Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MARK RUDTKE

MASTER THESIS

REALIZATION OF A FINE-GRAINED DOCUMENT HISTORY OVERLAY

Eingereicht am 22. Juli 2019

Betreuer: Dipl.-Inf. Hannes Dohrn, Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 22. Juli 2019

License

_____

Erlangen, 22. Juli 2019

Abstract

Wikis enable the maintenance of a collective knowledge base on the Internet and although they already have some prerequisites for collaborative work, they lack essential functions for the productive handling of shared articles.

The web-based collaboration platform Sweble uses the so-called Wiki Object Model as an alternative to common markup languages and allows a graph based storage of documents. Thus Sweble provides the potential to visualize the editing process of documents and to raise collective knowledge management to a new level.

The goal of this thesis is therefore to extend the Sweble software in a way that history information of a document can be processed and displayed in a user-friendly way. The user interface should offer users intuitive and clear access to additional information and statistics.

Zusammenfassung

Wikis ermöglichen die Pflege einer kollektiven Wissensbasis im Internet und obwohl sie bereits einige Voraussetzungen für kollaboratives Arbeiten mitbringen, fehlt es ihnen an wesentlichen Funktionen für den produktiven Umgang mit gemeinsamen Artikeln.

Die webbasierte Kollaborationsplattform Sweble nutzt das sogenannte Wiki Object Model als Alternative zu gängigen Auszeichnungssprachen und ermöglicht eine graphbasierte Speicherung von Dokumenten. Damit liefert Sweble das Potential den Bearbeitungsverlauf von Dokumenten zu visualisieren und kollektives Wissensmanagement auf eine neue Ebene zu heben.

Ziel dieser Arbeit ist es deshalb, die Sweble-Software dahingehend zu erweitern, dass Informationen zur Historie eines Dokuments verarbeitet und in benutzerfreundlicher Weise dargestellt werden können. Die Benutzeroberfläche soll Nutzern so einen intuitiven und übersichtlichen Zugang zu zusätzlichen Informationen und Statistiken bieten.

Inhaltsverzeichnis

1 Einführung

Die moderne Welt des 21. Jahrhunderts ist von einem stetigen Zuwachs an Daten und Informationen geprägt, der nicht zuletzt dem Internet und dem World Wide Web geschuldet ist. Beliebige Informationen können dort unter anderem in unzähligen Artikeln zur Verfügung gestellt werden. Spätestens mit dem Web 2.0 ist das Internet kein einseitiger Informationskanal mehr sondern ermöglicht vor allem eine gemeinschaftliche Nutzung mit unzähligen Kollaborationselementen.

Mithilfe sogenannter Wikis können Informationen bequem bereitgestellt und verwaltet werden. Die populärste Website, die Nutzern auf diese Weise Millionen Wissensartikel zur Verfügung stellt, ist Wikipedia[1]. Seit 2001 kann man sich hier nicht nur über beliebige Dinge informieren sondern auch eigene Änderungen in bereits bestehenden Artikeln anfügen oder gänzlich neue erschaffen. Obwohl sich Wikis stetig weiterentwickelt haben, fehlt es ihnen an einigen grundlegenden Funktionen, die nicht zuletzt auch das kollaborative Element erschweren oder zumindest nicht optimal unterstützen.

In Wikipedia beispielsweise, werden sämtliche Artikel in Wikitext verfasst, einer vereinfachten Auszeichnugssprache zur Formatierung von Beiträgen. Da es keinen allgemein akzeptierten Standard hierfür gibt, variiert dabei die Syntax, sodass es letztlich zur Existenz unterschiedlichster Arten dieser Auszeichnungssprachen kommt. Nach Dohrn und Riehle (2013) birgt die Verwendung von Wikitext viele Nachteile. Das Fehlen der Möglichkeit zur Abstraktion der Inhalte, der automatisierten Inhaltstransformation oder der Möglichkeit zur visuellen Darstellung der Bearbeitung konkreter Artikel sind nur einige Beispiele. Dohrn und Riehle haben deshalb eine Alternative entwickelt. Mithilfe des *Wiki Object Model* (WOM) können Inhalte innerhalb einer wiki-unabhängigen Baumstruktur dargestellt und gespeichert werden.

[1] https://www.wikipedia.org/

Die webbasierte Kollaborationsplattform Sweble ist Wikipedia in ihrer Funktion und Zielsetzung in vielerlei Hinsicht ähnlich. Die Möglichkeit Inhalte nicht in der textuellen Form ihrer jeweiligen Auszeichnungssprache sondern in einer standardisierten Baumstruktur speichern zu können, ist eine ihrer wesentlichen Funktionen und dient dieser Arbeit als Ausgangspunkt. Im weiteren Verlauf soll die Sweble Plattform um ein *Document History Overlay* erweitert werden, durch das Nutzer produktiver mit der Historie eines Dokuments arbeiten können.

