on his or her keyboard, a caret must be seen on the screen to know where the input will be inserted. The caret also needs to be responsive to the user's interaction. In particular, the user must be able to click anywhere in the editable text and use the arrow keys to move it (possibly using modifier keys, which's behavior depends on the operating system used).

Selection

The user must be able to draw a text selection using his or her mouse and change the selection using shift and the arrow keys. Most systems allow double clicks to select words and sometimes tripple clicks to select entire paragraphs. Other systems, for example OS X, allow holding the option key to draw are rectangular text selection, independent of line breaks.

[1] `http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html`, last checked on 07/21/2015

Context menu

The context menu is different in text inputs from other elements on a website. Most importantly, it offers an option to paste text, that is only available in native text inputs or elements in editing mode.

Keyboard shortcuts

Text inputs usually allow keyboard shortcuts to format the text and to perform clipboard operations. Formatting the text is possible through DOM manipulation, pasting text however only works on text inputs or elements in editing mode.

Undo / Redo

Undo and redo are common functions of text processing and it may be frustrating to users if they were missing.

Behavior

Rich-text editors (usually) share a certain behavior on user input. When writing a bulleted list, pressing the enter key usually creates another bullet point instead of inserting a new line. Pressing enter inside a heading will insert a new line. However pressing enter when the caret is located at the end of a heading commonly creates a new text paragraph after heading.

9.5 Approaches for imitating native components

These components are natively available for text inputs across all browsers. Switching an element to editing mode enables these components too. That means users can click in a text to place a caret and move it with the keyboard's arrow keys. They can copy and paste text. The browser offers a native context menu that allows pasting on input elements as well as on element in editing mode. All major browsers implement a behavior for the users' input that is common for rich-text editing.

When not using editing APIs, all of this must be implemented with JavaScript. This requires a lot of trickery and many components must be imitated to make it *seem* there is an input field, where there is none. The users must be convinced they are using a native input and must not notice they are not.

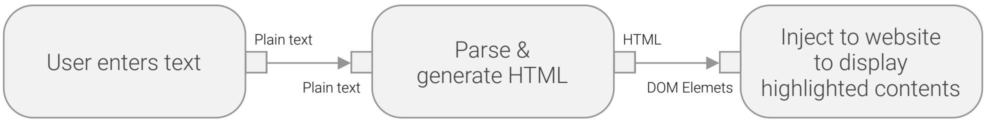


Figure 9.1: Rendering of highlighted source code in Ace and CodeMirror

Web-based code editors like ”Ace”[2] and ”CodeMirror” demonstrate that this is possible. They display syntax-highlighted source code editable by the user. The user seemingly writes inside the highlighted text and is also presented with a caret as well as the above mentioned components. In reality, the content that the user sees is a ”regular” part of the DOM—non-editable text, colored and formatted using HTML and CSS. When the user enters text, the input will be read with JavaScript. Based on the input Ace and CodeMirror generate HTML and add it to the editors contents, to show a properly syntax-highlighted representation (see figure 9.1). A `div` element that is styled to look like a caret is shown and moved with the user's keyboard and mouse input. The user's text input will be inserted at the according text offset. Amongst others, Ace and CodeMirror use DOM elements like `divs` to display a text selection and a *hidden* `textarea` to fetch keyboard inputs, to recreate the behavior and capabilities of a native text input.

Using tricks and *faking* elements or behavior is common in web front end development. This applies to JavaScript as well as to CSS. For instance, long before CSS3 has been developed, techniques have been discussed on how to implement rounded corners without actual browser support. Only years later, this has become a standard. This not only enables features long before the creators of browsers implement them, this *feedback* by the community of web developers also influences future standards. Incorporating feedback is a core philosophy of the WHATWG, the original creators of HTML5.

[2] `http://ace.c9.io/`, last checked on 08/22/2015

9.6 Implementation

The library implemented for this thesis uses similar techniques as Ace and CodeMirror to create a rich-text editor. The contents of the editor are represented by by a `div` element that contains the formatted text using HTML. The caret as a `div` element styled to mimic a native caret. The text selection is also displayed by using `div` elements that are styled accordingly. The keyboard and mouse input will be read with JavaScript and the contents of the editor will be changed accordingly using DOM Level 1 methods. The user's input will also be read to move the mimicked caret on the website with JavaScript. The specifics on the implementation of these components and how they interact will be discussed in **13: Implementation**.

Chapter 10

Software design

10.1 Implementation as pure library

Most rich-text editors are implemented and distributed as user interface components. That means instead of only providing a library that offers methods to format the selected text and leaving the implementation of the user interface to the respective developer, most libraries are distributed as input fields with a default editor interface that is, at best, customizable.

This can be unfitting for many situations. The user interface of an editor highly depends on the software it will be integrated in. Within the software the interface may even vary depending on its specific purpose. For instance, a content management system may require an editor with a menubar offering many controls while a comment form on a blog requires only very little controls. Medium.com uses an interface that only shows controls when the user selects text and has no menubar at all. Assuming there are many implementations of editors that are functional, it can be argued, that choosing between editors is often really a choice of the desired user interface.

Customizing a user interface can be just as complex as writing an interface from scratch. The latter affords to add HTML elements and call JavaScript methods while both require styling. While adding HTML elements to a website is not a complex task, in a worst case scenario, it can be more complicated to customize an interface to specific needs than writing an interface from scratch and being able to define just the elements as they are required.

As for the current state of the internet, web developers cannot easily implement a rich-text for themselves, they have to make a choice between pre-made solutions and customize them. Apart from the perspective of the user interface, integrating a fully featured editor into a software project can be invasive to the structure of the project.

For these reasons the library of this thesis will be implemented and distributed as a pure software library, offering developers an API to create a rich-text editor, rather than a fully implemented rich-text editor as a user interface component.

10.2 API

The library should be capable of any method implemented by HTML editing APIs. However the API design can differ to improve they way it will be worked with. In particular the API and aims at providing a quick and simple way to create editable areas and connecting a user interfaces to it.

API Design

The API of this library must be *well-designed*. That means it must be simple, effective and fit the developers' needs. The methods it offers should be simple in the sense that they conceal possibly complex tasks with understandable high-level concepts. They should be effective and fit the developers' needs in the sense that the API should be designed so that any requirement to of the developers should be matched with as little effort as possible. The API should create a workflow for developers that allows them to do what they intend to do and is as easy to use and as plausible as possible. jQuery is an example of incorporating an API that comes close to these goals.

The library's API will have two basic use cases. On the one hand, web developers must be enabled to implement rich-text editors with it. On the other hand, the library should offer interfaces for enabling web developers to extend the library and add features.

Extension For extension, web developers should have precise access to as many components and functions of the library, providing as much freedom and options as possible. This will include low-level access to components while control and explicitness is more important than simplicity.

All components of the library will be implemented as classes. To provide as much capabilities as possible to other developers, all classes of the library will be exposed in a designated namespace. The classes should conform the best practices of object-oriented programming to support developers in extending the library. The class design should not only consider the specific needs of the core library but also potential use cases for other developers.

For example, with a designated class to show and move a caret, multiple carets can be instantiated for an extension that allows real-time collaboration with multiple users. All available classes will be discussed in chapter **13: Implementation**.

Editor implementation For web developers implementing an editor, the API should be designed to offer methods for the most common tasks related to rich-text editing to allow fast and easy integration in a website. This should be high-level methods as compared to methods required for extending the library. Simplicity is more important than precise control over low-level behavior. For implementing a rich-text editor the exposed methods should cover

1. Formatting and removing formats
2. Insertion
3. Deletion
4. Controlling the caret
5. Controlling the text selection
6. Controlling the clipboard
7. Controlling settings
8. Undo / redo commands

jQuery demonstrates an effective and simple approach to API design, conforming the principles as discussed above. In jQuery all methods remain in a flat hierarchy within the root of a jQuery collection. Any method that is not a getter allows chaining and most methods are overloaded to allow passing various kinds of parameters, to determine what the function should do. Following these and the above-mentioned principles, the components listed above can be expressed in 11 functions:

```
 1  editor.caret([options]);
 2  editor.selection([options]);
 3  editor.insert([options]);
 4  editor.format([options]);
 5  editor.remove([options]);
 6  editor.settings([options]);
 7  editor.copy();
 8  editor.cut();
 9  editor.paste();
10  editor.undo();
11  editor.redo();
```

Listing 10.1: API for implementing a rich-text editor

The functions in lines 1 through 6 can take various overloaded parameters to determine the specific action. The selection command, for instance, can be called with two numbers to draw a selection from one character offset to another. To draw a selection from characters 10 to 20 `editor.selection(10, 20)` can be called. The function can also be called without passing any parameters to read the selection. `editor.selection()` will return the currently selected contents. A full API description can be found in tables .2 through .8.

Handling use cases

We can call programmers extending the library ”developers of the library” and programmers using the library to implement editors ”users of the library”. To account for both use cases and maintain a clear software architecture as well as a separation of concerns, all classes that provide functionality to the library must remain in a designated namespace which the library has access to. Developers of the library have access to the namespace and can utilize any of its classes to extend its functions.

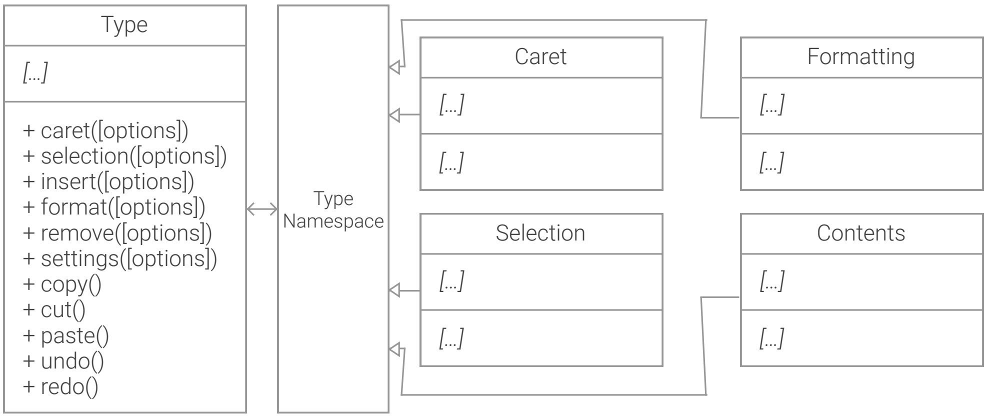


Figure 10.1: Diagram of the Type library and its internally used classes (excerpt)

The classes within the namespace will be used by a globally accessible class called ”Type” which is the entry point for the users of the library to implement rich-text editors. The Type class provides an API with all of the above-mentioned methods and uses the classes inside the namespace for their implementation. It must be instantiated and be passed an element on the website (for example a `div` element) which it will then use as its ”editor contents”. The users of the library can build an interface for this editable element and use the instance’s API to edit its rich-text contents.

10.3 Distribution

The library will be distributed as a single JavaScript file. Extensions by third-party developers will each be distributed as independent and separate (JavaScript) files. By exposing Type and its classes as discussed in section **10.2: API** they can be accessed from other files. This provides a modular ”plug and play” system for distributing and loading extensions. To improve loading times, web developers can concatenate Type and its extensions to a single file in a web project.

Chapter 11

Architecture

11.1 Model–view–controller

Model–view–controller (MVC) is a common approach for implementing user interfaces and it can be applied to user interface components too. While this approach can provide clear responsibilities, the problem is that most components, like the caret or the selection, serve a clear atomic purpose and would need to be broken apart into model, view and controller parts themselves, making the architecture fuzzy and complex instead of simplifying it.

Following the MVC architecture, the contents of the editor (the text) can be represented in a model (holding the text data and allowing methods to be performed on) and be rendered with a view (displaying the text in the browser). In contrast to the beforementioned components, this would be a a very clean model for implementing the editor's contents. It is even imaginable to implement multiple renderers in the view layer, turning the editor from rich-text into a Markdown editor, for instance.

Unfortunately, this approach would make the contents of the editor only editable through the API of the "Type" library. If any other script on a website would change its contents, the library's renderer would overwrite the changes with the next rendering the data of the internal model. As discussed in **10.2: API**, the library shall leave as much freedom as possible to the developers. This would create a bottleneck and restrict other developers. For this reason, the MVC architecture will not be used.

11.2 Modular and object-oriented programming

jQuery and CKEditor demonstrate a software architecture in which a base object, which is exposed to other developers as the library, provides an environment to extend its functionality, but does not offer many methods itself[1]. The actual functionality of both libraries is implemented through extensions while the libraries are usually bundled with a set of ”core extensions” that provide basic features. CKEditor makes use of modular programming techniques by implementing a major part of its editor as plugins that communicate via strictly defined interfaces. jQuery established a paradigm calling any extension a ”plugin” but instead of using strictly defined interfaces, developers are encouraged to add arbitrary methods to jQuery's base object, which can then be directly accessed. Extending a base object has many advantages:

1. It provides a namespace for the library
2. It provides a structure for extensions to access each other
3. It approaches modular programming and strong decoupling

Strict modular programming could create a system in which other developers can exchange any component easily to improve performance or enrich functionality. The disadvantage this approach would be that the need for well-defined interfaces can diminish flexibility. Formalizing interfaces would create complex structures and could make it harder for other developers to contribute to the library instead of inviting them. jQuery uses another approach and encourages arbitrary extensions. jQuery's approach demonstrates that this flexibility, in practice, can withstand possible conflicts. In turn, the low barrier for extending jQuery has spawned a rich collection of extensions and a big community of developers. While jQuery technically allows to be extended with complex libraries, it is designed to be extended with simple methods. It is difficult to establish complex interactions between extensions.

[1] CKEditor provides a framework for implementing components for it, but does not offer any rich-text functionality in its core. jQuery provides low-level utility methods for JavaScript.

Constructor Pattern & modularized structure

To close the gap between CKEditor's modular programming approach and jQuery's simple extension paradigms, object-oriented programming (OOP) can be used. JavaScript does not offer classes and classical inheritance, however the same functionality can be achieved using the constructor pattern and prototypal inheritance (see **13.14: OOP**). Functions following the constructor pattern are often called classes or pseudo-classes. Hereinafter the term classes will be used.

As discussed in **10.2: Handling use cases**, the base class will be globally accessible with the name ”Type”. It will provide the namespace for classes that extend the library and implement its functionality. A set of core extensions will provide all components needed for a rich-text editor. The ”Type” base class can be instantiated and will be the entry point for *users* of the library (see **10.2: Handling use cases**) to implement a rich-text editor. Like CKEditor and jQuery, will implement as little functionality as possible itself. The implementation of the base class as well as the interaction of its extensions will be discussed in detail in chapter **13: Implementation**. Implementing the library's extensions as classes has many benefits:

1. As compared to CKEditor and modular programming, strictly defined interfaces are not a necessity. This can improve flexibility and lower the barrier for other developers to contribute.

2. As compared to jQuery, classes can have complex interfaces, which allows rich functionality and possibilities in interaction.

3. Classes are a proven concept for encapsulating functionality and data, protecting access and structuring code as well as making it readable.

4. Through JavaScript's prototypical inheritance, the class can be instantiated as often as desired, but will only be allocated once in the browser's memory. Thereby the performance will be improved. Instance variables still allow to reuse a class in different contexts with different inherent data.

Chapter 12

Goals

The library of this thesis shall enable web developers to implement rich-text editors themselves. The following features shall be implemented:

Writing and deleting text Users of an editor shall be able to type inside the text using hardware as well as virtual keyboards. Behavior to support the users' input as discussed in **9.4: Native text input imitation** will be an optional ”nice-to-have” requirement.

Text formatting Users shall be able to format the text. It must be distinguished between inline and block formattings. Block formattings break the text flow and always create new lines before and after the formatted text. Block formattings include headings, paragraphs and quotations amongst others. Rich-text editors often show a different behavior for applying block formattings as compared to inline formattings. Block formattings will change any text block (i.e. block-formatted parts of a text) affected by the formatting command into the format that is being applied, even if only a part of a text block has been affected. Inline formattings will strictly apply the formatting to the part of the text that has been selected to be formatted. The text shall be formattable with any HTML tag and HTML attributes.

Caret Users shall be able to control a caret with a mouse and the arrow keys on a keyboard.

Selection Users shall be able to create a text selection and copy its contents.

Undo and redo Users shall be able to use keyboard shortcuts to undo and redo changes they made.

Clipboard control Developers shall define which contents can be pasted from the clipboard and apply rules to remove unwanted formattings.

Real-time collaboration The library shall provide basic real-time collaboration between editors over the network.

Media Editing images and other non-text content is not part of this thesis.

Browser support The library shall be stable in at least one major browser.

Part IV

Implementation

Chapter 13

Implementation

13.1 Overview

As discussed in **11.2: Constructor Pattern & modularized structure**, Type's implementation relies on a base class that provides a high-level API for implementing rich-text editors. The library's functionality is implemented through various other classes encapsulated in a designated namespace. Section **13.2: Technology** will discuss the tools used to develop and build the library. Section **13.3: Base class** will discuss the base class and the namespace it creates. Sections **13.5: Input flow** and following discuss the functionality, architecture and the classes involved in Type by explaining how the user's input will be read, processed and written to the website.

13.2 Technology

Overview

There are no pre-made solutions or conventions suggesting how to design, structure and concatenate components for a JavaScript library. The most popular JavaScript libraries on GitHub[1] each implement custom and different solutions. Angular.js implements a custom module-system and uses the tool ”Grunt” to concatenate multiple files into one. D3.js uses a Makefile and var-

[1] `https://github.com/search?l=JavaScript&q=stars%3A%3E1&s=stars&type=Repositories`

ious Node.js modules for building and concatenation. jQuery uses Grunt for concatenation and runs custom scripts implemented using Node.js to manipulate and clean up the resulting source code file. To support development, the library is split up into multiple files. One file each contains one class. The tools to concatenate, build and check the sources will be discussed in the following sections.

Gulp

Gulp is a Node.js-based task runner that is widely used as a build system for web applications. For this library, Gulp tasks are used to lint the source code, check code style rules and to concatenate and minify the library into a single file, that can be distributed to other developers.

Building

CommonJS specifies a system to define a so-called module in a JavaScript file that can be loaded by other modules by referencing the file name. The module that will be returned when loaded by other modules can be an arbitrary object or a primitive data type. For this library, each file is written as a CommonJS module containing and returning a single class. All files will be concatenated using RequireJS[2] in a designated Gulp task to a single file. The resulting file contains code structures generated by RequireJS that will allow the modules that have been defined across multiple files to access each other within a single file. This will increase the library's file size. AMDclean[3] is used to remove as much of this supporting code as possible while maintaining the functionality. To further decrease the file size UglifyJS2[4] is used to shorten variable names, remove whitespace and compress the file using ”gzip”.

Linting

JSLint is used to lint the source code before any concatenation happens. It will also check if the code conforms the widely acclaimed JavaScript code

[2] `http://requirejs.org/`, last checked on 8/18/2015

[3] `http://gregfranko.com/amdclean/`, last checked on 8/18/2015

[4] `https://github.com/mishoo/UglifyJS2`, last checked on 8/18/2015

conventions introduced by Douglas Crockford in his book ”JavaScript: The Good Parts” [Crockford, 2008]. The code mostly conforms these conventions, differing in the way the constructor pattern is implemented to favor better readability. Classes using the constructor pattern are implemented using the conventions of the Ace library and use the prefix convention[5] for private members.

Code formatting

JSCS is used to check the code to conform specific formatting conventions—for instance the number of spaces used for indentation or a consistent use of CamelCase. Along with JSLint checking for Douglas Crockford's conventions this ensures a consistent coding style across all files and classes.

13.3 Base class

Overview

The `Type` class is the base class (see **11.2: Modular and object-oriented programming**) is the starting point for users and developers of the library. It provides 4 purposes:

1. It can be instantiated to offer a high-level API to manipulate text and perform other rich-text related functions.
2. It provides a namespace for the library's internal classes.
3. A Type instance provides mutual access for the instances of the internally used classes.
4. It exposes its `prototype` as a public shorthand attribute.

Instantiation and usage

To develop an editor with the library, the `Type` base class must be instantiated. As discussed in **10.2: Handling use cases**, all of the library's functionality is

[5] `https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contributor_s_Guide/Private_Properties#Using_Prefixes`

provided by this class. It exposes its features as high-level instance methods. The `Type` class is exposed to the `window` namespace and is thereby globally accessible. On instantiation it must be passed an `HTMLElement`, which's contents will act as the editor's rich-text content and all rich-text operations will be performed on it. Just like with HTML editing APIs, developers can build user interfaces around the editable element. An optional second parameter can be passed to the class to define settings for the editor's instance.

```
var element = document.getElementById("myElement");
var editor = new Type(element, { paste: "text" });
```

Listing 13.1: Type instantiation

The `editor` instance variable now offers methods to format, insert and remove text, manipulate the caret and the selection, dynamically change settings and control undo/redo capabilities as well as trigger clipboard commands. The complete API is listed in tables .2 through .8. For example, to format the characters 10 to 20 as bold and move the caret behind the formatted text, the following methods can be executed:

```
editor.selection(10, 20);
editor.format("<strong />");
editor.caret(20);
```

Listing 13.2: Example commands to format text

This is just an example to demonstrate the API. It should be noted that the API allows to specify a text range in the format command as well as chaining and the above code can be simplified to a single line.

```
editor.format("<strong />", 10, 20).caret(20);
```

Listing 13.3: Example chaining

With the second optional parameter, settings can be passed to the Type class on instantiation to determine the editor's behavior. As for the implementation of this thesis two settings are available: To determine the behavior on paste events and to turn off default keyboard shortcuts. Passing options to the editor can be useful for extensions which can access any option passed be accessing the Type base class.

Namespace and references

The Type class creates a namespace for other classes used in the library. In JavaScript, functions are first-class objects, namespaces are objects and any object is a namespace. This way, the Type Function object (the class itself) can act as a valid namespace. CKEditor takes the same approach and attaches each module to the CKEDITOR base class. Any of the classes listed in figures .3 through .6 are attached to the Type class this way. As an example, listing 13.4 shows the declaration of the Type base class as well as the declarations of the classes Caret, Range and Environment:

```
// Declaration of the Type base class
function Type() {};

// Classes defined within the namespace created by Type
Type.Caret = function () {};
Type.Range = function () {};
Type.Environment = function () {};
```

Listing 13.4: Declaration of Caret, Range and Environment classes

The namespace provides a structure for the classes of the library, prevents the pollution of the global namespace and possible name conflicts. In terms of structure, this does not only encapsulate classes related to the library but also allows nesting. This way sub-namespaces can be created, which is especially important for other developers extending the library. The Type base class already creates a sub-namespace Type.Events for events.

On instantiation, the Type class, in turn, will instantiate classes of its namespace that implement its rich-text editing functionality. The Type instance will pass a reference to itself to all classes and offers getters for every class instance it created, to provide mutual access for each class of the library.

Exposal of Type's `prototype`

While Type's functionality is implemented through classes within its namespace, this does not expose its functionality to its instance-API. With the constructor pattern, all methods in the `prototype` of the `Type` class will be available as instance methods. In a simple approach, the library's API can be implemented directly in the implementation of the `Type` class. This contradicts the modular approach of the library. jQuery established an effective principle for extending its API. It exposes the library's `prototype` with a shorthand attribute as `jQuery.fn`. This way, other modules can extend the `prototype` easily and add methods to the library's API. Type follows the same principle and exposes its `prototype` as `Type.fn`. The `prototype` could also be accessed without exposing it with a shorthand attribute, but this is intended to clarify its purpose similar to jQuery and encourage developers to extend it.

13.4 Api

The methods of Type instances can be added by any of the classes of the library. However, to achieve a clear separation of concerns, maintainability and to conform the modularized structure, Type's instance API is implemented through a designated module that adds all methods to the API.

The module for this, called "CoreApi", is the only exception in Type's implementation—it extends `Type.fn` directly with all methods listed in tables .2 through .8, *without* using a class.

13.5 Input flow

To enable reading the user's input and writing it to the editor's contents, the classes `Input`, `Contents`, `Writer`, `Formatter`, `Caret` and `Selection` will be instantiated by the `Type` base class.

The `Input` class will listen to keyboard input and mouse input. It is responsible for setting the caret and the selection using the `Caret` and `Selection` classes and uses them to determine which part of the text should be changed or formatted when the user enters text, uses keyboard shortcuts or uses his or her mouse or touch device. It passes formalized edit operations to the `Contents`

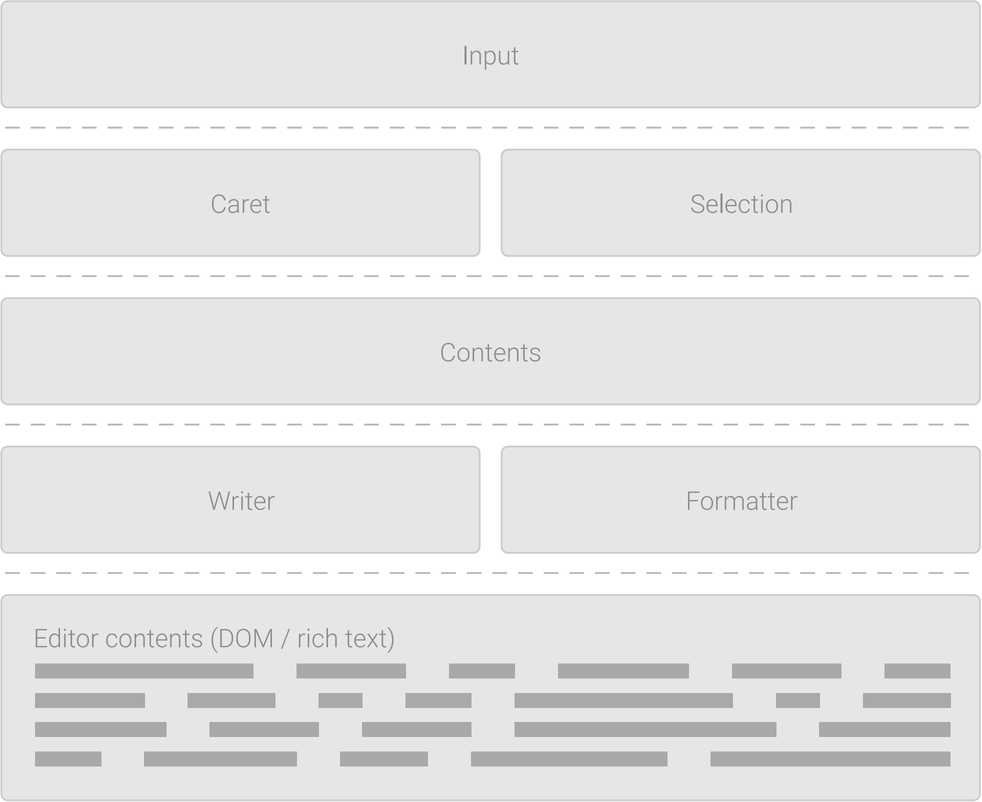


Figure 13.1: Components instantiated by the Type base class

class which will emit events for an `UndoManager` that enables undo and redo operations. The `Contents` class uses the `Writer` and `Formatter` instances to manipulate the visible text on the website. These classes perform the actual DOM operations on the contents of the element passed to Type on instantiation (see **13.3: Instantiation and usage**).

Usually, text input fields contain one caret and display one text selection at a time. For this reason the `Type` base class instantiates the `Caret` and `Selection` classes for shared usage within an editor's instance. Of course, this behavior can be extended, for example by instantiating multiple `Caret`s for real-time text collaboration.

13.6 Input reading

There are various input methods with which users can interact with native inputs. This includes using hardware devices as well as virtual (on screen) devices:

- Hardware keyboard input
- Virtual keyboard input
- Mouse input
- Touch input
- Game controller input (on game consoles)
- Remote control input (on smart TVs)

When mimicking a native input, in a best-case scenario, all these input methods should be accounted for. Fetching input includes two scenarios: The user clicks, touches or focuses the input in any way and does so at any position inside the input. If the user points (touches, clicks, etc.) in the middle of the text, the caret should move to that position. In environments without hardware keyboards, the library must ensure that a virtual keyboard shows up. Once the input is focused, text input must be fetched and written to the contents. There are various options to fetch user input, which will be discussed in the following sections.

Events

One way to fetch user input is by listening to events.

Keyboard Text input can be read through `KeyboardEvent`s. Keyboard events will be triggered for virtual keyboards and for hardware keyboards. When the user presses a key, the event can be stopped and the according characters can be inserted at the offset of the caret. As a downside, listeners for keyboard events cannot be bound to an element that is not a native text input, that means keyboard events must be listened to on the `document` level.

This does not only have (minor) performance downsides but als requires more logic to decide whether a keyboard input should be processed and ultimately stopped or ignored and allowed to bubble to other event listeners of a website. In particular, there can be edge cases, where even though a keyboard event should write contents to the editor, the event itself is supposed to trigger other methods that are not part of the editor. Keyboard events are supported by all major browsers across all devices.

Mouse and touch To support clicking or touching inside the editor's contents `MouseEvent`s and `TouchEvent`s can be used. Mouse events are supported on all major desktop browsers and all mobile browsers support touch events. Both event types support reading the coordinates indicating where the click or touch has been performed.

Remote controls Although some smart TVs offer keyboards, mice, pointers similar to Nintendo's Wii remote, input via smartphone apps and many other input divides, button-based remote controls are offered with almost any smart TV and remain a edge case for interacting with a text editor. In such an environment, users commonly switch between elements by selecting focusable elements with a directional pad. Only using events would not account for this since there would be no focusable element representing the editor. Recent browsers on Samsung's and LG's smart TVs are based on WebKit[6] while Sony's TVs use Opera. Before 2012 Samsung's browser was based on Gecko. All of these browsers and browser engines support keyboard events triggered by virtual keyboards to fetch their input.

Clipboard Another problem with relying entirely on events is the lack of native clipboard capabilities. Unless a native text input (including elements with enabled editing mode) is focused, shortcut keys for pasting will not trigger a paste event and the mouse's context menu will not offer an option for pasting.

[6] `http://www.samsungdforum.com/Devtools/Sdkdownload`, last checked on 07/22/2015

Hidden native input fields

As discussed in **9.5: Approaches for imitating native components**, the source code editors Ace and CodeMirror use a hidden (native) input field to fetch the users' keyboard input. While it appears to the users they are entering text in a syntax-highlighted representation of the source code, in reality users enter their text in a *hidden* `textarea` element. The input will be read from the `textarea`, processed and displayed with syntax-highlighting using HTML. This solves many problems that occur with relying solely on events:

- The hidden `textarea` can be focused with the tab key.
- The hidden `textarea` can be focused with remote controls.
- Virtual (on screen) keyboards will show up when the `textarea` is focused.
- Keyboard shortcuts for clipboard events work.
- It can display a native context menu that allows pasting.

Implementation

The `textarea` is created when the editor gets instantiated. Since browsers scroll the `textarea` into view when it receives the focus, it is positioned in the visual representation of the editor, scrolling the editor into view[7]. This perfectly mimics the browser's native behavior. To maintain the illusion that the user actually writes inside the visual representation of the editor the `textarea` is hidden.

Focus

Whenever the user clicks inside the editors visible contents, the mimicked caret will be moved (see **13.9: Caret**) to the according text position. To enable text input, the hidden `textarea` will be focused on click or touch events. The `textarea` is natively focusable using the tab key or a remote control on a smart TV. It will also trigger focus and blur events. This way, it is possible to display

[7] This does not mean the editor will be scrolled into view on instantiation, but when the user focuses ist, for example with the tab key.

the caret when the `textarea` receives the focus and read its input as well as hiding the caret on blur and thereby perfectly mimic the native behavior for input events.

Virtual (on screen) keyboard support

The `textarea` will be focused when the user clicks or touches inside the editor as well as with the tab key and remote controls. Focusing a text input triggers the display of native virtual keyboards.

Pasting

When a the textarea is focused, pasting via keyboard shortcuts is natively available. To enable pasting with the context menu, CodeMirror implements a technique where the `textarea` will be moved to the pointer's position on a `mousedown` event. Following the order of `MouseEvents`, this will be completed before the context menu will be triggered. This way it will be triggered on the `textarea` and contain a paste option. The paste event will insert the contents from the clipboard to the textarea from which the contents can be read.

Reading input

`textarea` elements support `input` events which can be used to read the text entered by the user. The input can be processed as discussed in **13.5: Input flow** and be removed from the textarea. In practice, this means that once a single character has been entered in the `textarea` it will be read from the `textarea`, inserted into the editor's contents and the `textarea` will be cleared again. Input reading requires further processing before it can be passed on to trigger a change in the editor's contents. The specifics on processing the input will be discussed in section **13.7: Input Pipeline**.

Editing mode

Using a `textarea` element allows plain-text input only. This is not a problem for regular keyboard input but rich-text contents pasted from the clipboard will be inserted as plain text and all formattings will be removed. To come around this issue, a `div` element in editing mode can be used instead of a `textarea` element.

As discussed in **Part II: Discussion** HTML editing APIs are very problematic and a key factor of this thesis is implementing a rich-text editor without using them. However using an element in editing mode only for input reading is not affected by these issues. Whenever a single character will be entered it will be read and immediately removed from the editable element. Formatting commands will not be used at all. Problematic text input behavior, for instance different markup that will be generated by the entering a line break, will not occur since the the editable area will only be used for reading input, the text that will be inserted in the editor will be generated by the library (see **13.7: Input Pipeline**). The only difference between a `textarea` and the editable element lies in the different contents it accepts for pasting. **6.3: Clipboard** discusses the problem that text pasted from the clipboard cannot be processed with native APIs across all browser. In this case, the clipboard contents will be pasted to a designated field from which it can be read, isolated from the editors contents, which solves this problem (see section **13.8: Pasting**).

13.7 Input Pipeline

Overview

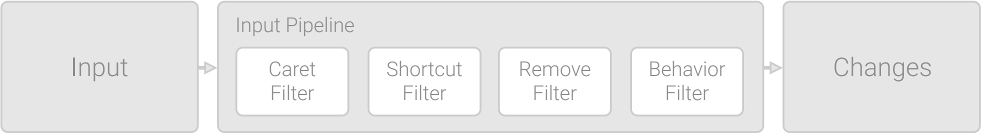


Figure 13.2: Input pipeline with sample filters

Before the text read from the hidden input field will be passed on to the `Contents` class, it will be passed though a series of `Filters`, called the ”input pipeline”. The input pipeline has 3 basic responsibilities.

- Stop and dispatch input that that should trigger functions of the library
- Implement rich-text editing behavior

- Filter and transform text

The input pipeline is part of the `Input` class. The pipeline itself is an array of input filters that can be added, removed and ordered with a designated API.

Input filters have a public API that specifies for which input they should be called. For instance, a filter can specify to be called when the user presses the ctrl s key combination. A filter can specify a handler for the input. For pressing ctrl s, a filter can specify to call a "save" function. The input is passed to the filters as an event (see **13.13: InputEvent**) that can be stopped from bubbling. This way it can be prevented that pressing ctrl s will also insert an "s" character to the editor. The order in which filters will be called is important. Some filters process the input and must cancel the event not only to prevent a character to be inserted, but also to prevent other filters from taking action.

The basic filters implemented for this library will be listed hereinafter, grouped by the responsibility as listed above.

Triggering functions

Caret Commonly, when a user presses one of the arrow keys inside a text, the caret will be moved and of course, in a browser environment, this is not different. To mimic this behavior arrow key input will be intercepted by the `Caret` input filter class, which will move the caret that has been instantiated by the `Type` base class (see **13.5: Input flow**). It will also account for modifier keys and the operating system used and move the caret accordingly.

Command The `Command` input filter class checks for and intercepts keyboard shortcuts commonly used for text formatting.

- ctrl b Formats the currently selected text bold.
- ctrl i Formats the currently selected text italic.
- ctrl u Formats the currently selected text underlined.

To format the text, by default, the tags `strong`, `i` and `u` will be used. Since in some cases this might not be desired, these default keyboard shortcuts must be opted-in by setting an option on instantiation (see **13.3: Instantiation and usage**).

The input event implements an abstraction to either check for the ctrl key or the cmd key depending on the operating system of the user (see **13.13: InputEvent**) so that the above-mentioned shortcuts can (only) be triggered with the cmd key instead of the ctrl key on the OS X systems.

Rich-text behavior

As discussed in **9.4: Native text input imitation**, most rich-text editors support the user with a common behavior reacting to the user's input. This behavior can be abstracted in a simple way using the input pipeline. The following paragraphs describe how the behavior discussed in section 9.4 can be implemented using filters. Being a ”nice-to-have”-feature, other filters was given more priority and these filers have not been implemented.

Headlines When a user presses enter while the caret is located at the end of a headline, a new text paragraph can be created behind the headline and the caret can be placed inside it.

Lists Pressing enter inside a list item can create a new list item behind the current item and the caret can be placed inside it.

Filterring and transformation

Line Breaks To display a line break in the contents of the editor a `br` or `p` tag must be inserted. When the user presses the enter key it does not suffice to insert a carriage return and/or line feed. This input will be intercepted and instead a `br` tag will be inserted to the editor's contents. As discussed in **13.7: Overview**, it is important that this filter will be invoked after the `Headlines` and `Lists` filters, so they can apply their behavior and prevent this filter's behavior.

Remove To delete text from the editor the `Remove` filter checks for backspace and del key inputs. Depending on whether there is a text selection or not, it either deletes the selection's contents or one character left/right of the caret.

Spaces Browsers display adjacent spaces as a single space. This is an unusual behavior for text editing. The `Spaces` input filter checks if adjacent spaces are being entered and inserts non-breaking spaces.

Events

As an alternative approach input filters could be implemented as input event handlers. With this, the same functionality could be achieved without an input pipeline. A designated pipeline however provides a clearer mental model for processing the input as it allows a separation of concerns. An input filter has a specific purpose and context as compared to an arbitrary input event handler. It can also be made sure that filters will be called in the right order and that any input filter will be run before triggering an input event. This way input event handlers only receive actual text input for the editors contents while keyboard shortcuts and other keypresses have been filtered out.

Extendability

The input pipeline is intended to be extended. It serves as an entry point for other developers to process input. For this, the `Input` class provides an API to add, remove and reorder input filters.

13.8 Pasting

As discussed in **13.6: Hidden native input fields**, all text input will be read from a designated input field. It is useful to distinguish between regular keyboard input from input pasted from the clipboard since the clipboard can contain rich text contents. Developers implementing an editor with Type should be able to determine which formattings should be allowed in the editor, i.e. pasted from the clipboard. This requires two steps.

1. Determine if an input has been made through typing or pasted from the clipboard.

2. Process the clipboard contents and make them accessible to developers.

Paste detection As discussed in **6.3: Clipboard**, modern browsers trigger paste events, which can be listened to. Not all browsers allow reading contents from a paste event, but this is not necessary. The paste paste event will insert its contents into the designated input element from which its contents can be read, after the event has completed. Some older browsers, specifically Opera versions older than 12.1 do not trigger paste events at all. For legacy support, the browser can be tested for an available clipboard API and in case it is missing, the text input can be checked for its text length. With the system discussed in **13.6: Hidden native input fields**, an input will always have the length of a single character. If the input is longer than that, this either means more than one character has been inserted or a single formatted character has been inserted[8]. These cases can only happen when contents have been pasted from the clipboard. However, if a single unformatted character has been pasted, it cannot be distinguished from a regular text input. It is to be noted that the use case this feature is designed for, is to sanitize pasted input, which is not necessary for a single plain text character input, although it must be acknowledged that there can be use cases requiring to register any paste event for other reasons.

