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 APIs history, it can be argued that its design has never been revisited only to ensure compatibility to current browsers. This thesis explains the APIs downsides and demonstrates that rich-text editing can be achieved without requiring the `contenteditable`-attribute with library "Type", thus solving many problems that can be found contemporary third-party rich-text editors.

Acknowledgements

Contents

II Concept   **34**

4 Approaches for enabling rich-text editing   **35**

III Implementation   **46**

5 Implementation   **47**

IV Evaluation

List of Figures

Listings

Bibliography

V Appendix

Chapter 1

Introduction

1.1 Motivation

Rich-text editors are commonly used by many on a daily basis. Often, this happens knowingly, for instance in an office suite, when users wilfully format text. But often, rich-text editors are being used without notice. For instance when writing e-mails, entering a URL inserts a link automatically in many popular e-mail-applications. Also, many applications, like note-taking apps, offer rich-text capabilites that go unnoticed. Many users do not know the difference between rich-text and plain-text writing. Rich-text editing has become a de-facto standard, that to many users is *just there*. Even many developers do not realise that formatting text is a feature that needs special implementation, much more complex than plain-text editing.

While there are APIs for creating rich-text input controls in many desktop programming environments, web-browsers do not offer native rich-text inputs. However, third-party JavaScript libraries fill the gap and enable developers to include rich-text editors in web-based projects.

The libraries available still have downsides. Most importantly, only a few of them work. As a web-developer, the best choices are either to use CKEditor or TinyMCE. Most other editors are prone to bugs and unwanted behaviour. Piotrek Koszuliński, core developer of CKEditor comments this on StackOverflow as follows:

> *"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 and guys from other... two main editors (guess why only three exist) are working on this for years and we still have full bugs lists ;).[sop,]"*

A lot of the bugs CKEditor and other editors are facing are due to the fact that they rely on so-called "HTML Editing APIs" that have been implemented in browsers for years, but only been standardized with HTML5. Still, to this present day, the implementations are prone to noumerous bugs and behave inconsitently across different browsers. And even though these APIs are the de-facto standard for implementing rich-text editing, with their introduction in Internet Explorer 5.5, it has never been stated they have been created to be used as such.

It's a fact, that especially on older browsers, rich-text editors have to cope with bugs and inconsistencies, that can only be worked around, but not fixed, as they are native to the browser. On the upside, these APIs offer a high-level API to call so-called "commands" to format the current text-selection.

However, calling commands will only manipulate the document's DOM tree, in order to format the text. This can also be achieved without using editing APIs, effectively avoiding unfixable bugs and enabling a consistend behaviour across all browsers.

Furthermore CKEditor, TinyMCE and most other libraries are shipped as user interface components. While being customizable, they tend to be invasive to web-projects.

This thesis demonstrates a way to enable rich-text editing in the browser without requiring HTML Editing APIs, provided as a GUI-less software library. This enables web-developers to implement rich-text editors specific to the requirements of their web-projects.

Rich-text editing on the web is a particularly overlooked topic. Most libraries use contenteditable without questioning its benefits. The literature on this topic is thin. It is rarely written about in books and papers and noone really examines alternative ways for implementation. ACE and CodeMirror show techniques how to do it. However looking at its history, it seems very questionable. People who implement editors using it often rant about its disad-

vantages. <- der letzte Satz sollte einer der kernpunkte der introduction sein. Überahupt, das questioning sollte im kern stehen.

1.2   Terminilogy

rich-text, WYSIWYG, word-processing, WYSIWYM

1.3   Structure

The first part of this thesis explains rich-text editing on desktop PCs. The second part explains how rich-text editors are currently being implemented in a browser-environment and the major technical differences to the desktop. Part three will cover the downsides and the problems that arise with the current techniques used. Part four will explain how rich-text editing can be implemented on the web bypassing these problems. Part five dives into the possibilities of web-based rich-text editing in particular when using the techniques explained in this thesis.

Part I

Theory

Chapter 2

Text editing in desktop environments

2.1 Basics of plain-text editing

caret selection input

2.2 Basics of rich-text editing

document tree formatting algorithms

2.3 Libraries for desktop environments

It is no longer needed to implement basic rich-text editing components from the ground up. Rich-text editing has become a standard and most modern Frameworks, system APIs or GUI libraries come with built-in capabilites. Table 2.1 lists rich-text text components for popular languages and frameworks.

Environment	Component
Java (Swing)	JTextPane / JEditorPane
MFC	CRichEditCtrl
Windows Forms / .NET	RichTextBox
Cocoa	NSTextView
Python	Tkinter Text
Qt	QTextDocument

Table 2.1: Rich-text components in desktop environments

Chapter 3

Text editing in browser environments

3.1 Overview

Developing a text editor in a browser environment differs from implementing a text editor in a desktop einvironment. Any implementation is based on the axioms of the browsers. Development is generally restricted to the components and APIs offered by the HTML5 standard and experimental features that are usually implemented in a subset of browsers[1]. However, the boundaries of these restrictions can be pushed. It is common practice to combine native elements and APIs[2] in ways they have not been designed for to enable features not natively offered. These techniques are often reffered to as "hacks" and, despite their terminolagy, are generally not regarded as a bad practice.

This chapter will describe the basics of text and rich-text editing in browsers as well as the APIs and techniques used. The origins and the history of rich-text editing pose the question if the paradigms rich-text editing is based on have been thoroughly reviewed and if alternative ways for an implementation, possibly using hacks, could be considered.

---

[1] If a specific component or API can be used is determined by the intended audience and browser market share.

[2] Native to the browser, not the operating system.

3.2 Plain-text inputs

Text input components for browsers have been introduced with the specification of HTML 2.0[3]. 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.

3.3 Rich-text editing

Major browsers, i.e. any browser with a market share above 0.5%[ag,], do not offer native input fields that allow rich-text editing. Neither the W3C's HTML5 and HTML5.1 specifications nor the WHATWG HTML specification recommend such elements. However, by being able to display HTML, browsers effectively are rich-text viewers. By the early 2000s, the first JavaScript libraries emerged, that allowed users to interactively change (parts of) the HTML of a website, to enable rich-text editing in the browser. The techniques used will be discussed in section 3.4 through section 3.9.

3.4 HTML Editing APIs

In July 2000, with the release of Internet Explorer 5.5, Microsoft introduced the IDL attributes `contentEditable` and `designMode` along with the content attribute `contenteditable`[4,5]. These attributes were not part of the W3C's HTML 4.01 specifications[6] or the ISO/IEC 15445:2000[7], the defining standards of that time. Table 3.1 lists these attributes and possible values.

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

Listing 3.1: An element set to editing mode

---

[3] `https://tools.ietf.org/html/rfc1866`, last checked on 07/15/2015
[4] `https://msdn.microsoft.com/en-us/library/ms533720(v=vs.85).aspx`, last checked on 07/10/2015
[5] `https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx`, last checked on 07/10/2015
[6] `http://www.w3.org/TR/html401/`, last checked on 07/14/2015
[7] `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=27688`, last checked on 07/14/2015

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

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` 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[8]. These elements include "divs", "paragraphs" and the docuement's "body" element amongst others. In editing mode

1. Users can interactively click on and type inside texts

2. An API is enabled that can be accessed via JScript and JavaScript

Setting the caret by clicking on elements, accepting keyboard input and modifying text nodes is handled entirely by the browser. No further scripting is necessary.

