RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Anbindung eines open-source Rich-Text-Editors an Middleware zum gemeinsamen Arbeiten an verschlüsselten Dokumenten

Theodros Zelleke

Bachelorarbeit – 14. Juli 2017.
Lehrstuhl für Netz- und Datensicherheit.

1. Prüfer: Prof. Dr. Jörg Schwenk
2. Prüfer: M. Sc. Dennis Felsch

hgi Lehrstuhl für Netz- und Datensicherheit

Zusammenfassung

Der *Lehrstuhl für Netz- und Datensicherheit* der *Ruhr-Universität Bochum* hat kürzlich mit *SECRET* [1] eine Anwendung vorgestellt, die Echtzeit-Kollaboration an verschlüsselten Textinhalten ermöglicht. Die Anwendung verwendet das Konzept der *Operational Transforms* [2], um zu gewährleisten, das Repräsentationen auf mehreren Clients, die den Textinhalt gleichzeitig bearbeiten, konsistent bleiben. SECRET verknüpft dieses Konzept mit XML-Verschlüsselung, um zu vermeiden, dass sensible Textinhalte gegenüber der zentralen Serverkomponente offenbart werden müssen. Im Gegensatz zu populären Anwendungen zur Echtzeit-Kollaboration wie *Google Docs* oder *Microsoft Office Online* fehlt SECRET eine mächtige grafische Benutzerschnittstelle, die es ermöglicht Rich-Text Formatierung vorzunehmen. Das Ziel dieser Bachelorarbeit soll es sein, einen geeigneten *open-source* Rich-Text Editor zur Anbindung an die Codebasis von SECRET zu recherchieren, die Anbindung zu konzipieren und die konkrete Implementierung soweit wie es gelingt voranzutreiben.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Datum	Theodros Zelleke

Inhaltsverzeichnis

1 Einleitung

Anwendungen zur Echtzeit-Kollaboration an Office-Dokumenten wie *Google Docs* oder *Microsoft Office Online* erfreuen sich privat großer Beliebtheit und gewinnen im modernen Arbeitsumfeld zunehmend an Bedeutung. Diese Anwendungen verwenden das Konzept der Operational Transforms (OT) [2] um gleichzeitige und möglicherweise zueinander widersprüchliche Editieranweisungen verschiedener Bearbeiter eines Dokuments zu integrieren. So wird gewährleistet, dass alle Bearbeiter eines Dokuments den gleichen, konsistenten Inhalt betrachten.

Im Hinblick auf den Datenschutz teilen die vorgenannten Anwendungen den erheblichen Nachteil dass alle Daten im Klartext an den Cloud-Server gesandt werden. Eine Lösung die Echtzeit-Kollaboration an verschlüsselten Dokumenten ermöglicht ist daher sehr wünschenswert und Gegenstand aktueller Forschung und Entwicklung. Da die Datenformate der vorgenannten Anwendungen intern auf dem XML-Format beruhen bietet sich die Technologie der XML-Verschlüsselung zur Verschlüsselung eines Dokuments oder ausgewählter Teilbereiche an. Der *Lehrstuhl für Netz- und Datensicherheit* (NDS) der *Ruhr-Universität Bochum* (RUB) hat kürzlich mit *SECRET* [1] eine Anwendung vorgestellt, die diesen Ansatz verfolgt und Echtzeit-Kollaboration mit XML-Verschlüsselung kombiniert.

1.1 Motivation

Die Benutzerschnittstelle von SECRET umfasst ein Texteingabefeld, in das der Benutzer Klartext eingeben kann. Es gibt keine Möglichkeit den eingegebenen Text zu formatieren oder zu strukturieren. Diese Funktionalität ist sehr weit von dem vertrauten Standard der vorgenannten, populären Applikationen entfernt. Solche, fehlende *Rich-Text*-Funktionalität muss aber nicht zwingend von Grund auf selbst implementiert werden[1]. Stattdessen gibt es heutzutage sehr viele, frei verfügbare *open-source* Projekte, die einen Rich-Text Editor implementieren. Daher ist es naheliegend die SECRET-Anwendung, die ohnehin bereits modular aufgebaut ist, an einen solchen Rich-Text Editor anzubinden und so um eine mächtige, grafische Editorkomponente zu erweitern.

[1] Tatsächlich würde ein solches Vorhaben bei weitem auch den Rahmen einer Masterarbeit sprengen.

1.2 Zielsetzung

Die Zielsetzung dieser Arbeit ist es, einen modernen Rich-Text Editor an die Codebasis von SECRET anzubinden. Dazu muss zunächst ein geigneter Editor aus dem grossen Angebot verfügbarer Editoren ausgewählt werden. Anschließend muss eine konkrete Anbindung konzipiert werden. Wie diese im Detail zu gestalten ist, hängt letztlich von vielen, auch kleineren Details der beteiligten APIs ab. In Prinzip ist die gestellte Aufgabe vergleichbar zu dem, was die SECRET zugrunde liegende Kernkomponente bereits leistet. Diese Komponente hält mehrere Repräsentationen eines XML-Dokuments „auf gleichem Stand“. Die Anbindung des Editors erfordert es zusätzlich dessen, HTML-basierte, Repräsentation eines Rich-Text Dokuments ebenfalls „auf gleichem Stand“ zu halten. Dazu werden keine OT benötigt, da die JavaScript-Laufzeitumgebung „single-threaded“ ist, es also nicht „gleichzeitig“ zu Modifikation beider Repräsentationen innerhalb eines Webbrowser-Prozesses kommen kann. Trotzdem können Inkompatibilitäten der beteiligten APIs und Besonderheiten der JavaScript-Laufzeitumgebung zu unvorhergesehenen Hindernissen führen.

1.3 Eigener Beitrag

Im Rahmen dieser Arbeit wurde zunächst eine eingehende Recherche durchgeführt um einen geeigneten Rich-Text Editor für die Anbindung an die Codebasis von SECRET zu finden. Nachdem ein geeigneter *open-source* Editor gefunden worden war, wurde ein neues Entwicklungsprojekt gestartet, welches auf der Codebasis von SECRET beruhte und in welches die Codebasis des Editors eingebunden wurde. Bevor mit der Implementierung der fehlenden Programmlogik begonnen werden konnte, war es zunächst notwendig den „Build-Prozess“ des gesamten Projekts an die gesteigerte Komplexität anzupassen und auch grundsätzlich zu verbessern. Nachdem sich der Autor mit den internen Details und Vorgängen sowohl des gewählten Editors, alsauch von SECRET, vertraut gemacht hatte, wurde eine Implementierung konzipiert und deren Umsetzung vorangetrieben.

1.4 Aufbau dieser Arbeit

Dieses Kapitel beschreibt die Problemstellung sowie die Motivation und Zielsetzung dieser Bachelorarbeit. Darüberhinaus werden wichtige Begriffe wie „Operational Transforms“ eingeführt und erläutert.

Im folgenden Kapitel werden zunächst die Softwarearchitektur von SECRET allgemein und die bereitgestellte API zur Verwaltung von XML-basierten Strukturen näher dargestellt. Anschließend werden die Kriterien dargelegt anhand derer die

Entscheidung für einen geeigneten Rich-Text Editor getroffen wurde und der ausgewählte Editor wird vorgestellt.

Danach werden in Kapitel 3 die verwendeten Werkzeuge und Technologien vorgestellt. Insbesondere wurde ein neues Werkzeug eingesetzt, um die Anwendung zu „bauen“.

In Kapitel 4 wird dann die Implementierung der benötigten Programmlogik zur Anbindung des gewählten Editors ausführlich beschrieben und anhand von Codebespielen erläutert.

Kapitel 5 enthält zum Schluss eine Zusammenfassung über die Themen dieser Bachelorarbeit.

2 Hintergrund

Diese Bachelorarbeit hatte die wechselseitige Anbindung zweier Softwarekomponenten zum Ziel. Diese beiden Komponenten, SECRET und ein Webbrowser-basierter Rich-Text Editor werden in diesem Kapitel folgerichtig näher vorgestellt. Bevor auf den verwendeten Rich-Text Editor eingegangen wird, werden die Kriterien seiner Auswahl erläutert.

2.1 SECRET

SECRET ist eine Erweiterung von *ShareJS* [3], einer OT-basierten *open-source* Echtzeit-Kollaborationsanwendung. ShareJS besitzt eine Client-Server-Architektur wobei der Server unter *NodeJS* betrieben wird und der Client in jedem modernen Webbrowser läuft. Die Kommunikation zwischen beiden Komponenten findet über WebSockets statt. ShareJS selbst unterstützt OT auf Text- und JSON-Datenstrukturen, besitzt aber einen modularen Aufbau der die Erweiterung um andere Datenstrukturen mit wenig Aufwand zulässt.

Die Autoren von SECRET haben ShareJS um eine Erweiterung ergänzt, die OT auf XML-Datenstrukturen ermöglicht. Dazu war er erforderlich einen neuartigen Ansatz zur Implementierung der benötigten OT-Operationen für den XML-Datentyp abzuleiten, der die Baumstruktur eines XML-Dokuments berücksichtigt und diese zusätzlich mit XML-Verschlüsselung kombiniert. Der in SECRET intergrierte, primitive grafische Editor kommuniziert mit der API dieser XML-Erweiterung um Editieroperationen des Nutzers an die ShareJS Programmlogik weiterzureichen. Im Folgenden wird diese API näher vorgestellt.

2.1.1 API der XML-Erweiterung von ShareJS

Die API des XML-Erweiterung stellt – unter anderem – Funktionen für die Modifikation des DOMs eines von ShareJS verwalteten XML-Dokuments bereit. Einige wichtigste Funktionen dieser API sind in Quellcode 2.1 aufgelistet.