Processing To process the pasted contents and possibly prevent inserting the contents to the editor an `InputEvent` will be generated and passed through the input pipeline. Any filter can be implemented to treat or ignore paste events. Users of the library can set an option on instantiation to determine how to treat pasted contents. These options include to allow plain text only, to allow any formatted text or specifying rules to allow specific formattings only. A full API description can be found at ABC. These options are implemented in the `Paste` filter, that will either let any contents pass through (allow any formatting) or filter out specific or all HTML tags.

13.9 Caret

The `Caret` class provides all functionality to place and move a caret in a text. It provides methods to be moved left, right, up and down in a text as well as

[8] The input field will contain markup of more than one character

to be placed at at specific position in a text. The visual representation of the caret is a `div` element, styled to imitate a text caret. Using a CSS3 animation, it imitates the ”blinking” common for native text carets.

The elements for carets as well as for the text selection will not be written to the editor’s contents. The editor’s contents should not contain any markup other than for the text itself. Instead this and all other elements will be stored in a designated `div` element at the end of the website’s `body` and be positioned using CSS.

The challenge with this class is that it must be able to be moved within text and in any kind of formatting, represented by any combination of DOM nodes. To be moved across letters and text lines, the caret must take into account that:

1. Letters have different widths and heights
2. Different fonts have different letter dimensions
3. Different formattings like a headline, italicized text or text with a specifically set font size, result in different letter dimensions

`CodeMirror` solves these problems by measuring each letter with the use of text ranges. Browsers offer a `Range` interface, a construct that has a start- and end-offset in a text. A range has methods to read its x- and y-coordinates on the website. These methods can be used to span a range over a single character, read its offsets and place the caret next to it using CSS and giving it the same height as the character.

To move the caret left or right, the according characters left and right of its current offset will be measured using this method. To move it up and down across text lines, the caret must check the offsets of every character, starting from the character of the current offset, until it reaches the character above or below it that is closest to its horizontal position. As discussed in section **13.15: Cache**, a cache to store positions cannot be applied. The complexity of this is method $O(n)$, however in practice, the number if characters this will affect is limited by readability and usability of the text editor. Mobile devices, that generally have less performance than desktop machines, have smaller screens displaying less characters per line. While this is not necessarily the case, it can be expected by good software design.

13.10 Selection

Using a designated input element for input reading comes with the cost of having to emulate the text selection. When the input field is focused, any selection on the web site, including that of the editor, will be removed. When text is selected, the input field does not necessarily have to be focused. To read inputs, it can be focused on a ”keydown” event, which will only remove the text selection when the user enters text. This is not problematic since selections will be removed on native inputs when a user enters text too. However, if the user right-clicks in the editor, the input element will be focused to enable pasting from the context menu (see **13.6: Pasting**). This will remove the text selection on any right click.

The W3C specifies an API to add multiple ranges to a selection, which should appear as multiple selections to the user. This way the element for input reading could be focused while other parts of the website, i.e. the editor's contents, could display a selection at the same time. However, while the API is available across all major browsers, it is dysfunctional and documented to not be working.

CodeMirror, ACE and Google's document editor each implement text selections by displaying `div` elements that mimic the look of a native selection. Type uses the same technique to show a text selection while the input element is focused. This mimicked selection replaces native selections entirely and will be created dynamically when the user clicks in the text and drags his or her mouse across the text.

The downside of this technique is that copy commands will not work anymore due to the fact that there is no actual text selection that can be copied, even though it appears to the users there is one. To treat this issue, CodeMirror adds the contents of the imitated selection to the hidden input field and selects these contents with a native selection. This allows the user to use keyboard shortcuts at any time and to copy text with the context menu. When the user types, the selected contents in the input field will be overwritten by the browser, so this does not affect input reading. Type uses the same technique.

13.11 Contents

The `Contents` class provides an API to add, remove and format text. This functionality is implemented through the `Writer` and `Formatter` classes. Its central responsibility is to proxy commands to these classes and to create ”actions” to pass them on to the `UndoManager`. An action describes the formatting, change or deletion of contents in a formalized way that can be undone by the `UndoManager` (see **13.12: Undo Manager**).

Writer

The `Writer` class implements functionality to add and remove contents to and from the editor. Along with the `Formatter` class, this is the lowest layer of the editor that will perform DOM operations to modify the contents in the browser.

Formatter

The `Formatter` is one of the key classes of the Type library. As discussed in **7.4: Formatting** it must generate *well-formatted* markup while being able to work on any *ill-formatted* markup it will be given. There is a virtually infinite number of edge-cases for markup that formatting commands can be applied to. Assume we have the following string.

```
<p>Lorem ipsum <em>dolor<u> sit amet</u></em> consec</p>
```

Listing 13.5: Markup with highlighted target for formatting

Listing 13.5 represents markup for the formatted string ”Lorem ipsum *dolor sit amet*”. The highlighted part (yellow) represents the part of the text that should be formatted using a formatting command.

Figure 13.3 shows the DOM representation of the markup of *Listing 13.5*. We can split up the text node ”Lorem ipsum” into two text nodes ”Lorem” and ” ipsum” to create a distinct node in which the formatting (yellow highlight)

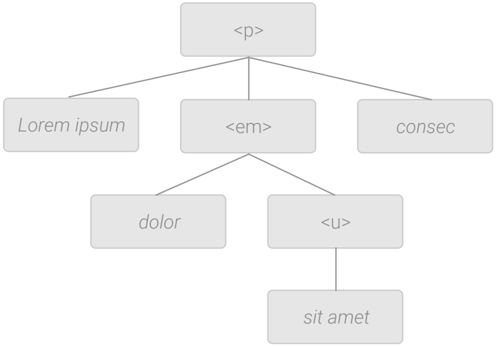


Figure 13.3: DOM representation of figure 13.5

starts and do the same with the ending text node ”sit amet”. This gives us a distinct nodes for the start and the end of the selection. When we traverse each node from the start to the end, every traversed node falls on either of the following cases:

1. It is the start node
2. It is the end node
3. It is a node between start and end node (but is not and does not contain the start or end node)
4. It contains end node

To generate *well-formatted* markup as discussed in **7.4: Formatting**, the following 2 rules must be followed:

1. The markup must conform the validation rules of HTML as specified by the W3C.
2. The markup must use as little DOM nodes as possible.

Algorithm 5 demonstrates a recursive algorithm, that will start iterating from the given start node over all its subsequent siblings until there are no siblings anymore *or* it found the end node *or* it found a node containing the end node.

Algorithm 5 Recursive algorithm to apply text formatting

```
procedure FORMAT($s, e$)    ▷ s and e are the distinct start and end nodes
    $c \leftarrow s$
    Let $a$ is an empty array
    while $c \neq null$ and $c \neq e$ and $c$ does not contain $e$ do
        $a$.push($c$)
        $c \leftarrow c$.nextSibling
    end while
    if $c = e$ then
        $a$.push($c$)
    end if
    if $c$ contains $e$ then
        FORMAT($c$.firstChild, $e$)
    end if
    if $c = null$ then
        $n \leftarrow$ next node in document flow
        FORMAT($n, e$)
    end if
    Wrap all nodes from $a$ with DOM node to apply formatting
    Connect siblings to wrapping node if they have the same tag
end procedure
```

- If the end node was found, the algorithm will wrap all all nodes it found with a node that applies the desired formatting.

- If there are no more siblings, it means it has reached the last sibling inside the node containing the start node. By the validation rules of the W3C, nodes are not allowed to intersect. The algorithm will wrap all nodes it found so far with a node that applies the desired formatting and recursively applies itself with the start node being the next node in the document flow and the end node remaining the same end node.

- If an element has been found that contains the end node, the algorithm will wrap all nodes it found so far with a node that applies the desired formatting (to avoid intersecting nodes). It will then apply the formatting algorithm recursively to the first child of the containing node. If this node in return is another container to the end node, the recursion will repeat until a sibling of the end node or the end node itself has been found.

Each rule performs the minimally necessary steps to format contents while conforming the HTML validation rules of not intersecting DOM nodes. By adding only the minimum of nodes, this will ensure simple (and valid) markup.

To format the nodes that have been collected by the algorithm, they will be wrapped by a DOM node that applies the desired formatting. The wrapping function will also remove any nodes between the start and end node that have the same tag as the node used for the formatting to clean up the markup. If the start or the end node has been contained by a node of the same tag as the node the text should be formatted with, the containing nodes will be used as start or end node. As a last step to improve the markup, the nodes left and right of the formatting node will be unified with the formatting node, if they have the same tag. These steps will simplify potentially *ill-formatted* markup that the formatting command affects. All DOM manipulations will be performed at the end of the algorithm, when all nodes have been read to improve performance.

13.12 Undo Manager

The `UndoManager` implements the undo and redo functionality of the editor. It can receive ”actions”, which are instances of classes inheriting from the `Type` action base class. Each instance has the methods `execute` and `undo` to apply and revoke its particular functionality. There are 3 types of actions.

Insert

The `Insert` action will insert text on execution and delete the text when its `undo` method will be called. It utilizes the `Writer` class for these operations. It must be instantiated with the according text so it can be executed and undone at any time.

Remove

The `Remove` action will remove text on execution and store it to insert it again when it is being undone. Just like the `Insert` action it uses the `Writer` class for this.

Format

The `Format` action applies formattings using the `Formatter` class. It will store references to the nodes the `Formatter` created to remove them when the action will be undone.

Stack

The `UndoManager` stores each action to an array (the undo stack) so that each action can be undone and redone in order. In some cases, consecutive actions, like multiple insertions, should be undone in a single undo command invoked by the user. For this, each action must implement the methods `mergable` and `merge`. When an action get added to the `UndoManger` the `mergable` of the highest element in the undo stack will return if it accepts the new action to be merged with it. The `Insert` action will accept other instances of the `Insert` action class so that consecutive character input can be undone in chunks and not only in characters. Each action will implement merging with other instances in its `merge` method to which a new action will be passed if `mergable` returned `true`. Otherwise the new action will be added to the undo stack. The `UndoManager` will only merge actions that have been passed within a time frame of 500 milliseconds, to generate a distinct undo history.

Action sources

The Etherpad extension (see **13.16: Real-time collaboration with Etherpad**) allows multiple users to modify the same text. The undo methods must only undo changes from the user that initiated the action. Furthermore, undoing and redoing an action must account for changes that affect the action, for instance, when the offset at which a text has been inserted has shifted, because another user has inserted text before that offset. To manage this, each action contains a source identifier that will be generated by the `UndoManager`. With this identifier the `UndoManager` can choose to undo actions of a particular user and account for changes of actions from other users.

13.13 Events

Overview

It is possible to trigger custom (native) events using the `CustomEvent` interface on modern browsers [Mozilla, 2015b]. Internet Explorer 9 and below allow this through similar interfaces. These interfaces could be used for triggering events for components of Type. However, this would trigger events that that are only relevant within the library in the global namespace. To avoid this, Type implements its own event system that populates events only within the library. Events from within the editor that can be useful to the website or web application should still be triggered as native events in the browser's global namespace.

Event Api

The `EventApi` class provides an API to add and remove event listeners as well as to trigger events. It provides instance and static methods.

```
// Static methods
EventApi.on(eventName, eventHandler);
EventApi.off(eventName, eventHandler);
EventApi.trigger(eventName, eventObject);

// Instance methods
EventApi.prototype.on(eventName, eventHandler);
EventApi.prototype.off(eventName, eventHandler);
EventApi.prototype.trigger(eventName, eventObject);
```

Listing 13.6: EventApi methods

Using the `OOP` class (see **13.14: OOP**), these methods will be inherited by the `Type` class. This way, using the `trigger` method, events can be triggered within the scope of a `Type` instance and be observed using the the `on` method. Event handlers can be removed using `off` method.

The static methods will also be inherited by the `Type` class. This is necessary to trigger events that are *Type* specific but not *instance* specific. Most importantly a ”ready” event will be triggered on every instantiation of the `Type` class. This way plugins and other third-party scripts can run initialization routines.

Plugins and third-party libraries can trigger arbitrary events and pass along arbitrary data. As a paradigm and in terms of a *good programming style*, event objects should be passed. Event objects are inherited from the `TypeEvent` or conform its API.

TypeEvent

```
// Gets or sets data
TypeEvent.data([options]);

// Stops the event from bubbling
TypeEvent.cancel();
```

Listing 13.7: TypeEvent API

The `TypeEvent` is a generic, general-purpose event. It can store arbitrary data and offers an API to be stopped from bubbling.

InputEvent

The Input `InputEvent` will be triggered by the `Input` class after a keyboard input has passed the input pipeline. It inherits all methods from the `TypeEvent` to conform the event system and contains information on the key and the modifier keys pressed. The key is represented with its key code and a key name. They key name will be mapped from the key code and is implemented with a list of readable names including ”backspace”, ”enter”, ”space” and others. The list not complete but can be extended during further development. The modifier keys include ”shift”, ”alt”, ”ctrl” and ”meta”. ”meta” is the browser's name for the cmd key on OS X systems. OS X uses the cmd key as modifier the same way Windows and Linux use the ctrl key. To support developing an editor for both platforms the ”cmd” modifier will be set when a user holds the cmd key on OS X *or* the ctrl key on other platforms.

PasteEvent

The `PasteEvent` will be triggered when the user pastes contents from the clipboard *before* it will be inserted to the editor's contents. It contains the clipboard's contents and can be cancelled so that other developers are free to manipulate and stop a paste event.

13.14 Utility classes

OOP

The `OOP` class extends the constructor pattern with basic classical inheritance. It provides the method `inherits` that will duplicate and copy the `prototype` from one `Function` object to another. It also copies attributes and methods defined on the `Function` object itself to implement inheritance of static definitions. It adds the attribute `_super` to the inheriting `Function` object referencing its parent class to enable child classes to access their respective superclasses.

Range

The `Range` class is an abstraction for the native `Range` interface. The native implementation is prone to bugs on many browsers. Instead of fixing the API by shimming its methods, the `Range` class implements all methods related to ranges while trying to interact as little as possible with the native API. On top of the methods of the native `Range` interface, this class implements additional methods required for Type.

Dom Walker

Working with text implies having to traverse the DOM, i.e. the nodes inside the text often. The `DomWalker` utility class solves this problem. The DOM API offers methods to access a node's siblings, children and parents, but it must always account for cases when any of these are `null` (there is no parent, sibling or child) or when they overflow the bounds of the editor's contents. But more importantly, for text editing, it is usually necessary to access the next (or previous) node in the document's content flow which can either be the parent, sibling, child or a node that can only be accessed by traversing multiple nodes. Also, it is often the case that it is not only necessary to fetch the next node, but to apply a filter to only fetch a specific node, for instance a text node or only a text node that has contents visible to the user[9]. Browsers offer a native API for this, called `TreeWalker`, but it is said to be slow[10], only partially supported by Internet Explorer 9 [Mozilla, 2015e] and has been criticized for its verbose API[11].

The `DomWalker` can be instantiated by providing a starting node and the type of nodes it should traverse. The latter argument can either be a string identifying a pre-made filter of the Type library or a custom filter function. Pre-made filters include:

- ”text” - A text node with visible contents
- ”textNode” - Any text node, visible or invisible
- ”visible” - Any visible DOM node

A `DomWalker` instance offers the high-level methods `next`, `previous`, `first` and `last` (amongst others) for traversal.

Text Walker

The `TextWalker` class acts as a container for all functions related to measuring text offsets. It provides utility methods to determine the character offset from one text node to another or, vice versa, which text node can be found at a text offset, starting from another given node. Both methods are required by various classes and are thus, centralized.

Dom Utilities

The `DomUtilities` class encapsulates common methods for all DOM operations other than traversal. It has no inherent purpose but many other classes

[9] Any text node consisting of whitespace only will not be displayed by any major browser

[10] `http://jsperf.com/qsa-vs-node-iterator`

[11] By John Resig, author of jQuery, `http://ejohn.org/blog/unimpressed-by-nodeiterator/`, last checked 08/19/2015

perform the same DOM operations, which hence reside in a common library to avoid code duplication.

Utilities

The `Utilities` class is a general-purpose class that contains methods to extend JavaScript's features. It contains methods to work with data structures and to detect object types.

Environment

The `Environment` class checks and provides informations on the current browser environment and its features. This class is especially important to mimic native behavior for user interaction. For instance, as discussed before, either the control key or the command key should be used to implement keyboard shortcuts depending on the operating system. To check for specific feature support, it is favorable to use duck typing within each class.

Settings

The settings class stores settings required for Type's modules, for instance the `id` of the DOM-container which all helper DOM-nodes from other classes will be appended to.

Development

The `Development` class is intended to contain utility methods to support the development of the library. As for the development of this thesis, it was sufficient to implement logging methods.

13.15 Cache

For traversing the text, for example when the caret moves, the text will need to be measured. All measurements can be stored to a cache to only perform the same measurement operations once. When the user edits the contents of the editor, these texts will change and the cache must be updated. A cache must also account for external changes. The DOM3 Events specification [W3C, 2015] offers `MutationObservers` to check for DOM changes. This feature is not supported by Internet Explorer version 10 or less[12]. Internet Explorer 9 and 10 offer an implementation for `MutationEvents`[13]. The W3C states that ”The MutationEvent interface [...] has not yet been completely and interoperably implemented across user agents. In addition, there have been critiques that the interface, as designed, introduces a performance and implementation challenge.” [W3C, 2015, Legacy MutationEvent events]. For this reason, the editor does not use any caching. Implementing an editor that is stateless in regards of its contents can also improve stability.

13.16 Real-time collaboration with Etherpad

Overview

To achieve real-time collaboration with multiple Type editors, Etherpad[14] can be used. Etherpad is a web-based collaborative real-time text editor with rich-text capabilities. It provides a server, written in JavaScript using Node.js as well as a web-based rich-text editor. Both components are distributed in one package and are meant to be used together.

To achieve real-time rich-text collaboration, multiple web-based clients communicate with a server via WebSockets using socket.io. Each client owns a local version of the document that it needs to sync with the server. The server uses an operational transformation algorithm to merge each change to the document accounting for all changes and then urges each client to update their local contents according to the final document.

Changesets

For this, Etherpad uses the concept of so-called ”changesets”. Each client sends its local changes, debounced to an interval of 500 milliseconds, as a serialized string—the changeset—to the server. The changeset includes all text

[12] `http://caniuse.com/#search=mutation`, last checked on 07/21/2015

[13] `http://help.dottoro.com/ljfvvdnm.php#additionalEvents`, last checked on 07/21/2015

[14] Etherpad has been completely rewritten under the name Etherpad Lite. However, its official website no longer links to its former source code. For simplicity, the name Etherpad will be used, referring to its rewrite as Etherpad Lite

insertions, removals and formattings of the last time frame. Along with the changeset, it sends a document revision number that the changeset is based on to the server. The document revision number increases with every changeset that has been accepted and applied to the document on the server side. The document on the server side is saved as a stack of changesets, which ultimately form the current document. For performance reasons, snapshots can be taken that save the document as formatted text.

Based on the revision number that the client provides with the changeset, the server can apply it to the version of the document the client was working on. The server will apply the resulting changes to all newer revisions of the document (if present) and send a changeset and the latest revision number back to the client. The changeset sent to the client includes all operations it needs to perform to update its local version to the newest version on the server.

As a last step, the client must apply the changeset it got from the server to its local document to display the most recent version to the user and update its local revision number to what it got from the server.

Merging

In a collaborative environment, it can happen that two (or more) clients send different changesets to the server that are based on the same document revision. It is the responsibility of the server to merge both changes so that it preserves either intend. As explained in the "Etherpad and EasySync Technical Manual"[citation needed], to solve this, for a document X with the conflicting changesets A and B, the server computes the new changesets A' and B' such that

$$XAB' = XBA' = Xm(A, B)$$

where $Xm(A, B)$ is the merge of A and B applied to the document X. The changesets A' and B' will be sent to the respective clients, which will apply it to their local documents to sync with the document on the server. To compute a changeset A'

- Insertions in B become retained characters in A'

- Insertions in A stay insertions in A'
- Retain whatever characters are retained in *both* A and B

For B' this applies vice versa.

Etherpad Client implementation

Clients interact with the server via WebSockets using socket.io. To sync their own changes with other clients, a client does 4 things.

- Request a the full document from the server
- Send a changeset to the server
- Receive acknowledgement from the server for a submitted changeset
- Receive a changeset from the server submitted by another client

When Etherpad's client connects to the server it receives an initial snapshot of the entire document as a string. To submit changes, the client uses a three-step architecture. The client stores any local changes that have not been sent to the server yet in a changeset Y. Any changeset sent to the server must be acknowledged by the server as it has applied the changeset to its document. Any changeset that has been sent and not been acknowledged yet will be stored in a as the changeset X. The document as it is acknowledged by the server is stored in a changeset A. The document visible to the users can be expressed by the representation of $Y \cdot X \cdot A$, i.e. applying each changeset to the next.

Whenever a user applies a local change, the changeset Y will be updated. Every 500 milliseconds, but not before a changeset submitted to the server as been acknowledged, the changeset Y will be sent to the server and Y will be assigned to X. Y will be set to a changeset that contains no changes. When the client hears the acknowledgement for X from the server, X will be applied the changeset A and X will be set to contain no changes.

This architecture supports receiving changesets from other clients as they must be applied to a client's local changes (committed and uncommitted) as well as the document version as acknowledged by the server. When a client receives another client's changeset B it will perform 4 steps.

1. Compute a new changeset by merging $Y(X \cdot B)$ and apply it to the document visible to the user.

2. Apply B to A.

3. Compute a new changeset by merging B and X and assign it to X

4. Compute a new changeset by merging $(X \cdot B) \cdot Y$ and assign it to Y

The operations needed to merge the changesets on the client, are the same operations for merging changesets on the server.

Type Client implementation

Etherpad's technology can be used to enable real-time collaboration for Type. While Etherpad offers a web-based client, its implementation has three flaws:

1. It cannot be integrated easily in other web applications.

2. It does not generate semantic markup. It is cluttered with control sequences.

3. It is hard to extend.

Etherpad does not provide a documentation on its client–server protocol, but it can be reverse engineered. It is possible for third-party libraries to communicate with an Etherpad server alongside Etherpad's ”native” clients, as long as a third-party library (like Type) conforms the protocol.

Etherpad's collaboration functionality comes with a cost in file size for Type and may only be used in specific use cases. This is why this feature is implemented as an optional extension (compare **13.17: Extending**) in a separate file. To enable collaboration the designated JavaScript file needs to be added to the website.

```
<script src="type.js"></script>
<script src="type.etherpad.js"></script>
```

Listing 13.8: Enabling real-time collaboration to Type

`type.etherpad.js` adds the classes it requires to the `Type` namespace and adds a static constructor to the Type library:

```
1 var element = document.getElementById("myElement");
2 var editor = Type.fromEtherpad(element, "http://example.com/editor/myEditorId");
```

Listing 13.9: Static constructor to generate a collaborative Type instance

The constructor used in line 2 of *Listing 13.9* will connect to an Etherpad server and append the contents from the server to the element given as first parameter.

Type's implementation for collaboration is simpler than Etherpad's native implementation. The `EtherpadClient` class sends and receives changes to and from an Etherpad server. The `EtherpadChangeset` class translates changesets from the server to edit commands for the local editor and vice versa. A changeset can contain multiple commands. For every command there is class representing it. Text insertions are represented by the `Insertion` class, deletions are represented by the `Removal` class and formattings are represented by the `Formatting` class. Every class implements a function to apply its changes to the editor as well as to return a fragment of a changeset representing its change.

When a changeset will be received it will be translated to these classes and they will be applied to the editor. When the local user changes the contents, the according class will be instantiated and added to a `EtherpadChangeset` class. Every 500ms this changeset will be serialized and sent to the Etherpad server using the fragments as returned by the command instances.

When a submitted change had to be merged with another change on the server, Type's client will receive a change like any other change from other clients and it will be applied to the local editor.

This architecture provides an unobtrusive way to integrate real-time collaboration in the Type library. It does not depend on a specific implementation of an editor. Developers are free to implement any editor specific to their needs with integrated real-time collaboration.

13.17 Extending

Overview

Type’s modular structure is designed for extension. Type’s `prototype` has been exposed as `Type.fn` and all its classes in the `Type` namespace. This provides other developers with all of Type’s functionality in a structured and accessible manner. Type is designed to lower the barrier for and encourage developers to extend Type by giving freedom and possibilities in how to implement an extension and trying to avoid compulsorily use of interfaces or configurations.

jQuery demonstrates a similarly liberal approach for writing extensions and experience shows that name conflicts are minimal and ”good” extensions are naturally favored over ”bad” extensions by the community of web developers.

API extension

```
Type.fn.myMethod = function () {};
```

Listing 13.10: Example Type instance API extension

As discussed in **13.3: Exposal of Type’s `prototype`**, to add a method to Type’s public API, its base class’ prototype can be extended with a function using the `Type.fn` shorthand attribute. Static constructors can be added by extending the `Type Function` object.

```
Type.myConstructor = function () {
  return new Type();
};
```

Listing 13.11: Example custom static constructor

Namespace extension

As discussed in **13.3: Namespace and references**, to implement extensions for Type, the `Type` namespace can be used to add custom classes or sub-namespaces.

```
Type.MyClass = function () {
  var caret = new Type.Caret();
};
```

Listing 13.12: Example Type namespace extension and usage of a built-in class.

All other classes that Type uses are exposed in this namespace and can be used by extensions.

Plugin API

A plugin may need to be initialized when an editor will be instantiated. To support this, Type will trigger an event on instantiation and pass the Type instance to the event handler

```
Type.on('ready', function(typeInstance) {});
```

Listing 13.13: Example event handler for a Type instantiation

To store and read data specific to an instance, Type offers the `data` method, that will return an `Object` for arbitrary access.

```
Type.fn.myMethod = function () {
  this.data("myPlugin").foo = 'bar';
  var bar = this.data("myPlugin").foo;
};
```

Listing 13.14: Example calls to format text

To give each plugin an own namespace, an arbitrary identifier must be passed as a `String` to the `data` method, which will provide a unique `Object` for different string identifiers. This can possibly cause name conflicts if two plugins choose to use the same string. Developers are advised to always use their own extension name as identifier. Experience with jQuery's plugin system as well as jQuery's `data` method shows that while this cannot prevent name conflicts, it is rarely a problem.

Part V

Conclusion

Chapter 14

Evaluation

The library of this thesis demonstrates a way to implement a web-based rich-text editor without using HTML editing APIs. Developers can instantiate editable areas and manipulate its contents with an API. They can enable their users to:

1. Type and delete text.
2. Apply formattings.
3. Place and move a caret.
4. Create a selection.
5. Copy and paste text.
6. Use undo and redo commands.

Unfortunately, not all features are perfectly stable and not all of the goals could be met. On the client side, the selection shows glitches and formatting commands are not fully implemented. They do not distinguish between inline and block formattings and tags, to format the content with, cannot contain attributes. This means that text cannot be styled with CSS, which takes away many formatting options, and that links cannot be added. The contents pasted from the clipboard cannot be sanitized yet.

An editor that is connected to an Etherpad server will add strings sent from other clients as well as remove text that has been removed by others.

It can receive text formattings, although this throws errors in some cases. Text that has been added and removed locally will be added and removed on other clients, although the algorithm breaks on new lines (i.e. text cannot be changed after a new line) and formattings will not be sent.

The best experience can be achieved on Google Chrome, other browsers require further testing.

The library lays a foundation to work with rich text and manage all components that are necessary for it. It provides a rich set of classes to improve and extend its functionality. The Etherpad extension, enabling real-time collaboration that can be used with any editor implemented with the library, is an example of that.

Chapter 15

Outlook

The library implements basic features of rich-text editing. In order to distribute the library for productive use some features need be completed and added.

15.1 Stability

Currently, basic rich-text editing is possible, but is unstable in some edge cases. The current features need to be tested and fixed for all major desktop browsers.

15.2 Features

The library only supports a basic set of rich-text editing features. The goals that have not been met—most importantly distinction between block and inline formattings and formattings with attributes—must be implemented. Above that, features known from other rich-text editors, especially adding media must be added.

15.3 Mobile support

The APIs the library uses for editing and formatting the text are available on mobile browsers, but interaction with touch devices has not been implemented.

It should not be difficult to add support for touch events and thereby add mobile support for the library.

15.4 Bi-directional text support

The library does not support bi-directional text yet. The Unicode Consortium published an algorithm [UC, 2015] on how to derive the visual order from a unicode string containing bi-directional text, which can be used to move the caret accordingly.

15.5 Markdown editing

The library can be used to implement a syntax highlighter for Markdown using input filters. For example, an input filter can insert a headline when a user types a hash mark at the beginning of a line or italicize text between two asterisks. This functionality can be distributed as an optional extension.

15.6 MVC & Document model

Towards the end of this thesis, Marijn Haverbeke, author of CodeMirror, started a crowd-funding campaign for supporting him to build a rich-text editor that does not rely on HTML editing APIs (called ProseMirror[1]) just like the library of this thesis. As discussed in **11.1: Model–view–controller**, Type does not rely on an architecture that abstracts the contents of the editor from the DOM. ProseMirror, by contrast, takes this approach. This restricts working with the editor to the capabilities of the internal model of the document and might make extending the editor complicated. On the other hand, it allows for a very stable rendering of the contents and switching between HTML or Markdown output. It is good to see both approaches being realized. By contrasting both ideas in a practical manner it might be easier to decide which approach is better for which purposes.

[1] `http://prosemirror.net/`, last checked on 08/24/2015

List of Figures

Listings

Bibliography

[Berners-Lee, 1995] Berners-Lee, T. (1995). Hypertext markup language - 2.0. RFC 1866.

[Coombs et al., 1987] Coombs, J. H., Renear, A. H., and DeRose, S. J. (1987). Markup systems and the future of scholarly text processing. *Commun. ACM*, 30(11):933–947. `http://doi.acm.org/10.1145/32206.32209`.

[Crockford, 2008] Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly Media / Yahoo Press.

[Goldfarb, 1990] Goldfarb, C. F. (1990). *The SGML Handbook*. Oxford University Press, Inc., New York, NY, USA.

[Harris, 2010] Harris, J. (2010). Google drive blog: What's different about the new google docs? `http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html`. [Online; accessed 2015-07-25].

[ISO, 1986] ISO (1986). Standard generalized markup language (sgml). Standard Generalized Markup Language (SGML).

[ISO, 2012] ISO (2012). Iso/iec 15445:2000. ISO/IEC 15445:2000.

[Koszuliński, 2012] Koszuliński, P. (2012). Paste as plain text contenteditable div & textarea (word/excel...) - stack overflow. `http://stackoverflow.com/questions/11240602/paste-as-plain-text-contenteditable-div-textarea-word-excel`. [Online; accessed 2015-07-25].

[Koszuliński, 2013] Koszuliński, P. (2013). javascript - contenteditable div vs. iframe in making a rich-text/wysiwyg editor - stack overflow. http://stackoverflow.com/questions/10162540/contenteditable-div-vs-iframe-in-making-a-rich-text-wysiwyg-editor. [Online; accessed 2015-07-25].

[Microsoft, 2000a] Microsoft (2000a). contenteditable property msdn. https://msdn.microsoft.com/en-us/library/ms533720(v=vs.85).aspx. [Online; accessed 2015-07-25].

[Microsoft, 2000b] Microsoft (2000b). designmode property msdn. https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx. [Online; accessed 2015-07-25].

[Microsoft, 2015a] Microsoft (2015a). Command identifiers (internet explorer). https://msdn.microsoft.com/en-us/library/ms533049(v=vs.85).aspx. [Online; accessed 2015-07-25].

[Microsoft, 2015b] Microsoft (2015b). Methods (internet explorer). https://msdn.microsoft.com/en-us/library/hh772123(v=vs.85).aspx. [Online; accessed 2015-07-25].

[Microsoft, 2015c] Microsoft (2015c). selection object (internet explorer). https://msdn.microsoft.com/en-us/library/ms535869(v=VS.85).aspx. [Online; accessed 2015-07-25].

[Mozilla, 2003] Mozilla (2003). Rich-text editing in mozilla | mdn. https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla. [Online; accessed 2015-07-25].

[Mozilla, 2015a] Mozilla (2015a). Content editable - web developer guide | mdn. https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_Editable. [Online; accessed 2015-07-25].

[Mozilla, 2015b] Mozilla (2015b). Customevent - web api interfaces | mdn. https://developer.mozilla.org/en/docs/Web/API/CustomEvent. [Online; accessed 2015-07-25].

[Mozilla, 2015c] Mozilla (2015c). Document.execcommand() - web api interfaces | mdn. `https://developer.mozilla.org/en-US/docs/Web/API/document/execCommand`. [Online; accessed 2015-07-25].

[Mozilla, 2015d] Mozilla (2015d). Midas - mozilla | mdn. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Midas`. [Online; accessed 2015-07-25].

[Mozilla, 2015e] Mozilla (2015e). Nodeiterator - web api interfaces | mdn. `https://developer.mozilla.org/en-US/docs/Web/API/NodeIterator`. [Online; accessed 2015-07-25].

[Mozilla, 2015f] Mozilla (2015f). Selection - web api interfaces | mdn. `https://developer.mozilla.org/en/docs/Web/HTML/Element`. [Online; accessed 2015-07-25].

[Mozilla, 2015g] Mozilla (2015g). Selection - web api interfaces | mdn. `https://developer.mozilla.org/en-US/docs/Web/API/Selection`. [Online; accessed 2015-07-25].

[Santos, 2014] Santos, N. (2014). Why contenteditable is terrible — medium engineering — medium. `https://medium.com/medium-eng/why-contenteditable-is-terrible-122d8a40e480`. [Online; accessed 2015-07-25].

[UC, 2015] UC, U. C. (2015). Unicode bidirectional algorithm. UBA.

[W3C, 1998] W3C (1998). Document object model (dom) level 1 specification.

[W3C, 1999] W3C (1999). Html 4.01 specification.

[W3C, 2012] W3C (2012). Web idl candidate recommendation.

[W3C, 2014] W3C (2014). Html5 specification.

[W3C, 2015] W3C (2015). Ui events w3c specification.

[Watson, 1992] Watson, D. G. (1992). Brief history of document markup. `http://chnm.gmu.edu/digitalhistory/links/pdf/chapter3/3.19a.pdf`. [Online; accessed 2015-07-28].

[WHATWG, 2015] WHATWG (2015). Html living standard.

Part VI

Appendix

Method	Description
execCommand	Executes a command.
queryCommandEnabled	Returns whether or not a given command can currently be executed.
queryCommandIndeterm	Returns whether or not a given command is in the indeterminate state.
queryCommandState	Returns the current state of a given command.
queryCommandSupported	Returns whether or not a given command is supported by the current document's range.
queryCommandValue	Returns the value for the given command.

Table .1: HTML Editing API

Example call	Description
type.caret()	Returns the offset of the caret.
type.caret('show')	Shows the caret.
type.caret('hide')	Hides the caret.
type.caret(10)	Moves the caret to the 10th character.
type.caret(10, 20)	Convenience function for type.select(10, 20).

Table .2: Type instance API: caret command

Example call	Description
type.selection()	Same as type.selection('html').
type.selection('text')	Returns the unformatted (plain) contents of the current selection.
type.selection('html')	Return the currently selected HTML.
type.selection(10)	Convenience function for type.caret(10).
type.selection(10, 20)	Selects characters 10 to 20.
type.selection(element)	Select an element.
type.selection(el1, el2)	Creates a selection between 2 elements.
type.selection(jQuery obj.)	Creates a selection between the first and last element in a jQuery object.
type.selection('save')	Returns an object that can be passed to type.selection('restore') to store and recreate a selection.
type.selection('restore', sel)	Takes an object returned by type.selection('save') as a second argument to recreate a stored selection.

Table .3: Type instance API: selection command

Example call	Description
type.insert(str)	Inserts formatted text at the caret's position. Will overwrite the current selection if there is one.
type.insert(str, 'text')	Inserts plain text at the caret's position removing all formattings from str. Will overwrite the current selection if there is one.
type.insert(str, 10)	Inserts str after the 10th character in the text.
type.insert(str, 10, 'text')	Same as type.insert(str, 10) but inserts unformatted text.

Table .4: Type instance API: insert command

Example call	Description
type.format(htmlString)	Formats the currently selected text with the markup passed as htmlString.
type.format(htmlString, 10, 20)	Formats the characters 10 to 20 in the text with the markup passed as htmlString.

Table .5: Type instance API: format command

Example call	Description
type.remove()	Deletes the currently selected text. Does nothing if there is no selection.
type.remove(5)	Removes 5 characters right of the caret's offset. Removes the first 5 characters of the selection if there is a text selection.
type.remove(-5)	Removes 5 characters left of the caret's offset. Removes the last 5 characters of the selection if there is a text selection.
type.remove(10, 20)	Will remove the text between the 10th and 20th character.

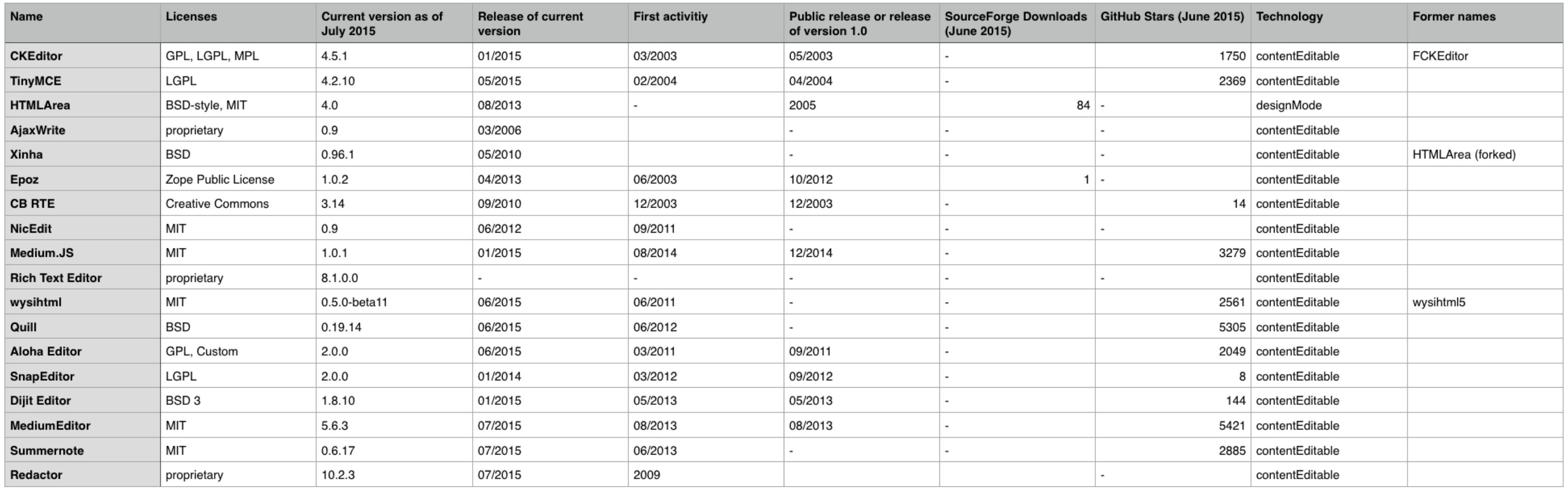
Table .6: Type instance API: remove command

Example call	Description
type.undo()	Revokes the user's last action.
type.undo(5)	Revokes the user's last 5 actions.
type.redo()	Reapplies a revoked action.
type.redo(5)	Reapplies 5 revoked actions.