The API enabled can be called globally on the `document` object, but will only execute when the user's selection or caret is focussed inside an element in editing mode. Table **??** lists the full HTML editing API. To format text, the method `document.execCommand` can be used. Calling

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

Listing 3.2: Emphasizing text using the HTML editing API

will wrap the currently selected text inside an element in editing mode with `<i>` tags. The method accepts three parameters. The first paramter is the "Command Identifier", that determines which command to execute. For instance, this can be `italic` to italicize the current selection or `createLink` to create a link with the currently selected text as label.

---

[8]`https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx`, last checked on 07/10/2015

```
1   document.execCommand('createLink', false, 'http://google.de/'
        );
```

Listing 3.3: Creating a link using the HTML editing API

The *third* parameter will be passed on to the internal command given as first parameter. In the case of a `createLink` command, the third parameter is the URL to be used for the link to create. The *second* paramter determines if executing a command should display a user interface specific to the command. For instance, using the `createLink` command with the second parameter set to `true` and not passing a third parameter, the user will be prompted with a system dialog to enter a URL. A full list of possible command identifiers can be found on MSDN[9].

3.5   Discussion of HTML Editing APIs

HTML editing APIs have been poorly designed. Sections V to W discuss the origins and the reasons why HTML editing APIs have been standardized and are supported by all major browser. It can be seen, that they did not came to be because of their good design. In sections X and Y will discuss the design of these APIs.

3.6   Browser support

With the release of Internet Explorer 5.5 and the introduction of editing capabilities, Microsoft released a short documentation[10], 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 Web Hypertext Application

---

[9] `https://msdn.microsoft.com/en-us/library/ms533049(v=vs.85).aspx`,   last checked on 07/10/2015

[10] `https://msdn.microsoft.com/en-us/library/ms537837(VS.85).aspx`, last checked on 07/10/2015

Technology Working Group (WHATWG), also states that the API's first use case has been for rich-text editing[11].

In March 2003, the Mozilla Foundation introduced an implementation of Microsoft's designMode, named Midas, for their release of Mozilla 1.3. Mozilla already named this "rich-text editing support" on the Mozilla Developer Network (MDN)[12]. In June 2008, Mozilla added support for contentEditable IDL and contenteditable content attributes with Firefox 3.

Mozilla's editing API mostly resembles the API implemented for Internet Explorer, however, to this present day, there are still differences (compare[ad,][am,]). This includes the available command identifiers[am,][ad,] as well as the markup generated by invoking commands[ai,].

In June 2006, Opera Software releases Opera 9[ap,], providing full support for contentEditable and designMode[aq,], followed by Apple in March 2008[ar,] providing full support Safari 3.1[can,]. MDN lists full support in Google Chrome since version 4[as,], released in January 2010[at,].

Starting in November 2004, WHATWG members have started actively discussing to incorporate these editing APIs in the HTML5 standard. Through reverse engineering, the WHATWG developed a specification based on Microsoft's implementation[ah,] and finally decided to include it in HTML5. With W3C's coorporation and the split in 2011, similar editing APIs based on this work are now included in W3C's HTML5 Standard[aw,] and WHATWG's HTML Standard[av,].

3.7 Emergence of HTML editing JavaScript libraries

Around the year 2003[13] the first JavaScript libraries emerged that made use of Microsoft's and Mozilla's editing mode to offer rich-text editing in the browser. Usually these libraries were released as user interface components (text fields) with inherent rich-text functionality and were only partly customizable.

---

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

[12]`https://developer.mozilla.org/en/docs/Rich-Text_Editing_in_Mozilla`, last checked on 07/10/2015

[13]compare *Meine Tabelle aller Editoren*

In May 2003 and March 2004 versions 1.0 of "FCKEditor"[14] 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 Wordpress and CKEditor is listed as the most popular rich-text editor for Drupal[15].

Since the introduction of Microsoft's HTML editing APIs, a large number of rich-text editors have been implemented. While many have been abandoned, GitHub lists about 600 JavaScript projects related to rich-text editing[16]. However, it should be noted, that some projects only use other projects' editors and some projects are stubs. Popular choices on GitHub include "MediumEditor", "wysihtml", "Summernote" and others.

3.8   Standardization of HTML Editing APIs

It makes sense to use HTML editing APIs for rich-text editing. Microsoft's demos published with the release of these API suggest to do so. Mozilla picked up on it and called their implementation "rich-text editing API". Other browsers followed with APIs based on Microsoft's idea of editable elements. However, it would have been imaginable for browsers to offer a native user interface component, a dedicated rich-text input field.

The HTML editing APIs have only been standardized with HTML5, which itself introduces 13 new types of input fields[17], but none with rich-text capabilites. The WHATWG discussed various ways to specify rich-text editing for the upcoming HTML5 standard, including dedicated input fields. The issues that have been faced with that idea are

1. Finding a way to tell the browser which language the rich-text input should generate. E.g. should it output (the then popular) "bb" code, (X)HTML, Textile or something else?

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

---

[14]Now distributed as "CKEditor"

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

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

[17]`https://developer.mozilla.org/en/docs/Web/HTML/Element/Input`, last checked on 07/16/2015

Ian Hickson, editor of WHATWG and author of the HTML5 specification adresses these main issues in a message from November 2004[18]. He states

> *Realistically, I just can't see something of this scoped[sic] [the ability to specify a input 'language' for a text-area 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, 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?*

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 milestone of the decision to integrate Microsoft's HTML editing APIs in the HTML5 standard.[19].

3.9   Usage of HTML Editing APIs for rich-text editors

Most rich-text editors use HTLM editing APIs as their basis. CKEditor and TinyMCE dynamically create an `iframe` on instantiation and set its `body` to editing mode using `contenteditable`. Doing so, users can type inside the

---

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

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

`iframe` so it effectively acts as text input field. Both libraries wrap the `iframe` in a user interface with buttons to format the `iframe`'s contents. When a user clicks on a button in the interface `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` instead. The advantages and disadvatanges of this technique will be discussed in Sections XY.

3.10   HTML Editing APIs are questionable

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 secifically well. In fact, all major browsers mimicked the API as implemented in Internet Explorer 5.5, even though there was no specification on it. It had to be reverse-engineered. The reasons for this have not been publically discussed. A reason may have been to be able to compete with the 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". However Mozilla adopted Microsoft's API applying practically no change to it. It can be argued that this has been part of the browser wars. At this time, it was essential for any browser to be able to be compatible with as many websites as possible. Many websites were only 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 the 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 stronger Internet Explorer.

As described in section ABC, other now popular browsers, i.e. Chrome, Safari and Opera, implemented these APIs only years later, when JavaScript libraries based on them have already been popular and widely used, which can be seen as a reason for these decisions. As described in section YZ, it has clearly been stated (see section 3.8), that the reason for standardizing these APIs for rich-text editing has been to ensure browser support.

The API itself stems from the time when the usage of the web was different from today, its future was still unknown and web applications like Google Docs have not even been thought of. It should be discussed if this API really is the answer to all problem and if it still fits (or ever fit) modern requirements for content management systems or web application. The advantantages, disadvantages and practical issues will be discussed in sections x y z.

3.11 Advantages of HTML Editing APIs

HTML Editing APIs have some notable advantages which will be discussed in this section.

Browser support

A fair reason for using HTML editing APIs is its wide browser support. caniuse.com lists browser support for 92.78% of all used web browsers[can,]. I.e. 92.78% of all people using the web use browsers that have full support for HTML editing APIs.

High-level API

HTML editing APIs offer high-level commands for formatting text. It requires little setup to implement basic rich-text editing. 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.