Diese Arbeit ist in sechs Kapitel untergliedert. In Kapitel 2 wird die Zielsetzung genauer definiert und zugehörige Anforderungen werden erarbeitet. Kapitel 3 gibt einen kurzen Überblick über relevante Grundlagen, auf denen sich im Anschluss weitere Überlegungen hinsichtlich der Architektur stützen. Ferner wird das erarbeitete Design anhand eines Prototypen exemplarisch beschrieben. Kapitel 4 beschreibt nachfolgend zugehörige Implementierung. Schließlich werden die Ergebnisse in Kapitel 5 evaluiert und in Kapitel 6 zusammengefasst.

2 Anforderungen

In diesem Kapitel werden Ziele und Anforderungen an das *Document History Overlay* definiert. Dabei wird zwischen funktionalen und nichtfunktionale Anforderungen unterschieden. Der jeweilig aufgelisteten Beschreibung folgt grundsätzlich ein Abnahmekriterium, welches später genutzt wird, um zu entscheiden, ob die Anforderung erfüllt wurde.

2.1 Zielsetzung und Absicht

Das Ziel dieser Arbeit ist es, das Potential der graphbasierten Speicherung zu erforschen und zu nutzen, um den Bearbeitungsverlauf von Dokumenten in einer produktiven Weise visuell darzustellen und damit kollaboratives Wissensmanagement auf eine neue Ebene zu heben. Zu diesem Zweck soll die Sweble-Software so erweitert werden, dass das Backend Informationen über die Historie des Dokuments in einem geeigneten Format zur Verfügung stellen kann. Diese Informationen sollen vom Frontend genutzt werden können, um eine benutzerfreundliche Darstellung und einen produktiven Umgang mit diesen zu ermöglichen. Dadurch soll ebenso ein Mehrwert für den Nutzer der Software generiert werden.

2.2 Funktionale Anforderungen

Die Historie eines Dokuments kann visualisiert weden.

Allen Autoren eines Artikels können Farben zugewiesen werden. Die zuletzt bearbeiteten Textstellen des jeweiligen Autors werden mit der jeweiligen Farbe versehen.

Das Überfahren oder Klicken von Text eines Artikels gibt zusätzliche Informationen über die Historie der entsprechenden Stelle.

Das Klicken eines Textelements öffnet ein Fenster, in dem Commit-Informationen bezüglich der Textstelle bereitgestellt werden. Diese enthalten mindestens Commit-Nachricht und -Zeitpunkt.

Zusätzliche Statistiken können angezeigt werden.

Eine Heatmap kann aktiviert werden, um die Änderungshäufigkeit von Textbereichen des Artikels zu visualisieren. Textstellen mit einer höheren Commit-Rate werden dementsprechend auch „heißer“ dargestellt.

Die Informationen über die Historie erlauben dem Nutzer eine schnelle Navigation zu anderen relevanten Informationen.

Relevante Informationen in diesem Kontext entsprechen zum einen der Profilseite eines Autors und zum anderen der Commit-Seite eines entsprechenden Textbereichs oder Commits. Zu diesen Seiten existieren Links, die in unmittelbarer Nähe von Autoren- beziehungsweise Commit-Informationen platziert sind.

2.3 Nichtfunktionale Anforderungen

Das Aussehen und die Handhabung aller Funktionalitäten ist einheitlich gestaltet.

Die Benutzerschnittstelle ist responsiv entwickelt und auch geeignet für mobile Endgeräte. Die Anordnung aller Elemente ist daher stimmig. Eine klare Abgrenzung von Inhalten ist gegeben.

Das Design aller Funktionalitäten ist für jede Art von Nutzer gut verständlich und bedienbar.

Die Verständlichkeit ist für Nicht-Experten durch ein umfangreicheres UX-Design gewährleistet. Die Bedienbarkeit ist für Beeinträchtigte durch eine barrierefreie Gestaltung gewährleistet.

Eine Anpassung am Design kann leicht umgesetzt werden.

Der Stil der HTML-Elemente der Benutzerschnittstelle ist leicht änderbar uns so an neue Gegebenheiten anpassbar.

3 Architektur und Design

Gegenstand dieses Kapitels ist die Herleitung von Architektur und Design der zu entwickelnden Erweiterungen der Sweble-Software. Zuerst wird ein Überblick über das zugrunde liegende *Wiki Object Model* gegeben. Anschließend werden im Kontext von Backend und Frontend notwendige Schritte erörtert, die die Grundlage für die Realisierung der genannten Anforderungen bildet. Die detaillierte Darlegung und Umsetzung dieser Überlegungen wird im nachfolgenden Kapitel behandelt. Abschließend wird auf das notwendige Design eingegangen, indem die Konzeption der Benutzerschnittstelle anhand eines Prototypen erläutert wird.