Table .7: Type instance API: undo and redo commands

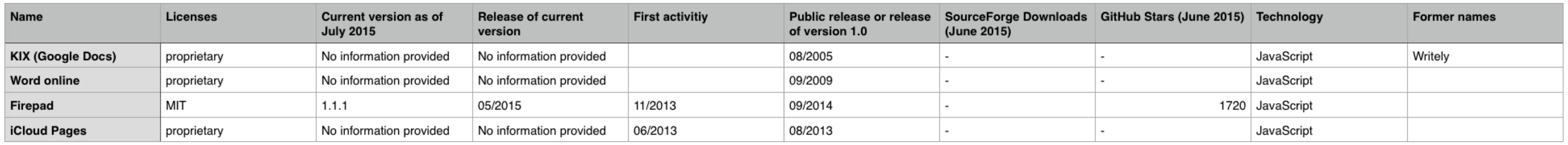
Example call	Description
type.options()	Returns all settings of an instance.
type.options(name)	Getter for a specific setting of an instance.
type.options(name, values)	Setter for a specific setting of an instance.
type.options({name: value})	Pass an object to set multiple settings of an instance.

Table .8: Type instance API: options command



Name	Licenses	Current version as of July 2015	Release of current version	First activitiy	Public release or release of version 1.0	SourceForge Downloads (June 2015)	GitHub Stars (June 2015)	Technology	Former names
CKEditor	GPL, LGPL, MPL	4.5.1	01/2015	03/2003	05/2003	-	1750	contentEditable	FCKEditor
TinyMCE	LGPL	4.2.10	05/2015	02/2004	04/2004	-	2369	contentEditable	
HTMLArea	BSD-style, MIT	4.0	08/2013	-	2005	84	-	designMode	
AjaxWrite	proprietary	0.9	03/2006		-	-	-	contentEditable	
Xinha	BSD	0.96.1	05/2010		-	-	-	contentEditable	HTMLArea (forked)
Epoz	Zope Public License	1.0.2	04/2013	06/2003	10/2012	1	-	contentEditable	
CB RTE	Creative Commons	3.14	09/2010	12/2003	12/2003	-	14	contentEditable	
NicEdit	MIT	0.9	06/2012	09/2011	-	-	-	contentEditable	
Medium.JS	MIT	1.0.1	01/2015	08/2014	12/2014	-	3279	contentEditable	
Rich Text Editor	proprietary	8.1.0.0	-	-	-	-	-	contentEditable	
wysihtml	MIT	0.5.0-beta11	06/2015	06/2011	-	-	2561	contentEditable	wysihtml5
Quill	BSD	0.19.14	06/2015	06/2012	-	-	5305	contentEditable	
Aloha Editor	GPL, Custom	2.0.0	06/2015	03/2011	09/2011	-	2049	contentEditable	
SnapEditor	LGPL	2.0.0	01/2014	03/2012	09/2012	-	8	contentEditable	
Dijit Editor	BSD 3	1.8.10	01/2015	05/2013	05/2013	-	144	contentEditable	
MediumEditor	MIT	5.6.3	07/2015	08/2013	08/2013	-	5421	contentEditable	
Summernote	MIT	0.6.17	07/2015	06/2013	-	-	2885	contentEditable	
Redactor	proprietary	10.2.3	07/2015	2009			-	contentEditable	

Figure .1: Editors using HTML editing APIs (selection)



Name	Licenses	Current version as of July 2015	Release of current version	First activitiy	Public release or release of version 1.0	SourceForge Downloads (June 2015)	GitHub Stars (June 2015)	Technology	Former names
KIX (Google Docs)	proprietary	No information provided	No information provided		08/2005	-	-	JavaScript	Writely
Word online	proprietary	No information provided	No information provided		09/2009	-	-	JavaScript	
Firepad	MIT	1.1.1	05/2015	11/2013	09/2014	-	1720	JavaScript	
iCloud Pages	proprietary	No information provided	No information provided	06/2013	08/2013	-	-	JavaScript	

Figure .2: Editors not using HTML editing APIs (selection)

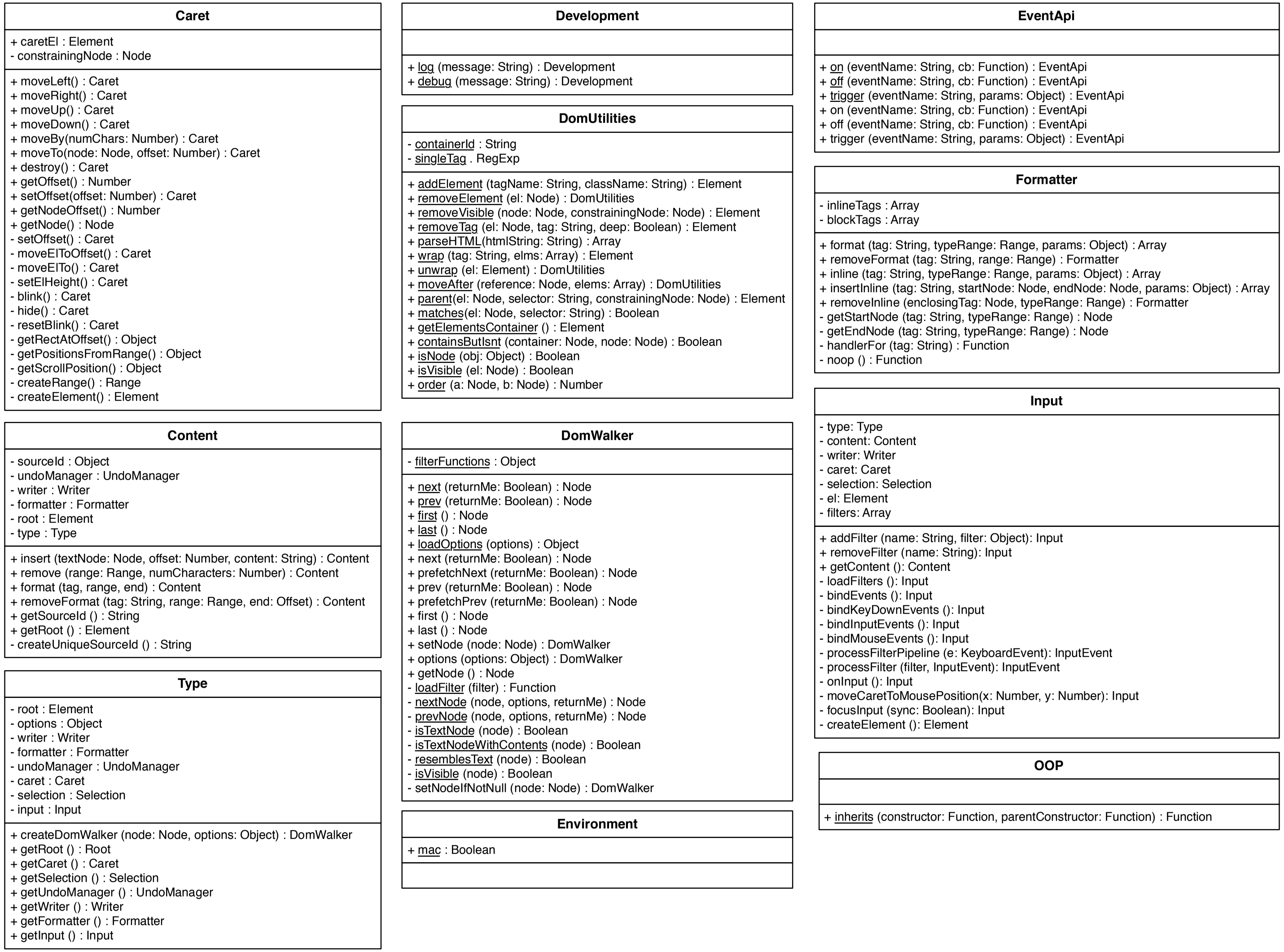

Caret
+ caretEl : Element
- constrainingNode : Node
+ moveLeft() : Caret
+ moveRight() : Caret
+ moveUp() : Caret
+ moveDown() : Caret
+ moveBy(numChars: Number) : Caret
+ moveTo(node: Node, offset: Number) : Caret
+ destroy() : Caret
+ getOffset() : Number
+ setOffset(offset: Number) : Caret
+ getNodeOffset() : Number
+ getNode() : Node
- setOffset() : Caret
- moveElToOffset() : Caret
- moveElTo() : Caret
- setElHeight() : Caret
- blink() : Caret
- hide() : Caret
- resetBlink() : Caret
- getRectAtOffset() : Object
- getPositionsFromRange() : Object
- getScrollPosition() : Object
- createRange() : Range
- createElement() : Element
Development
+ log (message: String) : Development
+ debug (message: String) : Development
EventApi
+ on (eventName: String, cb: Function) : EventApi
+ off (eventName: String, cb: Function) : EventApi
+ trigger (eventName: String, params: Object) : EventApi
+ on (eventName: String, cb: Function) : EventApi
+ off (eventName: String, cb: Function) : EventApi
+ trigger (eventName: String, params: Object) : EventApi
DomUtilities
- containerId : String
- singleTag . RegExp
+ addElement (tagName: String, className: String) : Element
+ removeElement (el: Node) : DomUtilities
+ removeVisible (node: Node, constrainingNode: Node) : Element
+ removeTag (el: Node, tag: String, deep: Boolean) : Element
+ parseHTML(htmlString: String) : Array
+ wrap (tag: String, elms: Array) : Element
+ unwrap (el: Element) : DomUtilities
+ moveAfter (reference: Node, elems: Array) : DomUtilities
+ parent(el: Node, selector: String, constrainingNode: Node) : Element
+ matches(el: Node, selector: String) : Boolean
+ getElementsContainer () : Element
+ containsButIsnt (container: Node, node: Node) : Boolean
+ isNode (obj: Object) : Boolean
+ isVisible (el: Node) : Boolean
+ order (a: Node, b: Node) : Number
Formatter
- inlineTags : Array
- blockTags : Array
+ format (tag: String, typeRange: Range, params: Object) : Array
+ removeFormat (tag: String, range: Range) : Formatter
+ inline (tag: String, typeRange: Range, params: Object) : Array
+ insertInline (tag: String, startNode: Node, endNode: Node, params: Object) : Array
+ removeInline (enclosingTag: Node, typeRange: Range) : Formatter
- getStartNode (tag: String, typeRange: Range) : Node
- getEndNode (tag: String, typeRange: Range) : Node
- handlerFor (tag: String) : Function
- noop () : Function
Content
- sourceId : Object
- undoManager : UndoManager
- writer : Writer
- formatter : Formatter
- root : Element
- type : Type
+ insert (textNode: Node, offset: Number, content: String) : Content
+ remove (range: Range, numCharacters: Number) : Content
+ format (tag, range, end) : Content
+ removeFormat (tag: String, range: Range, end: Offset) : Content
+ getSourceId () : String
+ getRoot () : Element
- createUniqueSourceId () : String
DomWalker
- filterFunctions : Object
+ next (returnMe: Boolean) : Node
+ prev (returnMe: Boolean) : Node
+ first () : Node
+ last () : Node
+ loadOptions (options) : Object
+ next (returnMe: Boolean) : Node
+ prefetchNext (returnMe: Boolean) : Node
+ prev (returnMe: Boolean) : Node
+ prefetchPrev (returnMe: Boolean) : Node
+ first () : Node
+ last () : Node
+ setNode (node: Node) : DomWalker
+ options (options: Object) : DomWalker
+ getNode () : Node
- loadFilter (filter) : Function
- nextNode (node, options, returnMe) : Node
- prevNode (node, options, returnMe) : Node
- isTextNode (node) : Boolean
- isTextNodeWithContents (node) : Boolean
- resemblesText (node) : Boolean
- isVisible (node) : Boolean
- setNodeIfNotNull (node: Node) : DomWalker
Input
- type: Type
- content: Content
- writer: Writer
- caret: Caret
- selection: Selection
- el: Element
- filters: Array
+ addFilter (name: String, filter: Object): Input
+ removeFilter (name: String): Input
+ getContent (): Content
- loadFilters (): Input
- bindEvents (): Input
- bindKeyDownEvents (): Input
- bindInputEvents (): Input
- bindMouseEvents (): Input
- processFilterPipeline (e: KeyboardEvent): InputEvent
- processFilter (filter, InputEvent): InputEvent
- onInput (): Input
- moveCaretToMousePosition(x: Number, y: Number): Input
- focusInput (sync: Boolean): Input
- createElement (): Element
Type
- root : Element
- options : Object
- writer : Writer
- formatter : Formatter
- undoManager : UndoManager
- caret : Caret
- selection : Selection
- input : Input
+ createDomWalker (node: Node, options: Object) : DomWalker
+ getRoot () : Root
+ getCaret () : Caret
+ getSelection () : Selection
+ getUndoManager () : UndoManager
+ getWriter () : Writer
+ getFormatter () : Formatter
+ getInput () : Input
Environment
+ mac : Boolean
OOP
+ inherits (constructor: Function, parentConstructor: Function) : Function

Figure .3: Classes of the library 1/3

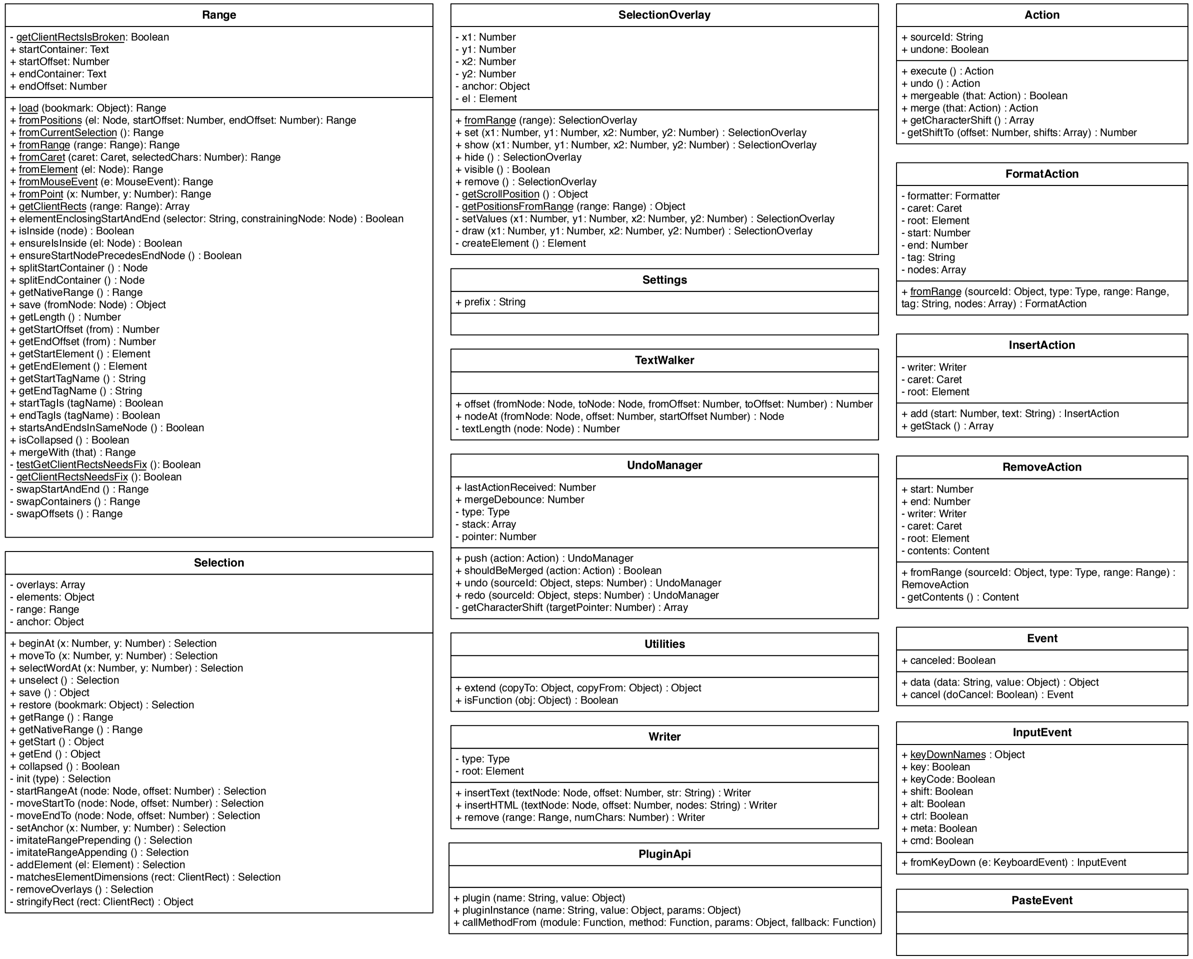

Range
- getClientRectsIsBroken: Boolean
+ startContainer: Text
+ startOffset: Number
+ endContainer: Text
+ endOffset: Number
+ load (bookmark: Object): Range
+ fromPositions (el: Node, startOffset: Number, endOffset: Number): Range
+ fromCurrentSelection (): Range
+ fromRange (range: Range): Range
+ fromCaret (caret: Caret, selectedChars: Number): Range
+ fromElement (el: Node): Range
+ fromMouseEvent (e: MouseEvent): Range
+ fromPoint (x: Number, y: Number): Range
+ getClientRects (range: Range): Array
+ elementEnclosingStartAndEnd (selector: String, constrainingNode: Node) : Boolean
+ isInside (node) : Boolean
+ ensureIsInside (el: Node) : Boolean
+ ensureStartNodePrecedesEndNode () : Boolean
+ splitStartContainer () : Node
+ splitEndContainer () : Node
+ getNativeRange () : Range
+ save (fromNode: Node) : Object
+ getLength () : Number
+ getStartOffset (from) : Number
+ getEndOffset (from) : Number
+ getStartElement () : Element
+ getEndElement () : Element
+ getStartTagName () : String
+ getEndTagName () : String
+ startTagIs (tagName) : Boolean
+ endTagIs (tagName) : Boolean
+ startsAndEndsInSameNode () : Boolean
+ isCollapsed () : Boolean
+ mergeWith (that) : Range
- testGetClientRectsNeedsFix (): Boolean
- getClientRectsNeedsFix (): Boolean
- swapStartAndEnd () : Range
- swapContainers () : Range
- swapOffsets () : Range
Selection
- overlays: Array
- elements: Object
- range: Range
- anchor: Object
+ beginAt (x: Number, y: Number) : Selection
+ moveTo (x: Number, y: Number) : Selection
+ selectWordAt (x: Number, y: Number) : Selection
+ unselect () : Selection
+ save () : Object
+ restore (bookmark: Object) : Selection
+ getRange () : Range
+ getNativeRange () : Range
+ getStart () : Object
+ getEnd () : Object
+ collapsed () : Boolean
- init (type) : Selection
- startRangeAt (node: Node, offset: Number) : Selection
- moveStartTo (node: Node, offset: Number) : Selection
- moveEndTo (node: Node, offset: Number) : Selection
- setAnchor (x: Number, y: Number) : Selection
- imitateRangePrepending () : Selection
- imitateRangeAppending () : Selection
- addElement (el: Element) : Selection
- matchesElementDimensions (rect: ClientRect) : Selection
- removeOverlays () : Selection
- stringifyRect (rect: ClientRect) : Object
SelectionOverlay
- x1: Number
- y1: Number
- x2: Number
- y2: Number
- anchor: Object
- el : Element
+ fromRange (range): SelectionOverlay
+ set (x1: Number, y1: Number, x2: Number, y2: Number) : SelectionOverlay
+ show (x1: Number, y1: Number, x2: Number, y2: Number) : SelectionOverlay
+ hide () : SelectionOverlay
+ visible () : Boolean
+ remove () : SelectionOverlay
- getScrollPosition () : Object
- getPositionsFromRange (range: Range) : Object
- setValues (x1: Number, y1: Number, x2: Number, y2: Number) : SelectionOverlay
- draw (x1: Number, y1: Number, x2: Number, y2: Number) : SelectionOverlay
- createElement () : Element
Settings
+ prefix : String
TextWalker
+ offset (fromNode: Node, toNode: Node, fromOffset: Number, toOffset: Number) : Number
+ nodeAt (fromNode: Node, offset: Number, startOffset Number) : Node
- textLength (node: Node) : Number
UndoManager
+ lastActionReceived: Number
+ mergeDebounce: Number
- type: Type
- stack: Array
- pointer: Number
+ push (action: Action) : UndoManager
+ shouldBeMerged (action: Action) : Boolean
+ undo (sourceId: Object, steps: Number) : UndoManager
+ redo (sourceId: Object, steps: Number) : UndoManager
- getCharacterShift (targetPointer: Number) : Array
Utilities
+ extend (copyTo: Object, copyFrom: Object) : Object
+ isFunction (obj: Object) : Boolean
Writer
- type: Type
- root: Element
+ insertText (textNode: Node, offset: Number, str: String) : Writer
+ insertHTML (textNode: Node, offset: Number, nodes: String) : Writer
+ remove (range: Range, numChars: Number) : Writer
PluginApi
+ plugin (name: String, value: Object)
+ pluginInstance (name: String, value: Object, params: Object)
+ callMethodFrom (module: Function, method: Function, params: Object, fallback: Function)
Action
+ sourceId: String
+ undone: Boolean
+ execute () : Action
+ undo () : Action
+ mergeable (that: Action) : Boolean
+ merge (that: Action) : Action
+ getCharacterShift () : Array
- getShiftTo (offset: Number, shifts: Array) : Number
FormatAction
- formatter: Formatter
- caret: Caret
- root: Element
- start: Number
- end: Number
- tag: String
- nodes: Array
+ fromRange (sourceId: Object, type: Type, range: Range, tag: String, nodes: Array) : FormatAction
InsertAction
- writer: Writer
- caret: Caret
- root: Element
+ add (start: Number, text: String) : InsertAction
+ getStack () : Array
RemoveAction
+ start: Number
+ end: Number
- writer: Writer
- caret: Caret
- root: Element
- contents: Content
+ fromRange (sourceId: Object, type: Type, range: Range) : RemoveAction
- getContents () : Content
Event
+ canceled: Boolean
+ data (data: String, value: Object) : Object
+ cancel (doCancel: Boolean) : Event
InputEvent
+ keyDownNames : Object
+ key: Boolean
+ keyCode: Boolean
+ shift: Boolean
+ alt: Boolean
+ ctrl: Boolean
+ meta: Boolean
+ cmd: Boolean
+ fromKeyDown (e: KeyboardEvent) : InputEvent
PasteEvent

Figure .4: Classes of the library 2/3

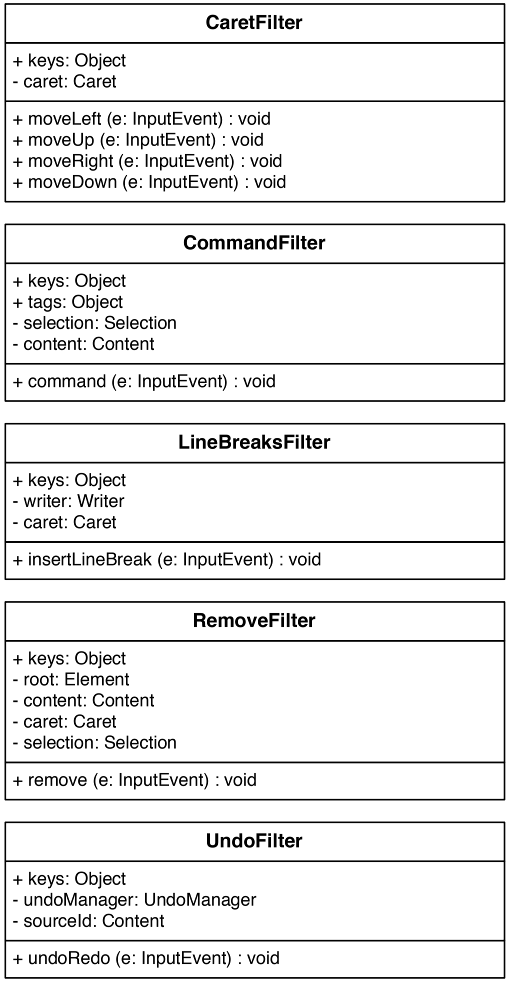


Figure .5: Classes of the library 3/3

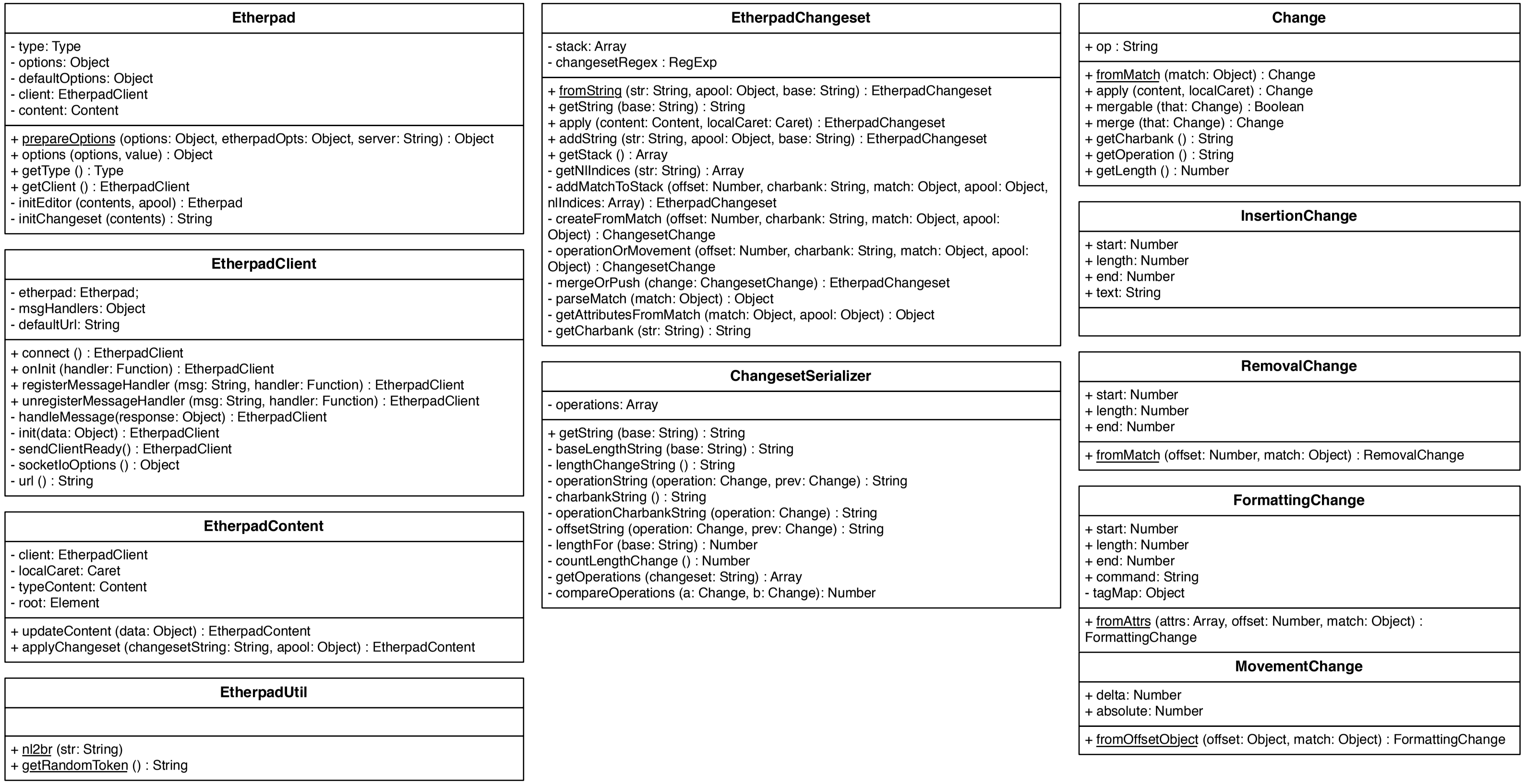

Etherpad
- type: Type
- options: Object
- defaultOptions: Object
- client: EtherpadClient
- content: Content
+ prepareOptions (options: Object, etherpadOpts: Object, server: String) : Object
+ options (options, value) : Object
+ getType () : Type
+ getClient () : EtherpadClient
- initEditor (contents, apool) : Etherpad
- initChangeset (contents) : String
EtherpadClient
- etherpad: Etherpad;
- msgHandlers: Object
- defaultUrl: String
+ connect () : EtherpadClient
+ onInit (handler: Function) : EtherpadClient
+ registerMessageHandler (msg: String, handler: Function) : EtherpadClient
+ unregisterMessageHandler (msg: String, handler: Function) : EtherpadClient
- handleMessage(response: Object) : EtherpadClient
- init(data: Object) : EtherpadClient
- sendClientReady() : EtherpadClient
- socketIoOptions () : Object
- url () : String
EtherpadContent
- client: EtherpadClient
- localCaret: Caret
- typeContent: Content
- root: Element
+ updateContent (data: Object) : EtherpadContent
+ applyChangeset (changesetString: String, apool: Object) : EtherpadContent
EtherpadUtil
+ nl2br (str: String)
+ getRandomToken () : String
EtherpadChangeset
- stack: Array
- changesetRegex : RegExp
+ fromString (str: String, apool: Object, base: String) : EtherpadChangeset
+ getString (base: String) : String
+ apply (content: Content, localCaret: Caret) : EtherpadChangeset
+ addString (str: String, apool: Object, base: String) : EtherpadChangeset
+ getStack () : Array
- getNlIndices (str: String) : Array
- addMatchToStack (offset: Number, charbank: String, match: Object, apool: Object, nlIndices: Array) : EtherpadChangeset
- createFromMatch (offset: Number, charbank: String, match: Object, apool: Object) : ChangesetChange
- operationOrMovement (offset: Number, charbank: String, match: Object, apool: Object) : ChangesetChange
- mergeOrPush (change: ChangesetChange) : EtherpadChangeset
- parseMatch (match: Object) : Object
- getAttributesFromMatch (match: Object, apool: Object) : Object
- getCharbank (str: String) : String
ChangesetSerializer
- operations: Array
+ getString (base: String) : String
- baseLengthString (base: String) : String
- lengthChangeString () : String
- operationString (operation: Change, prev: Change) : String
- charbankString () : String
- operationCharbankString (operation: Change) : String
- offsetString (operation: Change, prev: Change) : String
- lengthFor (base: String) : Number
- countLengthChange () : Number
- getOperations (changeset: String) : Array
- compareOperations (a: Change, b: Change): Number
Change
+ op : String
+ fromMatch (match: Object) : Change
+ apply (content, localCaret) : Change
+ mergable (that: Change) : Boolean
+ merge (that: Change) : Change
+ getCharbank () : String
+ getOperation () : String
+ getLength () : Number
InsertionChange
+ start: Number
+ length: Number
+ end: Number
+ text: String
RemovalChange
+ start: Number
+ length: Number
+ end: Number
+ fromMatch (offset: Number, match: Object) : RemovalChange
FormattingChange
+ start: Number
+ length: Number
+ end: Number
+ command: String
- tagMap: Object
+ fromAttrs (attrs: Array, offset: Number, match: Object) : FormattingChange
MovementChange
+ delta: Number
+ absolute: Number
+ fromOffsetObject (offset: Object, match: Object) : FormattingChange

Figure .6: Classes of the Etherpad Extension

Part VII

Source Code

Source Code

```
'use strict';

/**
 * Creates a new Type editor and sets up the core
 * modules used for WYSIWYG editing. The core
 * class only holds methods for setting and retrieving
 * options as well getters and setters for instances
 * of core modules.
 *
 * @param {{}|Element} options - Either pass
 *     an associative array with options for this
 *     editor or the root element that should be
 *     used to modify its contents for WYSIWYG
 *     editing
 * @param {Element} options.el The root element
 *     that should be used to modify its contents
 *     for WYSIWYG editing
 * @constructor
 */
function Type(options) {

  // Allow passing an element as only parameter
  if (Type.DomUtilities.isNode(options)) {
    options = { el: options };
  }

  // If no element has been passed, interrupt
  if (!options.el) {
    throw new Error('You must provide an element as root node for the editor\'s TypeContents.');
  }

  // Set settings for this editor
  this._root = null;
  this.options(options);

  // Set up core modules
  this._writer       = new Type.Writer(this);
  this._formatter    = new Type.Formatter(this);
  this._undoManager  = new Type.UndoManager(this);
  //this._content      = new Type.Content(this);
  this._caret        = new Type.Caret(this);
  this._selection    = new Type.Selection(this);
  this._input        = new Type.Input(this);

  // Trigger events
  Type.trigger('ready', this);

}

(function () {

  /**
   * Allows fast detection if an object is a Type Editor
   * instance (or class)
   *
   * @type {boolean}
   */
  this.typeEditor = true;

  /**
   * Holds the default options for every editor. These options
   * will be extended by the options passed to each instance
   * on instantiation.
   *
   * @type {{el: null, undoSteps: number}}
   * @private
   */
  this._defaultOptions = {
    el        : null,
    undoSteps : 20
  };

  /**
   * Sets or gets the options to be used by this Type instance.
   * Parameters can be passed as you know it from jQuery:
   *
   * Pass a single string to get an option:
   * this.options('el')
   * returns your editor's TypeContents baseelement
   *
   * Pass a name value combination to set a specific option
   * this.options('el', myElement)
   * sets the base element
   *
   * Pass an object to set multiple options
   * this.options({el: myElement, foo:bar})
   * sets both parameters
   *
   * @param {(string|Object)} options - Either a plain object
   *     with keys and values to be set or a string that will
   *     be used as a name for a option. If you pass a string
   *     pass a second parameter to set that option or no
   *     second parameter to retrieve that option.
   * @param {*} [value] - If the first parameter is a string,
 *     this value will be set to the key of the given first
 *     parameter. Any arbitrary value can be set.
 * @returns {Type|*} Returns the type instance if you set an
 *     option or the according value if you get an option
 */
this.options = function (options, value) {

  // Load default options if there are no instance options yet
  this._options = this._options || Type.Utilities.extend({}, this._defaultOptions);

  // Pass a single option name to fetch it
  if (typeof options === "string" && arguments.length === 1) {
    return this._options[options];
  }

  // Pass an option name and a value to set it
  if (typeof options === "string" && arguments.length === 2) {
    options = {options: value};
  }

  // Pass an object of key-values to set them
  if (typeof options === "object") {
    Type.Utilities.extend(this._options, options);
  }

  // If the el option has been passed copy it for quick access
  if (options.el) {
    this._root = options.el;
  }

  // Chaining
  return arguments.length ? this : this._options;
  //return this;

};

/**
 * Creates a {Type.DomWalker} that ist constrained to this
 * instance's root element unless you explicitly pass a
 * constrainingNode as argument. All other DomWalker options
 * can also be passed to this as usual.
 *
 * @param {Node} node - Any DOM {Node} to be set as starting
 *     node for the DomWalker
 * @param {Node|string|Function|{constrainingNode: Node, filter: string|Function}} [options]
 *     See {Type.DomWalker} for a description of possible arguments
 * @returns {Type.DomWalker}
 */
this.createDomWalker = function (node, options) {
  options = Type.DomWalker.loadOptions(options || {});
  options.constrainingNode = options.constrainingNode || this._root;
  return new Type.DomWalker(node, options);
};

/**
 * Getter for this instance's root element, i.e. the
 * element that contains this editor's text.
 * @returns {Element}
 */
this.getRoot = function () {
  return this._root;
};

/**
 * Getter for this instance's caret.
 * @returns {Type.Caret}
 */
this.getCaret = function () {
  return this._caret;
};

/**
 * Getter for this instance's selection.
 * @returns {Type.Selection}
 */
this.getSelection = function () {
  return this._selection;
};

/**
 * Getter for this instance's content.
 * @returns {Type.UndoManager}
 */
this.getUndoManager = function () {
  return this._undoManager;
};

/**
 * Getter for this instance's content.
 * @returns {Type.Content}
 */
```

```
  //this.getContent = function () {
  //  return this._content;
  //};

  /**
   * Getter for this instance's writer.
   * @returns {Type.Writer}
   */
  this.getWriter = function () {
    return this._writer;
  };

  /**
   * Getter for this instance's formatter.
   * @returns {Formatter}
   */
  this.getFormatter = function () {
    return this._formatter;
  };

  /**
   * Getter for this instance's input.
   * @returns {Type.Input}
   */
  this.getInput = function () {
    return this._input;
  };

}).call(Type.prototype);

/**
 * Exposes Type's prototype as jQuery-style shorthand variable
 * @type {Object}
 */
Type.fn = Type.prototype;

/**
 * The namespace for Type events
 * @type {{}}
 */
Type.Events = {};

/**
 * The namespace for Type actions
 * @type {{}}
 */
Type.Actions = {};

/**
 * Module Exports for CommonJs
 * @type {Type}
 */
module.exports = Type;
```

Listing 1: core.js

```
'use strict';

var Type = require('./core');

/**
 * Holds messages for developing and debugging Type
 * @constructor
 */
Type.Development = function () {
};

(function () {

  /**
   * Prints a message to the console if the browser's
   * console offers the log method.
   *
   * @param {...*} messages - Any number and type of arguments
   *      you want to pass to console.debug
   */
  Type.Development.log = function (messages) {
    if (console && console.log) {
      console.log.apply(console, arguments)
    }
    return Type.Development;
  };

  /**
   * Prints a debug message to the console if the browser's
   * console offers a debug method.
   *
   * @param {...*} messages - Any number and type of arguments
   *      you want to pass to console.debug
   */
  Type.Development.debug = function (messages) {
    if (console && console.debug) {
      console.debug.apply(console, arguments)
    }
    return Type.Development;
  };

}).call(Type.Development);

module.exports = Type.Development;
```

Listing 2: development.js

```
'use strict';

var Type = require('./core');

Type.Settings = {
  prefix : 'typejs-'
};

module.exports = Type.Settings;
```

Listing 3: settings.js

```
'use strict';

var Type = require('./core');

Type.OOP = function () {};

(function () {

  /**
   * Implements classical inheritance for the constructor pattern
   *
   * @param {Function} constructor - The child class that shall
   *     inherit attributes and methods
   * @param {Function} parentConstructor - The parent class that
   *     shall be inherited from
   * @returns {Function} The child class that inherited
   */
  Type.OOP.inherits = function (constructor, parentConstructor) {

    // Required variables
    var key;

    // Inherit instance attributes and methods
    constructor.prototype = Object.create(parentConstructor.prototype);
    constructor.prototype.constructor = constructor;

    // Inherit static attributes and methods
    for (key in parentConstructor) {
      if (parentConstructor.hasOwnProperty(key))
        constructor[key] = parentConstructor[key];
    }

    // Add parent / super property
    constructor._super = parentConstructor;

    // Return the inheriting class for convenience
    return constructor;

  };

}).call(Type.OOP);

module.exports = Type.OOP;
```