No need for language definitions

The WHATWG discussed dedicated rich-text inputs, for instance as an extension of the `textarea` component. Offering a native input for general rich-text input brings up the question which use-cases this input conforms. For a forum software, it might be useful to generate "BB" code, while for other purposes

other languages might be needed. Offering HTML editing APIs offers a semantically distinct solution, while still enabling a way to implement rich-text editing.

Possbile third-party solutions for other languages

While HTML editing APIs can be used to generate HTML only, third-party libraries can build on top of that by implementing editors that write "BB" code (for instance) and use HTML only for displaying it as rich-text.

3.12   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 implememtations of all major browsers. Calling the `italic` command, this is the output of Internet Explorer, Firefox and Chrome:

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

Listing 3.4: Markup of italic command in Internet Explorer

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

Listing 3.5: Markup of italic command in Firefox

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

Listing 3.6: Markup of italic command in Chrome

This is a *major* problem for web development, because it makes processing input very difficult. Given the number of possible edge cases, it is very hard to normalize the input. Apart from that Internet Explorer's output is semantically incorrect for most use cases[20] while Firefox's output is breaking sementics

---

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

entirely and considered a bad style regarding the principle of the separation of concerns[21]. Different browsers will not only generate differnent 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.

Flawed API

The original and mostly unaltered API is limited and not very effective. MDN lists 44 commands available for their `execCommand` implementation[22]. While other browsers do not match these commands exactly, their command lists are mostly similar. 17 of those commands format the text (for instance to italicize or make text bold) by wrapping the current selection with tags like `<em>` or `<strong>`. The only difference between any of 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 instance to apply a custom class to it (`<span class="highlight">Lorem ipsum</span>`). All 17 commands could be summarized by a single command that allows to pass custom tags or markup and wraps the selected text with it. This would not only simplify the API, but would also give it enormously more possibilities. The same goes for inserting elements. 7 commands insert different kinds of HTML elements, this could be simplified and extended by allowing to insert any kind of (valid) markup or elements with a single command.

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

Not extendable

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

---

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

[22]`https://developer.mozilla.org/en-US/docs/Web/API/document/execCommand#Commands`, last checked on 07/17/2015

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[23]. 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 workaround 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.

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 asthetical reasons, a comment form can be designed to allow bold and italic font formatting, but not headlines or colored text.

Implementing a rich-text editor with HTML editing APIs, unwanted formatting can be prevented by simply not offering input controls for these formattings (assuming no malicious behavior by the user). However any content, containing any formatting, can be pasted into elements in editing mode from the clipboard. There is no option to define filtering of some sort.

Recent browser versions allow listening to paste events. Chrome, Safari, Firefox and Opera grant full read access to the clipboard contents from paste events, which can then be stopped and the contents processed as desired. Internet Explorer allows 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. 82.78% of internet users support listening to and reading from

---

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

clipboard events[24].

When dealing with the clipboard, especially older browsers show an unexpected behavior. Older WebKit-based browsers insert so-called "Apple style spans"[25] on copy and paste commands. "Apple style spans" are pieces of markup that have no visible effect, 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 editor.

Bugs

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

> *"First of all... Don't try to make your own WYSIWYG editor if you're thinking about commercial use. [...] I've seen recently some really cool looking new editors, but they really doesn't[sic] work. Really. And that's not because their developers suck - it's because browsers suck."*[26]

Mozilla lists 1060 active issues related to its "Editor" component[bi,]. Google lists 420 active issues related to "Cr-Blink-Editing"[bh,]. The WebKit project lists 641 active issues related to "HTML Editing"[bg,]. Microsoft and Opera Software allow public access to their bug trackers. Some rich-text editors like CKEditor have been developed for over 10 years and still need to fix bugs related to the editing API[bf,][?]. Some bugs have caused big websites to block particular browsers entirely[med,]. Medium however has contacted Microsoft and lead them to fix this bug.

Given the argument that editing APIs provide easy to use and high-level methods to format text, in practice, the number of bugs and work-arounds required, renders a "easy and quick" implementation impossible. Also, browser

---

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

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

[26]`http://stackoverflow.com/questions/10162540/contenteditable-div-vs-iframe-in-making-a-rich-text-wy 11479435#11479435`, last checked on 07/18/2015

bugs cannot be fixed by web developers. At best they can be worked around, enforcing particular software design on developers and possibly spawning more bugs.

3.13 Treating HTML editing API related issues

The issues arising with HTML editing APIs cannot be fixed. Many libraries find workarounds to treat them. CKEditor, TinyMCE, that framework. Google Docs finds another way and does not use HTML editing APIs.

huge editor libraries, developed for 10 years trying to fix stuff libraries, not editors targeting inconstiencies they all can never know what's gonna happen medium

Clipboard

CKEditor "fixes" paste problems by implementing a custom fake context menu and opening a modal with some instruction that the user should press ctrl+v. This is a UX nightmare. Other editors like retractor (oder so) sanitize any change to the editors contents for the case of input by paste events.

3.14 Rich-text editing without editing APIs

HTML editing APIs are the recommended way for implementing a web-based rich-text editor. There is no native text input that can display formatted text. 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 DOM to display their rich-text contents too. Only the editing (of the DOM), commonly phrased "DOM manipulation", is implemented with HTML editing APIs.

Manipulation via the DOM APIs

Manipulating the DOM has been possible since the first implementations of JavaScript and JScript. It has been standardised in 1998 with the W3C's "Doc-

ument Object Model (DOM) Level 1 Specification" as part of the "Document Object Model (Core) Level 1"[27].

The DOM and its API is the recommended[28] way to change a website's contents and—apart from HTML editing APIs—the only possibility *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. Any DOM manipulation that can be achieved with HTML editing APIs can also be achieved using this API.

Algorithm 1 Simplified text formatting pseudocode

1: Split text nodes at beginning and end of selection
2: Create a new tag before the beginning text node of the selection
3: Loop over all nodes inside selection
4: Move each node inside new tag

Algorithm 5 demonstrates a simplified procedure to wrap a text selection in a tag. To implement the `bold` commmand of `execCommand`, this procedure can be implemented using the `strong` tag. The text selection can be read with the browser's selection API[29,30].

Algorithm 2 Simplified element insertion pseudocode

1: If{Selection is not collapsed}{
2: Split text nodes at beginning and end of selection
3: Loop over all nodes inside selection
4: Remove each node
5: Collapse selection}
6: Insert new tag at beginning of selection

Algorithm 2 demostrates a simplified procedure to insert a new tag and possibly overwriting the current text selection and thereby mimicking `execCommand`'s command based on insertion.

Algorithm 3 demonstrates a procedure to mimick the `delete` command of `execCommand`.

---

[27]`http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`, last checked on 07/10/2015
[28]recommended by the W3C and WHATWG
[29]`https://developer.mozilla.org/en-US/docs/Web/API/Selection`, last checked on 07/18/2015
[30]Internet Explorer prior version 9 uses a non-standard API `https://msdn.microsoft.com/en-us/library/ms535869(v=VS.85).aspx`, last checked on 07/18/2015

Algorithm 3 Simplified text removal pseudocode

1: If{Selection is not collapsed}{
2: Loop over all nodes inside selection
3: Remove each node
4: Collapse selection}
5: Else{
6: Remove one character left of the beginning of the selection}

Algorithm 4 Simplified element unwrapping pseudocode

1: Loop over all nodes inside element
2: Move each node before element
3: Remove element

Algorithm 4 demonstrates a procedure to unwrap an element, mimicking the `removeFormat` and `unlink` commands of `execCommand`. 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. Aside from this, editing APIs offer commands for undo/redo and clipboard capabilites as well as a command for selecting the entire contents of the editable elements. The latter can easily be implemented with the designated selection API, whereas chapter ch:implementation demonstrates an implementation for undo, redo and clipboard capabilities. Table **??** lists all commands available for `execCommand` and possible equivilents with DOM Level 1 methods. However, these are just simplified examples. A specific implementation accounting for all commands and all edge-cases will be discussed in chapters ch:concept and ch:implementation.