Die Methoden der API erwarten stets als erstes Argument einen XPath-Ausdruck im Stringformat. Dieser dient dazu, den zu modifizierenden Knoten in der Baumstruktur des XML-Dokuments eindeutig zu identifizieren. Weiterhin ist zu beachten, dass die Methode `insertElementAt` als dritten Parameter einen String erwartet,

der den einzuhängenden DOM-Knoten als serialisierten String erwartet. Dieser wird dann intern geparst und der entsprechende DOM-Knoten oder DOM-Zweig wird *neu* erstellt und anschließend in das von SECRET verwaltete DOM eingehängt. Schliesslich übernehmen alle Methoden der API als letzten Parameter einen Verweis auf eine *Callback*-Funktion. Dies ist ganz typisch in der modernen JavaScript-Programmierung und der asynchronen Natur der JavaScript-Umgebung, sowohl im Webbrowser alsauch auf dem Server, geschuldet.

```
1 insertElementAt(XPath, pos, str, cb);
2 removeElement(XPath, cb);
3
4 insertTextAt(XPath, pos, str, cb);
5 removeTextAt(XPath, pos, len, cb);
6 replaceTextAt(XPath, pos, value, cb);
```

Quellcode 2.1: Zur Identifizierung des zu modifizierenden DOM-Knotens erwartet die XML-Datentyp-API stets als erstes Argument einen XPath-Ausdruck. Weiterhin übernimmt jede Funktion der API als letztes Argument einen Verweis auf eine Callback-Funktion.

Im weiteren Text soll die API der XML-Erweiterung als *ShareJS-API* referenziert werden um sie von der API des Rich-Text Editors zu unterscheiden. Die Auswahl eines geeigneten solchen Editors aus dem vielfältigen Angebot an *open-source*-Projekten, die diese Funktionalität auf ein oder andere Weise implementieren, ist der Fokus des nächsten Abschnitts.

2.2 Recherche eines geeigneten *open-source* Rich-Text Editors

Eine Suche nach JavaScript-basierten Rich-Text Editoren auf *Github*[1] lieferte zum Zeitpunkt der Erstellung dieser Bachelorarbeit (Mai 2017) über 300 Treffer. Die verschiedenen Projekte unterscheiden sich nicht nur erheblich in Grösse und Funktionsumfang, sondern auch in der verwendeten JavaScript-Version, ihrer internen Architektur und Abhängigkeiten von externen Frameworks (z. B. *React*, *Angular*), und nicht zuletzt im Umfang und der Qualität der bereitgestellten Dokumentation.

Gerade der letzgenannte Faktor ist von enormer Bedeutung angesichts einer Grösse der Projekte von mehreren tausend Zeilen Quellcode bis zu mehr als hunderdttausend Zeilen. Ohne eine umfangreiche und qualitativ hochwertige Dokumentation ist es kaum möglich sich innerhalb des Zeitrahmens einer Bachelorarbeit in eine solche Codebasis einzuarbeiten.

[1] https://github.com/search?l=JavaScript&p=1&q=rich+text+editor&type=Repositories

Auch die verwendete JavaScript-Version beinflusst die Einarbeitung in eine neue Codebasis erheblich. JavaScript hat in den vergangenen Jahren eine enorme Entwicklung genommen und mit dem neuen Sprachstandard ECMAScript 2015 (ES2015) gravierende Veränderungen erfahren. Hier ist vorrangig die Einführung neuer Syntaxelemente wie Klassen und Module hervorzuheben, welche speziell die Entwicklung komplexerer Anwendungen erheblich vereinfachen. Im Hinblick auf diese Bachelorarbeit ist dies insoweit relevant, als dass sich der Quellcode im Allgemeinen wesentlich flüssiger lesen lässt und die gesamte Architektur der Anwendung schneller nachzuvollziehen ist.

Die Kehrseite dieser Entwicklung ist das selbst modernste Webbrowser (noch) nicht alle neuen Features unterstützen. Daher ist es erforderlich noch einen *Transpilierungsvorgang* durchzuführen und den Quellcode in den gegenwärtig in allen Webbrowsern unterstützten Sprachstandard ECMAScript 5 (ES5) umzuwandeln. Dieser Umwandlungsschritt ist ein wichtiger Bestandteil eines individuell für das jeweilige Projekt zu konfigurierenden Build-Prozesses, welcher im nachfolgenden Kapitel 3 näher erläutert wird.

Tabelle 2.2 vergleicht populäre *open-source* Rich-Text Editoren im Hinblick auf ihre Eignung für die Bachelorarbeit. Basierend auf den vorgenannten Kriterien wurde für die Umsetzung dieses Bachelorarbeit der Rich-Text Editor *Quill* ausgewählt.

Tabelle 2.2: Die Tabelle vergleicht populäre *open-source* Rich-Text Editoren im Hinblick auf ihre Eignung für das Projekt. Die Anzahl der Sterne des Projekts auf Github diente als Indikator für die Popularität des Projekts. Weitere wichtige Kriterien bei der Auswahl des Editors waren die verwendete JavaScript Version – ES5, ES2015 oder TypeScript (TS) – die Grösse des Projekts, abgeschätzt anhand der Anzahl der Quellcodezeilen (SLOC) und die Qualität und der Umfang der Dokumentation.

	Github Sterne[a]	JS Version	SLOC[a]	Dokumentation
CKEditor[b]	3369	ES5	661911	++
TinyMCE[c]	4276	ES5	146499	+++
QuillJS [4]	13262	ES2015/TS	28814	+++

[a] Daten abgerufen am 08.05.2017
[b] https://github.com/ckeditor/ckeditor-dev
[c] https://github.com/tinymce/tinymce

2.3 Quill

Quill ist ein moderner Rich-Text-Editor dessen Fokus auf Browser-Kompatibilität und Erweiterbarkeit liegt. Das Projekt wurde von Jason Chen und Byron Milligan

begründet und schliesslich von *Salesforce.com* an die open-source Gemeinde übergeben. Quill ist modular aufgebaut und beruht seinerseits intern auf zwei weiteren Softwarekomponenten, Parchment [5] und Delta [6]. Beides sind eigenständige Projekte mit separater Codebasis und API-Schicht.

3 Werkzeuge und Technologien

Dieses Kapitel stellt die wichtigsten verwendeten Technologien uns Softwarewerkzeuge vor. Zunächst wird allerdings auf einige Eigenschaften der JavaScript-Laufzeitumgebung eingegangen. Daraus ergeben sich Einschränkungen, die die Konzeption und Implementierung dieses Bachelorprojektes maßgeblich beeinflussen. Anschliessend werden die in dieser Arbeit verwendeten JavaScript-Dialekte vorgestellt. Zum Schluss wird eine in modernen Webbrowsern nativ implementierte API zur Überwachung von Änderungen am DOM eines Dokuments vorgestellt. Um die Verwaltung solcher DOM-Modifikationen zu vereinfachen wurde eine auf dieser API aufsetzende JavaScript-Bibliothek verwendet.

3.1 Die JavaScript-Laufzeitumgebung

Die JavaScript-Laufzeitumgebung legt alle auftretenden Ereignisse in einer speziellen Ereignis-Queue in der Reihenfolge ihres Auftretens ab [7]. Verknüpft mit dem jeweiligen Ereignis sind die Argumente, die den entsprechenden Callback-Funktionen bei deren Ausführung zur Verfügung gestellt werden. Die Verarbeitung der generierter Ereignisse wird nebenläufig in einem separaten Thread vorgenommen. Dieser Thread überprüft permanent, ob die Ereignis-Queue zu verarbeitende Ereignisse enthält. Wenn dies der Fall ist wird das „vorderste“ Ereignis-Objekt samt der hinterlegten Argumente aus der Ereignis-Queue entfernt und alle regisrierten Callback-Funktionen werden in der Reihenfolge ihrer Registrierung ausgeführt. Der individuelle Code den eine Callback-Funktion ausführt kann nicht preemptiv unterbrochen werden, sondern wird zu Ende ausgeführt bevor die nächste Callback-Funktion aus der Ereignis-Queue genommen und ausgeführt wird. Diese Zusammenhänge werden ausführlich in [8] dargestellt.

3.2 JavaScript-Dialekte

JavaScript ist heutzutage zweifelsohne eine der bedeutendsten Programmiersprachen. Allerdings konnten die heutigen Einsatzszenarien bei der Erschaffung der Sprache nicht annähernd vorhergesehen werden. Daher leidet JavaScript in der gegenwärtig flächendeckend unterstützten Version ES5 an vielen syntaktischen Fallstricken und weiteren Problemen, die andere populäre Programmiersprachen nicht

aufweisen. Um die Probleme zu überdecken und JavaScript um fehlende Merkmale zu erweitern, sind in der Vergangenheit verschiedene Mini-Sprachen entstanden.

CoffeeScript ist eine solche Mini-Sprache und bietet gegenüber gewöhnlichem JavaScript eine leichtgewichtigere Syntax und ist daher angenehmer zu lesen und zu programmieren. Dies erleichtert die Codeübersicht und damit Entwicklung und Fehlerfindung. CoffeeScript-Dateien werden von einem Compiler in JavaScript-Code übersetzt, um auf dem entsprechenden Client zur Ausführung gebracht zu werden. Der Quellcode von ShareJS und von SECRET liegt in CoffeeScript vor.

ES2015 stellt die gegenwärtig aktulle Spezifikation für JavaScript dar. Allerdings wird dieser Standard nicht flächendeckend und in allen Merkmalen von aktuellen Webbrowsern unterstützt. Dennoch markiert ES2015 eine signifikante Verbesserung der Sprache und adressiert einige der grösseren Unzulänglichkeiten. Der Quellcode von Quill liegt in ES2015 vor. Daher wurde ES2015 für die Implementierung dieser Bachelorarbeit gewählt. Genau wie CoffeeScript-Dateien müssen auch ES2015-Dateien gegenwärtig noch von einem Compiler übersetzt werden.