Listing 4: oop.js

```
'use strict';

var Type = require('./core');

// todo Write a method that takes an object (Type and Type.prototype) and attaches these event methods to it <- no
// todo Eine Klasse schreiben die die unteren funktionen im Prototype hat. Dann kann Type core seinen prototype und sein function object mit dem prototype extenden (dafuer muss es eine extend methode geben die nur diese funkionen und nicht den ganzen autmoatischen muell mitkopiert).

/**
 * Methods for Type.js instance events
 */
(function () {

  /**
   * Register a callback for a Type specific event
   *
   * @param {String} eventName - The name of the event on which you wish the
   *     function to be called
   * @param {Function} cb - The function you wish to be called on the event
   * @returns {Type}
   */
  this.on = function (eventName, cb) {
    this.eventCallbacks = this.eventCallbacks || {};
  this.eventCallbacks[eventName] =
       this.eventCallbacks[
      eventName] || [];
  this.eventCallbacks[eventName].
      push(cb);
  return this;
};

/**
 * Unregister a callback for a
     Type specific event
 *
 * @param {String} eventName - The
      name of the event on which
     you wish the
 *     for which you no longer
     wish to call the function
 * @param {Function} cb - The
     function you no longer wish
     to be called
 * @returns {Type}
 */
this.off = function (eventName, cb
    ) {
  this.eventCallbacks = this.
      eventCallbacks || {};
  var index = this.eventCallbacks[
      eventName] ? this.
      eventCallbacks[eventName].
      indexOf(cb) : -1;
  if (index > -1) {
    this.eventCallbacks[eventName
        ].splice(index, 1);
  }
  return this;
};

/**
 * Trigger a Type specific event
     to call all callbacks for
 *
 * @param {String} eventName - The
      name of the event on which
     you wish to
 *     call its callbacks for
 * @param {...*} params -
     Arbitrary parameters you
     wish to pass to the
 *     callbacks
 * @returns {Type}
 */
this.trigger = function (eventName
    , params) {
  var i;
  this.eventCallbacks = this.
      eventCallbacks || {};
  if (this.eventCallbacks[
      eventName]) {
    for (i = 0; i < this.
        eventCallbacks[eventName
        ].length; i += 1) {
      this.eventCallbacks[
          eventName][i].apply(
          this, params);
    }
  }
  return this;
};

}).call(Type.fn);


/**
 * Global Type.js events
 * Todo Remove code duplication
 */
(function () {

  /**
   * Register a callback for a
       global Type event
   *
   * @param {String} eventName - The
        name of the event on which
       you wish the
   *     function to be called
   * @param {Function} cb - The
       function you wish to be
       called on the event
   * @returns {Type}
   */
  this.on = function (eventName, cb)
      {
    this.eventCallbacks = this.
        eventCallbacks || {};
    this.eventCallbacks[eventName] =
         this.eventCallbacks[
        eventName] || [];
    this.eventCallbacks[eventName].
        push(cb);
    return this;
  };

  /**
   * Unregister a callback for a
       global Type event
   *
   * @param {String} eventName - The
        name of the event on which
       you wish the
   *     for which you no longer
       wish to call the function
   * @param {Function} cb - The
       function you no longer wish
       to be called
   * @returns {Type}
   */
  this.off = function (eventName, cb
      ) {
    this.eventCallbacks = this.
        eventCallbacks || {};
    var index = this.eventCallbacks[
        eventName] ? this.
        eventCallbacks[eventName].
        indexOf(cb) : -1;
    if (index > -1) {
      this.eventCallbacks[eventName
          ].splice(index, 1);
    }
    return this;
  };

  /**
   * Trigger a global Type event to
       call all callbacks for
   *
   * @param {String} eventName - The
        name of the event on which
       you wish to
   *     call its callbacks for
   * @param {...*} params -
       Arbitrary parameters you
       wish to pass to the
   *     callbacks
   * @returns {Type}
   */
this.trigger = function (eventName
    , params) {
  var i;
  this.eventCallbacks = this.
      eventCallbacks || {};
  if (this.eventCallbacks[
      eventName]) {
    for (i = 0; i < this.
        eventCallbacks[eventName
        ].length; i += 1) {
      this.eventCallbacks[
          eventName][i].apply(
          this, params);
    }
  }
  return this;
};

}).call(Type);
```

Listing 5: event_api.js

```
'use strict';

var Type = require('./core');

(function () {

  /**
   * Get or set a plugin. Will
       return the plugin with the
       given
   * name. Pass a second parameter
       to set the plugin to the
   * given name.
   *
   * @param {string} name - The name
        of the plugin that should
   *     be gotten or set
   * @param {*} [value] - The value
       to be set for the plugin
   * @returns {*}
   */
  this.plugin = function (name,
      value) {
    this._plugins = this._plugins ||
        {};
    if (value !== null) {
      this._plugins[name] = value;
    }
    return this._plugins[name];
  };

/**
 * Get or set a plugin. There are
     2 essential differences to
 * this.plugin.
 *
 * 1) If the plugin given as name
     already exists, it will not
 * be set, even if you pass
     subsequent parameters.
 *
 * 2) If the value passed is a
     Function object (not an
     instance)
 * it will be instantiated with
     the given params and saved
 * under the given name. If value
     is an instantiated object it
 * will simply be written to name,
      just as this.plugin would.
 *
 * @param {string} name - The name
      of the plugin that should
 *     be gotten and set
 * @param {*} [value] - The value
     to be set for the plugin.
 *     If you pass an instance of
     a function, this instance
 *     will be set. If you pass an
      uninstantiated function,
 *     it will be instantiated.
 * @param {...*} [params] -
     Arguments passed to the
       instance
 *     that will be created for
       value
 * @returns {*}
 */
this.pluginInstance = function (
    name, value, params) {

  params = Array.prototype.slice.
      call(arguments, 2);
  this._plugins = this._plugins ||
      {};

  if (this._plugins[name]) {
    return this._plugins[name];
  }

  if (value instanceof Function) {
    this._plugins[name] = new (
        Function.prototype.bind.
        apply(value, params));

  } else {
    this._plugins[name] = value;
  }

  return this._plugins[name];
};

/**
```

```
    * Call a method from an object (
        usually a plugin). If the
        called
    * method returns the plugin,
        return this type instance
        instead.
    * If the given method name is not
         a method in the object,
        call the
    * given callback.
    *
    * callMethodFrom purpose is to
        provide a shorthand way to
        expose
    * the API of a plugin as API of
        Type
    *
    * @param module
    * @param method
    * @param params
    * @param fallback
    * @returns {Type|*}
    */
  this.callMethodFrom = function (
      module, method, params,
      fallback) {

    var result = null;

    if (module.hasOwnProperty(method
        )) {
      result = module[method].apply(
          module, params);

    } else if (fallback) {
      result = fallback.apply(module
          , [method].concat(params)
          );

    } else {
      throw new Error('Method ' +
          method + 'cannot be found
           in given module');
    }

    return result === module ? this
        : result;

  };

}).call(Type.fn);
```

Listing 6: plugin_api.js

```
'use strict';

var Type = require('./core');

Type.Environment = function () {
};

(function () {

/**
 * Is the user's computer a
     Macintosh computer
 * @type {boolean}
 */
Type.Environment.mac = navigator.
    appVersion.indexOf("Mac") !==
    -1;

}).call(Type.Environment);

module.exports = Type.Environment;
```

Listing 7: environment.js

```
'use strict';

var Type = require('./core');

/**
 * @constructor
 */
Type.Utilities = function () {
};

(function () {

  /**
   * This behaves similar to jQuery'
       s extend method. Writes all
       properties
   * from the objects passed as
       copyFrom to the object
       passed  as copyTo.
   * Copying starts from left to
       right and will overwrite
       each setting
   * subsequently.
   *
   * @param {Object} copyTo
   * @param {...Object} copyFrom
   * @returns {Object}
   */
  Type.Utilities.extend = function (
      copyTo, copyFrom) {
    var i, key;
    for (i = 1; i < arguments.length
        ; i += 1) {
      for (key in arguments[i]) {
        if (arguments[i].
            hasOwnProperty(key)) {
          arguments[0][key] =
              arguments[i][key];
        }
      }
    }
    return arguments[0];
  };

  /**
   * Tests and returns if a given object is a function instance
   * todo this should be called isFunctionInstance otherwise typeof obj === 'Function' should be used
   *
   * @param obj
   * @returns {boolean}
   */
  Type.Utilities.isFunction = function(obj) {
    return !!(obj && obj.constructor && obj.call && obj.apply);
  };

}).call(Type.Utilities);

module.exports = Type.Utilities;
```

Listing 8: utilities.js

```
'use strict';

var Type = require('./core');

/**
 * @constructor
 */
Type.DomUtilities = function () {
};

(function () {

  /**
   * The id attribute of the container element where all the helper
   * elements including carets and input fields of type will be
   * appended to
   *
   * @type {string}
   * @private
   */
  Type.DomUtilities._containerId = Type.Settings.prefix + 'container';


  /**
   * Matches a single HTML tag
   * @type {RegExp}
   * @private
   */
Type.DomUtilities._singleTag = /^<([\w-]+)\s*\/?>(?:<\/\1>|)$/;

/**
 * Todo Use me wherever you find document.createElement or this.elementsContainer
 * @param {string} tagName
 * @param {string} [className]
 * @returns {Element}
 */
Type.DomUtilities.addElement = function (tagName, className) {
  var el = document.createElement(tagName);
  if (className) el.className = Type.Settings.prefix + className;
  this.getElementsContainer().appendChild(el);
  return el;
};

/**
 * Removes a DOM element
 * @param {Element} el
 * @returns {*}
 */
Type.DomUtilities.removeElement = function (el) {
    el.parentNode.removeChild(el);
    return this;
  };

  /**
   * Will remove a node and each parent (recursively) if removing
   * leaves the parent with no *visible* content
   *
   * @param {Node} node - The node to remove
   * @param {Node} [constrainingNode] - The algorithm will stop and
   *     not remove this node if it reaches it
   * @returns {Node|null} - Will return the parent node where this
   *     algorithm stopped (The node it did *not* delete)
   */
  Type.DomUtilities.removeVisible = function (node, constrainingNode) {
    var parent = node.parentNode;
    if (node === constrainingNode) return node;
    if (node === document.body) return node;
    if (parent === null) return null;
    parent.removeChild(node);
    if (!this.isVisible(parent))
      return this.removeVisible(
          parent, constrainingNode)
          ;
    return parent;
  };

  /**
   * Recursively unwraps the given
       tag from the element passed
       an all its children
   * Note to self and future
       developers, querySelectorAll
        can be used for this when
   * we drop IE 8 support.
   *
   * @param el
   * @param tag
   * @param deep
   * @returns {Type.DomUtilities}
   */
  Type.DomUtilities.removeTag =
      function (el, tag, deep) {
    var i;
    if (deep && el.childNodes.length
        ) {
      for (i = 0; i < el.childNodes.
          length; i += 1) {
        this.removeTag(el.childNodes
            [i], tag, deep);
      }
    }
    if (el.nodeType === 1 && el.
        tagName.toLowerCase() ===
        tag.toLowerCase()) {
      this.unwrap(el);
    }
    return this;
  };

  /**
   * Converts a string of HTML to a
       corresponding {NodeList}
   *
   * @param {String} htmlString - A
       string containing HTML
   * @returns {NodeList} - The
       elements represented by the
       string
   */
  Type.DomUtilities.parseHTML =
      function(htmlString) {
    var fragment = document.
        createDocumentFragment(),
      div = fragment.appendChild(
          document.createElement('
          div'));
    div.innerHTML = htmlString;
    return div.childNodes;
  };

  /**
   *
   * By Dave Atchley, taken (and
       modified) from
   * {@link https://gist.github.com/
       datchley/11383482}
   * No license given. I asked for
       the license by mail.
   * Still waiting.
   *
   * @param tag
   * @param elms
   * @returns {Element}
   */
  Type.DomUtilities.wrap = function
      (tag, elms) {

    // Even out parameters
    elms  = elms.length ? elms : [
        elms];

    // Prepare vars and cache the
        current parent
    // and sibling of the first
        element.
    var el    = elms[0],
      parent  = el.parentNode,
      sibling = el.nextSibling,
      wrapper = document.
          createElement(tag),
      i;

    // If the first element had a
        sibling, insert the wrapper
         before the
    // sibling to maintain the HTML
        structure; otherwise, just
        append it
    // to the parent.
    if (sibling) {
      parent.insertBefore(wrapper,
          sibling);
    } else {
      parent.appendChild(wrapper);
    }

    // Move all elements to the
        wrapper. Each element is
    // automatically removed from
        its current parent and
    // from the elms array.
    for (i = 0; i < elms.length; i
        += 1) {
      wrapper.appendChild(elms[i]);
    }

    // Remove the tag we want to
        wrap from TypeContents
    // so we don't have the same tag
         nested
    for (i = 0; i < elms.length; i
        += 1) {
      this.removeTag(elms[i], tag,
          true);
    }

    // Return newly created element
    return wrapper;
  };

  /**
   * Todo use this.moveAfter()
   * @param {Node} el
   * @returns {Type.DomUtilities}
   */
  Type.DomUtilities.unwrap =
      function (el) {

    var next       = el.nextSibling,
      parent       = el.parentNode,
      childNodes = el.childNodes;

    if (next) {
      while (childNodes.length) {
        parent.insertBefore(el.
            lastChild, next);
      }
    } else {
      while (childNodes.length) {
        parent.appendChild(el.
            firstChild);
      }
    }

    parent.removeChild(el);
    parent.normalize();

    return this;
  };

  /**
   *
   * @param reference
   * @param elems
   * @returns {*}
   */
  Type.DomUtilities.moveAfter =
      function (reference, elems) {

    var i;

    var next = reference.nextSibling
      ,
      parent = reference.parentNode;

    elems = !elems.length ? [elems]
        : Array.prototype.slice.
        call(elems, 0);

    if (next) {
      for (i = 0; i < elems.length;
          i += 1) {
        parent.insertBefore(elems[i
            ], next);
      }
    } else {
      for (i = 0; i < elems.length;
          i += 1) {
        parent.appendChild(elems[i])
            ;
      }
    }

    return this;
  };

  /**
   * Todo move to dom walker??
   *
   * @param {Node} el
   * @param {String} selector
   * @param {Node} [constrainingNode
   *   ]
   * @returns {HTMLElement|null}
   */
  Type.DomUtilities.parent =
      function(el, selector,
      constrainingNode) {
    while (el.parentNode && (!
        constrainingNode || el !==
        constrainingNode)) {
      if (this.matches(el, selector)
          ) {
        return el;
      }
      el = el.parentNode;
    }
    return null;
  };

  /**
   * Returns true if el matches the
       CSS selector given as second
        argument,
   * otherwise false
   *
   * Todo http://davidwalsh.name/
       element-matches-selector
   *
   * @param el
   * @param selector
   * @returns {boolean}
   */
  Type.DomUtilities.matches =
      function(el, selector) {
    var _matches = (el.matches || el
        .matchesSelector || el.
        msMatchesSelector || el.
        mozMatchesSelector || el.
        webkitMatchesSelector || el
        .oMatchesSelector);

    if (_matches) {
      return _matches.call(el,
          selector);
    } else {
      var nodes = el.parentNode.
          querySelectorAll(selector
          );
      for (var i = nodes.length; i
          --;) {
        if (nodes[i] === el)
          return true;
      }
      return false;
    }
  };

  /**
   *
   * @returns {Element}
   */
  Type.DomUtilities.
      getElementsContainer =
      function () {
```

```
    var container = window.document.getElementById(this._containerId);
    if (container === null) {
      container = window.document.createElement('div');
      container.setAttribute('id', this._containerId);
      window.document.body.appendChild(container);
    }
    return container;
  };

  /**
   *
   * @param {Node} container
   * @param {Node} node
   * @returns {boolean}
   */
  Type.DomUtilities.containsButIsnt = function (container, node) {
    return container !== node && container.contains(node);
  };

  /**
   *
   * @param obj
   * @returns {boolean}
   */
  Type.DomUtilities.isNode = function (obj) {
    return !!(obj && obj.nodeType);
  };

  /**
   * Returns true if the given node is visible to the user.
   *
   * @param {Element} el - The element to be checked
   * @returns {boolean}
   * @private
   */
  Type.DomUtilities.isVisible = function (el) {
    return !!el.offsetHeight;
  };

  /**
   * Compares the document positions of two DOM nodes
   *
   * @param {Node} a - A DOM node to compare with the given other node
   * @param {Node} b - A DOM node to compare with the given other node
   * @returns {number} - Returns -1 if a precedes b, 1 if it is the
   *   other way around and 0 if they are equal.
   */
  Type.DomUtilities.order = function (a, b) {
    if (a === b) {
      return 0;
    }
    if (a.compareDocumentPosition(b) & Node.DOCUMENT_POSITION_FOLLOWING) {
      return -1;
    }
    return 1;
  };

}).call(Type.DomUtilities);

module.exports = Type.DomUtilities;
```

Listing 9: dom_utilities.js

```
'use strict';

var Type = require('./core');

/**
 * @param {Node} node - The node to be used as the starting point for the
 *   first traversal operation.
 * @param {Object|Node} [options] - If an object is passed, it should
 *   contain settings determining what node to return, see specifics
 *   below. If a {Node} is passed, this acts as options.constrainingNode
 * @param {Function|string} [options.filter] - nextNode traverses
 *   the DOM tree and passes each node to this function. This function
 *   should return true if the node passed is a node that we look for
 *   or false otherwise. E.g. if we want to find the next text node
 *   in the tree, the function should check if the node passed is of
 *   nodeType === 3. If this parameter is not set, any node found
 *   will be returned.
 *   todo allow css selectors to be used for traversal
 * @param {Node} [options.constrainingNode] While traversing the DOM,
 *     this method will check nodes'
       parents and parents' parents.
       By
 *     passing a DOM node as this
      parameter, traversing up will
      stop at
 *     this node and return null.
      This is useful when you want
      to permit
 *     traversing outside the editor
      's root node.
 * @constructor
 */
Type.DomWalker = function (node,
    options) {
  this.setNode(node);
  this.options(options);
};

(function () {

  /**
   * Returns the next node in the
       document flow and sets the
       internal reference
   * to the current node to that
       node.
   * @returns {null|Node}
   */
  this.next = function (returnMe) {
    return this._setNodeIfNotNull(
        Type.DomWalker._nextNode(
        this._node, this._options,
        returnMe));
  };

  /**
   * Returns the next node in the
       document flow but does not
       set the internal
   * reference to the current node
       to that node.
   * @returns {null|Node}
   */
  this.prefetchNext = function (
      returnMe) {
    return Type.DomWalker._nextNode(
        this._node, this._options,
        returnMe);
  };

  /**
   * Returns the previous node in
       the document flow and sets
       the internal reference
   * to the current node to that
       node.
   * @returns {null|Node}
   */
  this.prev = function (returnMe) {
    return this._setNodeIfNotNull(
        Type.DomWalker._prevNode(
        this._node, this._options,
        returnMe));
  };

  /**
   * Returns the previous node in
       the document flow but does
       not set the internal
   * reference to the current node
       to that node.
   * @returns {null|Node}
   */
  this.prefetchPrev = function (
      returnMe) {
    return Type.DomWalker._prevNode(
        this._node, this._options,
        returnMe);
  };

  /**
   * Returns the first child node
       matching the given filter or
        the node passed itself
   * if it matches the filter too.
       Sets the internal reference
       for the current node to
   * the node found.
   * @returns {null|Node}
   */
  this.first = function () {
    var node = Type.DomWalker.first(
        this._node, this._options.
        filter);
    return this._setNodeIfNotNull(
        node);
  };

  /**
   * Returns the last child node
       matching the given filter or
        the node passed itself
   * if it matches the filter too.
       Sets the internal reference
       for the current node to
   * the node found.
   * @returns {null|Node}
   */
  this.last = function () {
    var node = Type.DomWalker.last(
        this._node, this._options.
        filter);
    return this._setNodeIfNotNull(
        node);
  };

  /**
   * Sets the internal node from
       which traversal is made to
       the given node.
   * @param {Node} node
   */
  this.setNode = function (node) {
    if (!node.nodeType) {
      throw new Error('The given
          node is not a DOM node');
    }
    this._node = node;
    return this;
  };

  /**
   * Sets the options used for
       traversal by this walker
   * @param options
   * @returns {*}
   */
  this.options = function (options)
      {
    this._options = Type.DomWalker.
        loadOptions(options);
    return this;
  };

  /**
   * Returns the current node the
       walker is on.
   * @returns {Node}
   */
  this.getNode = function () {
    return this._node;
  };

  /**
   * Will set this _node to the
       given node unless null is
       passed.
   * Will also return either null or
        the node, depending on what
   * has been passed. This method is
        used to process the return
   * values by the DomWalker
       traversal methods.
   *
   * @param {Node|null} node
   * @returns {Node|null}
   * @private
   */
  this._setNodeIfNotNull = function
      (node) {
    if (node === null) {
      return null;
    }
    this._node = node;
    return node;
  };

}).call(Type.DomWalker.prototype);

/**
 * todo replace Type.DomWalker with
     "this" where possible
 */
(function () {

  /**
   *
   * @type {Object}
   * @private
   */
  Type.DomWalker._filterFunctions =
      {
    text     : '
        _isTextNodeWithContents',
    textNode : '_isTextNode',
    textual  : '_resemblesText',
    visible  : '_isVisible'
  };

  /**
   *
   * @param node
   * @param options
   * @returns {null|Node}
   */
  Type.DomWalker.next = function (
      node, options) {
    return Type.DomWalker._nextNode(
        node, Type.DomWalker.
        loadOptions(options));
  };

  /**
   *
   * @param node
   * @param options
   * @returns {null|Node}
   */
  Type.DomWalker.prev = function (
      node, options) {
    return Type.DomWalker._prevNode(
        node, Type.DomWalker.
        loadOptions(options));
  };

  /**
   *
   * @param node
   * @param filter
   * @returns {null|Node}
   */
  Type.DomWalker.first = function (
      node, filter) {
    var options = Type.DomWalker.
        loadOptions(filter);
    options.constrainingNode = node;
    return Type.DomWalker._nextNode(
        node, options, true);
  };

  /**
   *
   * @param node
   * @param filter
   * @returns {null|Node}
   */
  Type.DomWalker.last = function (
      node, filter) {
    var options = Type.DomWalker.
        loadOptions(filter);
    options.constrainingNode = node;
    return Type.DomWalker._prevNode(
        node, options, true);
  };

  /**
   *
   * @param options
   * @returns {*}
   */
  Type.DomWalker.loadOptions =
      function (options) {

    // If no options parameter has
        been passed
    options = options || {};
    // If a node has been passed as options parameter
    if (options.nodeType) {
      options = {constrainingNode: options};
    }

    // If a function has been passed as ooptions parameter
    if (typeof options === 'string' || Type.Utilities.isFunction(options)) {
      options = {filter: options};
    }

    // Load internal filter function if filter param is a string
    if (options.filter) {
      options.filter = Type.DomWalker._loadFilter(options.filter);
    }

    // Return processed options
    return options;

  };

  /**
   *
   * @param filter
   * @returns {*}
   * @private
   */
  Type.DomWalker._loadFilter = function (filter) {
    var funcName;
    if (typeof filter === 'string') {
      funcName = Type.DomWalker._filterFunctions[filter];
      return Type.DomWalker[funcName];
    }
    return filter;
  };

  /**
   * Traverses the DOM tree and finds the next node after the node passed
   * as first argument. Will traverse the children, siblings and parents'
   * siblings (in that order) to find the next node in the DOM tree as
   * displayed by the document flow.
   *
   * @param {Node} node - The node from which the search should start
   * @param {Object|Node} [options] - If an object is passed, it should
   *     contain settings determining what node to return, see specifics
   *     below. If a {Node} is passed, this acts as options.constrainingNode
   * @param {Function} [options.filter] - nextNode traverses the
   *     DOM tree and passes each node to this function. This function
   *     should return true if the node passed is a node that we look for
   *     or false otherwise. E.g. if we want to find the next text node
   *     in the tree, the function should check if the node passed is of
   *     nodeType === 3. If this parameter is not set, any node found
   *     will be returned.
   * @param {Node} [options.constrainingNode] While traversing the DOM,
   *     this method will check nodes' parents and parents' parents. By
   *     passing a DOM node as this parameter, traversing up will stop at
   *     this node and return null. This is useful when you want to permit
   *     traversing outside the editor's root node.
   * @param {boolean} [returnMe] This should not be passed by the
   *     programmer, it is used internally for recursive function calls to
   *     determine if the current node should be returned or not. If the
   *     programmer passes a node and does *not* pass this argument, the
   *     node passed will not be considered for returning. After that,
   *     internally, this will be set to true and be passed on with the
   *     next node in the DOM to a recursive call. The node then passed to
   *     this method might be the node we are looking for, so having this
   *     set to true will return that node (given that the filter
   *     also returns true for that node)
   * @returns {null|Node} The next node in the DOM tree found or null
   *     if none is found for the options.filter criteria or
```

```
 *      options.constrainingNode
      has been hit.
 */
Type.DomWalker._nextNode =
    function (node, options,
    returnMe) {

  // For later use
  var parent = node.parentNode;

  // If a node is found in this
      call, return it, stop the
      recursion
  if (returnMe === true && (!
      options.filter || options.
      filter(node))) {
    return node;
  }

  // 1. If this node has children,
       go down the tree
  if (node.childNodes.length) {
    return Type.DomWalker.
        _nextNode(node.childNodes
        [0], options, true);
  }

  // 2. If this node has siblings,
       move right in the tree
  if (node.nextSibling !== null) {
    return Type.DomWalker.
        _nextNode(node.
        nextSibling, options,
        true);
  }

  // 3. Move up in the node's
      parents until a parent has
      a sibling or the
      constrainingNode is hit
  while (parent !== options.
      constrainingNode) {
    if (parent.nextSibling !==
        null) {
      return Type.DomWalker.
          _nextNode(parent.
          nextSibling, options,
          true);
    }
    parent = parent.parentNode;
  }

  // We have not found a node we
      were looking for
  return null;

};

/**
 * Traverses the DOM tree and
     finds the previous node
     before the node passed
 * as first argument. Will
     traverse the children,
     siblings and parents'
 * siblings (in that order) to
     find the next node in the
     DOM tree as
 * displayed by the document flow.
 *
 * @param {Node} node - The node
     from which the search should
      start
 * @param {Object|Node} [options]
     - If an object is passed, it
      should
 *     contain settings
     determining what node to
     return, see specifics
 *     below. If a {Node} is
     passed, this acts as options
     .constrainingNode
 * @param {Function} [options.
     filter] - nextNode traverses
      the
 *     DOM tree and passes each
     node to this function. This
     function
 *     should return true if the
     node passed is a node that
     we look for
 *     or false otherwise. E.g. if
     we want to find the next
     text node
 *     in the tree, the function
     should check if the node
     passed is of
 *     nodeType === 3. If this
     parameter is not set, any
     node found
 *     will be returned.
 * @param {Node} [options.
     constrainingNode] While
     traversing the DOM,
 *     this method will check
     nodes' parents and parents'
     parents. By
 *     passing a DOM node as this
     parameter, traversing up
     will stop at
 *     this node and return null.
     This is useful when you want
      to permit
 *     traversing outside the
     editor's root node.
 * @param {boolean} [returnMe]
     This should not be passed by
      the
 *     programmer, it is used
     internally for recursive
     function calls to
 *     determine if the current
     node should be returned or
     not. If the
 *     programmer passes a node
     and does *not* pass this
     argument, the
 *     node passed will not be
     considered for returning.
     After that,
 *     internally, this will be
     set to true and be passed on
      with the
 *     next node in the DOM to a
     recursive call. The node
     then passed to
 *     this method might be the
     node we are looking for, so
     having this
 *     set to true will return
```

```
       that node (given that the filter
 *     also returns true for that node)
 * @returns {null|Node} The next node in the DOM tree found or null
 *     if none is found for the options.filter criteria or
 *     options.constrainingNode has been hit.
 */
Type.DomWalker._prevNode = function (node, options, returnMe) {

  // For later use
  var parent = node.parentNode;

  // If a node is found in this call, return it, stop the recursion
  if (returnMe === true && (!options.filter || options.filter(node))) {
    return node;
  }

  // 1. If this node has children, go down the tree
  if (node.childNodes.length) {
    return Type.DomWalker._prevNode(node.lastChild, options, true);
  }

  // 2. If this node has siblings, move right in the tree
  if (node.previousSibling !== null) {
    return Type.DomWalker._prevNode(node.previousSibling, options, true);
  }

  // 3. Move up in the node's parents until a parent has a sibling or the constrainingNode is hit
  while (parent !== options.constrainingNode) {
    if (parent.previousSibling !== null) {
      return Type.DomWalker._prevNode(parent.previousSibling, options, true);
    }
    parent = parent.parentNode;
  }

  // We have not found a node we were looking for
  return null;

};

/**
 * Returns true if a given node is a text node
 *
 * @param {Node} node The node to be checked.
 * @returns {boolean}
 * @private
 */
Type.DomWalker._isTextNode = function (node) {
  return node.nodeType === Node.TEXT_NODE;
};

/**
 * Returns true if a given node is a text node and its contents are not
 * entirely whitespace.
 *
 * @param {Node} node The node to be checked.
 * @returns {boolean}
 * @private
 */
Type.DomWalker._isTextNodeWithContents = function (node) {
  return node.nodeType === Node.TEXT_NODE && /[^\t\n\r ]/.test(node.textContent);
};

/**
 * Returns true if a given node is displayed as text on the screen
 *
 * @param {Node} node The node to be checked.
 * @returns {boolean}
 * @private
 */
Type.DomWalker._resemblesText = function (node) {
  return node.nodeName.toLocaleLowerCase() === 'br' || Type.DomWalker._isTextNodeWithContents(node);
};

/**
 * Returns true if the given node is visible to the user.
 *
 * @param {Element} node - The node to be checked
 * @returns {boolean}
 * @private
 */
Type.DomWalker._isVisible = function (node) {
  return !!node.offsetHeight;
};
```

```
}).call(Type.DomWalker);


module.exports = Type.DomWalker;
```

Listing 10: dom_walker.js

```
'use strict';

var Type = require('./core');

Type.TextWalker = function () {
};

(function () {

  /**
   *
   * @param fromNode
   * @param toNode
   * @param fromOffset
   * @param toOffset
   * @returns {*}
   */
  Type.TextWalker.offset = function (fromNode, toNode, fromOffset, toOffset) {

    var dom = new Type.DomWalker(fromNode, 'textual'),
      node = dom.next(true),
      offsetWalked = 0;

    fromOffset = fromOffset || 0;
    toOffset = toOffset || 0;

    do {
      if (node === toNode) {
        return offsetWalked + toOffset - fromOffset;
      }
      //offsetWalked += node.nodeValue.trim().length;
      offsetWalked += Type.TextWalker._textLength(node);
    } while (node = dom.next());

    return null;

  };

  /**
   * todo constraining node
   * @param {Node} fromNode
   * @param {number} offset
   * @param {number} [startOffset]
   * @returns {{node:Node,offset:number}|null} - The node and the offset to its
   *      start or null if no node could be found
   */
  Type.TextWalker.nodeAt = function (fromNode, offset, startOffset) {

    var walker = new Type.DomWalker(fromNode, 'textual'),
    //var walker = new Type.DomWalker(fromNode, 'text'),
      node = walker.first(),//Type.DomWalker.first(fromNode, 'text'),
      offsetWalked = 0,
      length;

    startOffset = startOffset || 0;
    offset += startOffset;

    //if (fromNode.nodeType === 3 && offset >= 0 && offset <= fromNode.nodeValue.trim().length) {
    //  return { node: fromNode, offset: offset };
    //}

    //if (offset >= 0 && offset <= node.nodeValue.trim().length) {
    if (offset >= 0 && offset <= Type.TextWalker._textLength(node)) {
      return { node: node, offset: offset };
    }

    if (offset < 0) {
      while (node = walker.prev()) {
        //length = node.nodeValue.trim().length;
        length = Type.TextWalker._textLength(node);
        if (offsetWalked - length <= offset) {
          return { node: node, offset: length+(offset-offsetWalked) };
        }
        offsetWalked -= length;
      }

    } else {
      do {
        //length = node.nodeValue.trim().length;
        length = Type.TextWalker._textLength(node);
        if (offsetWalked + length >=
              offset) {
          return { node: node,
               offset: offset-
               offsetWalked };
        }
        offsetWalked += length;
      } while (node = walker.next())
          ;
    }

    return null;

  };

  Type.TextWalker._textLength =
      function (node) {
    if (node.nodeName.
        toLocaleLowerCase() === 'br
        ') {
      return 1;
    } else {
      return node.nodeValue.trim().
          length;
    }
  };

  /**
   *
   * @param a
   * @param b
   */
  //this.mergeTexts = function (a, b
      ) {
  //if (a.nodeType === Node.
      TEXT_NODE) {
  //
  //  }
  //  }
  //};

}).call(Type.TextWalker);


module.exports = Type.TextWalker;
```