Third-party alternatives

The only alternative 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.

3.15 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[31], they stated some of the reasons discussed in section XYZ. 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, abanding HTML editing APIs, enables features not possible before, for example tab stops for layouting, they state.

With their implementation, Google demonstrates it is possible to implement a fully featured rich-text editor using only JavaScript and not HTML editing APIs. However, fetching input and modifying text will not suffice to implement a text editor or even a simple text field. There is many more things, that need to be considered which will be discussed in chapter **??**.

Google's document however is proprietary software and its implementation has not been documented publically. Most rich-text editors still rely on HTML editing APIs. The editor "Firepad"[32] is an exception. It is based on "CodeMirror", a web-based code editor, and extends it with rich-text formatting. The major disadvantage of Firepad, is based on its origin as a code editor. It generates "messy" (unsemantic) markup with lots of control tags. It has a sparse API that is not designed for rich-text editing. It has no public methods to format the text. It is to be noted that Google's document editor generates lots of control tags too, but it is only used within Google's portfolio of office app and it may not be necessary that it creates *well-formatted*, semantic markup. A full list of rich-text editors using and not using HTML editing APIs can be found in tables **??** and **??**.

---

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

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

3.16 Advantages of rich-text editing without editing APIs

With a pure JavaScript implementation, many of the problems that editing APIs have, can be solved.

Generated output and flawed API

The generated markup can be chosen with the implementation of the editor. Furthermore, an editor can be implemented to allow developers using the editor to *choose* the output. Section AX describes the inconsistent output across various browsers as well as the problems of the API design of `execCommand`. Both issues can be adressed by offering a method to wrap the current selection in arbitrary markup. jQuery's `htmlString` implementation[33] demonstrates a simple and stable way to define markup in a string and pass it as an argument to JavaScript methods. A sample call could read as

```
1  // Mimicking document.execCommand('italic', false, null);
2  editor.format('<em />');
3
4  // Added functionality
5  editor.format('<span class="highlight" />');
```
<div align="center">Listing 3.7: Example calls to format text</div>

With an implementation like this, developers using the editor can choose which markup should be generated for italicising text. The markup will be consistent in the scope of their project unless chosen otherwise. Since the DOM manipulation is implemented in JavaScript and not by high-level browser methods, this will also ensure the same output on all systems and solve cross-browser issues. The second example function call in listing 5.5 demonstrates that custom formatting, fitting the needs of a specific project, can be achieved with the same API.

As described in section AB, many components native to text editing have to be implemented in JavaScript. This requires some effort but also enables

---

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

full control and direct over it. Ultimately, these components can be exposed in an API to other developers, enabling options for developing editors, not offered by HTML editing APIs. An example API will be discussed in sectionXImplementationn.

Not extendable

When implementing and editor in pure JavaScripts, the limitations aufgedrück by the editing mode, 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 docuement editor demonstrates rich functionality incliding layouting tools or floating images. Both of which are features that are hardly possible in an editing mode enabled environment[34]. SectionABC discusses some use cases exploring the possibilites of rich-text editing implemented this way.

Clipboard

In a pure JavaScript environment, clipboard functionality seems to be harder to implement than with the use of editing mode. Apart from filtering the input, pasting is natively available—via keyboard shortcuts as well as the context menu. However, as demonstrated in section IMPLEMENTATION, it is possible to enable native pasting—via keyboard and context menu—even without editing mode. Furthermore, it is possible to filter the pasted contents before inserting them in the editor.

Bugs

No software can be guaranteed to be bug free. However, refraining from using HTML editing APIs will render developing an editor independent from all of these APIs' bugs. Going a step further, the implementation can be aimed to minimize interaction with browser APIs, especially unstable ones. DOM manipulation APIs have been standardized for more than 15 years and tend to be well-proven and stable. This will minimize the number of "unfixable" bugs

---

[34]http://googledrive.blogspot.fr/2010/05/whats-different-about-new-google-docs.html, last checked on 07/19/2015

and ultimately free development from being dependent on browser development, update cycles and user adoption. Bugs can be fixed quicker and rolled out to websites. This means that, other than with HTML editing APIs, bugs that occur are part of the library can be fixed and not only worked around. Furthermore, with minimizing browser interaction, bugs probably occur indeopendently of the browser used, which makes finding and fixing bugs easier.

3.17 Disadvantages of rich-text editing without Editing APIs

Formatting

Editing APIs' formatting methods take away a crucial part of rich-text editing. Especially on the web, where a text may have many sources, formatting must account for many edge cases. Nick Santos, author of Medium's rich-text editor states in regeards of their editor implementation:

> *"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.[?]"*

An editor implemented *without* HTML editing APIs does not only need to account for content (HTML) that will be pasted into the editor[35] (in fact, content should be sanitized before it gets insterted in the editor, see sections A and b, paragraphs "clipboard"), 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 with as few tags as possible (and it conforms the standards of the W3C). The same visual representation of a text, can have many different—and valid—underlying DOM forms. Nick Santos gives the exaple of the following text[?]:

---

[35]Medium uses HTML editing APIs

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

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

```
1  <strong><em>Baggins</em></strong>
2  <em><strong>Baggins</strong></em>
3  <em><strong>Bagg</strong><strong>ins</strong></em>
4  <em><strong>Bagg</strong></em><strong><em>ins</em></strong>
```

Listing 3.8: 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 is not. It may even improve the markup it affects.

Mimicking native functionality

As described in section XY, rich-text editing consists of many components like the caret or the ability to paste text via a context menu. These basic components require complex implementations. HTML editing APIs offer these components natively without further scripting.

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[br,]. The same goes for mimicking some elements like the caret. To display a caret a DOM element like a `div` is needed and it needs to be moved by changing its style attribute. 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.

3.18 Comparison of these approaches

HTML editing APIs vs other method, pros and cons, conclusion. Also HTML editing APIs will get better, still the API sucks, fewer features, also still dependent on browser development in terms of possible bugs in the future and further features. Edge casese can be tackled and solutions rolled out immediately. Cross browser inconsistencies are less of a problem and can also be tackled as required.

Part II

Concept

Chapter 4

Approaches for enabling rich-text editing

4.1 Overview

Before discussing the software architecture in section **??**, sections 4.2 through 4.8 will discuss the apporaches, principles and goals for implementing this library without HTML editing APIs.

4.2 Approaches for enabling rich-text editing

This section will discuss the options to implement rich-text editing without relying (entirely) on HTML editing APIs and discuss the advantages and disadvantages of each method.

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 rerendered, 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-rendering engine. The `canvas` API is only capable of displaying unlayouted text with specifically set line breaks. On the other hand, making the editor accessible to other developers would be much more complex since the text only exists in an internal representaion inside the editor and would not be exposed as DOM component on the website.

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 accesed 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 flow.

Enabling editing mode without using its API

One way to enable editing but avoiding many bugs and browser inonstistencies, is to enable the editing mode on an element, but avoid 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 capabilites. All of this would be taken care of by the browser.