3.3 JavaScript-Entwicklung mit Webpack

Wie im vorigen Abschnitt erwähnt, müssen die Quelldateien dieser Bachelorarbeit vor Ausführung von verschiedenen Compilern kompiliert werden. Dies kann im Prinzip von Hand geschehen, was in der Praxis aber unzumutbar umständlich und fehleranfällig ist – eine automatisierte Lösung ist unerlässlich. Unabhängig davon wie der Kompilierungsprozess gestartet wird, ist das generierte Kompilat erheblich komplexer und daher schwieriger zu lesen und nachzuvollziehen. Daher ist es sehr hilfreich, wenn das Debugging der Anwendung im Kontext der ursprünglichen Quelldateien erfolgen kann. Um dies zu ermöglichen, müssen während des Kompilierungsprozesses sogenannte „Sourcemaps“ erstellt werden, die das Kompilat Zeile für Zeile mit dem Quellcode verknüpfen. JavaScript-Debugger, die in modernen Webbrowsern integriert sind, erkennen diese Sourcemaps und erlauben uneingeschränktes Debugging im Kontext der Quelldateien. Zu guter Letzt ist sehr wünschenswert, wenn der Kompilierungsprozess automatisch gestartet wird, sobald eine vorhandene Quelldatei editiert und gespeichert worden ist.

Webpack [9] ist ein modernes JavaScript-Entwicklungswerkzeug, das alle vorgenannten Merkmale unterstützt und daher für diese Bachelorarbeit verwendet wurde. Abbildung 3.1 zeigt den Chrome-Debugger bei Ausführung der Anwendung. Dieser erkennt die erzeugten Sourcemaps und erlaubt das Debugging im Kontext der CoffeeScript-Quelldateien.

Die in diesem Abschnitt beschriebenen Konfigurationen und Anpassungen stellen eine erhebliche Verbesserung gegenüber dem in SECRET vorgefundenen Status dar.

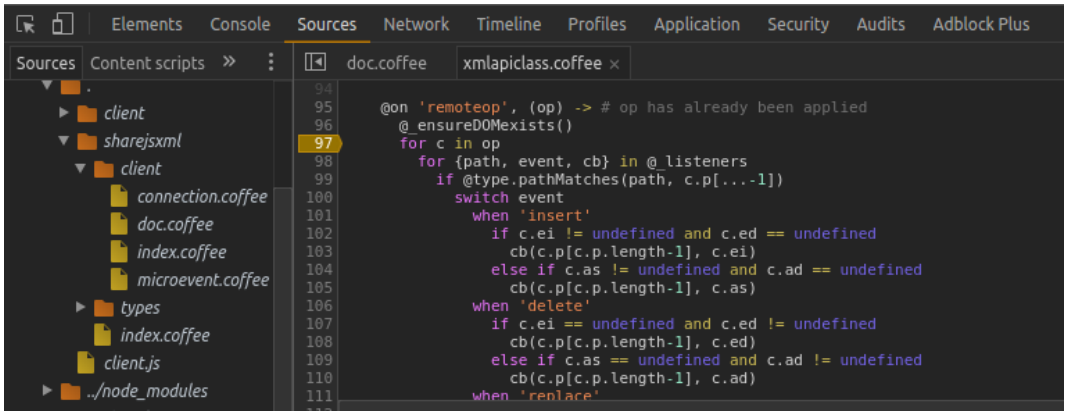


Abbildung 3.1: Der Chrome-Debugger erkennt die erzeugten Sourcemaps und erlaubt das Debugging im Kontext der CoffeeScript-Quelldateien.

Sie erleichtern, *über diese Bachelorarbeit hinaus*, die weitere Entwicklungsarbeit am Projekt.

3.4 Die MutationObserver-API

Die `MutationObserver`-API [10] moderner Webbrowser ermöglicht es, eine Callback-Funktion zu registrieren, die asynchron aufgerufen wird, wenn sich Änderungen in bestimmten Bereichen des DOM ereignen. DOM-Modifikationen die so überwacht werden können sind das Hinzufügen, Entfernen und Verschieben von DOM-Knoten, Modifikationen an den Attributen eines DOM-Knoten oder Änderungen am Inhalt von Textknoten. Eine konkrete DOM-Modifikation wird von einen `MutationObserver` in einem `MutationRecord`-Objekt aufgezeichnet, wenn sie sich ereignet. Zu einem späteren Zeitpunkt ruft die Ereignisverarbeitung der JavaScript-Laufzeitumgebung die registrierte Callback-Funktion auf und übergibt alle „ausstehenden“ `MutationRecord`-Objekte in einem Array als Argument.

3.5 Die MutationSummary-Bibliothek

Um für eine Gruppe von DOM-Modifikationen den „Netto-Effekt“ zu erhalten und um den Umgang mit `MutationRecord`-Objekten insgesamt zu vereinfachen wurde mit *Mutation Summary* [11] eine weitere JavaScript-Bibliothek eingebunden. Die Bibliothek stellt eine einzige Klasse zur Verfügung, deren Instanzen nach Initialisierung einen beliebigen Teilbereich eines DOM auf jegliche Änderungen hin überwachen, den „Netto-Effekt“ komplexer Änderungen berechnen, und das Ergebnis anschließend an eine Callback-Funktion weiterreichen.

Die Signatur des Konstruktors der Klasse ist in Quellcode 3.2 wiedergegeben. Welche Art von DOM-Modifationen konkret zu überwachen sind wird über die Konfigurations-Objekte, die der `queries`-Eigenschaft übergeben werden, spezifiziert. In dieser Arbeit wurden hier stets alle Arten von DOM-Modifikationen spezifiziert.

```
let observer = new MutationSummary({
  callback: Handler        // erforderlich
  rootNode: DOM-Knoten,    // optional
  observeOwnChanges:       // optional
  oldPreviousSibling:      // optional
  queries: [
    { /* query1 */ },
    { /* query2 */ },
      // ...
    { /* queryN */ }
  ]
});
```

Quellcode 3.2: Die Signatur des `MutationSummary`-Konstruktors erwartet eine Referenz auf eine Callback-Funktion und optional einen Wurzelknoten unterhalb dessen DOM-Modifikationen aufgezeichnet werden sollen.

4 Implementierung

Dieses Kapitel präsentiert die Implementierung. Zuerst wird eine generelle Konzeption für eine Implementierung ausgearbeitet. Dabei wird erkannt werden, dass die gestellte Aufgabe in zwei weitgehend unabhängig voneinander zu bearbeitende Teilaufgaben aufgeteilt werden kann. Die Implementierung für beide Teilaufgaben wird anschließend, soweit sie im Rahmen dieser Arbeit abgeschlossen werden konnte, präsentiert.

4.1 Konzeption der Implementierung

In diesem Abschnitt soll das Konzept für die Implementierung herausgearbeitet werden. Zu diesem Zweck präsentiert Abbildung 4.1 eine Übersicht der beteiligten Komponenten und deren Beziehung zueinander.

Ein konkretes Dokument liegt zu jedem Zeitpunkt in mehreren Repräsentationen vor. Quill repräsentiert das Dokument als DOM-Fragment des Webbrowsers mit einem <DIV>-Element als Wurzelknoten. Dieses HTML-Element ist mit der Klasse „ql-container“ ausgezeichnet und liegt innerhalb eines DOM-Bereichs, der von Quill kontrolliert wird und die gesamte Benutzerschnittstelle des Editors, samt der Werkzeugleiste, beinhaltet. Im weiteren Text wird dieses <DIV>-Element noch häufig erwähnt und soll daher als *Dokumentcontainer* referenziert werden. Der ShareJS-Client im Webbrowser, ebenso wie der ShareJS-Server und gegebenfalls weitere, entfernte ShareJS-Clients halten ebenfalls Repräsentationen des Dokuments vor.

In diesem Kontext ist zwischen lokalen und entfernten Editieroperationen zu unterscheiden. Lokale Editieroperationen werden von einem Benutzer, der mit Benutzerschnittstelle von Quill im lokalen Webbrowser interagiert, vorgenommen. Sie werden von Quill verarbeitet und führen zu Modifikationen am DOM unterhalb des Dokumentcontainers. Um die Repräsentation des Dokuments im gesamten ShareJS-Ökosystem auf aktuellem Stand zu halten, muss der lokale ShareJS-Client über die durchgeführten DOM-Modifikationen informiert werden. Entfernte Editieroperationen werden von entfernten ShareJS-Clients ausgelöst und über den zentralen ShareJS-Server an den lokalen ShareJS-Client übermittelt. Dieser integriert lokale und entfernte Editieroperationen unter Verwendung von OT. Anschließend muss die lokale Ansicht des Dokuments, welche von Quill verwaltet wird, aktualisiert werden.

Grundsätzlich lässt sich die Aufgabe der Anbindung des Quill-Editors an die ShareJS-Funktionalität in die folgenden beiden Unteraufgaben unterteilen.

- Es müssen alle DOM-Modifikation die infolge lokaler Editieroperationen an der Repräsentation des Dokuments in Quill entstehen an die ShareJS-API weitergeleitet werden.
- Es müssen entfernte Editieroperationen nach ihrer Verarbeitung durch den lokalen ShareJS-Client in die Repräsentation des Dokuments in Quill übernommen werden.

Diese beiden Teilaufgaben können prinzipiell völlig unabhängig voneinander bearbeitet werden. Ihnen ist gemeinsam, dass in der zu entwickelnden Programmkomponente jeweils zunächst die auslösenden Ereignisse registriert werden müssen, um auf diese zu reagieren. Weiterhin müssen die konkreten Lösungsansätze an der jeweils bereitgestellten Empfänger-API orientiert werden; sie gibt das „Ziel“ vor, gegen welches entwickelt werden muss. Daraus ergibt sich, dass die registrierten Modifikationen im Hinblick auf Format und Granularität gegebenenfalls vor Weiterleitung and die jeweilige Empfänger-API transformiert werden müssen.