Listing 11: text_walker.js

```
'use strict';

var Type = require('./core');

/**
 * Crates a new Type.Range
 *
 * Type.Range is a shim for the
     browsers' native {Range}
     objects and
 * is being used in Type for
     anything related to text
     ranges.
 *
 * Native ranges are often buggy,
     lack essential features and
     should
 * not be used other than for
     performance reasons. This
     class avoids
 * and / or fixes common issues with
      ranges and adds many methods
 * useful for text editing.
 *
 * Among many other factory methods,
      you can use the {Type.Range.
     fromRange}
 * method to create a {Type.Range}
     from a native {Range}.
 *
 * @param {Node} startContainer - A
     text node that the range
     should start in.
 * @param {number} startOffset - The
      offset (of characters) inside
      the
 *    startContainer where the
     range should begin.
 * @param {Node} endContainer - A
     text node that the range
     should end in.
 * @param {number} endOffset - The
     offset (of characters) inside
     the
 *    endContainer where the range
     should stop.
 * @constructor
 */
Type.Range = function (
    startContainer, startOffset,
    endContainer, endOffset) {

  this.startContainer =
      startContainer;
  this.startOffset    = startOffset;
  this.endContainer   = endContainer
      ;
  this.endOffset      = endOffset;

  this.
      ensureStartNodePrecedesEndNode
      ();

};

(function () {

  /**
   * If the startContainer and the
       endContainer are enclosed by
   * the same element matching the
       selector, that element will
   * be returned. Otherwise null
       will be returned.
   *
   * todo call this commonAncestor
       and make the selector
       optional
   *
   * @param {String} selector - This
        method will only return a
 *     common ancestor matched by
     this selector.
 * @param {HTMLElement} [
     constrainingNode] - If given
     , this
 *     method will stop traversing
      the DOM tree when it hits
 *     this element.
 * @returns {HTMLElement|null} -
     Will either return the
     common
 *     ancestor matching the
     selector or null otherwise.
 */
this.elementEnclosingStartAndEnd =
     function (selector,
    constrainingNode) {

  var tagEnclosingStartNode = Type
      .DomUtilities.parent(this.
      startContainer, selector,
      constrainingNode),
    tagEnclosingEndNode;

  if (tagEnclosingStartNode ===
      null) {
    return null;
  }

  tagEnclosingEndNode = Type.
      DomUtilities.parent(this.
      endContainer, selector,
      constrainingNode);

  if (tagEnclosingStartNode ===
      tagEnclosingEndNode) {
    return tagEnclosingStartNode;
  }

  return null;
};

/**
 * Will return whether or not the
     whole range (the
 * startContainer and the
     endContainer are both
     children
 * of the given element.
 *
 * @param {Node} node - The node
     to check if it
 *     is a parent to the start
     and endContainer.
 * @returns {boolean}
 */
this.isInside = function (node) {
  return node.contains(this.
      startContainer) && node.
      contains(this.endContainer)
      ;
};

/**
 * Will throw an error if the
     start and endContainer are
 * not children to the given
     element. Returns true if
 * they are.
 *
 * @param {HTMLElement} el - The
     element to check if it
 *     is a parent to the start
     and endContainer.
 * @returns {boolean}
 */
this.ensureIsInside = function (el
    ) {
  if (this.isInside(el)) {
    return true;
  }
  throw new Error('Range is not
      contained by given node.');
};

/**
 * Will swap start and end
     containers as well as
     offsets if
 * either the containers or the
     offsets are in the wrong
 * order (the start container /
     offset should precede the
     end)
 *
 * @returns {Type.Range} - This
     instance
 */
this.
    ensureStartNodePrecedesEndNode
     = function () {

  var startIsEnd, startPrecedesEnd
      ;
  startIsEnd = this.startContainer
       === this.endContainer;

  if (startIsEnd && this.
      startOffset <= this.
      endOffset) {
    return this;
  }

  if (startIsEnd && this.
      startOffset > this.
      endOffset) {
    return this._swapOffsets();
  }

  startPrecedesEnd = this.
      startContainer.
      compareDocumentPosition(
      this.endContainer);
  startPrecedesEnd =
      startPrecedesEnd & Node.
      DOCUMENT_POSITION_FOLLOWING
      ;

  if (!startPrecedesEnd) {
    this._swapStartAndEnd();
  }

  return this;
};

/**
 * Will split the startContainer
     text node at the startOffset
```

```
         and set
   * this' startContainer to the
        right node the resulting
        nodes of the
   * split and the startOffset to 0.
         Will return the new
        startContainer.
   *
   * @returns {Node} - The new
        startContainer
   */
  this.splitStartContainer =
      function () {

    var startsAndEndsInSameNode;

    if (this.startOffset === 0) {
      return this.startContainer;
    }

    startsAndEndsInSameNode = this.
        startsAndEndsInSameNode();
    this.startContainer = this.
        startContainer.splitText(
        this.startOffset);

    if (startsAndEndsInSameNode) {
      this.endContainer = this.
          startContainer;
      this.endOffset -= this.
          startOffset;
    }

    this.startOffset = 0;

    return this.startContainer;
  };

  /**
   * Will split the endContainer
        text node at the endOffset
        and set
   * this' endContainer to the left
        node the resulting nodes of
        the
   * split and the endOffset to the
        end of the endContainer.
   * Will return the new
        endContainer.
   *
   * @returns {Node} - The new
        endContainer
   */
  this.splitEndContainer = function
      () {
    if (this.endOffset !== this.
        endContainer.length) {
      this.endContainer = this.
          endContainer.splitText(
          this.endOffset).
          previousSibling;
      this.endOffset = this.
          endContainer.length;
    }
    return this.endContainer;
  };

  /**
   * Creates a native {Range} object
        and returns it.
   * @returns {Range}
   */
  this.getNativeRange = function ()
      {
    var range = document.createRange
        ();
    range.setEnd(this.endContainer,
        this.endOffset);
    range.setStart(this.
        startContainer, this.
        startOffset);
    return range;
  };

  /**
   * Looks up the number of
        characters (offsets) where
        this range starts
   * and ends relative to a given {
        Element}. Returns an {Object
        } containing
   * the element itself and the
        offsets. This object can be
        used to restore
   * the range by using the {@link
        Type.Range.load} factory.
   *
   * @param {Element} fromNode
   * @returns {{from: Element, start
        : number, end: number}}
   */
  this.save = function (fromNode) {
    var start, end;
    start = this.getStartOffset(
        fromNode);
    end = this.
        startsAndEndsInSameNode() ?
         start - this.startOffset +
         this.endOffset : this.
        getEndOffset(fromNode);
    return { from: fromNode, start:
        start, end: end };
  };

  /**
   * Returns the length of this
        range as numbers of
        characters.
   * @returns {number}
   */
  this.getLength = function () {
    return Type.TextWalker.offset(
        this.startContainer, this.
        endContainer, this.
        startOffset, this.endOffset
        );
  };

  /**
   * Returns the offset (number of
        visible characters) from the
         given node
   * to the startContainer and its
        startOffset. If no node has
        been passed
   * this will return the
        startOffset
   *
   * @param {Node} [from] - The node to start counting characters from
   * @returns {number|null}
   */
  this.getStartOffset = function (from) {
    if (from) {
      return Type.TextWalker.offset(from, this.startContainer, 0, this.startOffset);
    }
    return parseInt(this.startOffset, 10);
  };

  /**
   * Returns the offset (number of visible characters) from the given node
   * to the endContainer and its endOffset. If no node has been passed
   * this will return the endOffset
   *
   * @param {Node} [from] - The node to start counting characters from
   * @returns {number|null}
   */
  this.getEndOffset = function (from) {
    if (from) {
      return Type.TextWalker.offset(from, this.endContainer, 0, this.endOffset);
    }
    return parseInt(this.endOffset, 10);
  };

  /**
   * Returns the element containing the startContainer.
   *
   * @returns {Node}
   */
  this.getStartElement = function () {
    return this.startContainer.parentNode;
  };

  /**
   * Returns the element containing the endContainer.
   *
   * @returns {Node}
   */
  this.getEndElement = function () {
    return this.endContainer.parentNode;
  };

  /**
   * Returns the tag name of the element containing the
   * startContainer.
   *
   * @returns {string}
   */
  this.getStartTagName = function () {
    return this.getStartElement().tagName.toLowerCase();
  };

  /**
   * Returns the tag name of the element containing the
   * endContainer.
   *
   * @returns {string}
   */
  this.getEndTagName = function () {
    return this.getEndElement().tagName.toLowerCase();
  };

  /**
   * Returns whether or not the the element containing the
   * startContainer is of the given tagName.
   *
   * @param {string} tagName - The tag name to compare.
   * @returns {boolean}
   */
  this.startTagIs = function (tagName) {
    return this.getStartTagName() === tagName.toLowerCase();
  };

  /**
   * Returns whether or not the the element containing the
   * endContainer is of the given tagName.
   *
   * @param {string} tagName - The tag name to compare.
   * @returns {boolean}
   */
  this.endTagIs = function (tagName) {
    return this.getEndTagName() === tagName.toLowerCase();
  };

  /**
   * Returns whether or not the startContainer equals the
   * endContainer.
   *
   * @returns {boolean}
   */
  this.startsAndEndsInSameNode = function () {
    return this.startContainer === this.endContainer;
  };

  /**
   * Returns whether or not this range spans over no characters
   * at all.
   *
   * @returns {boolean}
   */
  this.isCollapsed = function () {
    return this.startOffset === this.endOffset && this.startsAndEndsInSameNode();
  };

  /**
   * Merges another range with this range and returns this range.
   *
   * @param {Type.Range} that - The range that should be added to
   *     this range.
   * @returns {Type.Range} - This instance
   */
  this.mergeWith = function (that) {

    var startOrder, endOrder;

    startOrder = Type.DomUtilities.order(this.startContainer, that.startContainer);
    endOrder = Type.DomUtilities.order(this.endContainer, that.endContainer);

    if (startOrder === 0) {
      this.startOffset = Math.min(this.startOffset, that.startOffset);
    } else if (startOrder === 1) {
      this.startContainer = that.startContainer;
    }

    if (endOrder === 0) {
      this.endOffset = Math.max(this.endOffset, that.endOffset);
    } else if (startOrder === -1) {
      this.endContainer = that.endContainer;
    }

    return this;

  };

  /**
   * Internal method to swap the start and end containers as well
   * as their offsets when it is initialized with the endContainer
   * preceding the startContainer.
   *
   * @returns {Type.Range} - This instance
   * @private
   */
  this._swapStartAndEnd = function () {
    this._swapContainers();
    this._swapOffsets();
    return this;
  };

  /**
   * Will swap the startContainer with the endContainer
   *
   * @returns {Type.Range} - This instance
   * @private
   */
  this._swapContainers = function () {
    var swapContainer = this.startContainer;
    this.startContainer = this.endContainer;
    this.endContainer = swapContainer;
    return this;
  };

  /**
   * Will swap the startOffset with the endOffset
   *
   * @returns {Type.Range} - This instance
   * @private
   */
  this._swapOffsets = function () {
    var swapOffset = this.startOffset;
    this.startOffset = this.endOffset;
    this.endOffset = swapOffset;
    return this;
  };

}).call(Type.Range.prototype);


(function () {

  /**
   * The implementation of {Range#getClientRects} is broken in WebKit
   * browsers. {@link Type.Range._getClientRectsNeedsFix} tests for
   * wrong behaviour and stores if it is broken in this variable.
   *
   * @type {null|boolean}
   */
  Type.Range._getClientRectsIsBroken = null;
```

```

/**
 * Will create a range spanning
     from the offset given as
     start to the
 * offset given as end, counting
     the characters contained by
     the given
 * el. This function should be
     used with the save method of
      {Type.Range}.
 *
 * @param {{from: HTMLElement,
     start: number, end: number}}
      bookmark -
 *     An object as returned by {
     Type.Range#save}
 * @param {HTMLElement} bookmark.
     from - The root element from
      which the
 *     start and end offsets
     should be counted
 * @param {number} bookmark.start
     - The offsets (number of
     characters)
 *     where the selection should
     start
 * @param {number} bookmark.end -
     The offsets (number of
     characters)
 *     where the selection should
     end
 * @returns {Type.Range} - A {Type
     .Range} instance
 */
Type.Range.load = function (
     bookmark) {
  return Type.Range.fromPositions(
      bookmark.from, bookmark.
      start, bookmark.end);
};

/**
 * Will create a range spanning
     from the offset given as
     start to the
 * offset given as end, counting
     the characters contained by
     the given
 * el.
 *
 * @param {HTMLElement|Node} el -
     The root element from which
     the start
 *     and end offsets should be
     counted
 * @param {number} startOffset -
     The offsets (number of
     characters) where the
 *     selection should start
 * @param {number} endOffset - The
      offsets (number of
     characters) where the
 *     selection should end
 * @returns {Type.Range} - A {Type
     .Range} instance
 */
Type.Range.fromPositions =
     function (el, startOffset,
     endOffset) {
  var start = Type.TextWalker.
      nodeAt(el, startOffset),
    end = Type.TextWalker.nodeAt(
        el, endOffset);
  return new Type.Range(start.node
      , start.offset, end.node,
      end.offset);
};

/**
 * Will read the current {
     Selection} on the document
     and create a {Type.Range}
 * spanning over the {Range}(s)
     contained by the selection.
     Will return
 * null if there is no selection
     on the document.
 *
 * todo Check if selection is
     actually inside editor and
     return null if not
 *
 * @returns {Type.Range|null} - A
     {Type.Range} instance or
     null
 */
Type.Range.fromCurrentSelection =
     function () {
  var sel = document.getSelection
      ();
  return sel.isCollapsed ? null :
      Type.Range.fromRange(sel.
      getRangeAt(0));
};

/**
 * Will create a {Type.Range}
     based on the start and end
     containers and
 * offsets of the given {Range}.
     This will also take care of
     browser
 * issues (especially WebKit) when
      the range is fetched from a
      selection
 * that ends at the end of an
     element.
 *
 * todo The "fix" is a solution
     for a single case
 * todo find the pattern of this
     and process all cases
 *
 * @param {Range} range - The {
     Range} that should be <em>
     migrated</em>
 *     to a {Type.Range}
 * @returns {Type.Range} - The {
     Type.Range} corresponding to
      the given
 *     {Range}
 */
Type.Range.fromRange = function (
     range) {
  var endContainer = range.
      endContainer,
    endOffset = range.endOffset;
  if (endOffset === 0 &&
      endContainer === Type.
      DomWalker.next(range.
      startContainer.parentNode.
      nextSibling, 'visible')) {
    endContainer = Type.DomWalker.
        last(range.startContainer
        .parentNode, 'text');
    endOffset = endContainer.
        length;
  }
  return new Type.Range(range.
      startContainer, range.
      startOffset, endContainer,
      endOffset);
};

/**
 * Will create a {Type.Range}
     spanning from the offset of
     the given {Caret}
 * over a number of characters
     passed as selectedChars. If
     selectedChars is
 * a positive number, the range's
     start will be set to the
     cursor position
 * and the end spanning to the
     characters to its right. If
     selectedChars is
 * negative it will span to the
     characters to its left.
 *
 * @param {Caret} caret
 * @param {number} selectedChars
 * @returns {Type.Range}
 */
Type.Range.fromCaret = function (
    caret, selectedChars) {
  var startNode = caret.getNode(),
    startOffset = caret.
        getNodeOffset(),
    end = Type.TextWalker.nodeAt(
        startNode, selectedChars,
         startOffset);
  return new Type.Range(startNode,
      startOffset, end.node, end
      .offset);
};

/**
 * Will create a {Type.Range}
     containing the given element
     's text by
 * finding the first and last text
      nodes inside the element
     and spanning
 * a range beginning at the start
     of the first text node and
     at the end
 * of the last text node.
 *
 * @param {HTMLElement} el - The
     element that should be <em>
     covered</em>
 *     by the returned {Type.Range
     }.
 * @returns {Type.Range} - A {Type
     .Range} spanning over the
     contents of the
 *     given element.
 */
Type.Range.fromElement = function
    (el) {
  var startNode = Type.DomWalker.
      first(el, 'text'),
    endNode = Type.DomWalker.last(
        el, 'text');
  return new Type.Range(startNode,
      0, endNode, endNode.
      nodeValue.length);
};

/**
 * Will return a new {Type.Range}
     at the position read from a
     given
 * {MouseEvent}. Will return null
     if the event was not
     triggerd from
 * within a text node.
 *
 * @param {MouseEvent} e - The
     mouse event to read
     positions from
 * @returns {Type.Range|null} -
     Returns a new Type.Range or
     null if the
 *     event has not been
     triggered from inside a text
      node
 */
Type.Range.fromMouseEvent =
    function (e) {
  return Type.Range.fromPoint(e.
      clientX, e.clientY);
};

/**
 * Will create a {Type.Range} at
     the offset and inside the
     text node
 * found at the x and y positions
     relative to the document.
     The range
 * will be collapsed. Will return
     null
 *
 * @param {number} x - The
     horizontal position relative
      to the document
 * @param {number} y - The
     vertical position relative
     to the document
 * @returns {Type.Range|null} -
     Returns a new Type.Range or
     null if the
 *     position is not inside a
     text node
 */
Type.Range.fromPoint = function (x
    , y) {

  var range, node, offset;

  if (document.
      caretPositionFromPoint) {
    range = document.
```

```
            caretPositionFromPoint(x,
             y);
        node = range.offsetNode;
        offset = range.offset;

    } else if (document.
          caretRangeFromPoint) {
        range = document.
            caretRangeFromPoint(x, y)
            ;
        node = range.startContainer;
        offset = range.startOffset;

    } else {
        Type.Development.debug('This
            browser does not support
            caretPositionFromPoint or
             caretRangeFromPoint.');
        return null;
    }

    // only split TEXT_NODEs
    if (node.nodeType === Node.
          TEXT_NODE) {
        return new Type.Range(node,
            offset, node, offset);
    }

    Type.Development.debug('User
          clicked in a non-text node,
           cannot create range');

    return null;

};

/**
 * WebKit browsers sometimes
     create unnecessary and
     overlapping {ClientRect}s in
 * {Range.prototype.getClientRects
     }. This method creates 2
     elements, creates a
 * range and tests for this
     behaviour.
 *
 * From {@link https://github.com/
     edg2s/rangefix}
 * (modified)
 *
 * Copyright (c) 2014 Ed Sanders
     under the
 * terms of The MIT License (MIT)
 *
 * @returns {boolean}
 * @private
 */
Type.Range.
      _testGetClientRectsNeedsFix =
      function () {

    var range = document.createRange
          (),
        p1 = Type.DomUtilities.
            addElement('p'),
        p2 = Type.DomUtilities.
            addElement('p'),
        needsFix;

    p1.appendChild(document.
          createTextNode('aa'));
    p2.appendChild(document.
          createTextNode('aa'));

    range.setStart(p1.firstChild, 1)
          ;
    range.setEnd(p2.firstChild, 1);

    needsFix = range.getClientRects
          ().length > 2;

    Type.DomUtilities.removeElement(
          p1);
    Type.DomUtilities.removeElement(
          p2);

    return needsFix;

};

/**
 * Will return if the browser has
     a broken model for {Range.
     prototype.getClientRects}.
 * This is usually the case with
     WebKit.
 *
 * @returns {boolean}
 * @private
 */
Type.Range._getClientRectsNeedsFix
      = function () {
    if (typeof Type.Range.
          _getClientRectsIsBroken !==
          'boolean') {
        Type.Range.
            _getClientRectsIsBroken =
             this.
            _testGetClientRectsNeedsFix
            ();
    }
    return Type.Range.
          _getClientRectsIsBroken;
};

/**
 * WebKit browsers sometimes
     create unnecessary and
     overlapping {ClientRect}s
 * in {Range.prototype.
     getClientRects}. This method
      takes a {Range}, fixes
 * the {ClientRect}s (if necessary
     ) and returns them.
 *
 * From {@link https://github.com/
     edg2s/rangefix}
 * (modified)
 *
 * Copyright (c) 2014 Ed Sanders
     under the
 * terms of The MIT License (MIT)
 *
 * @param {Range} range - A native
     {Range}
```

```
   * @return {ClientRect[]}
        ClientRectList or list of
   *     ClientRect objects
        describing range
   */
  Type.Range.getClientRects =
      function (range) {

    if (!Type.Range.
        _getClientRectsNeedsFix())
        {
      return range.getClientRects();
    }

    var partialRange = document.
        createRange(),
      endContainer = range.
        endContainer,
      endOffset = range.endOffset,
      rects = [];

    while (endContainer !== range.
        commonAncestorContainer) {

      partialRange.setStart(
          endContainer, 0);
      partialRange.setEnd(
          endContainer, endOffset);

      Array.prototype.push.apply(
          rects, partialRange.
          getClientRects());

      endOffset = Array.prototype.
          indexOf.call(
          endContainer.parentNode.
          childNodes, endContainer
          );
      endContainer = endContainer.
          parentNode;
    }

    partialRange = range.cloneRange
        ();
    partialRange.setEnd(endContainer
        , endOffset);
    Array.prototype.push.apply(rects
        , partialRange.
        getClientRects());

    return rects;

  };

}).call();

module.exports = Type.Range;
```

Listing 12: range.js

```
'use strict';

var Type = require('./core');

/**
 *
 * @param {Type} type
 * @constructor
 */
Type.Writer = function (type) {
  this._type = type;
  this._root = type.getRoot();
};

(function () {

  /**
   * Inserts a string in a text node
        at a given offset
   *
   * @param {Text} textNode - The
        text node into which str
        will be inserted.
   * @param {Number} offset - The
        character offset at which
        str will be inserted.
   * @param {String} str - The text
        that will be inserted
   * @returns {Type.Writer} - This
        instance
   */
  this.insertText = function (
      textNode, offset, str) {

    var nodeText = textNode.
        nodeValue;

    if (offset > 0) {
      textNode.nodeValue = nodeText.
          substring(0, offset) +
          str + nodeText.substring(
          offset, nodeText.length);
    } else {
      textNode.nodeValue = str +
          nodeText;
    }

    return this;
  };

  /**
   * Inserts DOM nodes at the offset
        of a text node
   *
   * @param {Text} textNode - The
        text node which will be
        split and in which
   *     the DOM will be inserted.
   * @param {Number} offset - The
        text offset at which the DOM
        should be
   *     inserted.
   * @param {Node|[Node]|NodeList|
        String} nodes - Either a {
        Node}, an array
   *      of {Node}s, a {NodeList} or
        a string containing HTML
       that will be
   *      inserted at the given
       offset in  a text node.
   * @returns {Type.Writer} - This
       instance
   */
  this.insertHTML = function (
      textNode, offset, nodes) {

    // Required variables
    var i, parent, insertBeforeNode;

    // Parse string (if given) to
        retrieve DOM nodes
    nodes = typeof nodes === 'string
        ' ? Type.DomUtilities.
        parseHTML(nodes) : nodes;

    // Make array if single DOM node
        was given
    nodes = nodes.length ? nodes : [
        nodes];

    // Make nodes an array (in case
        it is a NodeList)
    nodes = Array.prototype.slice.
        call(nodes);

    // Split text and prepare
        insertion
    insertBeforeNode = textNode.
        splitText(offset);
    parent = insertBeforeNode.
        parentNode;

    // If last given DOM node is a
        text, concat it with the
        text behind insertion
    if (nodes[nodes.length -1].
        nodeType === Node.TEXT_NODE
        ) {
      insertBeforeNode.nodeValue =
          nodes.pop().nodeValue +
          insertBeforeNode.
          nodeValue;
      if (!nodes.length) {
        textNode.nodeValue +=
            insertBeforeNode.
            nodeValue;
        Type.DomUtilities.
            removeElement(
            insertBeforeNode);
      }
    }

    // Insert DOM nodes between
        split texts
    for (i = nodes.length - 1; i >=
        1; i -= 1) {
      parent.insertBefore(nodes[i],
          insertBeforeNode);
      insertBeforeNode = nodes[i];
    }

    // If first given DOM node is a
        text, concat it with the
        text before insertion
    if (nodes.length && nodes[0].
        nodeType === Node.TEXT_NODE
        ) {
      textNode.nodeValue += nodes
          [0].nodeValue;
    } else if(nodes.length) {
      parent.insertBefore(nodes[0],
          insertBeforeNode);
    }

    // Chaining
    return this;

  };

  /**
   * todo refactor var names "a" and
       "b"
   * todo distinguish block from
       inline tags
   *
   * TODO CONSTRAIN TO TYPE ROOT !!!
       !   !    !!!
   *
   * remove(range)
   * remove(caret, -1)
   *
   * @param {Type.Range|Caret} range
   * @param {number} [numChars]
   */
  this.remove = function (range,
      numChars) {

    //var startNode, endNode,
        startParent, walker,
        current, prev, startRemoved
        , currentParent, a, b;
    var startNode, endNode, walker,
        current, startParent,
        startRemoved, currentParent
        , a, b;

    if (arguments.length === 2) {
      range = Type.Range.fromCaret(
          range, numChars);
    }

    startNode    = range.
        splitStartContainer();
    endNode      = range.
        splitEndContainer();
    startParent  = startNode.
        parentNode;
    walker       = this._type.
        createDomWalker(endNode, '
        textNode');
    //current      = endNode;
    startRemoved = false;

    //prev = endNode;

    if (!this._root.contains(
        startNode) || !this._root.
        contains(endNode)) {
      Type.Development.debug('The
          give startNode and
          endNode are not contained
```

```
            by the editor.');
      return this;
    }

    while (!startRemoved) {

      current = walker.getNode();
      walker.prev();

      a = (current === endNode && range.endOffset === 0);
      b = (current !== startNode && current === Type.DomWalker.first(current.parentNode, 'textNode'));

      if (a || b) {
        currentParent = current.parentNode;
        Type.DomUtilities.moveAfter(walker.getNode(), current.parentNode.childNodes);
        Type.DomUtilities.removeVisible(currentParent);
      }

      startRemoved = current === startNode;
      Type.DomUtilities.removeVisible(current);
      //current = walker.getNode();

    }


    /*while (!startRemoved) {

      prev = Type.DomWalker.prev(current, 'text');

      a = (current === endNode && range.endOffset === 0);
      b = (current !== startNode && current === Type.DomWalker.first(current.parentNode, 'text'));

      if (a || b) {
        currentParent = current.parentNode;
        Type.DomUtilities.moveAfter(prev, current.parentNode.childNodes);
        Type.DomUtilities.removeVisible(currentParent);
      }

      startRemoved = current === startNode;
      Type.DomUtilities.removeVisible(current);
      current = prev;

    }*/

    startParent.normalize();
    //startNode.parentNode.normalize();

    return this;

  };


}).call(Type.Writer.prototype);

module.exports = Type.Writer;
```

Listing 13: writer.js

```
'use strict';

var Type = require('./core');

/**
 *
 * @param {Type} type
 * @constructor
 */
Type.Formatter = function (type) {
  this._type = type;
};

(function () {

  /**
   * A list of tags that are displayed inline. We generate different markup
   * for inline and block tags. We use this array as reference to determine
   * what kind of markup to generate.
   *
   * todo move me to dom utils
   *
   * @type {string[]}
   * @private
   */
  this._inlineTags = ["strong", "em", "u", "s"];

  /**
   * A list of tags that are displayed as block elements. We generate different
   * markup for inline and block tags. We use this array as reference to determine
   * what kind of markup to generate.
   *
   * todo move me to dom utils
   *
   * @type {string[]}
   * @private
   */
  this._blockTags = ["h1", "h2", "
      h3", "h4", "h5", "h6", "
      blockquote"];

  /**
   * Will call either this.inline,
       this.block or this._noop
       depending on
   * whether the given tag is an
       inline or block element or
       we do not know
   * this tag yet (the latter would
       call _noop which would utter
        no action).
   *
   * @param {String} tag - The tag
       that we want to format the
       text with
   * @param {Type.Range} typeRange -
        An object containing data
       on which part
   *     of the text to format
   * @param {...*} params - Any
       number of arguments that
       specify attributes
   *     for the tag
   * @returns {Element[]} - The
       elements created by the
       formatting function
   */
  this.format = function (tag,
      typeRange, params) {
    typeRange.ensureIsInside(this.
        _type.getRoot());
    return this._handlerFor(tag).
        apply(this, arguments);
  };

  /**
   *
   * @param tag
   * @param range
   * @returns {*}
   */
  this.removeFormat = function (tag,
      range) {

    var startNode = this.
        _getStartNode(tag, range),
      dom = this._type.
          createDomWalker(startNode
          ),
      next;

    do {
      Type.DomUtilities.removeTag(
          dom.getNode(), tag, false
          );
      next = dom.next();
    } while(next && !next.contains(
        range.endContainer));// !==
        range.endContainer);

    return this;

  };

  /**
   *
   * @param tag
   * @param typeRange
   * @param params
   * @returns {Type.Formatter|
       Element[]}
   */
  this.inline = function (tag,
      typeRange, params) {

    var args, startNode, endNode,
        enclosingTag, selPositions;

    // If the selection is enclosed
        the tag we want to format
        with
    // remove formatting from
        selected area
    if (enclosingTag = typeRange.
        elementEnclosingStartAndEnd
        (tag)) {
      return this.removeInline(
          enclosingTag, typeRange);

    // Otherwise add formatting to
        selected area
    } else {
      startNode = this._getStartNode
          (tag, typeRange);
      endNode   = this._getEndNode(
          tag, typeRange);
      params    = Array.prototype.
          slice.call(arguments, 2);
      args      = [tag, startNode,
          endNode].concat(params);
      return this.insertInline.apply
          (this, args);
    }

  };

  /**
   * This method will wrap the given
        tag around (and including)
       all elements
   * between the startNode and
       endNode and try to maintain
       simple and valid
   * HTML. The tag should be an "
       inline"-element, for "block"
       elements use
   * {block}. Both methods have a
       different behaviour when
       generating markup.
   *
   * @param {String} tag
   * @param {Node} startNode
   * @param {Node} endNode
   * @param {...*} [params]
   * @returns {Element[]} - The
       elements created by the
       formatting function
  */
this.insertInline = function (tag,
     startNode, endNode, params)
    {

  // Required variables
  var currentNode = startNode,
    createdNodes  = [],
    nodesToWrap   = [],
    nextNode;

  // Collect the startNode and all
       its siblings until we
  // found the endNode or a node
       containing it
  while (currentNode && !
       currentNode.contains(
       endNode)) {
    nodesToWrap.push(currentNode);
    currentNode  = currentNode.
         nextSibling;
  }

  // If the node where we stopped
       is the endNode, add it
  // to our collection of nodes
  if (currentNode === endNode) {
    nodesToWrap.push(currentNode);
  }

  // If the node where we stopped
       contains the endNode,
  // apply this algorithm on it
       recursively
  if (currentNode && Type.
       DomUtilities.
       containsButIsnt(currentNode
       , endNode)) {
    createdNodes.concat(this.
         insertInline(tag,
         currentNode.firstChild,
         endNode));
  }

  // If we did not find the
       endNode but there are no
       more
  // siblings, find the next node
       in the document flow and
  // apply this algorithm on it
       recursively
  if (currentNode === null) {
    nextNode = Type.DomWalker.next
         (startNode.parentNode.
         lastChild, this._type.
         getRoot());
    createdNodes.concat(this.
         insertInline(tag,
         nextNode, endNode));
  }

  // Wrap the nodes we got so far
       in the provided tag
  createdNodes.push(Type.
       DomUtilities.wrap(tag,
       nodesToWrap));

  // Return all nodes that have
       been created
  return createdNodes;

};

/**
 *
 * @param {Node} enclosingTag
 * @param {Type.Range} typeRange
 * @returns {Type.Formatter}
 */
this.removeInline = function (
     enclosingTag, typeRange) {

  var tagName = enclosingTag.
       tagName,
    tagPositions = Type.Range.
         fromElement(enclosingTag)
         .save(this._type.getRoot
         ()),
    selPositions = typeRange.save(
         this._type.getRoot()),
    leftRange,
    rightRange;

  Type.DomUtilities.unwrap(
       enclosingTag);

  leftRange = Type.Range.
       fromPositions(this._type.
       getRoot(), tagPositions.
       start, selPositions.start);
  if (!leftRange.isCollapsed()) {
    this.inline(tagName, leftRange
         );
  }

  rightRange = Type.Range.
       fromPositions(this._type.
       getRoot(), selPositions.end
       , tagPositions.end);
  if (!rightRange.isCollapsed()) {
    this.inline(tagName,
         rightRange);
  }

  return this;

};

/**
 *
 * @param cmd
 * @param typeRange
 * @param params
 * @returns {Type.Formatter}
 * @private
 */
this.block = function (cmd,
     typeRange, params) {
  return this.inline.apply(this,
       arguments);
};

/**
 *
```

```
   * @param tag
   * @param typeRange
   * @returns {*}
   * @private
   */
  this._getStartNode = function (tag
      , typeRange) {
    return typeRange.startTagIs(tag)
         ? typeRange.
        getStartElement() :
        typeRange.
        splitStartContainer();
  };

  /**
   *
   * @param tag
   * @param typeRange
   * @returns {*}
   * @private
   */
  this._getEndNode = function (tag,
      typeRange) {
    return typeRange.endTagIs(tag) ?
         typeRange.getEndElement()
        : typeRange.
        splitEndContainer();
  };

  /**
   * Takes a tag name and returns
       the handler function for
       formatting
   * the DOM with this tag by
       checking if it is an inline
       or block tag.
   *
   * Todo Maybe use fallback http://
       stackoverflow.com/a
       /2881008/1183252 if tag is
       not found
   *
   * @param {String} tag - The name
       of the tag that the DOM
       should be
   *     formatted with.
   * @returns {inline|block|_noop} -
        The handler function for
       inline
   *     or block tags, or _noop if
       the tag is unknown.
   * @private
   */
  this._handlerFor = function (tag)
      {
    tag = tag.toLowerCase();
    if (this._inlineTags.indexOf(tag
        ) > -1) return this.inline;
    if (this._blockTags.indexOf(tag)
        > -1) return this.block;
    Type.Development.debug('Tag "' +
        tag + '" not implemented')
        ;
    return this._noop;
  };

  /**
   * Multi-purpose no-op handler
   *
   * @returns {Type.Formatter}
   * @private
   */
  this._noop = function () {
    return this;
  };

}).call(Type.Formatter.prototype);

module.exports = Type.Formatter;
```

Listing 14: formatter.js

```
'use strict';

var Type = require('./core');

/**
 * An editor's caret. We cannot use
     the browser's native caret
     since we do not utilize
 * native inputs (a textarea or an
     element that is set to
     contenteditable). We emulate
 * a caret with a blinking div. This
      class manages that div and
     provides methods to
 * position it.
 *
 * Creates a new Caret and adds a
     hidden div (visual
     representation of the caret)
     to
 * the DOM
 *
 * @param {Node|{constrainingNode:
     Node,color:string}|Type}
     options
 * @class Caret
 * @constructor
 */
Type.Caret = function (options) {

  options = options || {
      constrainingNode: null, color
      : null};

  if (options.typeEditor === true) {
    options = { constrainingNode:
        options.getRoot(), color:
        null };
  }

  if (Type.DomUtilities.isNode(
      options)) {
    options = { constrainingNode:
        options, color: null };
  }

  this.callbacks = {};

  this._constrainingNode = options.
      constrainingNode || document.
      body;
  this.caretEl = this._createElement
      (options.color);
  //this.moveTo(this.
      _constrainingNode);
  //this._hide();

};

(function () {

  /**
   * The id attribute of the caret
       container element as created
        by
   * _getElementContainer()
   *
   * @type {string}
   * @private
   */
  this._containerId = Type.Settings.
      prefix + 'caret-container';

  /**
   * Moves the caret left by one
       character
   *
   * @returns {Type.Caret}
   */
  this.moveLeft = function () {
    if (this.offset <= this.
        _visibleTextOffsets(this.
        textNode).start) {
      var prevTextNode = this.
          _prevTextNode(this.
          textNode);
      if(prevTextNode !== null) this
          .moveTo(prevTextNode,
          this._visibleTextOffsets(
          prevTextNode).end);
    } else {
      this._setOffset(this.offset -
          1);
    }
    return this;
  };

  /**
   * Moves the caret right by one
       character
   *
   * @returns {Type.Caret}
   */
  this.moveRight = function () {
    if (this.offset >= this.
        _visibleTextOffsets(this.
        textNode).end) {
      var nextTextNode = this.
          _nextTextNode(this.
          textNode);
      if(nextTextNode !== null) this
          .moveTo(nextTextNode,
          this._visibleTextOffsets(
          nextTextNode).start);
    } else {
      this._setOffset(this.offset +
          1);
    }
    return this;
  };

  /**
   * Moves the caret up by one line.
   * Tries to preserve horizontal
       position.
   *
   * Todo prevNode handling not nice
   * Todo should only walk 1 line
   *
   * Internally, this will create a
       collapsed range at the caret
       's offset and move
   * it left, character by character
       , and stop in the line above
       the caret when it's
   * horizontally aligned with it.
       The caret will then be moved
       to that position.
   *
   * @returns {Type.Caret}
   */
  this.moveUp = function () {

    // Shorthand variables
    var node = this.textNode,
      offset = this.offset,
      prevNode = node;

    // Initial range and positions
    var range  = this._createRange(
        node, offset),
      rangePos = this.
          _getPositionsFromRange(
          range),
      caretPos = this.
          _getRectAtOffset(this.
          offset),
      lastRangeLeft;

    // Move the range as described
        in the method's description
    while( prevNode !== null // &&
        offset > 0 &&
          && (!rangePos || (rangePos
              .top == caretPos.top
              || rangePos.left >
              caretPos.left)) ) {
      if(offset <= 0) {
        prevNode = this.
            _prevTextNode(node);
        if(prevNode !== null) {
          node = prevNode;
          offset = prevNode.length;
              // TODO Check auf !
              rangePos ist nicht
              noetig wenn
              _visibleTextOffsets
              verwendet werden, da
              unsichtbarer text nie
               selektiert wird
      }
    } else {
      offset--;
    }
    range.setStart(node, offset);
    range.collapse(true);
    lastRangeLeft = rangePos.left;
    rangePos = this.
        _getPositionsFromRange(
        range);
  }

  // If the range moved up, check
      2 characters above the
      caret to find a precise pos
      .
  if(rangePos.top < caretPos.top)
      {
    if(this._compareDeltaTo(
        caretPos.left,
        lastRangeLeft, rangePos.
        left) == -1) {
      offset += 1;
    }
    this.moveTo(node, offset);
  }

  // Chaining
  return this;
};

/**
 * Moves the caret down by one
     line.
 * Tries to preserve horizontal
     position.
 *
 * Todo nextNode handling not nice
 * Todo should only walk 1 line
 *
 * @returns {Type.Caret}
 */
this.moveDown = function () {

  // Shorthand variables
  var node = this.textNode,
    offset = this.offset,
    nextNode = node;

  // We are gonna create a range
      and move it through
  // the text until it is
      positioned 1 line below
  // the caret's position at
      around the same horizontal
  // position
  var range     = this.
        _createRange(node, offset),
    rangePos    = this.
        _getPositionsFromRange(
        range),
    caretPos    = this.
        _getRectAtOffset(this.
        offset),
    visibleText = this.
        _visibleTextOffsets(node)
        ;
    lastRangeRight;

  // Move the range right letter
      by letter. The range will
      start
  // in the same line and we keep
      moving it until it reaches
      the
  // next line and stop moving
      when it has moved further
      right
  // than the caret. That means
      the range will be one line
      below
  // the caret and in about the
      same horizontal position.
  while( nextNode !== null // &&
      offset < node.length &&
    && (!rangePos || (rangePos.
        bottom == caretPos.bottom
          || rangePos.right <
        caretPos.right))
  ) { // TODO gucken ob sich das
      noch irgendwie aufharngen
      kann wenn caret am ende des
       textes ist und rangePos
      nicht gesetzt ist
    if(offset >= visibleText.end
        /*node.length*/) {
      nextNode = this.
          _nextTextNode(node);
      if(nextNode !== null) {
        node = nextNode;
        visibleText = this.
            _visibleTextOffsets(
            node);
        offset = 0;
      }
    } else {
      offset++;
    }
    range.setEnd(node, offset);
    range.collapse(false);
    lastRangeRight = rangePos.
        right;
    rangePos = this.
        _getPositionsFromRange(
        range);
  }

  // The text might have only one
      line, we check to see if
      the range
  // has actually moved lower than
       the caret and then move
      the caret
  // In any case we moved the
      offset too far by 1
      character so we
  // we need to subtract it
  if(rangePos.bottom > caretPos.
      bottom) {
    if(this._compareDeltaTo(
        caretPos.right,
        lastRangeRight, rangePos.
        right) == -1) {
      offset -= 1;
      }
      this.moveTo(node, offset);
    }

    // Chaining
    return this;
  };

  /**
   * Moves the charet by the number
        of chars passed to as
        numChars
   * @param {number} numChars - The
        number of chars the caret
        should be moved by.
   *      Accepts negative values.
   * @returns {*}
   */
  this.moveBy = function (numChars)
      {
    var offset = this.getOffset();
    if (offset === null) return this
        ;
    this.setOffset(Math.max(0, this.
        getOffset() + numChars));
    return this;
  };

  /**
   * Places the caret in a text node
        at a given position
   *
   * @param {Node} node - The (text)
        {Node} in which the caret
        should be placed
   * @param {number} [offset=0] -
        The character offset where
        the caret should be moved to
   * @returns {Type.Caret}
   */
  this.moveTo = function (node,
      offset) {
    if (node.nodeType !== Node.
        TEXT_NODE) {
      node = Type.DomWalker.first(
          node, 'text');
    }
    if (node === null) {
      throw new Error('Node
          parameter must be or
          contain a text node');
    }
    if (node === this.textNode &&
        offset === null) {
      return this;
    }
    this.textNode = node;
    this._setOffset(offset || 0);
    return this;
  };

  /**
   * Inserts a given {string} at the
        caret's current offset in
      the caret's
   * current text node
   *
   * Todo this method needs to go
      somewhere else
   *
   * @param {string} str - The {
      string} that will be be
      inserted
   * @returns {Type.Caret}
   */
  this.insertText = function (str) {

    this._callbacksFor('insertText',
        str);

    if(/^[\n\r]+$/.test(str)) {

      var newNode = this.textNode.
          splitText(this.offset);
      newNode.parentNode.
          insertBefore(document.
          createElement('br'),
          newNode);
      this.moveTo(newNode, 0);
      return this;

    } else {

      var nodeText = this.textNode.
          nodeValue;
      if (this.offset > 0) {
        this.textNode.nodeValue =
            nodeText.substring(0,
            this.offset)
          + str
          + nodeText.substring(this.
              offset, nodeText.
              length);
      } else {
        this.textNode.nodeValue =
            str + nodeText;
      }
      this._setOffset(this.offset +
          str.length);
      return this;

    }


    /*
     var nodeText = this.textNode.
         nodeValue,
     splitText, i, newTextNodes =
         [],
     parentNode = this.textNode.
         parentNode,
     tmpNode;

     if (this.offset > 0) {
      nodeText = nodeText.substring
          (0, this.offset)
        + str
        + nodeText.substring(this.
            offset, nodeText.length
            );
    } else {
      nodeText = str + nodeText;
    }
    splitText = nodeText.split(/(?:\
        r\n|\r|\n)/g);

    this.textNode.nodeValue =
        splitText[0];

    for(i=1; i<splitText.length; i
        ++) {
      tmpNode = document.
          createTextNode(splitText[
          i]);
      parentNode.insertBefore(
          tmpNode, this.textNode.
          nextSibling);
      if(i < splitText.length - 1)
        parentNode.insertBefore(
            document.createElement
            ('br'), this.textNode.
            nextSibling);
    }

    this.moveTo(tmpNode, tmpNode.
        length);
    */
  };

  /**
   * Removes one character left from
       the current offset
   * and moves the caret accordingly
   *
   * Todo this method needs to go
       somewhere else
   *
   * @param {number} [numChars] -
       Home many characters should
       be removed
   *      from the caret's position.
       A negative number will
       remove
   *      characters left from the
       caret, a positive number
       from the right.
   * @returns {Type.Caret}
   */
  this.removeCharacter = function (
      numChars) {
    numChars = numChars || -1;
    if ( (this.offset <= 0 &&
        numChars < 0) || (this.
        offset >= this.textNode.
        length && numChars > 0) ) {
      return this;
    }
    this._callbacksFor('
        removeCharacter', numChars)
        ;
    var str = this.textNode.
        nodeValue;
    if(numChars < 0) {
      this.textNode.nodeValue = str.
          substring(0, this.offset
          + numChars)
        + str.substring(this.offset,
            str.length);
      this._setOffset(this.offset +
          numChars);
    } else {
      this.textNode.nodeValue = str.
          substring(0, this.offset)
        + str.substring(this.offset
            + numChars, str.length)
            ;
    }
    return this;
  };

  /**
   * Todo JSDOC
   *
   * @param functionName
   * @param callback
   * @returns {Type.Caret}
   */
  this.registerCallback = function(
      functionName, callback) {
    this.callbacks[functionName] =
        this.callbacks[functionName
        ] || [];
    this.callbacks[functionName].
        push(callback);
    return this;
  };

  /**
   * Removes the caret div from the
       DOM. Also removes the caret
   * container if there are no more
       carets in it
   *
   * @returns {Type.Caret}
   */
  this.destroy = function () {
    if (typeof this.caretEl !== "
        object") {
      return this;
    }
    var container = this.
        _getElementContainer();
    container.removeChild(this.
        caretEl);
    if (!container.hasChildNodes())
        {
      container.parentNode.
          removeChild(container);
    }
    this.caretEl = null;
    return this;
  };

  /**
   * Returns the offset of the caret
       in the text
   * To be specific, this returns
       the character offset
       relative to the
   * given constraining element.
   *
   * @returns {number|null}
   */
  this.getOffset = function () {
    if (!this.textNode) return null;
    return Type.TextWalker.offset(
        this._constrainingNode,
        this.textNode, 0, this.
        offset);
```

```
};

/**
 * todo unify with moveTo API
 * @param offset
 * @returns {*}
 */
this.setOffset = function (offset) {
  var t = Type.TextWalker.nodeAt(this._constrainingNode, offset);
  this.moveTo(t.node, t.offset);
  return this;
};

/**
 * Returns the offset of the caret relative to its current text node
 * todo Use this method on every public access to this variable
 * todo make offset private
 * @returns {number|null}
 */
this.getNodeOffset = function () {
  return this.offset;
};

/**
 * Getter for this instance's text node
 * todo Use this method on every public access to this variable
 * todo make textNode private
 * @returns {Node|null}
 */
this.getNode = function () {
  return this.textNode;
};

/**
 * Sets the offset and displays the caret at the according
 * position
 *
 * @param {number} offset - The offset that should be set
 * @returns {Type.Caret}
 */
this._setOffset = function (offset) {
  this.offset = offset;
  this._moveElToOffset();
  this._resetBlink();
  this._scrollIntoView();
  this._callbacksFor('_setOffset');
  return this;
};

/**
 * Moves the caret div to the position of the current offset
 *
 * @returns {Type.Caret}
 * @private
 */
this._moveElToOffset = function () {
  var rect = this._getRectAtOffset(this.offset);
  this._moveElTo(rect.left, rect.top);
  this._setElHeight(rect.bottom - rect.top);
  return this;
};

/**
 * Moves the caret to the given position
 *
 * @param {number} x Horizontal position the caret should be moved to
 * @param {number} y Vertical position the caret should be moved to
 * @returns {Type.Caret}
 * @private
 */
this._moveElTo = function (x, y) {
  this.caretEl.style.left = x + 'px';
  this.caretEl.style.top = y + 'px';
  return this;
};

/**
 * Todo jsdoc
 *
 * @param h
 * @returns {*}
 * @private
 */
this._setElHeight = function(h) {
  this.caretEl.style.height = h + 'px';
  return this;
};

/**
 * Scrolls page to show caret
 *
 * @returns {Type.Caret}
 * @private
 */
this._scrollIntoView = function () {
  //this.caretEl.scrollIntoView();
  return this;
};

/**
 * Makes the caret blink
 *
 * @returns {Type.Caret}
 */
this._blink = function () {
    this._removeClass(this.caretEl, 'hide');
    this._addClass(this.caretEl, 'blink');
    return this;
  };

  /**
   * Hides the caret
   *
   * @returns {Type.Caret}
   */
  this._hide = function () {
    this._removeClass(this.caretEl, 'blink');
    this._addClass(this.caretEl, 'hide');
    return this;
  };

  /**
   * Resets the blink animation by recreating the caret div element
   * Todo Maybe find a better way to reset the blink animation, DOM = slow
   *
   * @returns {Type.Caret}
   * @private
   */
  this._resetBlink = function () {
    var newCaret = this.caretEl.cloneNode(true);
    this.caretEl.parentNode.replaceChild(newCaret, this.caretEl);
    this.caretEl = newCaret;
    return this;
  };

  /**
   * Todo Maybe make a magic function that calls callbacks for functions automatically
   *
   * @param functionName
   * @param params
   * @private
   */
  this._callbacksFor = function(functionName, params) {
    var i;
    params = Array.prototype.slice.call(arguments, 1);
    if(this.callbacks[functionName]) {
      for(i=0; i<this.callbacks[functionName].length; i++) {
        this.callbacks[functionName][i].apply(this, params);
      }
    }
  };

  /**
   * TODO Possible code duplication with other code operating on the DOM like {BrowserInput}
   * TODO Caching instead of traversing every time
   * TODO We check for the _constrainingNode but this concept isn't really/properly used by other parts of the code
   *
   * @param el
   * @param returnMe
   * @returns {*}
   * @private
   */
  this._nextTextNode = function(el, returnMe) {

    var parent = el.parentNode;

    if(returnMe === true && this._isTextNodeWithContents(el)) {
      return el;
    }

    if(el.childNodes.length) {
      return this._nextTextNode(el.childNodes[0], true);
    }

    if(el.nextSibling !== null) {
      return this._nextTextNode(el.nextSibling, true);
    }

    while(parent !== this._constrainingNode) {
      if(parent.nextSibling !== null) {
        return this._nextTextNode(parent.nextSibling, true);
      }
      parent = parent.parentNode;
    }

    return null;
  };

  /**
   * TODO Possible code duplication with other code operating on the DOM like {BrowserInput}
   * TODO Caching instead of traversing every time
   * TODO We check for the _constrainingNode but this concept isn't really/properly used by other parts of the code
   *
   * @param el
   * @param returnMe
   * @returns {*}
   * @private
   */
  this._prevTextNode = function(el,
      returnMe) {

    var parent = el.parentNode;

    if(returnMe === true && this.
        _isTextNodeWithContents(el)
        ) {
      return el;
    }

    if(el.childNodes.length) {
      return this._prevTextNode(el.
          childNodes[el.childNodes.
          length - 1], true);
    }

    if(el.previousSibling !== null)
        {
      return this._prevTextNode(el.
          previousSibling, true);
    }

    while(parent !== this.
        _constrainingNode) {
      if(parent.previousSibling !==
          null) {
        return this._prevTextNode(
            parent.previousSibling
            , true);
      }
      parent = parent.parentNode;
    }

    return null;
  };

  /**
   * Todo: code duplication in
       browser.js, there should be
       a dom util module
   * @param node
   * @returns {boolean}
   * @private
   */
  this._isTextNodeWithContents =
      function(node) {
    return node.nodeType == 3 &&
        /[^\t\n\r ]/.test(node.
        textContent);
  };

  /**
   * Finds the whitespace at the
       beginning and the end of a
       text node and
   * returns their lengths
   *
   * @param textNode
   * @returns {{start: number, end:
       number}}
   * @private
   */
  this._visibleTextOffsets =
      function(textNode) {
    var startWhitespace = textNode.
        nodeValue.match(/^[\t\n\r
        ]+/g) || [''];
    var endWhitespace   = textNode.
        nodeValue.match(/[\t\n\r ]+
        $/g) || [''];
    return {
      start : startWhitespace[0].
          length,
      end   : textNode.nodeValue.
          length - endWhitespace
          [0].length
    }
  };

  /**
   * Utility method to add a class
       to an element
   * Todo There should be a separate
       utility module for stuff
       like this - yes
       dom_utilities
   *
   * @param {Element} el - The {
       Element} that the class
       should be added to
   * @param {string} className - The
       class to be removed
   * @returns {Type.Caret}
   * @private
   */
  this._addClass = function (el,
      className) {
    if (el.classList) {
      el.classList.add(className);
    } else {
      el.className += ' ' +
          className;
    }
    return this;
  };

  /**
   * Utility method to remove a
       class from an element
   * Todo There should be a separate
       utility module for stuff
       like this - yes
       dom_utilities
   *
   * @param {Element} el - The {
       Element} that the class
       should be removed from
   * @param {string} className - The
       class to be removed
   * @returns {Type.Caret}
   * @private
   */
  this._removeClass = function (el,
      className) {
    if (el.classList) {
      el.classList.remove(className)
          ;
    } else {
      var regex = new RegExp('(^|\\b
          )' + className.split(' ')
        .join('|') + '(\\b|$)', 'gi');
      el.className = el.className.replace(regex, ' ');
    }
    return this;
  };

  /**
   * Calculates The delta between a given the pivot (a {number}) and a as
   * well as b (both {number}s) and returns -1 if a is closer to pivot,
   * 1 of b is closer to pivot and 0 if both numbers are equally close.
   *
   *
   * @param {number} pivot - A number to which a and be will be compared to
   * @param {number} a - An arbitrary number
   * @param {number} b - An arbitrary number
   * @returns {number}
   * @private
   */
  this._compareDeltaTo = function (pivot, a, b) {
    var deltaA = Math.abs(pivot - a),
        deltaB = Math.abs(pivot - b);
    if (deltaA === deltaB) return 0;
    return deltaA < deltaB ? -1 : 1;
  };

  /**
   * Returns a {ClientRect} with the boundaries enclosing a character at a
   * given offset in a text node
   *
   * @param {Node} [node=this.textNode] - The text node which containing the
   *     character we which to fetch the boundaries of.
   * @param {number} offset - The offset of the character we which to fetch
   *     the boundaries of
   * @returns {{top: number, right: number, bottom: number, left : number}}
   * @private
   */
  this._getRectAtOffset = function (node, offset) {
    if (typeof node === "number") {
      offset = node;
      node = this.textNode;
    }
    return this._getPositionsFromRange(this._createRange(node, offset));
  };

  /**
   * Returns the positions from a {ClientRect} relative to the scroll
   * position
   *
   * @param {Range} range The {Range} that should be measured
   * @returns {{top: number, right: number, bottom: number, left : number}}
   * @private
   */
  this._getPositionsFromRange = function (range) {
    var scroll = this._getScrollPosition();
    var rect = range.getClientRects()[0];
    if(!rect) {
      return false;
    }
    return {
      top    : rect.top + scroll.top,
      right  : rect.right + scroll.left,
      bottom : rect.bottom + scroll.top,
      left   : rect.left + scroll.left
    };
  };

  /**
   * Return's the window's horizontal an vertical scroll positions
   *
   * @returns {{top: (number), left: (number)}}
   * @private
   */
  this._getScrollPosition = function () {
    return {
      top  : window.pageYOffset || document.documentElement.scrollTop,
      left : window.pageXOffset || document.documentElement.scrollLeft
    };
  };

  /**
   * Creates a {Range} and returns it
   *
   * @param {Node} startNode - The node in which the created range should begin
   * @param {number} start - The offset at which the range should start
   * @param {number} [end=start] - The offset at which the range should end
   *     Optional. Defaults to the start offset.
   * @param {Node} [endNode=node] - The node in which the created range should end.
   *     Optional. Defaults to the start node.
   * @returns {Range}
   * @private
   */
  this._createRange = function(startNode, start, end, endNode) {
    var range = window.document.createRange();
    range.setEnd(endNode || startNode, end || start);
    range.setStart(startNode, start);
    return range;
  };

  /**
   * Creates a div (the visual representation of the caret) and returns it.
   *
   * @returns {HTMLElement}
   * @private
   */
  this._createElement = function (color) {
    var container = this._getElementContainer(),
        el = window.document.createElement('div');
    el.className = Type.Settings.prefix + 'caret ' + color;
    container.appendChild(el);
    return el;
  };

  /**
   * All div representations of carets will be appended to a single
   * container. This method returns this container and creates it
   * if it has not been created yet.
   *
   * Todo use container from dom_utilites
   *
   * @returns {HTMLElement}
   * @private
   */
  this._getElementContainer = function () {
    var container = window.document.getElementById(this._containerId);
    if (container === null) {
      container = window.document.createElement('div');
      container.setAttribute('id', this._containerId);
      window.document.body.appendChild(container);
    }
    return container;
  }

}).call(Type.Caret.prototype);

module.exports = Type.Caret;
```