This sem-radical 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, the

---

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

input may break elements or simply get stuck. This was one of the reasons why Google decided to abandon editing APIs entirely[2]. It could be the source to many bugs, have the development of the library be dependent on browser development and ultimatively restrict the editors capabilites.

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. 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 most obvious part is probably the text caret. 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 behaviour even depends on the operating system used).

Selection Just like the caret, 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 rectengular text selection, independent of line breaks.

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 pos-

---

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

sible through DOM manipulation, pasting text however natively 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.

Behaviour   Rich-text editors (usually) share a certain behaviour on user input. When writing a bulleted list, pressing the enter key usually creates another bulleting 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 just after that heading. Other rules that need to be considered will be discussed in section [implementation].

4.3   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 a user can click in a text to place a caret and move it with the keyboard's arrow keys. He or she 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 behaviour for the user' input that is common for rich-text editing (as described above).

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 user must be convinced he or she is using a native input and must not notice he or she is not.

Figure 4.1: Rendering of highlighted source code in Ace and CodeMirror

Web-based code editors like "Ace" 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 inside the editor that a 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 4.1). 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 other elements like `div`s to display a text selection and a `textarea` to fetch keyboard inputs, to recreate the behavior and capabilites of a native text input. Google's document editor uses similar techniques too.

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 (often called "hacks") 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. Encorporating feedback is a core philisophy of the WHATWG, the creators of HTML5.

Hacks are a necessity for an editor like this. The following sections will discuss each part of the editor and describe ideas, techniques and hacks that can be used for an effective implementation. In prior, we will discuss goals and principles that these techniques should be oriented towards as well as restrictions of browsers these techniqes must adapt to.

4.4 Usage of HTML elements

Ace, CodeMirror and Google's document editor shre similar concepts for imitating a native text input. For the caret, each of the editors create a `div` element, style it to make it look like a caret and move it when the user clicks in the text or uses his or her arrow keys. Each editor also uses `div` elements to

display a text selection, styled to imitate the user's operating system's native selections.

The concept to use HTML elements for the visible components of text editing is inevitable. Anything that can be seen on a website must exist in the DOM. The only deviation to this approach would be to resort a `canvas` element, as discussed above. The apporach to use HTML elements will be used for the implementation of this editor.

4.5 Interaction with browser APIs

Interaction with the browser must be well chosen. In some cases browser interaction can cause bugs and slow down performance while in some cases it can increase performance. The library should conform these rules:

1. Minimize interaction with the DOM

2. Minimize interaction with unstable APIs

3. Use browser APIs if it improves performance or structure

DOM operations can trigger a browser reflow[3] which slows down the browser's performance. For this reason DOM operations should be avoided where possible.

Some APIs like the browser's `Range` or `Selection` interface are useful but known to be unstable. Libraries like Rangy try to tackle this by shimming parts of the interfaces. This requires complex methods and it is hard to account for possibly unknown bugs in the browsers' APIs. Ultimately this will also affect the library's file size. Instead of trying to fix native APIs, only as little as possible should be used. Instead of using possibly unstable APIs, pure JavaScript implementations should be used. Possibly unstable native APIs should only be used when it is inevitable.

Avoiding DOM operations and unstable APIs leads to a software architecture where the biggest part remains in its own business logic. Anything that can be implemented in pure JavaScript and does not cost performance

---

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

or memory *should* be implemented in pure JavaScript using own methods and data structues.

Facebook defined a similar goal for the user interface library React. React implements virtual DOM, an internal represenation of the actual DOM on which all operations should be performed, which in return does as little manipulation to the actual DOM as needed, to maximize performance. While the focus of this goal for this library is not solely performance, this part of it. However, there can be situations in which (even unstable) native browser methods may be faster than a JavaScript implementation. The text flow for instance should be left to the browser. There can also be cases in which a pure JavaScript implementation would be much more complex and would be at the expense of having a simple code structure. It must be decided in each particular case to use native and possible unstable APIs or a pure JavaScript implementation.

4.6 Markup

The editor should be a "good citizen" in the ecosystem of other editors (see **3.17: Formatting**) and should generate valid markup, expressing a formatting semantically correct with as few tags as possible. The algorithm for creating markup must be able to work on any markup parsable by the browser and in return must generate predictable, *simple* markup itself. It must not inject markup required for the internal workings of the editor[4].

4.7 User interface components

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 shipped as input fields with a default editor interface that is, at best, customizable.

---

[4]FirePad and Google' document editor inject markup for each text line, necessary for the internal functions of the editor

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 edtitor 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 in fact functional, it seems, choosing between editors is often just a choice of the desired ui. Customizing a ui can be just as complex as writing a ui from scratch. The latter affords to add HTML elememts and call JavaScript methods while both, customizing a user interface and writing a user interface from scratch require styling. In a worst case scenario, it can be more complex to customize an interface to specific needs then writing one from scratch and being able to define just the elements as they are required. That is why the library "Type" will be implemented and offered as a software library, rather than an editor and a user interface component.

4.8 API

Conformity with HTML Editing APIs

The library should be capable of any method implemented by HTML editing APIs. However the API can differ to improve they way it will be worked with.

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 want to do and is as easy to use and plausible as possible. jQuery is an example of encorporating an API conforming these principles.

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 API and add features.

Extension For extension, web developers should have specific 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 imporant than simplicity.

All components of Type will be implemented as classes. To provide as much capabilites as possible to other developers, all classes of Type 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 mutliple users. All available classes will be discussed in chapter **5: Implementation**.

Editor implementation For web developers implementing an editor (assuming the library offers all necessary features) 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 exact control over low-level behaviour. 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 chainging and most methods are overloaded to allow passing various kinds of parameters, to determine what the function should do. Following these and the abovementioned 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 4.1: 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). It can also be called without passing parameters, to read the selection. `editor.selection();` would return the currently selected text as a string. A full API description can be found at table TODO.

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.

Figure 4.2: Diagram of the Type library and its internally used classes (excerpt)

The library itself will be exposed as a class to the global namespace with the name "Type". It can be instantiated and then provides an own API, high-level methods, for implementing a rich-text editor.

Part III

Implementation

Chapter 5

Implementation

5.1 Technology

No IDEs, tools, not even conventions. Es gibt frameworks die eine architektur für webseiten und web-applikationen vorgeben, aber nichts dergleichen für bibliotheken.

Not for building: Big JS libraries all do it differently. Top 3 client side JavaScript repositories (stars) on github https://github.com/search?l=JavaScript&q=stars%3A%3E1&
Angular.js: Grunt d3 Makefile, also ein custom build script welche node packages aufruft jQuery custom scripts, mit grunt und regex und so

Not for Architecture Angular custom module system with own conventions d3 mit nested objects (assoc arrays) und funktionen jQuery mit .fn ACE mit Klassen, daraus habe ich gelernt

5.2 Ich habe verwendet

Gulp requireJs AMDClean Uglify

JSLint - Douglas Crockford coding dogmatas / conventions JSCS - JavaScript style guide checker

Livereload PhantomJs Mocha Chai

Durch Require und AMDClean schöne arbeitsweise (am ende über bord geworfen) und kleine Dateigröße, wenig overhead.

Automatisierte Client side Tests mit PhantomJs und Mocha/Chai

ECMAScript 6

ECMAScript 6 offers many features for JavaScript including classes and modules. It is not supported by most browsers yet however there are tools like Babel that can transpile ECMAScript 6 to ECMAScript 5. Babel is already used by popular websites like Netflix for JavaScript libraries and Frameworks like React.