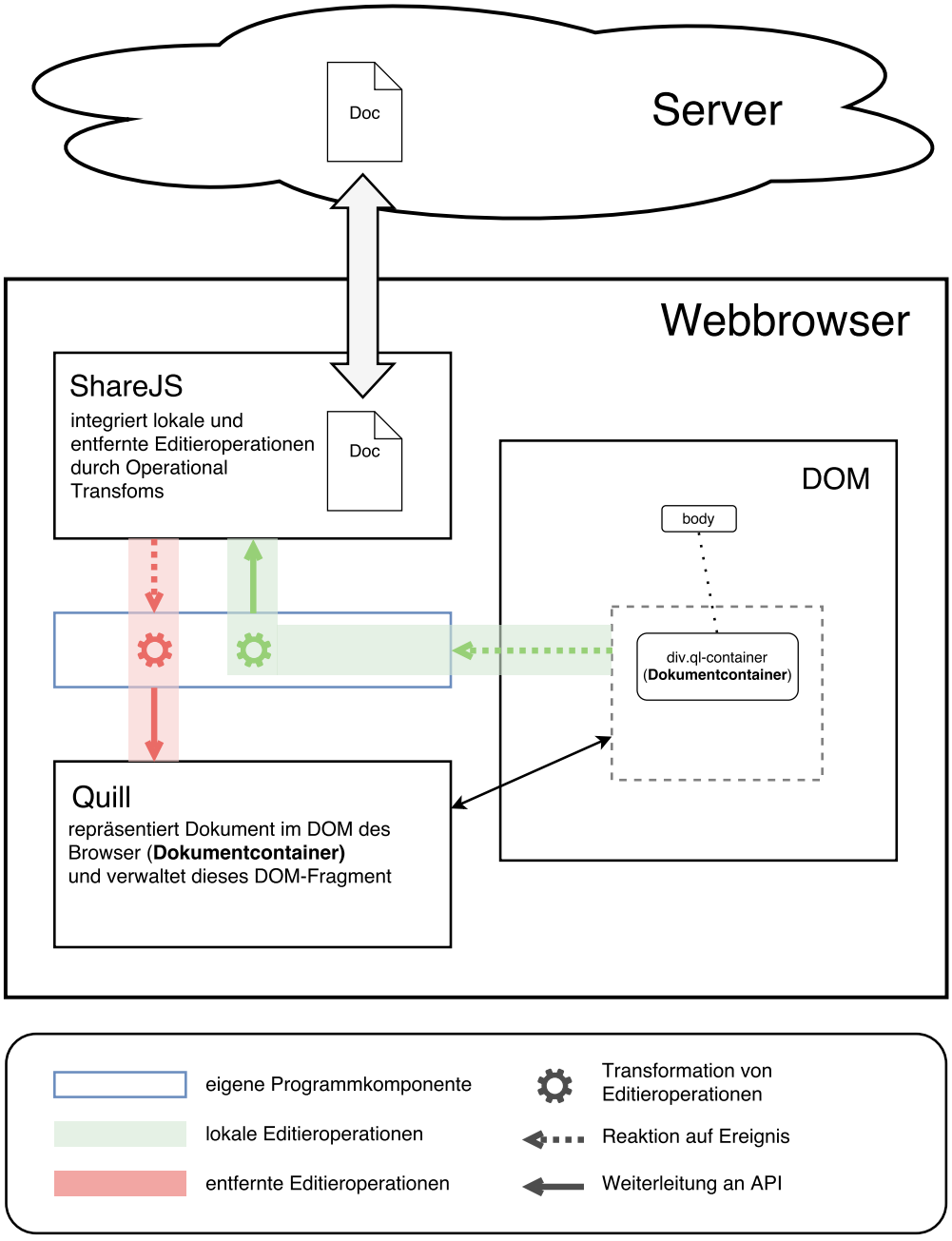


Abbildung 4.1: Die Grafik gibt einen Überblick über die beteiligten Komponenten und deren Beziehung untereinander.

4.2 Verarbeitung lokaler Editieroperationen

Zur Umsetzung der ersten Teilaufgabe ist es erforderlich auf alle DOM-Modifikationen am Dokumentcontainer innerhalb von Quill zu reagieren. Daher stellt sich zunächst die Frage, wie eigener Programmcode von diesen Modifikationen benachrichtigt werden kann. Diese Frage wird im folgenden Unterabschnitt beantwortet werden. Danach wird auf ein Problem eingegangen, das bei der Umsetzung dieser Arbeit auftauchte und auf eine fundamentale Limitierung der JavaScript-Laufzeitumgebung zurückzuführen ist. Ein Algorithmus, der zur Umgehung des Problems abgeleitet wurde, wird anschließend vorgestellt. Nachdem die Bereiche des DOM, die zu aktualieren sind, identifiziert wurden, wird erläutert, wie die entsprechenden Aufrufe der ShareJS-API generiert werden.

4.2.1 DOM-Modifikationen im Dokumentcontainer von Quill

Dieser Abschnitt beschäftigt sich mit der Frage wie die Modifikationen, die Quill am DOM des Dokumentcontainers als Reaktion auf Benutzerinteraktion vornimmt, in der eigenen Programmlogik registriert und verarbeitet werden können. Dazu werden in der Folge zwei konkrete Beispiele für Benutzerinteraktion detailiert betrachtet. Alle Modifikationen die Quill vornahm wurden in chronologischer Reihenfolge registriert und werden im entsprechenden Beispiel in tabellarischer Form wiedergegeben.

Zur Registrierung der atomaren DOM-Modifikationen wurde die in Abschnitt 3.4 vorgestellte `MutationObserver`-API moderner Webbrowser verwendet. Über diese API wurde ein Observer für den Dokumentcontainer definiert und in der übergebenen Callback-Funktion wurde dann die detailierte Auswertung der DOM-Modifikationen mit Hilfe des bereitgestellten `MutationRecord`-Objektes vorgenommen.

Das `MutationRecord`-Objekt unterscheidet zwischen strukturellen DOM-Modifikationen, bei denen Knoten hinzugefügt, entfernt oder verschoben worden sind und Text-Modifikation, bei denen lediglich der Inhalt eines vorhandenen Textknotens verändert worden ist. In diesem Sinne ist das Hinzufügen, Entfernen oder Verschieben eines Textknotens eine strukturelle DOM-Modifikation. `MutationRecord` spezifiziert noch einen weiteren Typ, der eine Modifikationen an den Attributen eines Knotens signalisiert. Dieser Typ wurde hier ausgelassen, da er erst bei komplexeren Editieroperationen eines Benutzer, wie Änderungen von Textfarbe und Textfont, eine Rolle spielt.

Bevor das erste Beispiel – die Eingabe eines simplen Textes – analysiert wird, soll zunächst die Struktur der zur Darstellung verwendeten Tabelle erläutert werden. Die erste Spalte enthält zur Referenzierung der DOM-Modifikationen eine Nummerierung. Die weiteren Spalten enthalten in Reihenfolge von links nach rechts folgende Angaben:

Typ gibt an um welche Art von Modifikation es sich handelt. Hierbei steht „N“ für eine strukturelle DOM-Modifikation während „T“ für eine reine Text-Modifikation steht.

Knoten gibt den HTML-Tag des Knotens an, der von der DOM-Modifikation unmittelbar betroffen ist. Bei einer strukturellen DOM-Modifikation ist dies stets der Elternknoten der hinzugefügten, entfernten oder verschobenen Knoten. Bei einer Text-Modifikation ist dies der betroffene Textknoten, der keinen HTML-Tag besitzt und von einem Webbrowser daher als `#text` angegeben wird.

Vorgängerknoten gibt den unmittelbaren Vorgängerknoten der hinzugefügten, entfernten oder verschobenen Knoten bei einer strukturellen DOM-Modifikation an.

Nachfolgeknoten gibt den unmittelbaren Nachfolgeknoten der hinzugefügten, entfernten oder verschobenen Knoten bei einer strukturellen DOM-Modifikation an.

Elternknoten gibt den Elternknoten des in der Spalte *Knoten* referenzierten Knotens an.

Veränderung enthält unterschiedliche Angaben in Abhängigkeit vom Typ der Modifikation. Bei einer strukturellen DOM-Modifikation enthält diese Spalte Angaben in der Form `+/-[Knoten(Textinhalt)<Elternknoten`, wobei `+/-` signalisiert ob Knoten hinzugefügt oder entfernt wurden. Bei einer Text-Modifikation ist die Angabe in der Form `[Text vorher<<Text nachher]`.

Dokument gibt den serialisierten Inhalt des Dokumentcontainers zur Laufzeit der entsprechenden Callback-Funktion an.

In der folgenden Diskussion muss zwischen den DOM-Modifikationen und den sie auslösenden Editieroperationen unterschieden werden. Eine konkrete Editieroperation, zum Beispiel das Tippen eines einzelnen Buchstaben oder der Klick auf einen Button der Werkzeugleiste von Quill, hat häufig mehrere atomare DOM-Modifikationen zur Folge. Die ursächlichen Ereignisse sind hier die Benutzerinteraktionen. Auf diese reagiert Quill durch Callback-Funktionen, die die jeweiligen „Umbauten“ am DOM vornehmen. In Abschnitt 3.1 wurde bereits näher erläutert, dass die jeweilige Callback-Funktion immer „zu Ende“ läuft und nicht preemptiv durch andere Callback-Funktionen unterbrochen werden kann. Um diesen Sachverhalt hervorzuheben sind in der nachfolgenden, tabellarischen Darstellung aufeinanderfolgende DOM-Modifikation die als Reaktion auf eine einzelne Editieroperation ausgelöst worden waren, *nicht* durch horizontale Linien getrennt.

In diesem Zusammenhang ist es wichtig zwischen Eigenschaften zu unterscheiden, die *direkt* auf dem `MutationRecord`-Objekt ausgewertet werden, und solchen die *indirekt* auf einem referenzierten Objekt ausgewertet werden. Wie in Abschnitt 3.4

erläutert, wird ein `MutationRecord`-Objekt *unmittelbar* bei Auftreten der DOM-Modifikation erzeugt danach nicht mehr modifiziert. Zur Laufzeit der entsprechenden Callback-Funktion spiegeln Eigenschaften, die direkt auf diesem Objekt ausgelesen werden, daher den Stand des DOM *zum Zeitpunkt des ursprünglichen DOM-Ereignisses* wider. Beispielsweise gilt dies für die `previousSibling`-Eigenschaft des `MutationRecord`-Objekts, aus der, bei strukturellen DOM-Modifikationen, eine Referenz auf den unmittelbaren Vorgängerknoten erhalten wird. Für Eigenschaften, die wiederum auf dem referenzierten Vorgängerknoten oder weiteren referenzierten Objekten ausgewertet werden, gilt dies nicht mehr. Diese können bis zur Laufzeit der Callback-Funktion sehr wohl weitere Veränderungen erfahren haben und so *nicht mehr den Stand des DOM zum Zeitpunkt des ursprünglichen DOM-Ereignisses widerspiegeln.*

4.2.1.1 Beispiel Simple Texteingabe

Tabelle 4.2 listet DOM-Modifikationen die Quill bei Eingabe des Textes „Hallo“ durch Tippen vornimmt. Die insgesamt sieben DOM-Modifikationen der Tabelle wurden von fünf Editieroperationen – den fünf Tastenklicks des Benutzers – ausgelöst.