Listing 15: caret.js

```
'use strict';

var Type = require('./core');

/**
 *
 * todo internal differenciation / abstraction of x and y *and scroll positions* for easier redrawing
 *
 * @param {number} [x1] - Horizontal position of the overlay
 * @param {number} [y1] - Vertical position of the overlay
 * @param {number} [x2] - x2 of the overlay
 * @param {number} [y2] - y2 of the overlay
 * @param {boolean} [show] - Set to false if you do not wish
 *     for the element to be shown. Defaults to true
 * @constructor
 */
Type.SelectionOverlay = function (x1, y1, x2, y2, show) {
  if (show !== false) {
    this.show(x1, y1, x2, y2);
  }
  this._setValues(x1, y1, x2, y2);
  this._anchor = {x: x1, y: y1};
};

(function () {

  /**
   * Will set the position and dimension values and update
   * the div styles
   *
   * @param {number|string} [x1] - Horizontal position of the overlay
   *     or either one of the
```

```
         strings 'left', 'right' or '
         line', which
   *       will span the overlay to
         the left side of the line of
          the
   *      textNode, the right side or
          the entire line
   * @param {number} [y1] - Vertical
          position of the overlay
   * @param {number} [x2] - x2 of
         the overlay
   * @param {number} [y2] - y2 of
         the overlay
   * @returns {Type.SelectionOverlay
         } - This instance
   */
  this.set  = function (x1, y1, x2,
      y2) {

    if (x1 === 'left') {
      x1 = this._textleft();
      x2 = null;
    }

    if (x1 === 'right') {
      x1 = null;
      x2 = this._textRight();
    }

    if (x1 === 'line') {
      x1 = this._textleft();
      x2 = this._textRight();
    }

    x1 = x1 === undefined ? null :
         x1;
    y1 = y1 === undefined ? null :
         y1;
    x2 = x2 === undefined ? null :
         x2;
    y2 = y2 === undefined ? null :
         y2;

    this._draw(x1, y1, x2, y2);
    this._setValues(x1, y1, x2, y2);

    return this;
  };

  /**
   *
   * @param x1
   * @param y1
   * @param x2
   * @param y2
   * @returns {Type.SelectionOverlay
         } - This instance
   */
  this.show = function (x1, y1, x2,
      y2) {
    this._el = this._createElement()
        ;
    this._draw(x1, y1, x2, y2);
    return this;
  };

  /**
   *
   * @returns {Type.SelectionOverlay
         } - This instance
   */
  this.hide = function () {
    Type.DomUtilities.removeElement(
         this._el);
    this._el = null;
    return this;
  };

  /**
   *
   * @param {number|string} x
   * @param {number} [y]
   * @returns {Type.SelectionOverlay
         } - This instance
   */
  /*this.anchor = function (x, y) {

    if (x === 'left') {
      x = this._textleft();
      y = null;
    }

    if (x === 'right') {
      x = this._textRight();
      y = null;
    }

    if (x !== null && x !==
        undefined) {
      this._anchor.x = x;
    }

    if (y !== null && y !==
        undefined) {
      this._anchor.y = y;
    }

    return this;

  };*/

  /**
   * Sets the horizontal start or
       end of this overlay
       depending
   * whether the value given is left
       or right of the anchor.
   * Will also set the other end to
       the anchor's position.
   *
   * @param {number} x - The
       horizontal position
   * @returns {Type.SelectionOverlay
       } - This instance
   */
  /*this.setXFromAnchor = function (
      x) {
    if (x === null || x ===
        undefined) {
      this.set(this._anchor.x, null,
           this._anchor.x, null);
    } else {
      if (x < this._anchor.x) this.
```

```
          set(x, null, this._anchor
          .x, null);
      if (x > this._anchor.x) this.
          set(this._anchor.x, null,
           x, null);
    }
    return this;
  };*/

  /**
   * Returns whether or not this
       overlay is actually visible
   *
   * @returns {boolean}
   */
  this.visible = function () {
    return !(this.x1 === this.x2 ||
        this.y1 === this.y2);
  };

  /**
   * Removes the overlay div and
       resets all position and
   * dimension values
   *
   * @returns {Type.SelectionOverlay
       } - This instance
   */
  this.remove = function () {
    if (this._el) {
      Type.DomUtilities.
          removeElement(this._el);
    }
    this._el = null;
    this.x1 = null;
    this.y1 = null;
    this.x2 = null;
    this.y2 = null;
    this._anchor = null;
    return this;
  };

  /**
   * Sets all dimension and position
       values to the given
   * values unless null is given
   *
   * @param {number} [x1] -
       Horizontal position of the
       overlay
   * @param {number} [y1] - Vertical
        position of the overlay
   * @param {number} [x2] - x2 of
       the overlay
   * @param {number} [y2] - y2 of
       the overlay
   * @returns {Type.SelectionOverlay
       } - This instance
   * @private
   */
  this._setValues = function (x1, y1
      , x2, y2) {
    if (x1 !== null) this.x1 = x1;
    if (y1 !== null) this.y1 = y1;
    if (x2 !== null) this.x2 = x2;
    if (y2 !== null) this.y2 = y2;
    return this;
  };

  /**
   * Sets dimension and position
       values to th element's style
   * unless they are not different
       to the current values.
   *
   * @param {number} [x1] -
       Horizontal position of the
       overlay
   * @param {number} [y1] - Vertical
        position of the overlay
   * @param {number} [x2] - x2 of
       the overlay
   * @param {number} [y2] - y2 of
       the overlay
   * @returns {Type.SelectionOverlay
       } - This instance
   * @private
   */
  this._draw = function (x1, y1, x2,
      y2) {

    if (!this._el) {
      return this;
    }

    // If x1 has changed, reposition
    if (x1 !== null && x1 !== this.
        x1) {
      this._el.style.left   = x1 + '
          px';
    }

    // If x1 or x2 have changed,
        recalculate the width
    if ((x1 !== null && x1 !== this.
        x1) || (x2 !== null && x2
        !== this.x2)) {
      x1 = x1 !== null ? x1 : this.
          x1;
      x2 = x2 !== null ? x2 : this.
          x2;
      this._el.style.width  = (x2-x1
          ) + 'px';
    }

    // If y1 has changed, reposition
    if (y1 !== null && y1 !== this.
        y1) {
      this._el.style.top    = y1 + '
          px';
    }

    // If y1 or y2 have changed,
        recalculate the height
    if ((y1 !== null && y1 !== this.
        y1) || (y2 !== null && y2
        !== this.y2)) {
      y1 = y1 !== null ? y1 : this.
          y1;
      y2 = y2 !== null ? y2 : this.
          y2;
      this._el.style.height  = (y2-
          y1) + 'px';
    }

    return this;
  };

  /**
   * Creates and returns the visible
       selection overlay element
   *
   * @returns {Element}
   * @private
   */
  this._createElement = function ()
      {
    return Type.DomUtilities.
        addElement('div', '
        selection');
  };

}).call(Type.SelectionOverlay.
    prototype);

/**
 *
 * @param {Range} range
 * @returns {Type.SelectionOverlay}
 */
Type.SelectionOverlay.fromRange =
    function (range) {
  var rect = Type.SelectionOverlay.
      _getPositionsFromRange(range)
      ;
  return new Type.SelectionOverlay(
      rect.left, rect.top, rect.
      right, rect.bottom, true,
      range.startContainer);
};

/**
 *
 * @param x
 * @param y
 * @returns {Type.SelectionOverlay}
 * @deprecated
 */
//Type.SelectionOverlay.fromPosition
    = function (x, y) {
//  var range = document.
    caretRangeFromPoint(x, y);
//  return Type.SelectionOverlay.
    fromRange(range)
//};

/**
 * Return's the window's horizontal
    an vertical scroll positions
 *
 * todo code duplication to caret.
    _getScrollPosition
 *
 * @returns {{top: (number), left: (
    number)}}
 * @private
 */
Type.SelectionOverlay.
    _getScrollPosition = function
    () {
  return {
    top  : window.pageYOffset ||
        document.documentElement.
        scrollTop,
    left : window.pageXOffset ||
        document.documentElement.
        scrollLeft
  };
};

/**
 * Returns the positions from a {
    ClientRect} relative to the
    scroll
 * position
 *
 * todo code duplication to caret.
    _getPositionsFromRange
 *
 * @param {Range} range The {Range}
    that should be measured
 * @returns {{top: number, right:
    number, bottom: number, left:
    number}}
 * @private
 */

Type.SelectionOverlay.
    _getPositionsFromRange =
    function (range) {
  var scroll = Type.SelectionOverlay
      ._getScrollPosition();
  var rect = range.getClientRects()
      [0];
  if(!rect) {
    return null;
  }
  return {
    top    : rect.top + scroll.top,
    right  : rect.right + scroll.
        left,
    bottom : rect.bottom + scroll.
        top,
    left   : rect.left + scroll.left
  };
};

module.exports = Type.
    SelectionOverlay;
```

Listing 16: selection_overlay.js

```
'use strict';

var Type = require('./core');

/**
 *
 * @param {Type} type
 * @constructor
 */
Type.Selection = function (type) {
  this._init(type);
};

(function () {

  /**
   * Resets (removes) the current
       selection if there is one,
       sets a new anchor at
   * the given coordinates and sets
       up a new selection at the
       node and offset found
   * at the coordinates.
   *
   * @param {number} x - Absolute
       horizontal position on the
       document
   * @param {number} y - Absolute
       vertical position on the
       document
   * @returns {Type.Selection} -
       This instance
   */
  this.beginAt = function (x, y) {
    this.unselect();
    this._setAnchor(x, y);
    return this._startRangeAt(this.
        _anchor.node, this._anchor.
        offset);
  };

  /**
   * Will move the end or the start
       of the selection to the node
       and offset found at
   * the given coordinates. Whether
       the start or the end will be
       moved depends on
   * whether the coordinates are on
       top / left of this selection
       's anchor or below /
   * right of it.
   *
   * @param {number} x - Absolute
       horizontal position on the
       document
   * @param {number} y - Absolute
       vertical position on the
       document
   * @returns {Type.Selection} -
       This instance
   */
  this.moveTo = function (x, y) {
    var range = Type.Range.fromPoint
        (x, y);
    this._addElement(range.
        endContainer);
    if (x < this._anchor.x || y <
        this._anchor.y) {
      this._moveStartTo(range.
          endContainer, range.
          endOffset);
    } else {
      this._moveEndTo(range.
          endContainer, range.
          endOffset);
    }
    return this;
  };

  /**
   * Returns the contents of the
       selection or null if there
       is no selection
   *
   * @returns {DocumentFragment}
   */
  this.getContent = function () {
    return this._range ? this._range
        .cloneContents() : null;
  };

  /**
   * todo we should really use type
       ranges and much of this
       implementation should go
       there
   * todo this does not work when
       spanning over multiple text
       nodes (for instance in case
       of formatting)
   * todo since multiple nodes
       making up a single text or
       sometimes even words, maybe
       there should be an
       abstraction and layer /
       class for this
   * @param x
   * @param y
   * @returns {*}
   */
  this.selectWordAt = function (x, y
      ) {

    var charAtStart, charAtEnd,
      whitespace = new RegExp('\\s')
        ,
      endLength = this._range.
          endContainer.nodeValue.
          length,
      startOffset = this._range.
          startOffset,
      endOffset = this._range.
          endOffset,
      startFound = false,
      endFound = false;

    this.beginAt(x, y);

    do {
      charAtStart = this._range.
          startContainer.nodeValue.
          charAt(this._range.
          startOffset - 1);

      if (startOffset > 1 && !
            whitespace.test(
            charAtStart)) {
          if (startOffset > 1) {
            startOffset -= 1;
            this._range.setStart(this.
                 _range.startContainer
                 , startOffset);
          }
        } else {
          startFound = true;
        }

      } while (!startFound);

      do {
        charAtEnd = this._range.
             endContainer.nodeValue.
             charAt(this._range.
             endOffset);

        if (endOffset < endLength && !
             whitespace.test(charAtEnd
             )) {
          if (endOffset < endLength) {
            endOffset += 1;
            this._range.setEnd(this.
                 _range.endContainer,
                 endOffset);
          }
        } else {
          endFound = true;
        }

      } while (!endFound);

      this._imitateRangeAppending();

      return this;
    };

    /**
     * Removes all selection overlays
         and resets internal
         variables.
     * @returns {Type.Selection} -
         This instance
     */
    this.unselect = function () {
      this._removeOverlays();
      this._elements = {};
      this._range = null;
      this._anchor = null;
      return this;
    };

    /**
     * Returns an object that can be
         used to recreate the current
     * selection using {@link Type.
         Selection#restore}
     * @returns {{from: Element, start
         : number, end: number}}
     */
    this.save = function () {
      return this.getRange().save(this
          ._root);
    };

    /**
     * Selects text from an object
         returned by {@link Type.
         Selection#save}
     * or {@link Type.Range#save}
     * @param bookmark
     * @returns {Type.Selection} -
         This instance
     */
    this.restore = function (bookmark)
         {
      this.unselect();
      this._range = Type.Range.load(
          bookmark).getNativeRange();
      this._imitateRangeAppending();
      return this;
    };

    /**
     * Returns a {Type.Range} spanning
         over the currently selected
         text.
     * @returns {Type.Range}
     */
    this.getRange = function () {
      return Type.Range.fromRange(this
          ._range);
    };

    /**
     * Returns the {Range} this
         selection spans over or null
         if nothing has been
     * selected yet.
     * @returns {Range|null}
     */
    this.getNativeRange = function ()
         {
      return this._range;
    };

    /**
     * Returns the start node and
         offset of this selection.
     * @returns {{node: Node, offset:
         number}|null}
     */
    this.getStart = function () {
      if (this._range) {
        return {node: this._range.
             startContainer, offset:
             this._range.startOffset};
      }
      return null;
    };

    /**
     * Returns the end node and offset
         of this selection.
     * @returns {{node: Node, offset:
         number}|null}
     */
    this.getEnd = function () {
      if (this._range) {
```

```
      return {node: this._range.endContainer, offset: this._range.endOffset};
    }
    return null;
  };

  /**
   * Returns whether or not this selection is visible. By checking if there currently
   * are any overlays and if the first overlay is actually visible. There should be
   * no case where there are visible overlays but the first overlay wouldn't be visible,
   * so this is a quick and performant way to check for the selection's visibility.
   *
   * @returns {boolean} - True if selection is hidden, false if there is a selection
   */
  this.collapsed = function () {
    return !this._overlays.length || !this._overlays[0].visible();
  };

  /**
   * Alias method for select() for better code readability. For initialization
   * all variables should be set to their default values. This is what select
   * does for us.
   *
   * @param {Type} type
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._init = function (type) {
    this._root = type.getRoot();
    return this.unselect();
  };

  /**
   * Creates a new {Range}, which will be the basis for drawing and this selection.
   * todo Use {Type.Range}? Should be cool if we don't use getRects or we make Type.Range more performant
   *
   * @param {Node} node - The text node that the selection should start in
   * @param {number} offset - The offset in the text node that the selection should start in
   * @returns {Type.Selection} - This instance
   */
  this._startRangeAt = function (node, offset) {
    this._range = window.document.createRange();
    this._range.setStart(node, offset);
    this._range.setEnd(node, offset);
    return this;
  };

  /**
   *
   * @param {Node} node - The text node that the selection should end in
   * @param {number} offset - The offset in the text node that the selection should end in
   * @returns {Type.Selection} - This instance
   */
  this._moveStartTo = function (node, offset) {
    this._range.setStart(node, offset);
    this._range.setEnd(this._anchor.node, this._anchor.offset);
    this._imitateRangePrepending();
    return this;
  };

  /**
   *
   * @param {Node} node - The text node that the selection should end in
   * @param {number} offset - The offset in the text node that the selection should end in
   * @returns {Type.Selection} - This instance
   */
  this._moveEndTo = function (node, offset) {
    this._range.setStart(this._anchor.node, this._anchor.offset);
    this._range.setEnd(node, offset);
    this._imitateRangeAppending();
    return this;
  };

  /**
   * Sets the anchor node, offset and position in this screen for this selection.
   * When a user draws a selection, what is being selected depends on whether he /
   * she moves his / her mouse before or behind the point he / she started to draw
   * the selection. The information in the anchor needs to be saved to implement
   * this behaviour.
   *
   * @param {number} x - Absolute horizontal position on the document
   * @param {number} y - Absolute vertical position on the document
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._setAnchor = function (x, y) {
    var range = Type.Range.fromPoint(x, y);
    this._anchor = {x: x, y: y, node: range.startContainer, offset: range.startOffset};
    this._addElement(this._anchor.node);
    return this;
  };

  /**
   * Creates {Type.SelectionOverlay}s that mimic the appearance of
   * the selection as drawn by {this._range}
   *
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._imitateRangePrepending = function () {

    // Required variables
    var rects = Type.Range.getClientRects(this._range),//this._range.getClientRects(),
      draw,
      overlay,
      i;

    // Resize and add overlays to match the range's rects
    for (i = rects.length - 1; i >= 0; i -= 1) {
      if (this._overlays[i]) {
        this._overlays[i].set(rects[i].left, rects[i].top, rects[i].right, rects[i].bottom);
      } else {
        draw = !this._matchesElementDimensions(rects[i]);
        overlay = new Type.SelectionOverlay(rects[i].left, rects[i].top, rects[i].right, rects[i].bottom, draw);
        this._overlays.unshift(overlay);
      }
    }

    // Remove overlays prepending the current range's rects
    while (this._overlays.length > rects.length) {
      this._overlays.shift().remove();
    }

    // Chaining
    return this;

  };

  /**
   * Creates {Type.SelectionOverlay}s that mimic the appearance of
   * the selection as drawn by {this._range}
   *
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._imitateRangeAppending = function () {

    // Required variables
    var rects = this._range.getClientRects(),
      draw,
      overlay,
      i;

    // Resize and add overlays to match the range's rects
    for (i = 0; i < rects.length; i += 1) {
      if (this._overlays[i]) {
        this._overlays[i].set(rects[i].left, rects[i].top, rects[i].right, rects[i].bottom);
      } else {
        draw = !this._matchesElementDimensions(rects[i]);
        overlay = new Type.SelectionOverlay(rects[i].left, rects[i].top, rects[i].right, rects[i].bottom, draw);
        this._overlays.push(overlay);
      }
    }

    // Remove overlays coming after the current range's rects
    while (this._overlays.length > rects.length) {
      this._overlays.pop().remove();
    }

    // Chaining
    return this;

  };

  /**
   * Todo scrolling
   *
   * @param {Node|Element} el - An element or a text node
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._addElement = function (el) {
    var rect, key;
    el = el.nodeType === 3 ? el.parentNode : el;
    rect = el.getBoundingClientRect();
    key = this._stringifyRect(rect);
    this._elements[key] = rect;
    return this;
  };

  /**
   *
   * @param {ClientRect} rect
   * @private
   */
  this._matchesElementDimensions = function (rect) {
    var key = this._stringifyRect(rect);
    return this._elements.hasOwnProperty(key);
  };

  /**
   * Removes all selection overlays
   *
   * @returns {Type.Selection} - This instance
   * @private
   */
  this._removeOverlays = function () {
    var i;
    this._overlays = this._overlays || [];
    for (i = 0; i < this._overlays.length; i += 1) {
      this._overlays[i].remove();
    }
    this._overlays = [];
    return this;
  };

  /**
   *
   * @param {ClientRect} rect
   * @returns {string}
   * @private
   */
  this._stringifyRect = function (rect) {
    var top    = rect.top.toString(),
      left   = rect.left.toString(),
      bottom = rect.bottom.toString(),
      right  = rect.right.toString();
    return top + left + bottom + right;
  };

}).call(Type.Selection.prototype);

module.exports = Type.Selection;
```

Listing 17: selection.js

```
'use strict';

var Type = require('./core');

/**
 * todo pasting
 * todo trigger events
 *
 * @param {Type} type
 * @constructor
 */
Type.Input = function (type) {

  this._type = type;
  //this._content = type.getContent();
  this._content = new Type.Content(type);
  this._writer = type.getWriter();
  this._caret = type.getCaret();
  this._selection = this._type.getSelection();
  this._el = this._createElement();
  this._elStyle = this._el.style;
  this._caretStyle = this._caret.caretEl.style;

  this._loadFilters();
  this._bindEvents();

};

(function () {

  /**
   * Adds a filter to the input pipeline
   *
   * @param {String} name - An identifier for the filter
   * @param {Object} filter - A filter
   * @returns {Type.Input}
   */
  this.addFilter = function (name, filter) {
    this._filters = this._filters || {};
    this._filters[name] = filter;
    return this;
  };

  /**
   * Removes a filter from the input pipeline
   *
   * @param {String} name - An identifier for the filter
   * @returns {Type.Input}
   */
  this.removeFilter = function (name) {
    this._filters = this._filters || {};
    if (this._filters.name) {
      delete this._filters.name;
    }
    return this;
  };

  /**
   * Getter for this instance's content.
   * @returns {Type.Content}
   */
  this.getContent = function () {
    return this._content;
  };

  /**
   *
   * @returns {Type.Input}
   * @private
   */
  this._loadFilters = function () {
    this._filters = this._filters || {};
    this._filters.undo = new Type.Input.Filter.Undo(this._type, this);
    this._filters.cmd = new Type.Input.Filter.Command(this._type, this);
    this._filters.caret = new Type.Input.Filter.Caret(this._type, this);
    this._filters.remove = new Type.Input.Filter.Remove(this._type, this);
    this._filters.lineBreaks = new Type.Input.Filter.LineBreaks(this._type, this);
    return this;
  };

  /**
   * Binds events on type's root element to catch keyboard
   * and mouse input.
   *
   * @returns {Type.Input}
   * @private
   */
  this._bindEvents = function () {
    this._bindKeyDownEvents();
    this._bindInputEvents();
    this._bindMouseEvents();
    return this;
  };

  /**
   * Some inputs needs to be interrupted and caught before it gets inserted
   * to the input element. This includes return keys for example
   *
   * @returns {Type.Input}
   * @private
   */
  this._bindKeyDownEvents = function () {

    this._el.addEventListener('keydown', function (e) {
      this._processFilterPipeline(e);
    }.bind(this), false);

    return this;
  };

  /**
   * Todo x-browser http://stackoverflow.com/a/8694125/1183252
   * Todo x-browser http://jsfiddle.net/MBags/ (?)
   *
   * @returns {Type.Input}
   * @private
   */
  this._bindInputEvents = function () {
    this._el.addEventListener('input', function (e) {
      this._onInput(e);
    }.bind(this), false);
    return this;
  };

  /**
   * Todo Legacy Internet Explorer and attachEvent https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
   *
   * @returns {Type.Input}
   * @private
   */
  this._bindMouseEvents = function
```

```
    () {

    var self = this;

    function dragSelection(e) {
      self._selection.moveTo(e.
          clientX, e.clientY);
    }

    function stopDraggingSelection()
        {
      document.removeEventListener('
          mousemove', dragSelection
          , false);
      document.removeEventListener('
          mouseup',
          stopDraggingSelection,
          false);
      self._el.innerHTML = '';
      self._el.appendChild(self.
          _selection.getContent());
      document.execCommand('
          selectAll', false, null);
    }

    function startDraggingSelection(
        e) {
      if (e.which === 1) {
        e.preventDefault();
        self._caret._hide();
        self._selection.beginAt(e.
            clientX, e.clientY);
        document.addEventListener('
            mousemove',
            dragSelection, false);
        document.addEventListener('
            mouseup',
            stopDraggingSelection,
            false);
      }
    }

    function caret(e) {
      if (self._selection.collapsed
          ()) {
        self.
            _moveCaretToMousePosition
            (e.clientX, e.clientY);
        self._caret._blink();
      }
      self._focusInput();
    }

    function selectWord(e) {
      self._selection.selectWordAt(e
          .clientX, e.clientY);
    }

    function contextmenu(e) {
      if (e.which === 3) {
        self.
            _moveCaretToMousePosition
            (e.clientX, e.clientY);
        self._caret._blink();
        self._moveElToPosition(e.
            clientX - 3, e.clientY
            - 3);
        self._el.focus();
        document.execCommand('
            selectAll', false, null
            );
      }
    }

    this._type.getRoot().
        addEventListener('
        contextmenu', contextmenu,
        false);
    this._type.getRoot().
        addEventListener('mousedown
        ', startDraggingSelection,
        false);
    this._type.getRoot().
        addEventListener('mouseup',
        caret, false);
    this._type.getRoot().
        addEventListener('dblclick
        ', selectWord, false);

    return this;

  };

  /**
   * Takes a {KeyboardEvent} and
       creates a {Type.Events.Input
       }. Then
   * iterates over all registered
       input filters in the
       pipeline and
   * has them process it in order.
       Will stop processing the
       event
   * when any handler of an input
       filter cancels the event.
       Returns
   * the resulting {Type.Events.
       Input}
   *
   * @param {KeyboardEvent} e
   * @returns {Type.Events.Input}
   * @private
   */
  this._processFilterPipeline =
      function (e) {
    var inputEvent = Type.Events.
        Input.fromKeyDown(e),
      name;
    for (name in this._filters) {
      if (this._filters.
          hasOwnProperty(name)) {
        this._processFilter(this.
            _filters[name],
            inputEvent);
        if (inputEvent.canceled) {
          e.preventDefault();
          break;
        }
      }
    }
    if (!inputEvent.canceled) {
      if (this._el.textContent.
          length > 2) {
        this._type.trigger('paste',
            [inputEvent]);
      } else {
        this._type.trigger('input',
            [inputEvent]);
      }
    }
    return inputEvent;
  };

  /**
   *
   * @param filter
   * @param {Type.Events.Input} e
   * @returns {Type.Events.Input}
   * @private
   */
  this._processFilter = function (
      filter, e) {
    var func = filter.keys[e.key];
    if (func) {
      filter[func](e);
    }
    if (!e.canceled && filter.keys.
        all) {
      filter[filter.keys.all](e)
    }
    return e;
  };

  /**
   *
   * @param {InputEvent} e
   * @returns {Type.Input}
   * @private
   */
  this._onInput = function (e) {
    this._content.insert(this._caret
        .textNode, this._caret.
        offset, this._el.
        textContent);
    this._caret._setOffset(this.
        _caret.offset + this._el.
        textContent.length); //
        todo better api
    this._el.innerHTML = '';
    return this;
  };

  /**
   *
   * @param x
   * @param y
   * @returns {*}
   * @private
   */
  this._moveCaretToMousePosition =
      function(x, y) {
    var range = Type.Range.fromPoint
        (x, y);
    if (range.startContainer.
        nodeType === 3) {
      this._caret.moveTo(range.
          startContainer, range.
          startOffset);
      this._caret._blink();
    }
    return this;
  };

  /**
   *
   * @param x
   * @param y
   * @returns {*}
   * @private
   */
  this._moveElToPosition = function(
      x, y) {
    this._el.style.left = x + 'px';
    this._el.style.top = y + 'px';
    return this;
  };

  /**
   *
   * @param sync
   * @returns {*}
   * @private
   */
  this._focusInput = function (sync)
      {
    if (sync) {
      this._el.focus();
    } else {
      window.setTimeout(function() {
           this._el.focus();}.bind(
          this), 0);
    }
    return this;
  };

  /**
   * Todo generalise and formalise
       and normalise adding
       elements to the domUtil.
       elementsContainer
   * @returns {Element}
   * @private
   */
  this._createElement = function ()
      {
    var div = document.createElement
        ('div');
    div.setAttribute('
        contenteditable', 'true');
    div.className = Type.Settings.
        prefix + 'input';
    Type.DomUtilities.
        getElementsContainer().
        appendChild(div);
    return div;
  };

}).call(Type.Input.prototype);

Type.Input.Filter = {};

module.exports = Type.Input;
```

Listing 18: input.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type event
 * @constructor
 */
Type.Events.Type = function () {
  this.canceled = false;
};

(function () {

  /**
   * Sets or gets data for this
       event. Parameters can be set
   * and retrieved like in jQuery:
   *
   * Call data with no params to
       retrieve all data set:
   * this.data() -> {}
   *
   * Pass a single string to get
       specific data:
   * this.data('foo')
   *
   * Pass a name value combination
       to set data
   * this.data('foo', 'bar')
   *
   * Pass an object to set multiple
       data
   * this.data({foo: 'foo', 'bar':'
       bar'})
   *
   * @param {(string|Object)} data -
        Either a plain object
   *     with keys and values to be
       set or a string that will
   *     be used as a name for a
       data setting. If you pass a
   *     string, pass a second
       parameter to set that data
   *     or no second parameter to
       retrieve that data.
   * @param {*} [value] - If the
       first parameter is a string,
   *     this value will be set to
       the key of the given first
   *     parameter. Any arbitrary
       value can be set.
   * @returns {Type.Events.Type
       |{}|*} Returns this instance
   *     if you set data or the
       according value if you get
   *     data. Will return all data
       in an object of you pass
   *     no parameters.
   */

  this.data = function (data, value)
      {

    // Initialize data object if not
        initialized yet
    this._data = this._data || {};