Type will not use ECMAScript 6 and Babel. Even though its features can support the development with syntactic sugar, most of its features can be approached using JavaScript design patterns. The benefit of not using ECMAScript 6 and transpilers are a smaller file size and more independence from third-party tools.

5.3 Coding conventions

Habe mich größtenteils an Crockfordstyle orientiert, aber die Klassen anders geschrieben. Habe den Stil von ACE editor verwendet, denn der ist gut lesbar. Lesbarkeit war mir wichtiger als Crockford style. Für private Eigenschaften und Methoden habe ich die prefix convention verwedendet. https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contributor_s_Guide/Private_Properties Sie bewirkt keine echte accessibility restriction, aber es ist eine allgemein anerkannten convention und ist auch viel besser lesbar.

5.4 Architecture

Model–view–controller

Model–view–controller (MVC) is a common apprach for implementing user interfaces and it can be applied to user interface components too. While this apporach can provide clear responsibilites, 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). As discussed in
4.8: API, the library shall leave as much freedom as possible to the devel-
opers. In and MVC system like this, the contents of the editor could only be
changed though the API, since arbitrarily modifying the HTML of the editor's
contents would change the view, but not the model. This would create an
artificial bottleneck, which cannot be desired.

Modular and object-oriented programming

jQuery and CKEditor demonstrate a software architecture in which a core
object mainly provides a system to extend its functionality, but does not of-
fer many methods itself[1]. The actual functionality of both libraries is imple-
mented through extensions while the libraries are usually bundled with a set of
"core extensions" that provide basic features. CKEditor makes use of modular
programming by implementing a major part of its editor as plugins that com-
municated via well-defined interfaces. jQuery established a paradigm calling
any extension a "plugin" but instead of using clearly defined interfaces, devel-
opers 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

Modular programming could create a system in which other developers
could exchange any component easily to improve performance or enrich func-
tionality. The disadvantage this apporach 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 goes the opposite direction and
encourages arbitrary extensions. jQuery's approach demonstrates that this

---

[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.

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 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.

Constructor Pattern 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. Functions following the constructor pattern are often called classes or pseudo-classes. Hereinafter the term classes will be used.

Modularized structure For this library a base class will provide the namespace to be extended with other classes implementing the functionality of this library. A set of core extensions provide all components needed for a rich-text editor. The base class will also provide a constructor that will be the library's entry point for other developers to implement a rich-text editor, but, like CKEditor and jQuery, will implement as little functionality as possible itself. The base class and its extensions will be discussed in detail in the sections **5.5: Base class** and **??: ??**. 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 possiblities in interoperation.

3. Classes are a proven concept for encapsulating functionality and data, protecting access and structing 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.

5. Instance variables still allow to reuse a class in different contexts with different inherent data.

As an alternative apporach, the module pattern can be used, which would also allow namespacing, encapsulation and protected access, but would make an implementation much more complicated and be much less readable.

5.5 Base class

Overview

The `Type` class is the base class (see **5.4: Modular and object-oriented programming**) and the starting point for users and developers of the library. It provides 4 purposes:

1. It can be instantiated and offer a high-level API to manipulate text and perform other rich-text related functions

2. It provides a namespace for all of the library's internal classes

3. As an instance, it exposes references to all instances of the internally used classe

4. It exposes its `prototype` as a public shorthand attribute

Instantiation and usage

To develop an editor with the library, it must be instantiated. As discussed in **4.8: Handling use cases**, all of the libary's functionality is encapsuled in a class (`Type`) that exposes its features as high-level functions. 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. An optional second parameter can be passed to define settings for the editor's instance.

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

<div align="center">Listing 5.1: Type instantiation</div>

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 capabilites as well as trigger clipboard commands. The complete API is listed in Figure ABC. For example, to format the characters 10 to 20 as bold and move the caret behind the formatted text, the following methods need to be executed:

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

<div align="center">Listing 5.2: Example commands to format text</div>

Although it should be noted that the API allows chaining and the above code can be simplified to a single line.

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

<div align="center">Listing 5.3: Example chaining</div>

With the second optional parameter for `Type` settings can be passed to determine the eiditor's behavior. As for the implementation of this thesis two settings are avaialable. One to determine the behavior on paste events, the other to turn off default keyboard shortcuts. A full description can be found in figure ABC. Section **5.27: Extending** discusses how extensions of the library can add further options for their own use.

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 apporach and attaches

each module to the `CKEDITOR` base class. Any of the classes listed in **??: ??** are attached to the `Type` class this way. As an exmple, listing 5.4 shows the declaration of the `Type` base class as well as the declarations of the classes `Caret`, `Range` and `Environment`:

```
1  // Declaration of the Type base class
2  function Type () {};
3
4  // Classes defined within the namespace created by Type
5  Type.Caret = function () {};
6  Type.Range = function () {};
7  Type.Environment = function () {};
```

Listing 5.4: 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 prepares a sub-namespace `Type.Events` for events.

Section **5.6: Module architecture** discusses how Type's classes interact to provide its rich-text functionality. One responsibility of the `Type` base class is to instanitate all other classes needed for its API and provide getters to each instance, so that each instance can access the other instantiated classes.

Exposal of Type's `prototype`

Type's functionality is implemented through classes within its namespace. However, this does not expose its functionality to its instances' 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 inside the implementation of the `Type` class. This contradicts the modular approach of the implementation. jQuery established an effective principle for extending its API. It exposes the library's `prototype` with a shorthand attribute as `jQuery.fn`. Other modules can extend the `prototype` easily and add methods to the libary's API. Type follows the same principle

and exposes its `prototype` the same way as `Type.fn`. jQuery's `prototype`—as well as Type's `prototype`—can be extended even without exposing it as an attribute, but apart from providing a more convenient way to do so, primarily, jQuery established a simple and yet powerful paradigm that encourages other developers to extend the library's API.

5.6   Module architecture

As discussed in **5.4: Modular and object-oriented programming**, Type's functionality is implemented through classes inside Type's namespace. The classes' functionality will be exposed through the base class' `prototype`. This section discusses the architecture and interaction of the classes of this library. The implementations and characteristics of each module will be discussed hereinafter.

When the `Type` base class will be instantiated, it in turn instatiates a number of classes passing each instance a reference to itself. As described in **5.5: Namespace and references** the base class provides setters for each class instance to allow the instantiated classes to access one another. For its basic functionality, Type instantiates the classes `Contents`, `Formatting`, `Caret`, `Selection` and `Input`.

The `Input` class will listen to keyboard input and mouse events. Utilizing the `Caret` and `Selection` classes it determines which part of the text should be changed or formatted. It passes the formalized edits to the `Changes` classes which will record the changes for undo and redo operations. The `Contents` class allows adding and removing text and the `Formatting` class allows to apply formattings, both using DOM operations. The `Changes` class will use these classes to to alter the text according to the change it it should apply.

Usually, text fields contain one caret and display one text selection at a time. This is why the `Type` base class instantiate 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.

Figure 5.1: Components instantiated by the Type base class

5.7   Type Api

As discussed in **5.5: Exposal of Type's `prototype`**, Type's public API for implementing editors is not hard-coded to the `Type` class. Instead it extends `Type`'s `prototype` dynamically. This conforms the paradigm of having a base class and implementing all its featues in modules. To still ensure maintanability and a clean software structure, the methods for Type's API will not be added arbitrarily by other classes. Instead, all API methods will be implemented by a single module that in turn calls the required methods of the corresponding classes. This allows for a clean and independent software architecture, a separation of concerns and decouples the API from the structure and the implementation of the library.