Tabelle 4.2: Die Tabelle listet DOM-Modifikationen die Quill bei Eingabe des Textes „Hallo“ durch Tippen vornimmt. Eine detailierte Beschreibung des Inhalts der einzelnen Spalten ist dem Fließtext zu entnehmen.

	Typ	Kn.	VKn.	NKn.	EKn.	Veränderung	Dokument
1	N	P	-	BR	DIV	+[#text(H)<P]	H
2	N	P	#text	-	DIV	−[BR()<null]	H
3	T	#text	-	-	P	[»H]	H
4	T	#text	-	-	P	[H»Ha]	Ha
5	T	#text	-	-	P	[Ha»Hal]	Hal
6	T	#text	-	-	P	[Hal»Hall]	Hall
7	T	#text	-	-	P	[Hall»Hallo]	Hallo

Nach Initialisierung enthält der Dokumentcontainer ein <P>-Element und darin ein
-Element. DOM-Modifikationen 1–3 sind alle mit dem ersten Tastenklick, der Eingabe des Buchstaben „H“ in den textfreien Dokumentcontainer verknüpft. In DOM-Modifikation 1 wurde zunächst ein leerer Textknoten in das <P>-Element vor dem
-Element eingefügt. In DOM-Modifikation 2 wurde das
-Element aus dem DOM entfernt. Schließlich wurde der Textinhalt des im ersten Schritt eingefügten Textknotens um den Buchstaben „H“ erweitert. In der Folge wird der Inhalt

des Textknoten innerhalb des <P>-Element um jeweils einen Buchstaben erweitert.

Hier fällt auf, dass bereits aus der Spalte *Veränderung* für DOM-Modifikation 1 hervorgeht, dass der in diesem Schritt eingehängte Textknoten den Textinhalt „H“ enthält. *Wie kann das sein, wenn doch aus DOM-Modifikation 3 folgt, dass das „H“ erst in diesem, späteren Schritt eingefügt wird*? In ähnlicher Weise nicht unmittelbar nachvollziehbar sind auch die Angaben der letzten Spalte, *Dokument*, die identisch sind für die ersten drei DOM-Modifikationen und den Status des DOM nach Abschluß der dritten DOM-Modifikation widergeben. *Wieso wird dieser Status bereits zur Laufzeit der ersten Callback-Funktion ermittelt*?

Die Antwort auf beide Fragen umfasst zwei Aspekte:

1. Im vorliegenden Beispiel wurde als Reaktion auf den ersten Tastenklick eine, Quill-interne, Callback-Funktion ausgeführt, die DOM-Modifikationen 1-3 vornahm. Erst nach ihrer Beendigung wurden die mit den DOM-Modifikationen verknüpften Callback-Funktionen ausgeführt. Zur Laufzeit der Callback-Funktion für DOM-Modifikation 1 war also DOM-Modifikation 3 bereits umgesetzt. Dies wird unmittelbar anhand der letzten Spalte, *Dokument*, deutlich. Diese gibt den Inhalt des Dokumentcontainers, ermittelt aus der `innerHTML`-Eigenschaft, wider[1].
2. Bei strukturellen DOM-Modifikationen enthält die Spalte *Veränderung* auch Information die nicht unmittelbar auf dem entsprechenden `MutationRecord`-Objekt, sondern indirekt auf referenzierten Objekten ausgewertet wurde. Für DOM-Modifikation 1 wurde der Inhalt des betroffenen Textknotens indirekt aus der `textContent`-Eigenschaft ausgelesen. Dies geschieht zur Laufzeit der Callback-Funktion, also wie im ersten Punkt erläutert, erst nachdem DOM-Mutation 3 stattgefunden hatte. Daher liefert die `textContent`-Eigenschaft den, zu diesem Zeitpunkt bereits gesetzten Inhalt „H“.

Festzuhalten bleibt, dass – ohne übermäßigen Aufwand – nur die begrenzte Information, die direkt auf dem `MutationRecord`-Objekt ausgelesen werden kann, den Zustand des DOMs bei Auftreten des DOM-Ereignisses referenziert. Im Folgenden soll ein komplexeres Beispiel nachvollzogen werden.

4.2.1.2 Beispiel Fettschrift

Tabelle 4.3 zeigt die Abfolge der DOM-Modifikation, die Quill durchführt, wenn der Benutzer ein zuvor markiertes Textfragment durch einen Klick auf den entsprechenden Button in Fettschrift setzt. Der Umbau innerhalb des Dokumentcontainers ist in diesem Beispiel so umfangreich, da das markierte Textfragment einen Zeilenumbruch

[1] Die Callback-Funktion hielt eine Referenz auf den Dokumentcontainer in einer *Closure*. Zur Definition einer Closure siehe [12]

überspannt. Abbildung 4.4 zeigt einen Screenshot des Editors unmittelbar nach dem Buttonklick.

Tabelle 4.3: Die Tabelle listet DOM-Modifikationen die Quill bei Aktivierung von Fettschrift für einen zuvor ausgewählten Textabschnitt vornimmt. Der markierte Textabschnitt überstreckt einen Zeilenumbruch. Eine detailierte Beschreibung des Inhalts der einzelnen Spalten ist dem Fließtext zu entnehmen.

	Typ	Kn.	VKn.	NKn.	EKn.	Veränderung
1	T	#text			P	[Hallo, dies»Hallo,]
2	N	P	#text		DIV	+[#text(dies)<STRONG]
3	N	P	#text		DIV	+[STRONG(dies)<P]
4	N	P	#text	STRONG	DIV	-[#text(dies)<STRONG]
5	N	STRONG			P	+[#text(dies)<STRONG]
6	T	#text			STRONG	[ist ein Test!»ist]
7	N	P	#text		DIV	+[#text(ein Test!)<P]
8	N	P	#text	#text	DIV	+[STRONG(ist)<P]
9	N	P		STRONG	DIV	-[#text(ist)<STRONG]
10	N	STRONG			P	+[#text(ist)<STRONG]

Die insgesamt 10 DOM-Modifikationen wurden durch eine einzige Editieroperation, den Buttonklick des Benutzers, ausgelöst und daher kommen die jeweiligen Callback-Funktionen, wie oben bereits näher erläutert, erst nach Beendigung der letzten DOM-Modifikation zur Ausführung. Daher wird in dieser Tabelle gegenüber der Darstellung in Tabelle 4.2 auf die Wiedergabe des gesamten Dokumentinhalts (letzte Spalte) verzichtet. Der, zur Laufzeit der Callback-Funktionen ermittelte Inhalt war in allen Fällen

```
<p>Hallo, <strong>dies</strong></p>
<p><strong>ist</strong> ein Test!</p>
```

was eben die Struktur des Dokuments nach Umsetzung aller DOM-Modifikationen wiedergibt. Konkret wurden von Quill die folgenden Schritte durchgeführt:

1. Die erste DOM-Modifikation verkürzte den Inhalt des Textknotens, der der ersten Zeile entsprach, von „Hallo, dies“ zu „Hallo, “. Eine Zeile wird in Quill durch einen <P>-Element abgebildet.
2. Anschließend wurde das zuvor entfernte Textfragment als neuer Textknoten an gleicher Position wieder eingefügt. Insgesamt wurde also mit den ersten beiden DOM-Modifikationen der Textinhalt der ersten Zeile auf zwei Textknoten aufgespalten.
3. Danach wurde unter das <P>-Element ein leeres <STRONG>-Element eingefügt, welches später den in Fettschrift zu rendernden Text der ersten Zeile

aufnehmen sollte. Aus Tabelle 4.3 geht hervor, dass das Element nach einem Textknoten (Spalte *VKn.*) am Ende der ersten Zeile eingefügt wurde (da kein *NKn.* vorhanden war).

4. In der vierten DOM-Modifikation wurde nun der im zweiten Schritt eingefügte Textknoten wieder entfernt, um

5. in der fünften DOM-Modifikation unter das im dritten Schritt hinzugefügte <STRONG>-Element verschoben zu werden. An dieser Stelle waren die Änderungen, die die erste Zeile betrafen, abgeschlossen.

6. In diesem Schritt wurde der Inhalt des Textknotens, der der zweiten Zeile entsprach, von „ist ein Test!“ zu „ist“ verkürzt.

7. Danach wurde der im vorigen Schritt entfernte Text als neuer Textknoten an gleicher Position wieder eingefügt. Wie in den ersten beiden Schritten wurde also in den vergangenen beiden Schritten der Textinhalt der zweiten Zeile auf zwei Textknoten aufgespalten.

8. Anschließend wird unterhalb des <P>-Elements der zweiten Zeile ein leeres <STRONG>-Element nach dem ersten Textknoten der Zeile (Spalte *VKn.*) und vor dem zweiten (Spalte *NKn.*) eingefügt.

9. In der neunten DOM-Modifikation wurde nun der im sechsten Schritt verkürzte Textknoten entfernt, um

10. in der letzten DOM-Modifikation unter das im achten Schritt hinzugefügte <STRONG>-Element verschoben zu werden. An dieser Stelle waren die Änderungen, die die zweite Zeile betrafen, abgeschlossen.