3.1 Wiki Object Model

Das *Wiki Object Model* (WOM) wurde von Dohrn und Riehle (2011) entwickelt und bildet die Grundlage für nachfolgende Architektur. Die Sweble-Software nutzt das WOM-Format, um Dokumente zu speichern. Sowohl die Speicherung, als auch die Darstellung kann in einer Baumstruktur, ähnlich dem DOM, abgebildet werden. Der Baum wird aus der Menge der Knoten des Dokuments gebildet, welche sich aus WOM-Knoten und einigen XHTML-Knoten zusammensetzt.

3.2 Backend

Zunächst stellt sich die Frage, woher Informationen über die Historie eines Dokuments kommen und wie diese aussehen müssen. Basis dafür ist zum einen natürlich das Dokument selbst, welches, wie in Abschnitt 3.1 beschrieben, im WOM-Format vorliegt. Zum anderen ist es die graphbasierte Speicherung dieses Formats, bei der grundsätzlich nicht der vollständige Baum gespeichert wird, sondern vielmehr nur ein Teilgraph mit Referenz auf den Vorgänger-Graph, um so Redundanzen minimieren zu können (Knogl, 2016). Wenn bei dieser Speicherung nun Informationen über die aktuelle Version des Graphen anhand eines Commits mitgespeichert werden könnte, so kann nachfolgend auch jeder einzelne Knoten diese erhalten. Commits lassen sich hier eindeutig bestimmen durch eine *Project ID* und eine *Commit ID*. So können auch Informationen über Commit-Nachricht und -Zeitpunkt abgerufen werden. Aufgrund dieser Überlegungen sollten Graphen zusammen mit diesen IDs im Speicher abgelegt werden. Sobald der Graph abgerufen und der zugehöriger *Resource Tree* gebaut wird, sollten hier ebenso beide IDs genutzt werden, um auch dieses Konstrukt mit den geeigneten Informationen zu versehen. Zur Speicherung auf die entsprechenden Knoten eignen sich hier Attribute mit geeignetem Namensraum. Der resultierende *Resource Tree* stellt somit ein geeignetes Format für das Frontend dar.

Das Backend muss daher sowohl die Repräsentation von Knoten in der Datenbank, als auch die Speicherung und Rekonstruierung der Graphen um Parameter für *Project ID* und *Commit ID* erweitern.

3.3 Frontend

Es stellt sich nun die Frage, welche Art von Komponenten benötigt wird um den Anforderungen gerecht zu werden. Die zugrunde liegende Funktione ist das Anzeigen eines Artikels beziehungsweise dessen Textelemente, welche, wie in vorherigem Abschnitt hergeleitet, im WOM-Format vorliegen und entsprechende *Project ID* und *Commit ID* mit sich bringen. Für diesen Zweck kann der *Article Renderer* des Sweble-Hub geeignete

Komponenten zur Visualisierung bereitstellen. Des Weiteren müssen Autoren in den Kontext gebracht und visualisiert werden. Aufgrund dieser Überlegungen würde sich eine Seite anbieten, auf der ein Artikel angezeigt werden kann und auf der sich gleichzeitig Schaltflächen befinden, welche der Darstellung von Autoren dienen. Diese Schaltflächen werden im Folgenden auch Autoren-Buttons oder *Author Buttons* genannt. Die Seite soll als zentrale Steuerung fungieren und wird in folgenden Ausführungen als *Document History Overlay* bezeichnet. Für die Färbung des Artikeltextes eignet sich ein Dekorierer nach dem Architekturmuster von Gamma et al. (1995), welcher hier vor den *Article Renderer* geschaltet werden soll. Schließlich sollte der Dekorierer auch eine Wrapper-Komponente nutzen, um den dekorierten Textelementen so eine gewisse Logik verleihen zu können. Diese Logik soll genutzt werden um Informationen über die Historie des jeweiligen Textelements, und so auch des ganzen Artikels, sinnvoll zu verarbeiten. Hierfür können nun *Project ID* und *Commit ID* aus dem Element gelesen werden.

Das Frontend muss daher um die drei beschriebenen Komponenten erweitert werden. Diese stellen die Grundlage für User Interface dar.

3.4 Document History Overlay UI

Der folgende Abschnitt beschreibt das User Interface des *Document History Overlay*. Zunächst wird sein grundlegender Aufbau und der zugrundeliegende Arbeitsablauf betrachtet. Darüber hinaus werden weitere Funktionalitäten vorgestellt, die kategorisch nicht den Hauptfunktionen zuzuordnen sind aber dennoch eine nützliche und hilfreiche Bereicherung darstellen. Abschließend werden konkrete UX-Design-Entscheidungen diskutiert und begründet.