The module for this, called "CoreApi", is the only exception to the rest of Type's implementation in the way that it is not encapsulated in a class. In

order to add a method to Type instances, `Type.fn` must be extended with a function. This will be done for all methods discussed in *Listing 4.1*. However this does not require a class contstruct.

Similar to the principles of MVC in which the controllers should be kept small and the implementation should reside in the model layer, the functionality of the API resides in their designated classes, while the API methods are being kept small. The classes are written in a way that allow the API methods to require not more than one function call for their implementation and at most translate their parameters for the class methods.

5.8   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:

1. Hardware keyboard input

2. Virtual keyboard input

3. Mouse (pointer) input

4. Touch input

5. Game controller input (on game consoles)

6. Remote control input (on smart TVs)

When simulating 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 selects the input in any way and does so at any position inside the input. If the user touches / clicks / etc in the middle of the text, the caret should move to that position and typing must be enabled. In environments without hardware keyboards, the libarary must ensure that a virutal keyboards show up. Once the input component is selected, text input must be fetched and written to the editor. There are various options to fetch user input, which will be discussed in the following paragraphs.

Events

Keyboad   One way to fetch user input is by listening to events. Text input can be read through `KeyboardEvent`s. Keyboard events will be triggered for virtual keyboards just as for hardware keyboards. When the user presses a key, the event can be stopped and the according characters can be inserted at the position of the caret. As a downside, listeners for keyboard events cannot be bound to a specific element that is not a native text input, that means keyboard events must be listened to on the `document` level. This not only has (minor) performance downsides but als requires more logic to decide whether a keyboard input should be processed and ultimatively stopped or ignored and allowed to bubble to other event listeners of a website. Furthermore, 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 (pointer) 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 others, 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 enviroment, users commonly switch between elements by selecting focusable elements with a directional pad. Using only events would not account for this case since there would be no focusable element representing the editor. Recent browsers on Samsung's and LG's smart TVs are based on WebKit[2] while Sony's TVs use Opera. Before 2012 Samsung's browser was based on Gecko. All of these browsers and browser engines supprt keyboard events to fetch input.

---

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

Clipboard   A problem with relying entirely on events is the lack of native clipboard capabilities. Unless a native text input (including elememts 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. *Recent versions of Chrome, Opera and Android Browser[3] allow triggering arbitrary paste events on elements in editing mode thereby read the cliboard contents. With this, shortcuts could be enabled with JavaScript and instead of the native context menu, a customly build context menu using HTML could be shown that allows the user to paste, but this only works on elements in editing mode and only in these three browsers.*

Hidden native input fields

As discussed in **4.3: Approaches for imitating native components**, the source code editors Ace and CodeMirror use a hidden (native) input field to fetch the user's keyboard input. While it appears to the user he or she is entering text in a syntax-highlighted representation of the source code, in reality the user enters his or her text in a hidden `textarea` element. The input will be processed and displayed with syntax-highlighting. This solves many problems of 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.

- Keyboard shortcuts for clipboard events work.

- It can display a native context menu that allows pasting.

Implementation   To achieve this, a `textarea` must be created when the editor gets instantiated. Since browsers scroll the `textarea` into view when it receives the focus, it must be positioned in the visual representation of the editor, so the editor will be scrolled into view. This perfectly mimicks the browser's native behavior. To maintain the illusion that the user acutally

---

[3]`http://caniuse.com/#feat=clipboard`, last checked on 07/22/2015

writes inside the visual represenation of the editor the `textarea` must be hidden.

Focus   Whenever the user clicks inside the editors visible contents, the (fake) caret will be set (see **5.11: Caret**) at the according text position. To enable text input, the hidden `textarea` will be focused. The `textarea` is 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 the caret when the `textarea` receives the focus and read its input as well as hiding the caret on blur and thereby perfectly mimick 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 foucsed, pastiing via keyboard shortcuts is natively available. To enable pasting with the context menu, the `textarea` must be moved to the pointer's position on a `mousedown` event. Following the order of `MouseEvent`s, this will be completed, before the context menu will be triggerd. This way it will be triggered on the `textarea` and contain a paste option.

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 **5.6: Module architecture** and be removed from the textarea. 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 **5.9: Input Pipeline**.

Figure 5.2: Input pipeline with sample filters

5.9 Input Pipeline

Overview

Before the text read from the hidden `textarea` will be passed on to the `Changes` class, it will be passed though a series of `Filter`s, called the "input pipeline". The input pipeline has 3 basic responsibilites.

- Stop and dispatch input that that shoud trigger functions of the library

- Implement rich-text editing behavior

- Filter and transform input

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

JavaScript does not support the concept of `Interfaces` and although the constructor pattern allows class-like structures, it does not support classical inheritance unless implemented with supporting constructs[4]. These constructs can be costly in terms of file size and performance. For this reason, input filters do not implement a formalized interface and do not inherit a base filter class. Instead, each input filter registers and specifies for which input should be called. For instance, a filter can register to be called when the user presses the `ctrl` `s` key combination and trigger a "save" function. The input is passed to the filters as an event 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.

---

[4]Interfaces can be implemented though helper classes too

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 eivironment, this is not different. To mimick 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 **5.6: Module architecture**). 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 can be turned off by setting the according options (see **5.5: 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 **5.21: Events**) so that the abovementioned 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 **4.2: Native text input imitation**, most rich-text editors support the user with a common behavior depending on the user's input. This behavior can be abstracted in a simple way using the input pipeline. The following paragraphs describe the rules and filters implemented by default. Further rules can easily implemented as input filter and being added to the input pipeline.

Headlines When a user presses $\boxed{\text{enter}}$ while the caret is located at the end of a headline, a new text paragraph will be created behind the headline and the caret will be placed inside it.

Lists Pressing $\boxed{\text{enter}}$ inside a list item creates a new list item behind the current one and the caret weill be placed inside it.

Filterring and transformation

Line Breaks To display a line break in the visible rich text (i.e. the markup) a `br` or `p` tag must be inserted[5]. When the user presses the $\boxed{\text{enter}}$ 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 **5.9: 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 $\boxed{\text{backspace}}$ and $\boxed{\text{del}}$ key inputs. Depending on whether there is a text selection or not, it either deltes 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 behavoir for text editing. The `Spaces` input filter checks if adjacent spaces are being entered and inserts non-breaking spaces.

Events

As an alternative apporach, for every input that the user makes, an event could be triggered. Input filters could be implemented as event handlers. With this, the same functionality could be achived without an input pipeline. A designated pipeline however provides a clearer mental model for processing the input, 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 any input filter will be run before triggering an

---

[5]To enforce a line break other tags. like the `div` tag can be used, but would not be semantically appropriate in this context

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.

5.10   Pasting

Kontrolle durch die Implementierung mit einem versteckten Eingabefeld. Erst
mal gibt es ein reliable Paste event, was es in keinem Editor basierend auf
contentEditable gibt. 2. Hat man volle Kontrolle über den Inhalt, was gepastet
wird und was nicht. Und das ganze mit einer einfachen API. Pasting sollte
internes event ausloesen.

5.11   Caret

The `Caret` class provides all functionality to place and move a caret in a text.