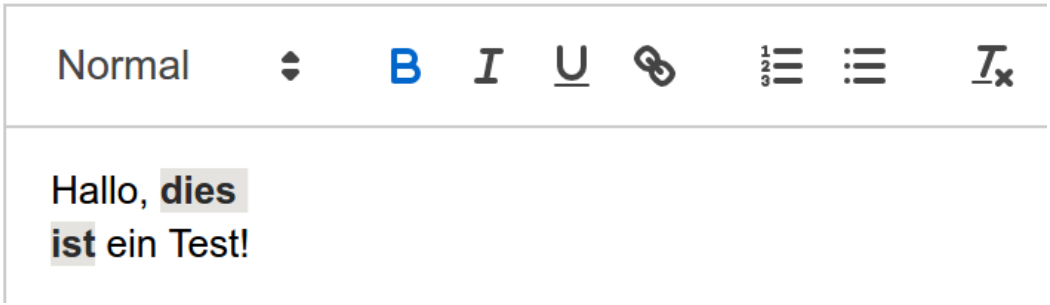


Abbildung 4.4: Die Abbildung zeigt einen Screenshot des Quill Editors nachdem der grau-hinterlegte, selektierte Textinhalt durch einen Klick auf den entsprechenden Button in Fettschrift gesetzt wurde. Die mit dieser atomaren Benutzeraktion verbundenen DOM-Modifikationen sind in Tabelle 4.3 aufgelistet.

Das Beispiel zeigt, dass Quill – gewiss im Sinne einer einfacheren Programmlogik – komplexe Textänderungen aus vielen einzelnen DOM-Modifikationen synthetisiert

und nicht in Bezug auf die Anzahl der DOM-Modifikationen optimiert. Man hätte im vorliegenden Beispiel das gleiche Endergebnis erzielt, wenn zuerst die beiden <STRONG>-Elemente erzeugt und erst danach die jeweiligen Textknoten gespalten und die dann erhaltenen Fragmente direkt unter die <STRONG>-Elemente eingefügt worden wären. Dies hätte gegebenenfalls drei DOM-Operationen „eingespart“, wie im nächsten Absatz gezeigt werden wird.

4.2.1.3 Netto-Bilanz komplexer Editieroperationen

Das vorige Beispiel wurde nun unter Verwendung der `MutationSummary`-Bibliothek wiederholt. Tabelle 4.5 listet die DOM-Modifikationen auf, die für die gleiche Benutzeraktivität erhalten wurden. Im Gegensatz zu der Darstellung in Tabelle 4.3 ist es hier nicht möglich, wie auch wenig sinnvoll, die DOM-Modifikationen chronologisch zu ordnen. Das `MutationSummary`-Objekt, welches der Callback-Funktion zur Laufzeit übergeben wird enthält jeweils ein Array für neu hinzugefügte und entfernte DOM-Knoten. Ausserdem enthält es ein Array für alle DOM-Knoten, deren Reihenfolge innerhalb ihres Elternknotens verändert wurde, und ein Array für DOM-Knoten, die innerhalb des DOM zu einem anderen Elternknoten verschoben wurden. Schließlich ist ein Array für Textknoten deren Textinhalt verändert wurde, enthalten. Diese Arrays enthalten lediglich die relevanten Knoten. Wenn also, wie in Tabelle 4.3, ein Textknoten zunächst an einer Stelle des DOM-Baums eingehängt wird (Schritt 2) und derselbe Textknoten in einem späteren Schritt wieder aus dem DOM entfernt und an einer anderen Stelle wieder eingehängt wird, berichtet `MutationSummary` nicht von den ersten beiden, redundanten DOM-Operationen.

Entsprechend listet Tabelle 4.5 nur noch 7 DOM-Modifikationen, während in Tabelle 4.3 noch insgesamt 10 DOM-Modifikation berichtet worden waren. Intern hat `MutationSummary` die gleichen DOM-Modifikationen wie in Tabelle 4.3 aufgezeichnet und daraus dann eine „Netto-Bilanz“ der Änderungen berechnet. Im vorliegenden Beispiel erkennt die Bibliothek, dass die Auswirkungen von 3 DOM-Modifikationen aufgrund nachfolgender DOM-Modifikationen hinfällig geworden sind und verwirft diese daher.

Tabelle 4.5: Die Tabelle listet DOM-Modifikationen die Quill bei Aktivierung von Fettschrift für einen zuvor ausgewählten Textabschnitt vornimmt. Der markierte Textabschnitt überstreckt einen Zeilenumbruch. Eine detailierte Beschreibung des Inhalts der einzelnen Spalten ist dem Fließtext zu entnehmen.

	Typ	Kn.	VKn.	NKn.	EKn.	Veränderung
1	+	#text			STRONG	+[#text(dies)<STRONG]
2	+	STRONG	#text		P	+[STRONG(dies)<P]
3	+	P	#text		DIV	+[#text(ein Test!)<P]
4	+	P	#text	#text	DIV	+[STRONG(ist)<P]
5	RP	#text			STRONG	RP[P»STRONG]
6	T	#text			P	[Hallo, dies»Hallo,]
7	T	#text			STRONG	[ist ein Test!»ist]

4.2.2 Reduktion auf DOM-Fragmente

Nachdem nun alle relevanten DOM-Modifikationen identifiziert sind, ist es naheliegend diese möglichst „Eins–zu–Eins“ in Aufrufe der ShareJS-API zu überführen. An diesem Punkt stellte sich während der Bearbeitung dieser Bachelorarbeit allerdings heraus, dass es *ohne erheblichen Aufwand nicht möglich ist die erforderlichen ShareJS-API Aufrufe korrekt zu generieren.*

4.2.2.1 Problem: Herleitung eines XPath-Ausdruck

Die ShareJS-API verlangt als erstes Argument stets einen XPath-Ausdruck. Wenn dieser Ausdruck generiert werden soll – zur Laufzeit der entsprechenden Callback-Funktion – können weitere, strukturelle DOM-Modifikationen bereits stattgefunden haben. Diese können die DOM-Struktur im Hinblick auf den betrachteten DOM-Knoten so modifiziert haben, dass der XPath-Ausdruck nicht „korrekt“ generiert werden kann.

Ein simples Beispiel ist das Löschen einer Zeile: Hier werden mindestens zwei DOM-Knoten entfernt – ein <P>-Element und (mindestens) ein Textknoten. Zur Laufzeit der Callback-Funktion sind beide Knoten – selbstverständlich – bereits aus dem DOM entfernt. Die Generierung eines XPath-Ausdruck ist für beide Knoten nicht unmittelbar möglich, allerdings besitzen sowohl das `MutationRecord`-Objekt alsauch das `MutationSummary`-Objekt eine Methode um für aus dem DOM entfernte Knoten den letzten Elternknoten zu erfahren. Dies ist aber für den Textknoten von geringem Nutzen, denn die Methode liefert das <P>-Element zurück, das ebenfalls bereits aus dem DOM entfernt. Natürlich kann die Methode erneut aufgerufen werden und dies kann auch in einem simplen Algorithmus generell iterativ oder rekursiv implementiert werden. Aber dieses Beispiel war auch recht simpel – die Situation wird

deutlich komplizierter wenn DOM-Knoten nicht nur entfernt sondern auch verschoben werden. Wenn ein Knoten unterhalb eines zuvor verschobenen Knotens entfernt wurde, müsste der Algorithmus der XPath-Generierung diesen Sachverhalt erkennen und sein Verhalten entsprechend anpassen. Noch komplizierter wird es, wenn Geschwister-Knoten untereinander oder durch Einfügen eines neuen Knotens ihre Position unterhalb des Elternknotens ändern. Kurzum, zur Laufzeit der Callback-Funktion ist es nicht möglich ohne komplexe Programmlogik einen korrekten XPath-Ausdruck für viele DOM-Knoten zu generieren.

Hier soll ausdrücklich betont werden, dass das Problem nicht davon abhängig ist ob die DOM-Modifikationen in chronologischer Reihenfolge bearbeitet werden (bei direkter Verwendung der `MutationObserver`-API) oder in beliebiger Reihenfolge (bei Verwendung der `MutationSummary`-Bibliothek). Das zugrundeliegende Problem ist, dass sich mehrere atomare DOM-Modifikation akkumulieren können bevor es möglich ist in einer Callback-Funktion mit eigener Programmlogik zu darauf reagieren.

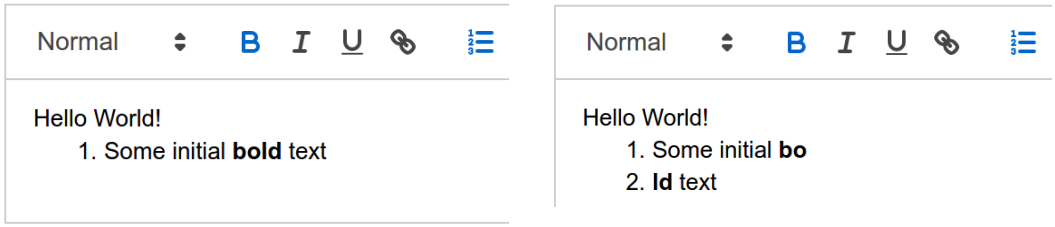
Daher war es erforderlich ein Ansatz zu finden, der die vorgenannten Probleme umgeht und eine einfache Programmlogik implementiert: Für jede strukturelle DOM-Modifikation wird ein übergeordneter DOM-Knoten identifiziert, der selbst keine übergeordneten DOM-Knoten oder Geschwister-DOM-Knoten besitzt, die ebenfalls modifiziert wurden. Das gesamte DOM-Fragment, das dieser DOM-Knoten mitsamt seiner Nachfahren-DOM-Knoten bildet, wird dann aus dem ShareJS-Dokument entfernt und mit dem aktuellen DOM-Fragment aus dem Dokumentcontainer ersetzt.

4.2.2.2 Algorithmus zur Reduktion

Zur Identifikation der zu aktualisierenden DOM-Fragmente wurde Algorithmus 4.6 hergeleitet und implementiert. Der Algorithmus nimmt DOM-Knoten, die mit strukturellen DOM-Modifikationen verbunden sind, als Eingabe entgegen und liefert die Wurzelknoten der zu aktualisierenden DOM-Fragmente zurück.