    // Pass a single option name to
        fetch it
    if (typeof data === "string" &&
        arguments.length === 1) {
      return this._data[data];
    }

    // Pass an option name and a
        value to set it
    if (typeof data === "string" &&
        arguments.length === 2) {
      data = {options: value};
    }

    // Pass an object of key-values
        to set them
    if (typeof data === "object") {
      Type.Utilities.extend(this.
          _data, data);
    }

    // Data of no params have been
        passed, otherwise this for
        chaining
    return arguments.length ? this :
        this._data;

  };

  /**
   * Sets this event instance to be
       cancelled
   *
   * @param {boolean} [doCancel] -
       Set to false to uncancel
   *     the event. All other values
       or no value at all
   *     will set the event to be
       cancelled
   * @returns {Type.Events.Type} -
       This instance
   */
  this.cancel = function (doCancel)
      {
    this.canceled = doCancel !==
        false;
    return this;
  };

}).call(Type.Events.Type.prototype);

module.exports = Type.Events.Type;
```

Listing 19: events/type.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type input event.
 * This is an abstraction for
     browser events that lead to an
      input in
 * the editor.
 *
 * @param {Object} options - Object
     holding parameters for the
     event
 * @param {string} [options.key] - A
      descriptive name for the key
 *     pressed as set in {@link Type
     .Events.Input.keyDownNames}.
 * @param {number} [options.keyCode]
      - The key code of the key
     pressed
 * @param {boolean} [options.shift]
      - Whether or not the shift key
      has
 *     been pressed together with
     the given key.
 * @param {boolean} [options.alt] -
     Whether or not the alt key has
 *     been pressed together with
     the given key.
 * @param {boolean} [options.ctrl] -
      Whether or not the control
     key has
 *     been pressed together with
     the given key.
 * @param {boolean} [options.meta] -
      Whether or not the command
     key has
 *     been pressed together with
     the given key (for os x users)
     .
 * @constructor
 */
Type.Events.Input = function (
    options) {

  options = options || {};

  this.key      = options.key      ||
      null;
  this.keyCode = options.keyCode ||
      null;
  this.shift    = options.shift    ||
      false;
  this.alt      = options.alt      ||
      false;
  this.ctrl     = options.ctrl     ||
      false;
  this.meta     = options.meta     ||
      false;
  this.cmd      = (!Type.Environment.
      mac && options.ctrl) || (Type
      .Environment.mac && options.
      meta);

  this.canceled = false;

};

/**
 * Inherit from general Type event
 */
Type.OOP.inherits(Type.Events.Input,
    Type.Events.Type);

/**
 * Maps character codes from key
     down events to readable names
 * @type {Object}
 */
Type.Events.Input.keyDownNames = {
  8  : 'backspace',
  9  : 'tab',
  13 : 'enter',
  32 : 'space',
  37 : 'left',
  38 : 'up',
  39 : 'right',
  40 : 'down',
  46 : 'del'
};

/**
 * Factory to create a {Type.Events.
     Input} from an {InputEvent}
 *
 * @param {InputEvent} e
 * @returns {Type.Events.Input}
 */
Type.Events.Input.fromInput =
    function (e) {
  return Type.Events.Input.
      fromKeyDown(e);
};

/**
 * Factory to create a {Type.Events.
     Input} from a {KeyboardEvent}
 *
 * @param {KeyboardEvent} e
 * @returns {Type.Events.Input}
 */
Type.Events.Input.fromKeyDown =
    function (e) {

  var charCode = (typeof e.which ===
       "number") ? e.which : e.
      keyCode,
    options = {
      key      : Type.Events.Input.
          keyDownNames[charCode] ||
           charCode,
      keyCode : charCode,
      shift    : e.shiftKey,
      alt      : e.altKey,
      ctrl     : e.ctrlKey,
      meta     : e.metaKey
    };

  return new Type.Events.Input(
      options);

};

module.exports = Type.Events.Input;
```

Listing 20: events/input.js

```
'use strict';

var Input = require('../input');

/**
 * Creates a caret filter. Will catch arrow key inputs,
 * move the editor's caret and cancel the event.
 *
 * @param {Type} type
 * @constructor
 */
Input.Filter.Caret = function (type) {
  this._caret = type.getCaret();
};

(function () {

  this.keys = {
    left  : 'moveLeft',
    up    : 'moveUp',
    right : 'moveRight',
    down  : 'moveDown'
  };

  /**
   * Moves the caret left
   *
   * @param {Type.Events.Input} e
   */
  this.moveLeft = function (e) {
    this._caret.moveLeft();
    e.cancel();
  };

  /**
   * Moves the caret up
   *
   * @param {Type.Events.Input} e
   */
  this.moveUp = function (e) {
    this._caret.moveUp();
    e.cancel();
  };

  /**
   * Moves the caret right
   *
   * @param {Type.Events.Input} e
   */
  this.moveRight = function (e) {
    this._caret.moveRight();
    e.cancel();
  };

  /**
   * Moves the caret down
   *
   * @param {Type.Events.Input} e
   */
  this.moveDown = function (e) {
    this._caret.moveDown();
    e.cancel();
  };

}).call(Input.Filter.Caret.prototype);

module.exports = Input.Filter.Caret;
```

Listing 21: input_filters/caret.js

```
'use strict';

var Input = require('../input');

/**
 * Creates a caret filter. Will catch arrow key inputs,
 * move the editor's caret and cancel the event.
 *
 * @param {Type} type
 * @param {Type.Input} [input]
 * @constructor
 */
Input.Filter.Undo = function (type, input) {
  this._undoManager = type.getUndoManager();
  this._sourceId = input.getContent().getSourceId();
};

(function () {

  this.keys = {
    90  : 'undoRedo' // z
  };

  /**
   * Performs undo and redo commands
   *
   * @param {Type.Events.Input} e
   */
  this.undoRedo = function (e) {

    if (e.cmd && e.shift) {
      this._undoManager.redo(this._sourceId);
      e.cancel();
    } else if (e.cmd) {
      this._undoManager.undo(this._sourceId);
      e.cancel();
    }

  };

}).call(Input.Filter.Undo.prototype);

module.exports = Input.Filter.Undo;
```

Listing 22: input_filters/undo.js

```
'use strict';

var Input = require('../input');

/**
 * Creates a command filter. Will
     fetch common
 * text formatting keyboard
     shortcuts and call
 * the according formatting methods.
 *
 * todo should listen for key codes
     and not keys
 *
 * @param type
 * @param {Type.Input} input
 * @constructor
 */
Input.Filter.Command = function (
    type, input) {
  this._selection = type.
      getSelection();
  //this._formating = type.
      getFormatter();
  this._content = input.getContent()
      ;
};

(function () {

  this.keys = {
    66 : 'command', // b
    73 : 'command', // i
    83 : 'command', // s
    85 : 'command'  // u
  };

  this.tags = {
    66 : 'strong',
    73 : 'em',
    83 : 's',
    85 : 'u'
  };

  /**
   * todo format stuff when nothing
       is selected
   * @param {Type.Events.Input} e
   */
  this.command = function (e) {

    var sel;

    if (e.cmd) {
      sel = this._selection.save();
      this._content.format(this.tags
          [e.key], this._selection.
          getRange());
      this._selection.restore(sel);
      e.cancel();
    }

  };

}).call(Input.Filter.Command.
    prototype);

module.exports = Input.Filter.
    Command;
```

Listing 23: input_filters/command.js

```
'use strict';

var Input = require('../input');

/**
 * Creates a remove filter. Will
     catch backspace and del key
 * inputs and remove either the
     currently selected text or
 * the character next to the caret.
 *
 * @param {Type} type
 * @param {Type.Input} [input]
 * @constructor
 */
Input.Filter.Remove = function (type
    , input) {
  this._root = type.getRoot();
  //this._writer = type.getWriter();
  this._content = input.getContent()
      ;
  this._caret = type.getCaret();
  this._selection = type.
      getSelection();
};

(function () {

  this.keys = {
    backspace : 'remove',
    del       : 'remove'
  };

  /**
   *
   * @param {Type.Events.Input} e
   */
  this.remove = function (e) {

    var range, newOffset,
        removeChars, moveChars;

    if (this._selection.collapsed())
        {
      removeChars = e.key === '
          backspace' ? -1 : 1;
      moveChars = e.key === '
          backspace' ? -1 : 0;
      range = Type.Range.fromCaret(
          this._caret, removeChars)
          ;
      newOffset = this._caret.getOffset() + moveChars;
    } else {
      range = this._selection.getRange();
      newOffset = range.getStartOffset(this._root);
      this._selection.unselect();
      this._caret._blink();
    }

    this._caret.setOffset(newOffset);
    this._content.remove(range);

    //e.cancel();

  };


}).call(Input.Filter.Remove.prototype);

module.exports = Input.Filter.Remove;
```

Listing 24: input_filters/remove.js

```
'use strict';

var Input = require('../input');

/**
 * Creates a caret filter. Will catch arrow key inputs,
 * move the editor's caret and cancel the event.
 *
 * @param {Type} type
 * @param {Type.Input} [input]
 * @constructor
 */
Input.Filter.LineBreaks = function (type, input) {
  this._writer = type.getWriter();
  this._caret = type.getCaret();
};

(function () {

  /**
   * This filter will react to enter keys
   * @type {{enter: string}}
   */
  this.keys = {
    enter  : 'insertLineBreak'
  };

  /**
   * Inserts a br tag
   * @param e
   */
  this.insertLineBreak = function (e) {
    var br = document.createElement('br');
    this._writer.insertHTML(this._caret.textNode, this._caret.offset, br);
    this._caret.moveRight();
    e.cancel();
  };


}).call(Input.Filter.LineBreaks.prototype);

module.exports = Input.Filter.LineBreaks;
```

Listing 25: input_filters/line_breaks.js

```
'use strict';

var Type = require('./core');

/**
 *
 * @param {Type} type
 * @constructor
 */
Type.UndoManager = function (type) {
  this._type = type;
  this._stack = [];
  this._pointer = 0;
  this.lastActionReceived = null;
  this.mergeDebounce = 500;
};

(function () {

  /**
   *
   * @param {Type.Actions.Type|Type.Actions.Insert|*} action
   * @returns {Type.UndoManager}
   */
  this.push = function (action) {
    this._stack.length = this._stack.length === 0 ? 0 : this._pointer + 1;
    if (this.shouldBeMerged(action))
```

```
        {
      this._stack[this._pointer].
          merge(action)
    } else {
      this._stack.push(action);
      this._pointer = this._stack.
          length - 1;
    }
    this.lastActionReceived = Date.
        now();
    return this;
  };

  /**
   *
   * @param action
   * @returns {boolean}
   */
  this.shouldBeMerged = function (
      action) {
    if (this.lastActionReceived ===
        null) {
      return false;
    }
    if (Date.now() > this.
        lastActionReceived + this.
        mergeDebounce) {
      return false;
    }
    return !!(this._stack.length &&
        this._stack[this._pointer].
        mergeable(action));
  };

  /**
   *
   * @param {*} [sourceId]
   * @param {number} [steps]
   * @returns {Type.UndoManager}
   */
  this.undo = function (sourceId,
      steps) {

    steps = steps === undefined ? 1
        : steps;

    //for (steps; steps > 0; steps
        -= 1) {
    //  if (this._pointer < 0) {
    //    this._pointer = -1;
    //    break;
    //  }
    //  this._stack[this._pointer].
        undo(this.
        _getCharacterShift());
    //  this._pointer--;
    //}

    while (steps > 0 && this.
        _pointer > -1) {
      if (this._stack[this._pointer
          ].sourceId === sourceId
          || sourceId === undefined
          ) {
        this._stack[this._pointer].
            undo(this.
            _getCharacterShift());
        steps -= 1;
      }
      this._pointer -= 1;
    }

    return this;
  };

  /**
   *
   * @param {*} [sourceId]
   * @param {number} [steps]1
   * @returns {Type.UndoManager}
   */
  this.redo = function (sourceId,
      steps) {

    var stackLen = this._stack.
        length;
    steps = steps === undefined ? 1
        : steps;

    //for (steps; steps > 0; steps
        -= 1) {
    //  this._pointer++;
    //  if (this._pointer > this.
        _stack.length - 1) {
    //    this._pointer = this.
        _stack.length - 1;
    //    break;
    //  }
    //  this._stack[this._pointer].
        execute(this.
        _getCharacterShift());
    //}

    while (steps > 0 && this.
        _pointer < stackLen) {
      this._pointer++;
      if (this._pointer > this.
          _stack.length - 1) {
        this._pointer = this._stack.
            length - 1;
        break;
      }
      if (this._stack[this._pointer
          ].sourceId === sourceId
          || sourceId === undefined
          ) {
        this._stack[this._pointer].
            execute(this.
            _getCharacterShift());
        steps--;
      }
    }

    return this;
  };

  /**
   * Will iterate through the stack
       (beginning from its end)
   * and collect all character
       insertions and removals and
   * return them. This can be used
```

```
        bei actions to shift the
   * their character offset to which
         they apply their changes.
   *
   * @param {number} [targetPointer]
        - The stack pointer
   *      to which all character
        shifts shall be collected
   * @returns {number[][]} - A map
        of insertions and removals
   *      First dimensions is at
        which offsets characters
        have
   *      changed. Second dimension
        is the number of characters
   *      that have been added or
        removed.
   * @private
   */
  this._getCharacterShift = function
       (targetPointer) {

    targetPointer = targetPointer
        === undefined ? this.
        _pointer + 1 :
        targetPointer;

    var len = this._stack.length -
        1,
      shifts = [],
      shift,
      i, j;

    for (i = len; i >= targetPointer
        ; i -= 1) {

      shift = this._stack[i].
          getCharacterShift();

      for (j = 0; j < shift.length;
          j++) {
        shifts.push( shift[j] );
      }

      //shifts.concat(shift);
    }

    return shifts;
  }

}).call(Type.UndoManager.prototype);

module.exports = Type.UndoManager;
```

Listing 26: undo_manager.js

```
'use strict';

var Type = require('./core');

/**
 * Creates a new Content class
 *
 * This class can be used to
     manipulate the editor's
 * contents and will make sure any
     action performed
 * is undoable and re-doable.
 *
 * @param {Type} type
 * @constructor
 */
Type.Content = function (type) {
  this._sourceId = this.
       _createUniqueSourceId();
  this._undoManager = type.
       getUndoManager();
  this._writer = type.getWriter();
  this._formatter = type.
       getFormatter();
  this._root = type.getRoot();
  this._type = type;
};

(function () {

  /**
   * Inserts text to the editor's
       contents and pushes an
   * action to the undo manager{}
   *
   * @param {Text|Number} textNode -
       The text node in which the
   *      contents should be inserted
   * @param {Number|String} offset -
       The character offset in the
   *      text node at which the
       contents should be inserted
   * @param {String} [content] - The
       text that should be
   *      inserted
   * @returns {Type.Content} - This
       instance
   */
  this.insert = function (textNode,
      offset, content) {

    // If only an offset and
        contents were given
    if (arguments.length === 2) {
      var nodeInfo = Type.TextWalker
          .nodeAt(this._root,
          textNode);
      content = offset;
      offset = nodeInfo.offset;
      textNode = nodeInfo.node;
    }

    // Change contents
    this._writer.insertText(textNode
        , offset, content);

    // Undo capabilities
    var absOffset = Type.TextWalker.
        offset(this._root, textNode
        , 0, offset);
  var insertion = new Type.Actions
      .Insert(this._sourceId,
      this._type, absOffset,
      content);
  this._undoManager.push(insertion
      );

  // Chaining
  return this;

};

/**
 * Removes the text inside a given
     range from the contents
 *
 * @param {Type.Range|Number}
     range - The text range that
     should
 *     be removed from the
     contents. This parameter can
      also be
 *     the start offset
 * @param {Number} numCharacters -
     If this parameter is set
     the
 *     first parameter will be
     interpreted as a number and
     is the
 *     start offset in the text.
     This parameter will be the
     number
 *     of character to be removed
     beginning from the start
     offset.
 * @returns {Type.Content} - This
     instance
 */
this.remove = function (range,
    numCharacters) {

  // If only an offset
      numCharacters were given
  if (arguments.length === 2) {
    range = Type.Range.
        fromPositions(this._root,
          range, range +
        numCharacters);
  }

  // Undo capabilities
  var removal = Type.Actions.
      Remove.fromRange(this.
      _sourceId, this._type,
      range);
  this._undoManager.push(removal);

  // Change contents
  this._writer.remove(range);

  // Chaining
  return this;

};

/**
 * Formats a given text range
 *
 * @param {String} tag - The HTML
     tag the text should
 *     be formatted with
 * @param {Type.Range|number}
     range - The range of text
 *     that should be formatted or
      a number that will be
 *     the start offset of the
     formatting
 * @param {number} [end] - If the
     second parameter that
 *     was given is a start offset
     , this will be the end
 *     offset in the text that
     will be formatted.
 * @returns {Type.Content} - This
     instance
 */
this.format = function (tag, range
    , end) {

  // If positions instead of a
      range were given
  if (arguments.length === 3) {
    range = Type.Range.
        fromPositions(this._root,
         range, end);
  }

  // Change contents
  var nodes = this._formatter.
      format(tag, range);

  // Undo capabilities
  var formatting = new Type.
      Actions.Format.fromRange(
      this._sourceId, this._type,
       range, tag, nodes);
  this._undoManager.push(
      formatting);

  // Chaining
  return this;

};

/**
 * Formats a given text range
 *
 * @param {String} tag - The HTML
     tag the text should
 *     be formatted with
 * @param {Type.Range|number}
     range - The range of text
 *     that should be formatted or
      a number that will be
 *     the start offset of the
     formatting
 * @param {number} [end] - If the
     second parameter that
 *     was given is a start offset
     , this will be the end
 *     offset in the text that
     will be formatted.
 * @returns {Type.Content} - This
     instance
 */
this.removeFormat = function (tag,
        range, end) {

      // If positions instead of a range were given
      if (arguments.length === 3) {
        range = Type.Range.fromPositions(this._root, range, end);
      }

      // Change contents
      this._formatter.removeFormat(tag, range);

      // Chaining
      return this;

    };

    /**
     * Getter for this content's source id
     * @returns {number}
     */
    this.getSourceId = function () {
      return this._sourceId;
    };

    /**
     * Getter for this instance's root element, i.e. the
     * element that contains this editor's text.
     * @returns {Element}
     */
    this.getRoot = function () {
      return this._root;
    };

    /**
     *
     * @returns {*|number}
     * @private
     */
    this._createUniqueSourceId = function () {
      Type.Content._lastSourceId = Type.Content._lastSourceId || 0;
      Type.Content._lastSourceId += 1;
      return Type.Content._lastSourceId;
    }

}).call(Type.Content.prototype);

module.exports = Type.Content;
```

Listing 27: content.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type action
 * @param {*} sourceId - An arbitrary key identifying the author
 *      of this action
 * @param {boolean} [undone] - The state of this action
 * @constructor
 */
Type.Actions.Type = function (sourceId, undone) {
  this.sourceId = sourceId;
  this.undone = undone || false;
};

(function () {

  /**
   * Performs this action
   *
   * @returns {Type.Actions.Type} - This instance
   */
  this.execute = function () {
    this.undone = false;
    return this;
  };


  /**
   * Revokes this action
   *
   * @returns {Type.Actions.Type} - This instance
   */
  this.undo = function () {
    this.undone = true;
    return this;
  };


  /**
   * Returns if a given action can be merged with this
   * action
   *
   * @param {Type.Actions.Type|*} that
   * @returns {boolean}
   */
  this.mergeable = function (that) {
    return false;
  };

  /**
   * Merges a given action with this action
   *
   * @param {Type.Actions.Type|*} that
   * @returns {Type.Actions.Type} - This instance
   */
  this.merge = function (that) {
    return this;
  };

  /**
   * Returns the offsets and number
       of characters
   * this actions inserts or removes
   *
   * @returns {number[][]}
   */
  this.getCharacterShift = function
      () {
    return [[0,0]];
  };

  /**
   * Calculates the number of
       characters a given
   * offset must be adjusted based
       on the given
   * character shifts.
   *
   * @param {number} offset - The
       character offset
   *     for which the number of
       characters must
   *     be added or removed to
       account for the
   *     given shifts
   * @param {number[][]} shifts -
       The character
   *     shifts that must be
       accounted for
   * @returns {number} - The number
       of characters
   *     that an offset must be
       addded to or
   *     removed from to account fot
       the given
   *     shifts.
   * @private
   */
  this._getShiftTo = function (
      offset, shifts) {
    var adjustment = 0,  len =
        shifts.length, i;
    for (i = 0; i < len; i += 1)
      if (shifts[i][0] <= offset)
          adjustment += shifts[i
          ][1];
    return adjustment;
  };

}).call(Type.Actions.Type.prototype)
    ;

module.exports = Type.Actions.Type;
```

Listing 28: actions/type.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type action
 * @param {*} sourceId - An
     arbitrary key identifying the
     author
 *     of this action
 * @param {Type} type - A type
     instance on which the action
 *     should be executed
 * @param {Number} offset - The
     character offset at which the
 *     text should be inserted
 * @param {String} text - The text (
     containing HTML) that
 *     should be inserted
 * @param {boolean} [undone] - The
     state of this action
 * @constructor
 */
Type.Actions.Insert = function (
    sourceId, type, offset, text,
    undone) {
  this.sourceId = sourceId;
  this.undone = undone || false;
  this._writer = type.getWriter();
  this._caret = type.getCaret();
  this._root = type.getRoot();
  this.add(offset, text);
};

/**
 * Inherit from general Type action
 */
Type.OOP.inherits(Type.Actions.
    Insert, Type.Actions.Type);

(function () {

  /**
   * Inserts text in the editor
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Insert}
       - This instance
   */
  this.execute = function (shifts) {
    var len = this._stack.length,
      nodeInfo, i, adj;
    for (i = 0; i < len; i += 1) {
      adj = this._getShiftTo(this.
          _stack[i].start, shifts);
      nodeInfo = Type.TextWalker.
          nodeAt(this._root, this.
          _stack[i].start + adj);
      this._writer.insertText(
          nodeInfo.node, nodeInfo.
          offset, this._stack[i].
          text);
    }
    this._caret.setOffset(this.
        _stack[len-1].end + adj);
    this.undone = false;
    return this;
  };

  /**
   * Revokes this action
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Insert}
       - This instance
   */

  this.undo = function (shifts) {
    var len = this._stack.length,
      range, i, adj;
    for (i = len - 1; i >= 0; i -=
        1) {
      adj = this._getShiftTo(this.
          _stack[i].start, shifts);
      range = Type.Range.
          fromPositions(this._root,
           this._stack[i].start+adj
          , this._stack[i].end+adj)
          ;
      this._writer.remove(range);
    }
    this._caret.setOffset(this.
        _stack[0].start + adj);
    this.undone = true;
    return this;
  };

  /**
   * Returns if a given action can
       be merged with this
   * action
   * @param {*} that
   * @returns {boolean}
   */
  this.mergeable = function (that) {
    return that instanceof Type.
        Actions.Insert;
  };

  /**
   * Merges a given action with this
       action
   * @param {Type.Actions.Insert|*}
       that
   * @returns {Type.Actions.Insert}
       - This instance
   */
  this.merge = function (that) {

    var stack = that.getStack(),
      length = stack.length,
      i;

    for (i = 0; i < length; i += 1)
        {
      this.add(stack[i].start, stack
          [i].text);
    }

    return this;

  };

  /**
   * Returns the offsets and number
       of characters
   * this actions inserts
   * @returns {number[][]}
   */
  this.getCharacterShift = function
      () {

    var shifts, shift, len, stck, i;

    if (this.undone) {
      return [[0,0]];
    }

    shifts = [];
    len = this._stack.length;

    for (i = 0; i < len; i += 1) {
      stck = this._stack[i];
      shift = [stck.start, stck.end
          - stck.start];
      shifts.push(shift)
    }

    return shifts;

  };

  /**
   *
   * @param {Number} start
   * @param {String} text
   * @returns {Type.Actions.Insert}
       - This instance
   */
  this.add = function (start, text)
      {

    // Required vars
    var length = text.length,
      end = start + length,
      stackText,
      insertPosition,
      i;

    // Create stack if not exists
    this._stack = this._stack || [];

    // Add to stack if stack is
        empty
    if (this._stack.length === 0) {
      this._stack.push({start:start,
          end:end, text:text});
      return this;
    }

    // Iterate over stack and insert
        maintaining order
    for (i = 0; i < this._stack.
        length; i++) {

      // Insert at beginning
      if (this._stack[i].start > end
          ) {
```

```
        this._stack.splice(i, 0, {start:start, end:end, text:text});
        break;
      }

      // Add to insertion if it overlaps with another instertion
      if (start >= this._stack[i].start && start <= this._stack[i].end) {
        stackText = this._stack[i].text;
        insertPosition = start - this._stack[i].start;
        this._stack[i].text = stackText.substr(0, insertPosition) + text + stackText.substr(insertPosition);
        this._stack[i].end += length;
        break;
      }

      // Add to end
      if (i+1 >= this._stack.length) {
        this._stack.push({start:start, end:end, text:text});
        break;
      }

      // Insert between other insertions
      if (this._stack[i].end < start && (this._stack[i+1].start < end)) {
        this._stack.splice(i, 0, {start:start, end:end, text:text});
        break;
      }

    }

    // Chaining
    return this;

  };

  /**
   * Getter for this instance's stack
   * @returns {Array}
   */
  this.getStack = function () {
    return this._stack || [];
  }

}).call(Type.Actions.Insert.prototype);

module.exports = Type.Actions.Insert;
```

Listing 29: actions/insert.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type action
 * @param {*} sourceId - An arbitrary key identifying the author
 *      of this action
 * @param {Type} type - A type instance on which the action
 *      should be executed
 * @param {Number} start - The character offset from which the
 *      text should be removed
 * @param {Number} end - The character offset to which the
 *      text should be removed
 * @param {boolean} [undone] - The state of this action
 * @constructor
 */
Type.Actions.Remove = function (sourceId, type, start, end, undone) {
  this.sourceId = sourceId;
  this.undone = undone || false;
  this._writer = type.getWriter();
  this._caret = type.getCaret();
  this._root = type.getRoot();
  this.start = start;
  this.end = end;
  this._contents = this._getContents();
};


/**
 * Inherit from general Type action
 */
Type.OOP.inherits(Type.Actions.Remove, Type.Actions.Type);

(function () {

  /**
   * Removes text from the editor
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Remove} - This instance
   */
  this.execute = function (shifts) {
    var adj = this._getShiftTo(this.start, shifts),
      range = Type.Range.
          fromPositions(this._root,
           this.start + adj, this.
          end + adj);
    this._writer.remove(range);
    this._caret.setOffset(this.start
        + adj);
    this.undone = false;
    return this;
  };

  /**
   * Inserts the removed text again
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Remove}
       - This instance
   */
  this.undo = function (shifts) {
    var adj = this._getShiftTo(this.
        start, shifts),
      nodeInfo = Type.TextWalker.
          nodeAt(this._root, this.
          start + adj);
    this._writer.insertHTML(nodeInfo
        .node, nodeInfo.offset,
        this._contents);
    this._caret.setOffset(this.end +
        adj);
    this.undone = true;
    return this;
  };

  /**
   * Returns if a given action can
       be merged with this
   * action
   * @param {*} that
   * @returns {boolean}
   */
  this.mergeable = function (that) {
    return false; // Deactivated
    return (that instanceof Type.
        Actions.Remove) &&
      ((that.end == this.start) || (
          that.start == this.end));
  };

  /**
   * Merges another remove action
       with this remove action
   * @param {Type.Actions.Remove|*}
       that
   * @returns {Type.Actions.Remove}
       - This instance
   */
  this.merge = function (that) {
    this.start = Math.min(this.start
        , that.start);
    this.end = Math.max(this.end,
        that.end);
    this._contents = this.
        _getContents();
    return this;
  };

  /**
   * Returns the offsets and number
       of characters
   * this actions inserts or removes
   * @returns {number[][]}
   */
  this.getCharacterShift = function
      () {
    var len = this.start - this.end;
    return this.undone ? [[0,0]] :
        [[this.start, len]];
  };

  /**
   * Returns the contents between
       the text offsets of
   * this action.
   * @private
   */
  this._getContents = function () {
    var range = Type.Range.
        fromPositions(this._root,
        this.start, this.end);
    return range.getNativeRange().
        cloneContents().childNodes;
  };

}).call(Type.Actions.Remove.
    prototype);


/**
 * Creates a new Type action
 * @param {*} sourceId - An
     arbitrary key identifying the
     author
 *     of this action
 * @param {Type} type - A type
     instance on which the action
 *     should be executed
 * @param {Type.Range} range - The
     text range that should
 *     be removed from the contents.
 * @constructor
 */
Type.Actions.Remove.fromRange =
    function (sourceId, type, range
    ) {
  var bookmark = range.save(type.
      getRoot());
  return new Type.Actions.Remove(
      sourceId, type, bookmark.
      start, bookmark.end);
};

module.exports = Type.Actions.Remove
    ;
```

Listing 30: actions/remove.js

```
'use strict';

var Type = require('../core');

/**
 * Creates a new Type action
 * @param {*} sourceId - An arbitrary key identifying the author
 *     of this action
 * @param {Type} type - A type instance on which the action
 *     should be executed
 * @param {Number} start - The character offset from which the
 *     text should be formatted
 * @param {Number} end - The character offset to which the
 *     text should be formatted
 * @param {Element[]} nodes - The initial elements that have been
 *     affected by performing this action
 * @param {Number} tag - The tag the text should be formatted
 *     with
 * @param {boolean} [undone] - The state of this action
 * @constructor
 */
Type.Actions.Format = function (sourceId, type, start, end, tag, nodes, undone) {
  this.sourceId = sourceId;
  this.undone = undone || false;
  this._formatter = type.getFormatter();
  this._caret = type.getCaret();
  this._root = type.getRoot();
  this._start = start;
  this._end = end;
  this._tag = tag;
  this._nodes = nodes;
};

/**
 * Inherit from general Type action
 */
Type.OOP.inherits(Type.Actions.Format, Type.Actions.Type);

(function () {

  /**
   * Removes text from the editor
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Format} - This instance
   */
  this.execute = function (shifts) {
    var adjStart = this._getShiftTo(this._start, shifts),
      adjEnd = this._getShiftTo(this._end, shifts),
      range = Type.Range.fromPositions(this._root, this._start+adjStart, this._end+adjEnd);
    this._nodes = this._formatter.format(this._tag, range);
    this.undone = false;
    return this;
  };

  /**
   * Inserts the removed text again
   * @param {Number[][]} shifts
   * @returns {Type.Actions.Format} - This instance
   */
  this.undo = function (shifts) {
    var len = this._nodes.length,
      i;
    for (i = 0; i < len; i += 1) {
      Type.DomUtilities.unwrap(this._nodes[i]);
    }
    this.undone = true;
    return this;

  };

}).call(Type.Actions.Format.prototype);


/**
 * Creates a new Type action
 * @param {*} sourceId - An arbitrary key identifying the author
 *     of this action
 * @param {Type} type - A type instance on which the action
 *     should be executed
 * @param {Type.Range} range - The text range that should be
 *     formatted.
 * @param {Number} tag - The tag the text should be formatted
 *     with
 * @constructor
 */
Type.Actions.Format.fromRange = function (sourceId, type, range, tag, nodes) {
  var bookmark = range.save(type.getRoot());
  return new Type.Actions.Format(sourceId, type, bookmark.start, bookmark.end, tag, nodes);
};

module.exports = Type.Actions.Format;
```

Listing 31: actions/format.js

```
'use strict';

var Type = require('./core');

(function () {

  /**
   * Returns the offset of the caret
   * type.caret()
   *
   * Show the caret
   * type.caret('show')
   *
   * Hides the caret
   * type.caret('hide')
   *
   * Moves the caret to the 10th character
   * type.caret(10)
   *
   * Convenience function for type.select(10, 20)
   * type.caret(10, 20)
   *
   * @param {...*} params - Various parameters are possible.
   *     See examples in the block comment.
   * @returns {Type} - The editor instance
   */
  this.caret = function (params) {

    // type.caret() todo was ist bei selektion?
    if (!arguments.length) {
      return this._caret.getOffset();
    }

    // type.caret('show')
    if (arguments[0] === 'show') {
      this._caret.show();
      return this;
    }

    // type.caret('hide')
    if (arguments[0] === 'hide') {
      this._caret.hide();
      return this;
    }

    // type.caret(10)
    if (arguments.length === 1 && typeof arguments[0] === "number") {
      this._caret.setOffset(arguments[0]);
      return this;
    }

    // type.caret(10, 20)
    if (arguments.length === 2) {
      return this.selection(arguments[0], arguments[1]);
    }

    return this;

  };

  /**
   * Same as type.selection('text')
   * type.selection()
   *
   * Returns the unformatted (plain) contents of the current selection
   * type.selection('text')
   *
   * Return the currently selected HTML
   * type.selection('html')
   *
   * Convenience function for type.caret(10)
   * type.selection(10)
   *
   * Selects characters 10 to 20
   * type.selection(10, 20)
   *
   * Select an element
   * type.selection(element)
   *
   * Creates a selection between 2 elements
   * type.selection(element1, element2)
   *
   * Creates a selection between the first and last element in the jQuery Collection
   * type.selection(jQueryCollection)
   *
   * Returns an object that can be passed to type.selection('restore') to store and recreate a selection
   * type.selection('save')
   *
   * Takes an object returned by type.selection('save') as a second argument to recreate a stored selection
   * type.selection('restore', sel)
   *
   * @param {...*} params - Various parameters are possible.
   *     See examples in the block comment.
   * @returns {Type} - The editor instance
   */
  this.selection = function (params) {

    // type.selection() || type.selection('text')
    if (!arguments.length || arguments[0] === 'text') {
      return Type.Range.fromCurrentSelection().text();
      }

      // type.selection('html')
      if (arguments[0] === 'html') {
        return Type.Range.
            fromCurrentSelection().
            html();
      }

      // type.selection(10)
      if (arguments.length === 1 &&
          typeof arguments[0] === "
          number") {
        return this.caret(arguments
            [0]);
      }

      // type.selection(10, 20)
      if (arguments.length === 2 &&
          typeof arguments[0] === "
          number") {
        new Type.Range(this.root,
            arguments[0], arguments
            [1]).select();
        return this;
      }

      // type.selection(element)
      if (DomUtil.isNode(arguments[0])
          ) {
        new Type.Range(arguments[0]).
            select();
        return this;
      }

      // type.selection(element1,
          element2)
      if (arguments.length === 2 &&
          DomUtil.isNode(arguments))
          {
        new Type.Range(arguments[0],
            arguments[1]).select();
        return this;
      }

    // type.selection(
        jQueryCollection) || type.
        selection([Array])
    if (arguments[0].jQuery) {
      new Type.Range(arguments[0],
          arguments[1]).select();
      return this;
    }

    // type.selection('save')
    if (arguments[0] === 'save') {
      return Type.Range.
          fromCurrentSelection().
          save();
    }

    // type.selection('restore', sel
        )
    if (arguments[0] === 'restore')
        {
      return Type.Range.
          fromCurrentSelection().
          restore(arguments[1]);
    }

    return this;

  };

  /**
   * Inserts plain text at the caret
       's position, regardless if
       str contains html. Will
       overwrite the current
   * type.insert(str)* selection if
       there is one.
   *
   *
   * Inserts formatted text at the
       caret's position. Will
       overwrite the current
       selection if there is one.
   * type.insert('html', str)
   *
   * Inserts str at the offset given
       as second parameter
   * type.insert(str, 10)

   * Same as type.insert(str, 10)
       but inserts formatted text
       given as html string
   * type.insert('html', str, 10)
   *
   * @param {...*} params - Various
       parameters are possible.
   *     See examples in the block
       comment.
   * @returns {Type} - The editor
       instance
   */
  this.insert = function (params) {

    // type.insert(str)
    if (arguments.length === 1) {
      this.getInput().getContent().
          insert(this.getCaret().
          getOffset(), arguments
          [0]);
      return this;
    }

    // type.insert(str, 'text')
    if (arguments.length === 2 &&
        arguments[1] === 'text') {
      // this._writer.insertText(
          arguments[0]);
      this.getInput().getContent().
          insert(this.getCaret().
          getOffset(), arguments
          [0]);
      return this;
    }

    // type.insert(str, 10)
    if (arguments.length === 2 &&
        typeof arguments[1] === '
        number') {
      this._writer.insertText(
          arguments[0], arguments
```

```
            [1]);
        return this;
      }

      // type.insert('html', str, 10)
      if (arguments.length === 3 &&
          arguments[0] === 'html') {
        this._writer.insertHTML(
            arguments[1], arguments
            [2]);
        return this;
      }

      return this;

    };

    /**
     * Formats the currently selected
         text with the given tag.
     * type.format(tagName, [...params
         ])* E.g. use type.cmd('
         strong') to format the
         currently selected text bold
     *
     * Applies type.format to a
         specific text range
     * type.format(startOffset,
         endOffset, tagName, [...
         params])
     *
     * @param {...*} params - Various
         parameters are possible.
     *     See examples in the block
         comment.
     * @returns {Type} - The editor
         instance
     */
    this.format = function (params) {

      var sel, range;

      if (arguments.length === 1) {
        sel = this._selection.save();
        this.getInput().getContent().
            format(arguments[0], this
            ._selection.getRange());
        this._selection.restore(sel);
        return this;
      }

      if (arguments.length === 3) {
        range = Type.Range.
            fromPositions(this.
            getRoot(), arguments[1],
            arguments[2]);
        this.getInput().getContent().
            format(arguments[0],
            range);
        return this;
      }

      return this;
    };

    /**
     * Deletes the currently selected
         text. Does nothing if there
         is no selection.
     * type.remove()
     *
     * Removes a number of characters
         from the caret's position. A
          negative number will remove
          characters left
     * of the caret, a positive number
          from the right. If there is
          a selection, the characters
          will be removed
     * from the end of the selection.
     * type.remove(numChars)
     *
     * Will remove characters between
         the given offsets
     * type.remove(startOffset,
         endOffset)
     *
     * @param {...*} params - Various
         parameters are possible.
     *     See examples in the block
         comment.
     * @returns {Type} - The editor
         instance
     */
    this.remove = function (params) {

      if (arguments.length < 2) {
        this.getInput().getContent().
            remove(this.getCaret().
            getOffset(), arguments[0]
             || -1)
      }

      if (arguments.length === 2) {
        this.getInput().getContent().
            remove(arguments[0],
            arguments[1]);
      }

      return this;
    };

    /**
     * Revokes the last user input
     *
     * @param {Number} [steps] - The
         number of actions to revoke
     * @returns {Type} - The editor
         instance
     */
    this.undo = function (steps) {
      var sourceId = this.getInput().
          getContent().getSourceId();
      this.getUndoManager().undo(
          sourceId, steps);
      return this;
    };