BiDi support

IME

http://marijnhaverbeke.nl/blog/browser-input-reading.html https://en.wikipedia.org/wiki/Input_met

5.12   Range

The `Range` class is a utility class and an abstraction for the native `Range`
interface. The native implementation is prone to bugs on many browsers. As
discussed in **4.5: Interaction with browser APIs**, 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 required by Type that are implemented by the native `Range`
interface, this class implements additional methods specific to the API.

5.13   Selection

Fake damit copy & past nativ funktioniert Obwohl eine selection mehrere
Ranges haben kann, wird das von keinem Browser unterstützt http://stackoverflow.com/questions/4777
select-multiple-divs-with-ranges-w-contenteditable/4780571#4780571 Habe ver-
sucht mit die Darstellung über getClientRects zu emulierten, aber das ist buggy
siehe https://github.com/edg2s/rangefix/blob/master/rangefix.js

Selection Overlay   overlay

5.14   Formatting

formatting

5.15   Change Listener

change

5.16   Contents

contents

5.17   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 suffi-
cient to implement logging methods.

5.18   Dom Utilities

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

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

5.19   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 accounted for cases when any of these is `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 or child. 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 specfic node, for instance a text node or only a text node that has contents visible to the user[6]. Browsers offer a native API for this, called `TreeWalker`, but it is said to be slow[7], is only partially supported by Internet Explorer 9[8] and has been critisized for its verbose API[9].

API

The `DomWalker` class offers high-level methods for traversal, designed to quickly apply the most common filters. I can be instantiated with multiple shorthand parameters.

Algorithm 5 Simplified text formatting pseudocode

1: new DomWalker(startNode, node);
2: new DomWalker(startNode, 'filtername');
3: new DomWalker(startNode, filterFunction(){});
4: new DomWalker(startNode,{optionsObject});

---

[6]Any text node consisting of whitespace only will not be displayed by any major browser
[7]`http://jsperf.com/qsa-vs-node-iterator`
[8]`https://developer.mozilla.org/en-US/docs/Web/API/NodeIterator#Browser_compatibility`
[9]By   John   Resig,   author   of   jQuery`http://ejohn.org/blog/unimpressed-by-nodeiterator/`

The API is not only designed for the development of this library but most importantly to be a utility too for developers extending the library or implementing editors.

5.20   Environment

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

Event Api

ev api

5.21   Events

Dazu entscheiden dass es ein Type-Internes Eventsystem gibt, dessen Events nur in Type existieren. Damit polluten die nicht den globalen (name)space mit events. Nur bestimmte events werden global (nativ) getriggert, und das sind genau die events, die nach aussenhin von Belang sind und abgegriffen werden koennen sollen (=API design)

Input   input event only event required so far

5.22   Settings

settings

5.23   Text Walker

text walker

5.24   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.

5.25   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. On the other hand, the library should be a "good citizen" on a website, which means it should be as unobtrusive and leave the developers as much freedom as possible. The library will essentially modify parts of the DOM that act as the editor's contents. It should be agnostic to the editor's contents at all times to give other developers the freedom to change the contents in any way needed without breaking the editor. A cache must account for external changes. The DOM3 Events specification[10] offers `MutationObserver`s to check for DOM changes. This feature is not supported by Internet Explorer version 10 or less[11]. Internet Explorer 9 and 10 offer an implementation for `MutationEvents`[12]. 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."[13]. Apart from that, the benefits of a cache may not signifcantly increase the library's performance. The actions that can be supported by a cache, most importantly moving the caret in the text, are not very complex and do not noticably afflict the CPU. Implementing an editor that is stateless in regards of its contents can also improve stability.

---

[10]`http://www.w3.org/TR/DOM-Level-3-Events/`, last checked on 07/21/2015

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

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

[13]`http://www.w3.org/TR/DOM-Level-3-Events/#legacy-mutationevent-events`, last checked on 07/21/2015

5.26 Etherpad integration

Overview

Etherpad[14] is a web-based collaborative real-time text editor with rich-text capabilites. 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 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 ther server side is saved as a stack of changesets, which ultimatively 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.

---

[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

The changeset it sends back includes the differences from the the client's current revision of the document (with its own changes already applied) to the the latest revision on the server that includes the result of all changes the server had to perfom along each revision as just described.

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. This implies that anything affected by the changesets and is not inserted or retained will be removed. *// CHECK ON THIS AGAIN!*

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. Any local changes that have not been sent to the server yet, the client stores 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 $A \cdot X \cdot Y$, 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 uncommited, 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 X and B and Y 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 and B and 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 client–server protocol can be reverse engineered.

5.27   Extending

Type's modular structure is designed for extension. Type's `prototype` has been exposed as `Type.fn` all its classes in the `Type` namespace to provide other developers with all of Type's functionality in a sctructured and easily accessible manner.

Plugin API

...and plugin cache

```
1   // Adding contructors
2   Type.fromEtherpad = function() { /* Implementation */ };
3
4   // Events to load plugins on instantiation
5   Type.on('ready', function(type) { /* Implementation */ });
6
7   // Lazy loading using plugin cache
8   Type.fn.myPluginMethod = function (tag, params) {
9     var cmd = this.plugin('cmd', /* Plugin instantiation */ );
10    /* Execute myPluginMethod */
11  };
12
13  // Static function calls just like jQuery
14  Type.fn.cmd = function (tag, params) { /* Implementation */
       };
```

Listing 5.5: Example calls to format text

Erstellung von Erweiterungen und Plugins soll so einfach und frei wie möglich sein. Es ist die Verantwortung der Programmierer diese Freiheit so zu nutzen, dass sie trotzdem sauber arbeiten. jQuery zeigt, dass Freiheit funk-

tioneren kann und nicht zu schlechtem nutzen führt. Schlechte Plugins sterben durch die Dynamik des "Marktes" aus. Die Vorteile ueberwiegen: Beim CKEditor ist es aufwändig (viel Code um sich in den Editor zu integrieren) und schwierig (Lernkurve) Erweiterungen zu schreiben. Der CKEditor ist aber auch ein Produkt und kein Framework. Ein Framework sollte Möglichkeiten geben und einfach (und painless) sein. jQuery hat für sich Konventionen erzeugt und Möglichkeiten gegen, die Programmierer effektiv arbeiten lassen können. Und jQuery zeigt: Es funktioniert.

Part IV

Evaluation

5.28 Mobile Support

5.29 Development / Meta

Crockford style is a bad idea. I will change it to Standard or Airbnb https://github.com/airbnb/javascri

List of Figures

Listings

Bibliography

[bf,] 2 active tickets by component – ckeditor.

[aw,] 7 user interaction — html5.

[ar,] Apple - press info - apple releases safari 3.1.

[bi,] Bug list.

[bg,] Bug list.

[med,] The bug that blocked the browser — medium engineering — medium.

[can,] Can i use... support tables for html5, css3, etc.

[at,] Chrome releases: Stable channel update.

[as,] Content editable - web developer guide | mdn.

[av,] Html standard.

[bh,] Issues - chromium - an open-source project to help move the web forward. - google project hosting.

[ad,] Methods (internet explorer).

[am,] Midas - mozilla | mdn.

[br,] Minimizing browser reflow | best practices | google developers.

[aq,] Opera 9.0 for windows changelog.

[ap,] Opera changelogs.

[sop,] Paste as plain text contenteditable div & textarea (word/excel...) - stack overflow.

[ai,] Rich-text editing in mozilla | mdn.

[ag,] Top 9 browsers from june 2014 to june 2015 | statcounter global stats.

[ah,] The whatwg blog — the road to html 5: contenteditable.

Part V

Appendix

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)