Algorithmus 4.6 : Reduktion auf relevante DOM-Fragmente

```
/* Array mit allen DOM-Knoten, die strukturelle DOM-Modifikationen
   referenzieren                                                    */
Daten : Array K_mod[K_1 ... K_nMod]
/* Array mit allen Wurzelknoten der zu aktualisierenden DOM-Fragmente
   */
Ergebnis : Array K_rel
1  Beginn
2      K_rel ← [] /* initialisiere leeres Array                     */
       /* initialisiere leeres Array, enthält DOM-Pfade von einem
          modifiziertem Knoten zum Dokumentcontainer                */
3      Pfade ← [];
4      für jedes K in K_mod tue
           /* ermittle den DOM-Pfad ausgehend vom Knoten K zum
              Dokumentcontainer                                     */
5          Pfad^K ← ermittleDOMPfad(K);
6          T^K ← Pfad^K[0] ;          /* unmittelbarer Elternknoten von K */
7          für jedes Pfad in Pfade tue
8              T ← Pfad[0];
9              wenn T in Pfad^K dann
                   /* Abbruch des äußeren Schleifendurchlaufs, fahre mit
                      nächstem Knoten K fort                        */
10             wenn T^K in Pfad dann
                   /* entferne Pfad aus Pfade                       */
           /* der ermittelte DOM-Pfad liegt nicht unterhalb eines bereits
              enthaltenen Pfades                                    */
11         n ← length(Pfade);
12         Pfade[n] ← Pfad^K;
       /* kopiere die unmittelbaren Elternknoten der relevanten Pfade in
          das finale Ausgabearray                                   */
13     für i ← 0 bis length(Pfade) − 1 tue
14         K_rel[i] ← Pfade[i][0];
```



(a) Vorher **(b)** Nachher

Abbildung 4.7: Die Abbildung zeigt den Editor unmittelbar bevor (a) und unmittelbar nachdem (b) ein Zeilenumbruch innerhalb des ersten Listenpunkts eingefügt worden war. Die damit verbundenen DOM-Modifikationen sind in Abbildung 4.8 dargestellt.

4.2.2.3 Beispiel Zeilenumbruch

Ein praktisches Beispiel für die Anwendung von Algorithmus 4.6 ist in Abbildung 4.8 dargestellt. Die Grafik dokumentiert die DOM-Modifikationen die Quill bei Einfügen eines Zeilenumbruchs in eine Auflistung vornimmt (siehe Abbildung 4.7). Die rechte Seite der Grafik zeigt die DOM-Struktur nach Aktualisierung durch Quill. Der Algorithmus findet zunächst alle Elternknoten der modifizierten DOM-Knoten. Im Anschluss werden aus dieser Gruppe Elternknoten gefiltert, die selbst übergeordnete DOM-Knoten als Bestandteil dieser Gruppe enthalten. Die übrig gebliebenen DOM-Knoten werden vom Algorithmus zurückgeliefert und nachfolgend als zusammenhängendes DOM-Fragment über die ShareJS-API aktualisiert. Im betrachteten Beispiel erhält man das <OL>-Element als einzig zu aktualisierendes DOM-Fragment.

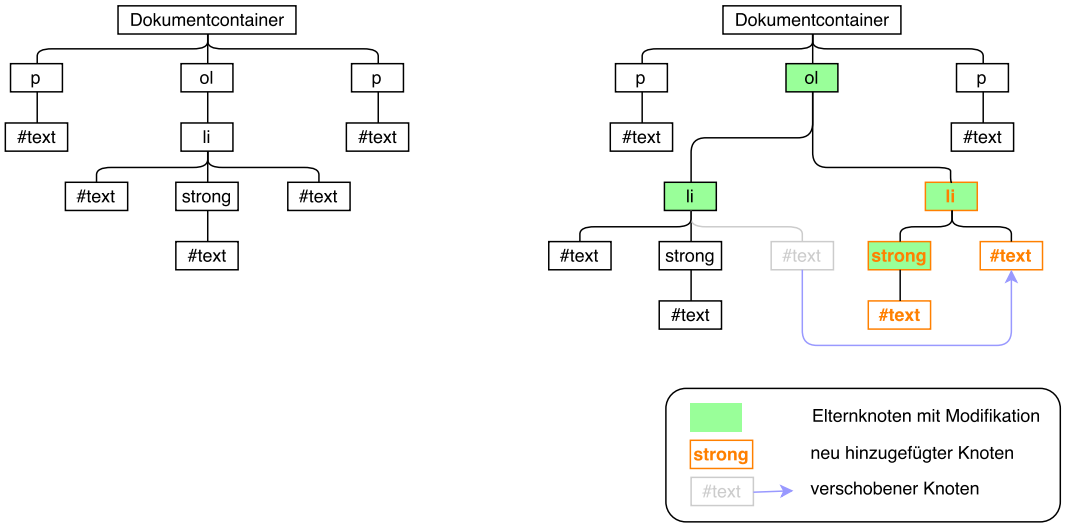


Abbildung 4.8: Die Grafik dokumentiert die DOM-Modifikationen die Quill bei Einfügen eines Zeilenumbruchs in eine Auflistung vornimmt[a]. Die linke Seite der Grafik zeigt das DOM-Fragment des Dokuments unmittelbar vor Einfügen des Zeilenumbruchs. Die rechte Seite der Grafik zeigt das DOM-Fragment unmittelbar danach. Neu eingefügte und innerhalb des DOM-Fragments verschobene DOM-Knoten sind farblich hervorgehoben. Weiterhin sind die von diesen Modifikationen betroffenen, unmittelbaren Elternknoten farblich hervorgehoben. Unter Verwendung von Algorithmus 4.6 werden die DOM-Knoten gefunden, die an die ShareJS-API weiterzureichen sind. Im vorliegenden Beispiel umfasst der -Knoten alle DOM-Modifikationen und wird daher als einziger, zu aktualisierender Knoten an die ShareJS-API weitergereicht.

[a]In der Tat sind hier nur die wichtigsten DOM-Modifikationen hervorgehoben. Welche Änderungen Quill am DOM vornimmt hängt auch vom Verlauf einer Editiersitzung, welcher Einfluß auf interne Cache-Strukturen hat, ab.

4.2.3 Aktualisierung des ShareJS-Dokuments

Nachdem die zu aktualisierenden DOM-Knoten erhalten wurden, zeigt dieser Abschnitt, wie die einzelnen DOM-Modifikationen in Anweisungen umzuwandeln sind, die die ShareJS-API entgegennimmt. Zunächst sollen einige Hilfsfunktionen vorgestellt werden, die implementiert wurden um die benötigten Argumente für die Verwendung der ShareJS-API zu erhalten.

4.2.3.1 Hilfsfunktionen

Die ShareJS-API erwartet zur Identifizierung des zu modifizierenden Elements stets als erstes Argument einen XPath-Ausdruck. Einerseits muss dieser XPath-Ausdruck

einen entsprechenden DOM-Knoten eindeutig identifizieren, andererseits lassen sich stets mehrere Ausdücke finden, die diese Bedingung erfüllen. Da Webbrowser selbst keine API zur Abfrage eines XPath-Ausdruck für einen DOM-Knoten bieten wurde eine entsprechende Funktion im Rahmen dieser Arbeit implementiert. Diese Funktion ist in Quellcode 4.9 gelistet. Die Funktion `getElementXPath` generiert für einen als erstes Argument übergebenen DOM-Knoten einen XPath-Ausdruck der Form:

`\Vorfahrenknoten[Index]\...\Elternknoten[Index]\Knoten[Index]`.

Die Funktion prüft in den Zeilen 3-5, dass der übergebene Startknoten ein Elementknoten ist. Als optionales zweites Argument übernimmt die Funktion einen Verweis auf einen weiteren DOM-Knoten. Dieser signalisiert, dass der XPath-Ausdruck *relativ* zu diesem DOM-Knoten abgeleitet werden soll. Dies ist erforderlich, da die Funktion sonst innerhalb des DOM bis zum Wurzelknoten hinaufsteigt und einen entsprechenden Ausdruck erzeugt. In der praktischen Verwendung wird der Funktion daher als zweites Argument stets der Dokumentcontainer übergeben.

```
1  function getElementXPath(elt, root = document.body) {
2      let path = '';
3      if (elt.nodeType !== 1) {
4          return path;
5      }
6      for (; elt; elt = elt.parentNode) {
7          let xname;
8          let pos = getSiblingPos(elt);
9          xname = elt.tagName + '[' + pos + ']';
10         path = '/' + xname + path;
11     }
12     return path;
13 }
```

Quellcode 4.9: Die Funktion `getElementXPath` generiert für einen als erstes Argument übergebenen DOM-Knoten einen eindeutigen XPath-Ausdruck.

Weiterhin erwartet die API bei Einfügen eines DOM-Fragments die Einfügeposition innerhalb der Gruppe der Geschwisterknoten. Die Funktion `getSiblingPos` (Quellcode 4.10) liefert für einen als erstes Argument übergebenen DOM-Knoten dessen eindeutige Position innerhalb der Gruppe seiner Geschwisterknoten.

```
1  function getSiblingPos(elt) {
2      let pos = 1;
3      for (let sib = elt.previousSibling; sib; sib = sib.previousSibling) {
4          if (sib.nodeType === 1) pos++
5      }
6      return pos;
7  }
```

Quellcode 4.10: Die Funktion `getSiblingPos` liefert für einen als erstes Argument übergebenen DOM-Knoten dessen eindeutige Position innerhalb der Gruppe seiner Geschwisterknoten.

Ausserdem erwartet die API bei Einfügen eines neuen DOM-Fragments, dieses als serialisierten String. Für ein einzufügendes DOM-Fragment ist ein solcher String unmittelbar in der `outerHTML`-Eigenschaft des betreffenden Wurzelknotens des DOM-Fragments verfügbar.