    /**
     * Reapplies a revoked input
     * @param {Number} [steps] - The
         number of actions to reapply
     * @returns {Type} - The editor
         instance
     */
```

```
  this.redo = function (steps) {
    var sourceId = this.getInput().getContent().getSourceId();
    this.getUndoManager().redo(sourceId, steps);
    return this;
  };

}).call(Type.fn);
```

Listing 32: core_api.js

```
'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad instance
 *
 * @param {Type} type - A Type instance Etherpad should
 *     use for collaboration
 * @constructor
 */
Type.Etherpad = function (type) {

  this.options(type.options('etherpad') || {});

  this._type = type;
  this._caret = type.getCaret();

  this._client = new Type.Etherpad.Client(this);
  this._client.onInit(this._initEditor.bind(this));
  this._client.connect();

  this._content = new Type.Etherpad.Content(this);

};

(function () {

  /**
   * Object that holds the default settings for communicating with an
   * Etherpad server.
   *
   * @type {{host: string, port: number, rootPath: string, apikey: null}}
   */
  this._defaultOptions = {
    host     : 'localhost',
    port     : 9001,
    rootPath : '/api/1.2.1/'
  };

  /**
   * Sets the options to be used for communicating with an
   * Etherpad server. Takes either a plain object or a key
   * value combination to set a single, specific option.
   * In the latter case, the key must be a {string}.
   *
   * @param {(string|Object)} options - Either a plain object
   *     with keys and values to be set or a string that will
   *     be used as a name for a option. If you pass a string,
   *     pass a second parameter to set that option or no
   *     second parameter to retrieve that option.
   * @param {*} [value] - If the first parameter is a string,
   *     this value will be set to the key of the given first
   *     parameter. Any arbitrary value can be set.
   * @returns {Type|*} Returns the type instance if you set an
   *     option or the according value if you get an option
   */
  this.options = function (options, value) {

    // Load default options if there are no instance options yet
    this._options = this._options || Type.Utilities.extend({}, this._defaultOptions);

    // Pass a single option name to fetch it
    if (typeof options === "string" && arguments.length === 1) {
      return this._options[options];
    }

    // Pass an option name and a value to set it
    if (typeof options === "string" && arguments.length === 2) {
      options = {options: value};
    }

    // Pass an object of key-values to set them
    if (typeof options === "object") {
      Type.Utilities.extend(this.
          _options, options);
    }

    // Chaining / Returning data
    return arguments.length ? this :
         this._options;

  };

  /**
   * Getter for the Type instance
   * @returns {Type}
   */
  this.getType = function () {
    return this._type;
  };

  /**
   * Getter for the Etherpad client
   * @returns {Type.Etherpad.Client}
   */
  this.getClient = function () {
    return this._client;
  };

  /**
   * Getter for the Etherpad content
   * @returns {Type.Etherpad.Content
       }
   */
  this.getContent = function () {
    return this._content;
  };

  /**
   * Will load the pad contents from
        an Etherpad connection
      message
   * to the Type editor contents.
   *
   * @param {{attribs: string, text:
        string}} contents - The
      contents
   *      of the editor sent by the
      server
   * @returns {Type.Etherpad} - This
        instance
   * @private
   */
  this._initEditor = function (
      contents, apool) {
    var changeset = this.
        _initChangeset(contents);
    Type.Development.log(changeset);
    this._type.getRoot().innerHTML =
         Type.Etherpad.Util.nl2br(
        contents.text);
    this._content.applyChangeset(
        changeset, apool);
    this._startSending();
    return this;
  };

  /**
   * Listens to Type Events and
       sends changesets to the
       connected Etherpad
   * Server.
   *
   * @returns {Type.Etherpad} - This
        instance
   * @private
   */
  this._startSending = function () {
    this._type.on('input', function
        (e) {

      var char, insertion, deletion,
           offset;

      char = String.fromCharCode(e.
          keyCode);
      char = e.shift ? char : char.
          toLowerCase();
      Type.Development.log(char, e.
          keyCode);

      if (e.keyCode == 8 || e.
          keyCode == 46) {
        offset = this._caret.
            getOffset();
        deletion = new Type.Etherpad
            .Changeset.Changes.
            Removal(offset, 1);
        this._client.send(deletion);
      } else if(/[a-zA-Z]/.test(char
          )) {
        offset = this._caret.
            getOffset();
        insertion = new Type.
            Etherpad.Changeset.
            Changes.Insertion(
            offset, char);
        this._client.send(insertion)
            ;
      }

    }.bind(this));
    return this;
  };

  /**
   * Creates a regular changeset
       from the initial attributes
       of an Etherpad
   * server.
   *
   * @param {{attribs: string, text:
        string}} contents - The
      contents
   *      of the editor sent by the
      server
   * @returns {string} - A regularly
        formatted changeset
   * @private
   */
  this._initChangeset = function (
      contents) {
    var prefix = 'Z:' + contents.
        text.length.toString(36) +
        '>0',
      body = contents.attribs.
          replace(/\+/g, '='),
      appendix = '$';
      return prefix + body +
          appendix;
  };

}).call(Type.Etherpad.prototype);

/**
 * Creates a new Type instance
     connected to an Etherpad
     server
 *
 * @param {{}|Element} options - The
      options you would pass to
     instantiate a
 *     Type instance
 * @param {{}} options.etherpad -
     The options for the Type.
     Etherpad
 *     constructor
 * @param {{}|string} [etherpadOpts]
      - Either the parameters for
     the
 *     Type.Etherpad constructor or
     a pad name as a string
 * @param {string} [server] - The
     URL for the Etherpad server
 * @constructor
 */
Type.fromEtherpad = function(options
    , etherpadOpts, server) {
  options = Type.Etherpad.
      prepareOptions(options,
      etherpadOpts, server);
  var type = new Type(options);
  new Type.Etherpad(type);
  return type;
};

/**
 * Used for the Type.fromEtherpad
     constructor to process its
     parameters
 *
 * @param {{}} options - The options
      you would pass to instantiate
      a
 *     Type instance
 * @param {{}} options.etherpad -
     The options for the Type.
     Etherpad
 *     constructor
 * @param {{}|string} [etherpadOpts]
      - Either the parameters for
     the
 *     Type.Etherpad constructor or
     a pad name as a string
 * @param {string} [server] - The
     URL for the Etherpad server
 * @returns {{}}
 */
Type.Etherpad.prepareOptions =
    function (options, etherpadOpts
    , server) {

  options = options || {};
  etherpadOpts = etherpadOpts || {};

  if (Type.DomUtilities.isNode(
      options)) {
    options = { el: options };
  }

  if (arguments.length === 3) {
    etherpadOpts = { pad:
        etherpadOpts, server:server
         };
  }

  if (typeof etherpadOpts === '
      string') {
    etherpadOpts = { pad:
        etherpadOpts };
  }

  options.etherpad = etherpadOpts;
  return options;

};

module.exports = Type.Etherpad;
```

Listing 33: /plugins/etherpad/type.etherpad.js

```
'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad.Util
     instance
 * Contains utility methods for Type
     .Etherpad
 * @constructor
 */
Type.Etherpad.Util = function () {
};

(function () {

  /**
   * Replaces newlines with <br />
       tags
   *
   * @param {string} str - The
       original string containing
       newlines
   * @returns {string} - The altered
       string containing <br />
       tags
   */
  Type.Etherpad.Util.nl2br =
      function(str)
  {
    return (str + '').replace(/([^>\
        r\n]?)(\r\n|\n\r|\r|\n)/g,
        '$1<br ' + '/>$2');
  };

  /**
   * Returns a random string
        starting with 't.' that can
        be used as a token for
   * connecting to an Etherpad
        server.
   *
   * @returns {string}
   */
  Type.Etherpad.Util.getRandomToken
      = function()
{
  var chars = '0123456789
      ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
      ',
    stringLength = 20,
    randNumber,
    str = '';
  for (var i = 0; i < stringLength
      ; i++)
  {
    randNumber = Math.floor(Math.
        random() * chars.length);
    str += chars.substring(
        randNumber, randNumber +
        1);
    return 't.' + str;
  };

}).call(Type.Etherpad.Util.prototype
    );


module.exports = Type.Etherpad.Util;
```

Listing 34: /plugins/etherpad/util.js

```
'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad.
     Client instance
 *
 * @param etherpad
 * @constructor
 */
Type.Etherpad.Client = function (
     etherpad) {
  this._etherpad = etherpad;
  this._msgHandlers = {};
  this._accepted = true;
  this._lastSent = Date.now();
  this._changeset = new Type.
      Etherpad.Changeset();
  this.registerMessageHandler('
      ACCEPT_COMMIT', this.
      _acceptCommit.bind(this));
};

(function () {

  /**
   * The default URL the client
       connects to if no URL has
       been set
   * @type {string}
   * @private
   */
  this._defaultUrl = 'http://
      localhost:9001/';

  /**
   * The interval in which chagesets
       will be sent
   *
   * @type {number}
   * @private
   */
  this._debounceTime = 0;

  /**
   * Connects to an Etherpad server
   * @returns {Type.Etherpad.Client}
       - This instance
   */
  this.connect = function () {
    this._socket = io.connect(this.
        _url(), this.
        _socketIoOptions());
    this._socket.once('connect',
        this._sendClientReady.bind(
        this));
    this._socket.on('message', this.
        _handleMessage.bind(this));
    return this;
  };

  /**
   * Sets a function that will be
       called when this client
       connects to
   * a server. The pad contents from
       the server will be passed
       to the
   * handler.
   * @param {Function} handler - The
       function that will be
       called
   * @returns {Type.Etherpad.Client}
       - This instance
   */
  this.onInit = function (handler) {
    this._onInitHandler = handler;
    return this;
  };

  /**
   * Registers a handler that will
       be called for a given
       message
 * @param {string} msg - The message on which the handler should be called
 * @param {Function} handler - The handler that should be called
 * @returns {Type.Etherpad.Client} - This instance
 */
this.registerMessageHandler = function (msg, handler) {
  this._msgHandlers[msg] = this._msgHandlers[msg] || [];
  this._msgHandlers[msg].push(handler);
  return this;
};

/**
 * Removes a handler for a given message
 * @param {string} msg - The message on which the handler is called
 * @param {Function} handler - The handler that should be removed
 * @returns {Type.Etherpad.Client} - This instance
 */
this.unregisterMessageHandler = function (msg, handler) {
  var index;
  if (this._msgHandlers[msg]) {
    index = this._msgHandlers[msg].indexOf(handler);
    if (index > -1) {
      this._msgHandlers[msg].splice(index, 1);
    }
  }
  return this;
};

/**
 *
 * @param change
 */
this.send = function (change) {

  var root, changestr;

  this._changeset._mergeOrPush(change);

  if (this._accepted && Date.now() - this._debounceTime > this._lastSent) {
    this._accepted = false;
    root = this._etherpad.getContent().getRoot();
    changestr = this._changeset.getString(root);
    this._sendChangeset(changestr);
    this._changeset = new Type.Etherpad.Changeset();
    this._lastSent = Date.now();
  }

  return this;

};

/**
 *
 * @param msg
 * @private
 */
this._acceptCommit = function (msg) {
  this._etherpad.getContent().setRevision(msg.newRev);
  this._accepted = true;
};

/**
 *
 * @param changeset
 * @private
 */
this._sendChangeset = function (changeset) {
  this._sendMessage({
    type: "USER_CHANGES",
    baseRev: this._etherpad.getContent().getRevision(),
    changeset: changeset,
    apool: {
      nextNum: 1,
      numToAttrib: {0: ["author", this._userId]}
    }
  });
};

/**
 *
 * @param msg
 */
this._sendMessage = function(msg) {
  this._socket.json.send({
    type: 'COLLABROOM',
    component: 'pad',
    data: msg
  });
};

/**
 * Will call the message handlers registered for Etherpad messages
 * @param {Object} response - The message from the server
 * @returns {Type.Etherpad.Client} - This instance
 * @private
 */
this._handleMessage = function(response) {
    // Required variables
    var msg = response.data.type,
      len, i;

    // Dev code
    Type.Development.debug('message
        ', response);

    // This message will be received
         when connecting to the
        server
    if(response.type === '
        CLIENT_VARS') {
      this._init(response.data);
      return this;
    }

    // Notify developers on
        unhandled messages from the
         server
    if (!this._msgHandlers[msg]) {
      Type.Development.debug('
          Unhandled etherpad
          message', response);
      return this;
    }

    // For all other messsages call
        the according message
        handlers
    len = this._msgHandlers[msg].
        length;
    for (i = 0; i < len; i += 1) {
      this._msgHandlers[msg][i](
          response.data);
    }

    // Chaining
    return this;

  };

  /**
   * Will be called when this client
        successfully connected to
        an
   * Etherpad server.
   * @param {Object} data - The data
        that ther server sent
   * @returns {Type.Etherpad.Client}
        - This instance
   * @private
   */
  this._init = function(data) {
    this._etherpad.getContent().
        setRevision(data.
        collab_client_vars.rev);
    this._userId = data.userId;
    if (this._onInitHandler) {
      this._onInitHandler(data.
          collab_client_vars.
          initialAttributedText,
          data.collab_client_vars.
          apool);
    }
    return this;
  };

  /**
   * Sends a 'CLIENT_READY' message
        to an Etherpad server
   * @returns {Type.Etherpad.Client}
        - This instance
   */
  this._sendClientReady = function()
      {
    var msg = {
      "isReconnect" : false,
      "component"   : 'pad',
      "type"        : 'CLIENT_READY
          ',
      "token"       : Type.Etherpad.
          Util.getRandomToken(),
      "padId"       : this._etherpad
          .options('pad'),
      "sessionID"   : "null",
      "password"    : null,
      "protocolVersion": 2
    };
    this._socket.json.send(msg);
    return this;
  };

  /**
   * Returns the options to connect
        to an Etherpad server with
        socket.io
   * @returns {{path: string,
        resource: string, max
        reconnection attempts:
        number, sync disconnect on
        unload: boolean}}
   * @private
   */
  this._socketIoOptions = function
      () {
    return {
      'path' : '/socket.io',
      'resource' : 'socket.io',
      'max reconnection attempts' :
          3,
      'sync disconnect on unload' :
          false
    };
  };

  /**
   * Returns the URL this client
        connects to
   * @returns {string}
   * @private
   */
  this._url = function () {
    return this._etherpad.options('
        url') || this._defaultUrl;
  }

}).call(Type.Etherpad.Client.
    prototype);


module.exports = Type.Etherpad.
```

```
    Client;
```

Listing 35: /plugins/etherpad/client.js

```
'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad.Content instance
 *
 * @param {Type.Etherpad} etherpad - The Etherpad instance this
 *      class will manage content collaboration for
 * @constructor
 */
Type.Etherpad.Content = function (etherpad) {
  this._client = etherpad.getClient();
  this._localCaret = etherpad.getType().getCaret();
  this._typeContent = new Type.Content(etherpad.getType());
  this._root = this._typeContent.getRoot();
  this._client.registerMessageHandler('NEW_CHANGES', this.updateContent.bind(this));
};

(function () {

  /**
   * Applies a change to the editor's contents
   *
   * @param {{newRev: number, changeset: string, apool:{}}} data - The data received from a NEW_CHANGES message
   * @returns {Type.Etherpad.Content} - This instance
   */
  this.updateContent = function(data) {
    this.revision = data.newRev;
    this.applyChangeset(data.changeset, data.apool);
    return this;
  };

  /**
   * Applies a serialized changeset to the editor's contents
   *
   * @param {string} changesetString - A serialized Changeset
   * @returns {*}
   */
  this.applyChangeset = function (changesetString, apool) {
    var changeset = new Type.Etherpad.Changeset.fromString(changesetString, apool, this._root.textContent);
    changeset.apply(this._typeContent, this._localCaret);
    return this;
  };

  /**
   * Getter for the document revision
   * @returns {number}
   */
  this.getRevision = function () {
    return this.revision;
  };

  /**
   * Setter for the document revision
   * @returns {Type.Etherpad.Content} - This instance
   */
  this.setRevision = function (rev) {
    this.revision = rev;
    return this;
  };

  /**
   * Getter for the root element
   * @returns {number}
   */
  this.getRoot = function () {
    return this._root;
  };

}).call(Type.Etherpad.Content.prototype);


module.exports = Type.Etherpad.Content;
```

Listing 36: /plugins/etherpad/content.js

```
'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad.Changeset instance
 *
 * @constructor
 */
Type.Etherpad.Changeset = function () {
  this._stack = [];
};

(function () {

  /**
   * Matches a changeset to an array of results for
   * an each operation
   *
   * @type {RegExp}
   * @private
   */
  this._changesetRegex = /((?:\*[0-9a-z]+)*)(?:\|([0-9a-z]+))?([-+=])([0-9a-z]+)|\?|/g;

  /**
   * Returns a serialized changeset string based on the length of
   * a given string or the text contents of a given element
   *
   * @param {string|Element} base - Either a string or an element
   * @returns {string}
   */
  this.getString = function (base) {
    var serializer = new Type.Etherpad.ChangesetSerializer(this);
    return serializer.getString(base);
  };

  /**
   * Applies this changeset to a given content
   *
   * Todo Insertions and removals must be executed in order
   * Todo not after one another
   *
   * @param {Type.Content} content - The content this changeset
   *     should be applied to
   * @param {Type.Caret} localCaret - The local user's caret
   * @returns {Type.Etherpad.Changeset} - This instance
   */
  this.apply = function (content, localCaret) {
    var i, len = this._stack.length;
    for (i = 0; i < len; i += 1) {
      this._stack[i].apply(content, localCaret);
    }
    return this;
  };

  /**
   * Adds a serialized changeset (as a string) to this
   * changeset instance
   *
   * @param {string} str - A serialized changeset
   * @param {Object} apool - An Etherpad attribute pool
   * @param {string} base - Text contents
   * @returns {Type.Etherpad.Changeset} - This instance
   */
  this.addString = function (str, apool, base) {

    var charbank = this._getCharbank(str),
      nlIndices = this._getNlIndices(base),
      offsets = { absolute: 0, stack: []},
      rawMatch, match;

    while ((rawMatch = this._changesetRegex.exec(str)) !== null) {
      if (match = this._parseMatch(rawMatch))
        this._addMatchToStack(offsets, charbank, match, apool, nlIndices);
    }

    return this;

  };

  /**
   * Returns the indices of newlines in a string
   *
   * @param {string} str - The string to return the indexes of newlines for
   * @returns {number[]} - An array in indexes of the newlines in the text
   * @private
   */
  this._getNlIndices = function (str) {
    var regex = /[\n]/gi, result, indices = [];
    while ( (result = regex.exec(str)) ) {
      indices.push(result.index);
    }
    return indices;
  };

  /**
   * Getter for the operation stack
   * @returns {Array}
   */
  this.getStack = function () {
    return this._stack;
  };

  /**
   *
   * @param {{ absolute: number, stack: number[] }} offset - An object
   *     containing offset information
   * @param {string} charbank - The charbank of a string changeset
   * @param {{attrs: string, operator: string, value: string, nl: number}} match
   *     A match as returned by _parseMatch
   * @param {Object} apool - An Etherpad attribute pool
   * @param {number[]} nlIndices -
   * @private
   */
  this._addMatchToStack = function (offset, charbank, match, apool, nlIndices) {

    var delta;

    this._mergeOrPush(this._createFromMatch(offset, charbank, match, apool));

    if (match.operator === '=') {
      delta = parseInt(match.value, 36);
      delta += match.nl ? 1 : 0;
      offset.absolute += delta;
      offset.stack.push(offset);
    }

    return this;

  };

  /**
   *
   * @param offset
   * @param charbank
   * @param match
   * @param apool
   * @returns {*}
   * @private
   */
  this._createFromMatch = function (offset, charbank, match, apool) {

    var attrs = this._getAttributesFromMatch(match, apool);

    switch(match.operator) {
      case '*':
        return Type.Etherpad.Changeset.Changes.Command.fromAPool(apool);
      case '=':
        return this._operationOrMovement(offset, charbank, match, attrs);
      case '+':
        return new Type.Etherpad.Changeset.Changes.Insertion(offset.absolute, charbank);
      case '-':
        return Type.Etherpad.Changeset.Changes.Removal.fromMatch(offset, match);
      default:
        Type.Development.debug('Cannot match operator ' + match.operator, match);
        return null;
    }
  };

  /**
   *
   * @param offset
   * @param charbank
   * @param match
   * @param attrs
   * @returns {Type.Etherpad.Changeset.Changes.Formatting}
   * @private
   */
  this._operationOrMovement = function (offset, charbank, match, attrs) {
    if (!attrs.length) {
      return Type.Etherpad.Changeset.Changes.Movement.fromOffsetObject(offset, match);
    } else {
      return Type.Etherpad.Changeset.Changes.Formatting.fromAttrs(attrs, offset.absolute, match);
    }
  };

  /**
   *
   * @param {Type.Etherpad.Changeset.Changes.Change} change - A change
   *     instance or an inheriting class
   * @returns {Type.Etherpad.
```

```
        Changeset} - This instance
  * @private
  */
this._mergeOrPush = function (
    change) {

  var last = this._stack[this.
      _stack.length - 1];

  if (!!last && last.mergable(
      change)) {
    last.merge(change);
  } else if (!(change instanceof
      Type.Etherpad.Changeset.
      Changes.Movement)) {
    this._stack.push(change);
  }

  return this;
};


/**
  * Parses a regex match and
      returns a readable object
  *
  * @param {Array|{index: number,
      input: string}} match - A
  *      RegEx match
  * @returns {{attrs: string,
      operator: string, value:
      string}}
  * @private
  */
this._parseMatch = function (match
    ) {

  if (match.index === this.
      _changesetRegex.lastIndex)
    this._changesetRegex.lastIndex
        ++;

  if(match[0] === '')
    return false;

  return {
    attrs    : match[1],
    nl       : match[2],
    operator : match[3],
    value    : match[4]
  }
};

/**
 * Returns the attributes from a
     match and an apool
 *
 * @param {{attrs: string,
     operator: string, value:
     string}} match - A
 *     match parsed by this.
     _parseMatch
 * @param {{numToAttrib: array}}
     apool - An attribute pool
     from an
 *     Etherpad server
 * @returns {*[]}
 * @private
 */
this._getAttributesFromMatch =
    function (match, apool) {
  var i;
  if (!match.attrs.length) return
      [];
  i = parseInt(match.attrs.substr
      (1));
  return [apool.numToAttrib[i]]
};

/**
 * Reads the charbank from a
     changeset string
 *
 * @param {string} str - A
     serialized changeset
 * @returns {string|null} - The
     charbank or null
 *     if there is no charbank
 * @private
  */
  this._getCharbank = function (str)
      {
    var opsEnd = str.indexOf('$')+1;
    return opsEnd >= 0 ? str.
        substring(opsEnd) : null;
  };

}).call(Type.Etherpad.Changeset.
    prototype);


/**
 * Creates a new {Type.Etherpad.
     Changeset} from a serialized
 * changeset string
 *
 * @param {string} str - A
     serialized changeset string
 * @returns {Type.Etherpad.Changeset
     }
 */
Type.Etherpad.Changeset.fromString =
    function (str, apool, base) {
  var changeset = new Type.Etherpad.
      Changeset();
  changeset.addString(str, apool,
      base);
  return changeset;
};


/**
 * Namespace for changes
 * @type {{}}
 */
Type.Etherpad.Changeset.Changes =
    {};


module.exports = Type.Etherpad.
```

```
      Changeset;
```

Listing 37: /plugins/etherpad/changeset.js

```
'use strict';

var Type = require('../../../core');

/**
 * Creates a new Type.Etherpad.
     Changeset instance
 *
 * @constructor
 */
Type.Etherpad.Changeset.Changes.
    Change = function () {

};

(function () {

  this.apply = function (content,
      localCaret) {
    return this;
  };

  this.mergable = function (that) {
    return false;
  };

  this.merge = function (that) {
    return this;
  };

  this.getCharbank = function () {
    return '';
  };

  this.getOperation = function () {
    return '';
  };

  this.getLength = function () {
    return 0;
  };

}).call(Type.Etherpad.Changeset.
    Changes.Change.prototype);

Type.Etherpad.Changeset.Changes.
    Change.fromMatch = function (
    match) {
  return new Type.Etherpad.Changeset
      .Changes.Change();
};

module.exports = Type.Etherpad.
    Changeset.Changes.Change;
```

Listing 38: /plugins/etherpad/changes/change.js

```
'use strict';

var Type = require('../../../core');

/**
 * Creates a new Type.Etherpad.
     Changeset instance
 *
 * @param {number} delta - The
     relative movement
 * @param {number} [absolute] - The
     absolute text position
 * @constructor
 */
Type.Etherpad.Changeset.Changes.
    Movement = function (delta,
    absolute) {
  this.delta = delta;
  this.absolute = absolute || null;
};

/**
 * Inherit from Etherpad change
 */
Type.OOP.inherits(Type.Etherpad.
    Changeset.Changes.Movement,
    Type.Etherpad.Changeset.Changes
    .Change);


(function () {


  /**
   * Etherpad's serialized string
       for this operation
   * @type {string}
   */
  this.op = '=';

}).call(Type.Etherpad.Changeset.
    Changes.Movement.prototype);

/**
 *
 * @param match
 * @returns {Type.Etherpad.Changeset
```

```
        .Changes.Movement}
 * @constructor
 */
Type.Etherpad.Changeset.Changes.
     Movement.fromMatch = function (
     match) {
  return new Type.Etherpad.Changeset
       .Changes.Movement(parseInt(
       match.value, 36));
};

/**
 * Creates a new Movement instance
       from an object containing the
       delta
 * and the absolute text offset.
 *
 * @param {{ absolute: number, stack
       : number[] }} offset - An
       object
 *      containing offset information
 */
Type.Etherpad.Changeset.Changes.
     Movement.fromOffsetObject =
     function (offset, match) {
  var delta = offset.stack[offset.
       stack.length -1];
  return new Type.Etherpad.Changeset
       .Changes.Movement(delta,
       offset.absolute + parseInt(
       match.value));
};


module.exports = Type.Etherpad.
     Changeset.Changes.Movement;
```

Listing 39: /plugins/etherpad/changes/movement.js

```
'use strict';

var Type = require('../../../core');

/**
 * Creates a new Type.Etherpad.
       Changeset instance
 *
 * @constructor
 */
Type.Etherpad.Changeset.Changes.
     Insertion = function (offset,
     text) {
  this.start  = offset;
  this.length = text.length;
  this.end    = offset + text.length
       ;
  this.text   = text;
};

/**
 * Inherit from Etherpad change
 */
Type.OOP.inherits(Type.Etherpad.
     Changeset.Changes.Insertion,
     Type.Etherpad.Changeset.Changes
     .Change);


(function () {

  /**
   * Etherpad's serialized string
       for this operation
   * @type {string}
   */
  this.op = '+';

  /**
   *
   * @returns {string}
   */
  this.getOperation = function () {
    return this.op + this.text.
         length;
  };

  /**
   *
   * @returns {number}
   */
  this.getLength = function () {
    return this.length;
  };

  /**
   * @param {Type.Content} content -
         The content this changeset
   *      should be applied to
   * @param {Type.Caret} localCaret
         - The local user's caret
   * @returns {Type.Etherpad.
         Changeset.Changes.Insertion}
          - This instance
   */
  this.apply = function (content,
       localCaret) {
    content.insert(this.start, this.
         text);
    localCaret.moveBy(this.length);
    return this;
  };

  /**
   *
   * @param {Type.Etherpad.Changeset
         .Changes.Insertion} that -
         Another Insertion
   *      instance
   * @returns {boolean}
   */
  this.mergable = function (that) {
    return that instanceof Type.
         Etherpad.Changeset.Changes.
         Insertion &&
        this.start <= that.start &&
            that.start <= this.end;
  };

  /**
   *
   * @param {Type.Etherpad.Changeset
       .Changes.Insertion} that -
       Another Insertion
   *     instance
   * @returns {Type.Etherpad.
       Changeset.Changes.Insertion}
       - This instance
   */
  this.merge = function (that) {
    var offset = that.start - this.
        start;
    this.text = this.text.substr(0,
        offset) + that.text + this.
        text.substr(offset);
    return this;
  }

}).call(Type.Etherpad.Changeset.
    Changes.Insertion.prototype);

module.exports = Type.Etherpad.
    Changeset.Changes.Insertion;
```

Listing 40: /plugins/etherpad/changes/insertion.js

```
'use strict';

var Type = require('../../../core');

/**
 * Creates a new Type.Etherpad.
     Changeset instance
 *
 * @constructor
 */
Type.Etherpad.Changeset.Changes.
    Removal = function (offset,
    length) {
  this.start  = offset;
  this.length = length;
  this.end    = offset + length;
};

/**
 * Inherit from Etherpad change
 */
Type.OOP.inherits(Type.Etherpad.
    Changeset.Changes.Removal, Type
    .Etherpad.Changeset.Changes.
    Change);


(function () {

  /**
   * Etherpad's serialized string
       for this operation
   * @type {string}
   */
  this.op = '-';

  /**
   *
   * @returns {string}
   */
  this.getOperation = function () {
    return this.op + this.length;
  };

  /**
   *
   * @returns {number}
   */
  this.getLength = function () {
    return this.length * -1;
  };

  /**
   * @param {Type.Content} content -
       The content this changeset
   *     should be applied to
   * @param {Type.Caret} localCaret
       - The local user's caret
   * @returns {Type.Etherpad.
       Changeset.Changes.Insertion}
       - This instance
   */
  this.apply = function (content,
      localCaret) {
    content.remove(this.start, this.
        length);
    if (this.end <= localCaret.
        getOffset())
      localCaret.moveBy(this.length
          * -1);
    return this;
  };



}).call(Type.Etherpad.Changeset.
    Changes.Removal.prototype);

/**
 *
 * @param {{ absolute: number, stack
     : number[] }} offset - An
     object
 *     containing offset information
 * @param {{attrs: string, operator:
     string, value: string}} match
 * @returns {Type.Etherpad.Changeset
     .Changes.Removal}
 */
Type.Etherpad.Changeset.Changes.
    Removal.fromMatch = function (
    offset, match) {
  return new Type.Etherpad.Changeset
```

```
      .Changes.Removal(offset.absolute, parseInt(match.value, 36));
};


module.exports = Type.Etherpad.Changeset.Changes.Removal;
```

Listing 41: /plugins/etherpad/changes/removal.js

```
'use strict';

var Type = require('../../../core');

/**
 * Creates a new Type.Etherpad.Changeset instance
 *
 * @constructor
 */
Type.Etherpad.Changeset.Changes.Formatting = function (command, offset, length, remove) {
  this.command = command;
  this.start   = offset;
  this.length  = length;
  this.end     = offset + length;
  this.remove  = !!remove;
};

/**
 * Inherit from Etherpad change
 */
Type.OOP.inherits(Type.Etherpad.Changeset.Changes.Formatting, Type.Etherpad.Changeset.Changes.Change);


(function () {

  /**
   * Etherpad's serialized string for this operation
   *
   * @type {string}
   */
  this.op = '=';

  /**
   * Maps Etherpad commands to tags to apply in the editor
   *
   * @type {{bold: string}}
   * @private
   */
  this._tagMap = {
    bold          : 'strong',
    italic        : 'em',
    underline     : 'u',
    strikethrough : 's'
  };

  /**
   * @param {Type.Content} content - The content this changeset
   *     should be applied to
   * @param {Type.Caret} [localCaret] - The local user's caret
   * @returns {Type.Etherpad.Changeset.Changes.Insertion} - This instance
   */
  this.apply = function (content, localCaret) {
    if (this.command !== 'author') {
      content.format(this._tagMap[this.command], this.start, this.end);
    }
    return this;
  };

}).call(Type.Etherpad.Changeset.Changes.Formatting.prototype);


/**
 *
 * @param attrs
 * @param offset
 * @param match
 * @returns {Type.Etherpad.Changeset.Changes.Formatting}
 */
Type.Etherpad.Changeset.Changes.Formatting.fromAttrs = function (attrs, offset, match) {
  return new Type.Etherpad.Changeset.Changes.Formatting(attrs[0][0], offset, parseInt(match.value), !attrs[0][1]);
};



module.exports = Type.Etherpad.Changeset.Changes.Formatting;
```

Listing 42: /plugins/etherpad/changes/formatting.js

```

'use strict';

var Type = require('../../core');

/**
 * Creates a new Type.Etherpad.
     ChangesetSerializer instance
 *
 * This class can be used to
     serialize a Type.Etherpad.
     Changeset
 * to a string for an Etherpad
     server
 *
 * @param {Type.Etherpad.Changeset}
     changeset
 * @constructor
 */
Type.Etherpad.ChangesetSerializer =
    function (changeset) {
  this._operations = this.
      _getOperations(changeset);
};

(function () {

  /**
   * Returns a serialized changeset
       string based on the length
       of
   * a given string or the text
       contents of a given element
   *
   * @param {string|Element} base -
       Either a string or an
       element
   * @returns {string} - The
       changeset string
   */
  this.getString = function (base) {

    var changeset, len, i;

    len = this._operations.length;

    changeset  = this.
        _baseLengthString(base);
    changeset += this.
        _lengthChangeString();
    changeset += this.
        _operationString(this.
        _operations[0], null, base)
        ;

    for (i = 1; i < len; i += 1) {
      changeset += this.
          _operationString(this.
          _operations[i], this.
          _operations[i-1]);
    }

    changeset += this.
        _charbankString();

    return changeset;

  };

  /**
   * Returns the length parameter
       for the changeset string
   *
   * Returns the length of either a
       string or the text inside
   * an element as a 36 base encoded
       number, prepended by 'Z:'
   *
   * @param {string|Element} base -
       Either a string or an
       element
   * @returns {string} - The 36 base
       encoded number
   * @private
   */
  this._baseLengthString = function
      (base) {
    return 'Z:' + this._lengthFor(
        base).toString(36);
  };

  /**
   * Returns the parameter for the
       changeset string that
       determines
   * the change in the length of the
       text.
   *
   * @returns {string}
   * @private
   */
  this._lengthChangeString =
      function () {
    var count = this.
        _countLengthChange();
    return (count > 0 ? '>' : '<') +
        Math.abs(count).toString()
        ;
  };

  /**
   * Returns a serialized operation
       as a string
   *
   * @param {{op: string, start:
       number, end: number, text:
       string}|{op: string, start:
       number, numChars: number}}
       operation
   *     An insertion or removal
       object
   * @param {{op: string, start:
       number, end: number, text:
       string}|{op: string, start:
       number, numChars: number}} [
       prev]
   *     The operation before this
       operation
   * @returns {string} - The
       serialized string for the
       operation
   * @private
   */
  this._operationString = function (
      operation, prev, base) {

    var offset, hack, operatorSnd;

    offset = this._offsetString(
        operation, prev, base);
    hack = operation.op == '+' ?
        '*0' : '';

    if(/^[\n\r]+$/.test(operation.
        text || '')) {
      operatorSnd = '|1+1'; // todo
          Only works if charbank
         === a single newline
    } else {
      operatorSnd = operation.
          getOperation();
    }

    return offset + hack +
        operatorSnd;

  };

  /**
   * Returns the serialized charbank
        string from all
   * operations
   *
   * @returns {string}
   * @private
   */
  this._charbankString = function ()
       {
    var charbank, len, i;
    charbank = '$';
    len = this._operations.length;
    for (i = 0; i < len; i += 1) {
      charbank += this.
          _operationCharbankString(
          this._operations[i]);
    }
    return charbank;
  };

  /**
   * Return the text of an operation
        or an empty string
   *
   * @param {{op: string, start:
        number, end: number, text:
        string}|{op: string, start:
        number, numChars: number}}
        operation
   *      An insertion or removal
        object
   * @returns {string} - The text of
         the operation or an empty
        string
   * @private
   */
  this._operationCharbankString =
      function (operation) {
    return operation.text ?
        operation.text : '';
  };

  /**
   * Returns a serialized string
        calculating the delta
   * offset of 2 operations.
   *
   * @param {{op: string, start:
        number, end: number, text:
        string}|{op: string, start:
        number, numChars: number}}
        operation
   *      An insertion or removal
        object
   * @param {{op: string, start:
        number, end: number, text:
        string}|{op: string, start:
        number, numChars: number}} [
        prev]
   *      The operation before this
        operation
   * @returns {string} - The
        serialized offset string for
         the operation
   *      relative to is prev
        operation
   * @private
   */
  this._offsetString = function (
      operation, prev, base) {
    var offset = operation.start - (
        prev ? prev.start : 0);
    return offset > 0 ? '=' + offset
        .toString(36) : '';
  };

  /**
   *
   * @param str
   * @returns {Array}
   * @private
   */
  this._getNlIndices = function (str
      ) {
    var regex = /[\n]/gi, result,
        indices = [];
    while ( (result = regex.exec(str
        )) ) {
      indices.push(result.index);
    }
    return indices;
  };

  /**
   * Returns the length of a string
        or the text inside an
        element
   *
   * @param {string|Element} base -
        Either a string or an
        element
   * @returns {number|null} - Will
        return the text length or
        null
   *      if the argument passed is
        not a string or an element
   * @private
   */
  this._lengthFor = function (base)
      {

    var change = 0, len, i;
    len = this._operations.length;
    for (i=0; i < len; i += 1) {
      change += this._operations[i].
          getLength() || 0;
    }

    if (change < 0) change += 3;

    if (typeof base === 'string') {
      return base.length + change;
    }
    if (base.textContent) {
      return base.textContent.length
          - 1 + change;
    }
    return null;
  };

  /**
   * Returns if the sum of all
       characters added and removed
        in this
   * changeset
   *
   * @returns {number} - The sum of
       all characters added and
       removed
   *      in this changeset
   * @private
   */
  this._countLengthChange = function
      () {
    var change = 0, len, i;
    len = this._operations.length;
    for (i=0; i < len; i += 1) {
      change += this._operations[i].
          getLength() || 0;
    }
    return change;
  };

  /**
   * Returns an array of all
       operations of a changeset
   * ordered by the start offset
   *
   * @param {Type.Etherpad.Changeset
       } changeset
   * @returns {Array}
   * @private
   */
  this._getOperations = function (
      changeset) {
    var operations = changeset.
        getStack().slice(0);
    operations.sort(this.
        _compareOperations);
    return operations;
  };

  /**
   * Compares the offsets of two
       insertions. This method can
       be
   * used with Array.prototype.sort
   *
   * @param {{start: number, end:
       number, text: string}|{start
       : number, numChars: number}}
        a
   *      An insertion or removal
       object
   * @param {{start: number, end:
       number, text: string}|{start
       : number, numChars: number}}
        b
   *      An insertion or removal
       object
   * @returns {number}
   * @private
   */
  this._compareOperations = function
      (a, b) {
    if (a.start < b.start) return
        -1;
    if (a.start > b.start) return 1;
    return 0;
  };

}).call(Type.Etherpad.
    ChangesetSerializer.prototype);



module.exports = Type.Etherpad.
    ChangesetSerializer;
```

Listing 43: /plugins/etherpad/changeset_serializer.js

Declaration of Academic Integrity

I hereby confirm that the present thesis on “A WYSIWYG Framework” is solely my own work and that if any text passages or diagrams from books, papers, the Web or other sources have been copied or in any other way used, all references – including those found in electronic media – have been acknowledged and fully cited.

....

(Name, Date, Signature)