4.2.3.2 Aufruf der ShareJS-API

Zur Aktualisierung des Dokuments in ShareJS wird folgende Strategie verfolgt:

1. Zunächst werden die strukturellen DOM-Modifikationen vorgenommen. Die im vorigen Abschnitt anhand ihrer Wurzelknoten identifizierten DOM-Fragmente werden zunächst aus dem ShareJS-Dokument entfernt und anschließend mit dem aktuellen Inhalt aus der Quill-Repräsentation des Dokuments wieder eingefügt. Zur Entfernung des DOM-Fragments wird die Funktion `removeElement(XPath, cb)` der ShareJS-API verwendet. Der benötigte XPath-Ausdruck wird mit der Funktion `getElementXPath` generiert. Wie in Unterabschnitt 4.2.2 ausführlich geschildert, werden nach Anwendung von Algorithmus 4.6 nur DOM-Knoten erhalten, deren Anordnung innerhalb des DOM unverändert ist und für die daher die Ableitung eines XPath-Ausdrucks im Kontext der ShareJS-Repräsentation des Dokuments unproblematisch ist. Anschließend wird der aktuelle Inhalt des DOM-Fragment in der Quill-Repräsentation aus der Eigenschaft `outerHTML` des betrachteten DOM-Knotens als serialisierter String ausgelesen. Dieser String wird der Funktion `insertElementAt(XPathToParent, pos, value, cb)` der ShareJS-API als drittes Argument übergeben, um das DOM-Fragment wieder einzusetzen. Die Ermittlung eines XPath-Ausdrucks für den Elternknoten (erstes Argument) und der Position innerhalb der Geschwisterknoten (zweites Argument) ist hier ebenfalls unproblematisch.

2. Nachdem alle strukturellen DOM-Modifikationen vorgenommen wurden, entspricht die DOM-Struktur der ShareJS-Repräsentation der DOM-Struktur der Quill-Repräsentation. Lediglich die von `MutationSummary` berichteten Text-Modifikationen müssen noch „eingearbeitet“ werden. Dazu wird die Funktion `replaceTextAt(XPath, pos, value, cb)` verwendet. Diese erhält als erstes Argument einen XPath-Ausdruck, der den entsprechenden Textknoten identifiziert. Dazu wird ein XPath-Ausdruck für den Elternknoten mit `getElementXPath` abgeleitet und um den String `/text()` erweitert. Als zweites Argument wird stets 0 übergeben (`pos = 0`), um zu signalisieren, dass der gesamte Inhalt des Textknotens ersetzt werden soll. Als drittes Argument wird der aktualisierte Inhalt übergeben, der aus der Quill-Repräsentation des Textknotens aus der Eigenschaft `textContent` ausgelesen wird.

4.3 Verarbeitung entfernter Editieroperationen

Zur Umsetzung der zweiten Teilaufgabe ist es erforderlich auf alle entfernten Editieroperationen zu reagieren, nachdem diese von ShareJS verarbeitet wurden und das DOM unterhalb des Dokumentcontainers entsprechend zu aktualisieren. Zunächst wird erklärt, wie vermieden werden kann, dass die von entfernten Editieroperationen ausgelösten DOM-Modifikationen unterhalb des Dokumentcontainers zur Ausführung der Callback-Funktionen führen, die zur Bearbeitung lokaler Editieroperationen registriert sind.

4.3.1 Problem: Verwechslung mit lokalen Editieroperationen

Die `MutationSummary`-Bibliothek bietet eine praktikable Lösung für dieses Problem. Bei erstmaligem Aufruf und Registrierung der eigenen Callback-Funktion liefert `MutationSummary` ein Objekt zurück, das im Nachhinein verwendet werden kann, um, unter anderem, die Überwachung des DOM zu unterbrechen oder nach Unterbrechung wider aufzunehmen. Bevor eigener Code entfernte Editieroperationen an Quill „übergibt“ wird auf diesem Objekt die Methode `disconnect` aufgerufen. Diese stoppt die Überwachung des DOM unmittelbar, sodass Modifikationen am DOM unterhalb des Dokumentcontainers vorgenommen werden können, ohne dass die registrierte Callback-Funktion später zur Ausführung kommt. Nachdem die entfernte Editieroperation in die Quill-Repräsentation eingearbeitet worden ist, wird die Methode `reconnect` aufgerufen, welche die Überwachung des DOM fortsetzt. Aufgrund der „single-threaded“-Natur der JavaScript-Laufzeitumgebung kann es während dieses Zeitraums nicht zu überlappenden DOM-Modifikationen durch Quill kommen.

4.3.2 Registrierung von DOM-Modifikationen in ShareJS

ShareJS bietet eine API zur Registrierung von Callback-Funktionen bei Modifikationen des internen DOM. Allerdings ist die API bei weitem nicht so leistungsfähig und angenehm in der Verwendung wie die vergleichbare API die `MutationObserver` oder `MutationSummary` bieten. In der praktischen Anwendung war es erforderlich, die Callback-Funktionen mit den zu überwachenden DOM-Knoten zu registrieren. Die „Verwaltung“ dieser Callback-Funktionen hätte einen viel zu grossen Aufwand bedeutet, als dass dieser im Rahmen der Bachelorarbeit hätte umgesetzt werden können.

In dieser Bachelorarbeit wurde folgender Ansatz gewählt: ShareJS löst ein Ereignis aus, nachdem eine entfernte Editieroperationen, gegebenenfalls nach Anwendung von OT, in das interne DOM übernommen wurde. Für dieses Ereignis wurde eine Callback-Funktion registriert. Diese schaltet zunächst die Überwachung des DOM

durch `MutationSummary` aus und übergibt danach den gesamten Inhalt des internen DOM als serialisierten String an die API von Quill. Anschließend wird die Überwachung des DOM durch `MutationSummary` wieder aktiviert.

4.3.3 Verwendung der Quill-API zur Aktualisierung des DOM

Im vorigen Abschnitt wurde erklärt, dass der gesamte Inhalt des ShareJS-internen DOM an die Quill-API übergeben wird. Diese stellt eine entsprechende Funktion zum Austausch des gesamten Inhalts des Dokumentcontainers bereit. Die Funktion ist im Untermodul `Clipboard` von Quill enthalten und trägt den bezeichnenden Namen `dangerouslyPastHTML`. Der Name der Funktion soll den unbedarften Entwickler auf das mögliche Einführen einer *Cross-Site-Scripting*-Schwachstelle aufmerksam machen. Quellcode 4.11 zeigt ein Beispiel für die Verwendung der Funktion.

```
1 quill.clipboard.dangerouslyPasteHTML('<p>Hello <b>World</b>!</p>');
```

Quellcode 4.11: Das Quellcode-Fragment zeigt die Verwendung der Funktion dangerouslyPasteHTML.

5 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde der *open-source* Rich-Text Editor Quill in die Anwendung SECRET integriert.

Dazu wurde in Kapitel 2 zunächst eine umfassende Recherche bestehender *open-source* Editoren auf GitHub durchgeführt. In deren Folge wurde nach sorgfältiger Abwägung der definierten Kriterien der Editor *Quill* aufgrund seiner umfangreichen Dokumentation und seiner angenehm zu lesenden Codebasis ausgewählt.

Anschließend wurden in Kapitel 3 die verwendeten Softwarewerkzeuge und -technologien vorgestellt. Zum Zusammenbau der Anwendung wurde eine neue Konfiguration unter Verwendung eines modernen Softwarewerkzeugs implementiert. Diese Konfiguration leistet nun die Kompilierung der unterschiedlichen JavaScript-Dialekte, die im Projekt verwendet werden. Dies geschieht automatisch im Hintergrund bei Änderungen des Quellcodes durch den Entwickler. Hervorzuheben ist, dass mit der geschilderten Konfiguration das gleichzeitige Debuggen des Server-Codes und des Client-Codes ermöglicht worden ist, und dass dies im Kontext des jeweiligen JavaScript-Dialektes geschieht.

In Kapitel 4 wurde die zu implementierende Anbindungslogik konzipiert und deren Implementierung dargelegt. Die Anbindungslogik wurde in die Verarbeitung *lokaler* und *entfernter* Editieroperationen unterteilt. Während der Implementierung der Anbindungslogik für lokale Editieroperationen tauchte ein Problem auf, das letztlich auf eine fundamentale Limitierung innerhalb der JavaScript-Laufzeitumgebung zurückzuführen ist. Um dieses Problem zu umgehen, wurde ein Algorithmus abgeleitet und implementiert, der häufig grössere DOM-Fragmente aktualisiert als ohne besagtes Problem nötig gewesen wäre.

Für die Implementierung der Anbindungslogik für entfernte Editieroperationen wurde ein Ansatz gewählt, der den gesamten Inhalt des Dokuments in Quill entfernt und mit dem aktualisierten Inhalt des Dokuments aus ShareJS ersetzt. Eine „elegantere“ Implementierung gelang trotz reichlicher Bemühungen nicht.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] D. Felsch, C. Mainka, V. Mladenov, and J. Schwenk. SECRET: On the Feasibility of a Secure, Efficient, and Collaborative Real-Time Web Editor. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 835–848. ACM, 2017.

[2] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD*, 18:399–407, 1989.

[3] ShareJS. Online. URL `https://github.com/josephg/ShareJS`.

[4] Quill. Online. URL `https://quilljs.com/`.

[5] Parchment. Online. URL `https://github.com/quilljs/parchment`.

[6] Delta. Online. URL `https://github.com/quilljs/delta`.

[7] E. Swenson-Healey. The JavaScript Event Loop: Explained. Online, 10 2013. URL `http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/`.

[8] J. Resig, B. Bibeault, and J. Maras. *Secrets of the JavaScript Ninja*, chapter 13. Manning Publications Co., .

[9] Webpack. Online. URL `https://webpack.js.org`.

[10] WHATWG. DOM – Mutation Observers. Online, 05 2016. URL `https://dom.spec.whatwg.org/#mutation-observers`.

[11] Mutation Summary. Online. URL `https://github.com/rafaelw/mutation-summary`.

[12] J. Resig, B. Bibeault, and J. Maras. *Secrets of the JavaScript Ninja*, chapter 5. Manning Publications Co., .