Die konkrete Umsetzung des *Document History Overlay* UI unter Berücksichtigung aller Anforderungen wird im Folgenden anhand eines Design-Prototypen aufgezeigt und beschrieben. Der dafür erstellte Beispiel-Artikel ist durch den Wikipedia-Artikel „Rainbow Coffee House“[2] inspiriert.

[2]https://de.wikipedia.org/wiki/Rainbow_Coffee_House

3.4.1 Aufbau und Arbeitsablauf

Dazu werden in diesem Kapitel alle Elemente der Benutzerschnittstelle genannt und im jeweiligen Kontext beschrieben. Zur besseren Darstellung werden alle Elemente mit entsprechenden Abbildungen untermalt.

Kontext Authoren

Abbildung 3.1 zeigt alle relevanten Schaltflächen, die für die Farbzuweisung und -änderung der Autoren relevant sind. Oben rechts befinden sich die beiden Buttons „Reassign colors“ und „Reset colors“. Bei Klick auf „Reassign colors“ werden die zugewiesenen Autorenfarben der Reihe nach durchgewechselt. Im unten abgebildeten Beispiel würde der Autor „Mona“ die Farbe Grün zugewiesen bekommen, der Autor „Robert“ wiederum würde blau umgefärbt werden. Der Button „Reset colors“ setzt alle Farben wieder auf die Standardwerte zurück.

Unten links befinden sich die Autoren-Schaltflächen, wobei jeder Autor mithilfe eines eigenen Buttons mit individueller Farbgebung visualisiert wird. Die vom jeweiligen Autor zuletzt hinzugefügte oder angepasste Textstelle wird entsprechend mit derselben Farbe markiert. Fährt man mit der Maus über einen der Autoren-Buttons werden alle dem Autor zugeordneten Textpassagen visuell hervorgehoben.

Abbildung 3.1: Schaltflächen für Farbzuweisung und -änderung

Abbildung 3.2 zeigt das Popup, das erscheint, sobald der Nutzer auf eine der Autoren-Schaltflächen klickt. Das Popup klappt unterhalb des Autors auf. Durch einen kleinen Pfeil im Rahmen wird seine Zugehörigkeit zum jeweiligen Autor dezent aber eindeutig visualisiert. Das Popup besteht im Wesentlichen aus drei separat bedienbaren Elementen:

Im obersten Bereich befindet sich die Schaltfläche „Profile Page“, die als Link zur Profilseite des jeweiligen Autors fungiert. Klickt der Nutzer auf den Button wird er dementsprechend zu den Profilinformationen weitergeleitet.

Der mittlere Bereich zeigt 12 verschiedenfarbige Kreise, die in zwei Reihen mit je sechs Farben angeordnet sind. Alle hier gelisteten Farben entsprechen den Standardfarben, die den Autoren initial zugeordnet werden. Sind bereits alle zwölf Farben vergeben, erhält jeder weitere Autor eine zufällig generierte Farbe. Die Hauptfunktion dieses Bereichs besteht in der Möglichkeit dem entsprechenden Autor nach eigener Präferenz eine andere Farbe zuzuweisen.

Möchte der Nutzer keine der 12 Standardfarben für den ausgewählten Autor verwenden, kann er mithilfe des ausklappbaren Color Pickers darunter eine neue Farbe aus einer umfangreichen Farbpalette wählen. Mit Klick auf die Schaltfläche „Color Picker“ wird die Farbpalette angezeigt und der Nutzer kann die bevorzugte Farbe festlegen.

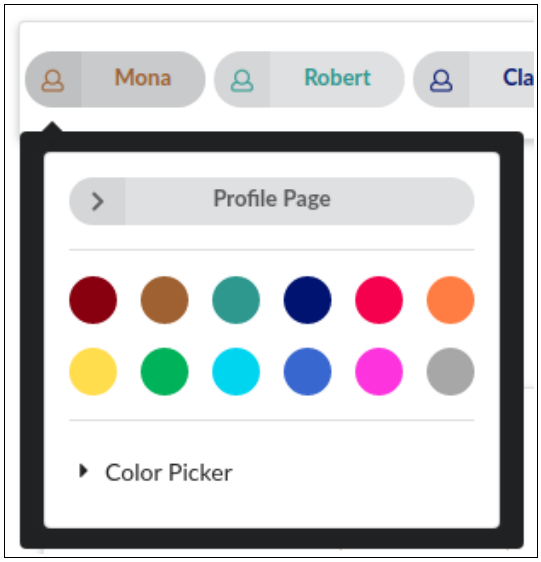

Abbildung 3.2: Popup eines Autoren-Button

Kontext Artikel

Wie in Abbildung 3.3 zu sehen ist, sind oberhalb des Artikels, der sich durch einen feinen Rahmen von den restlichen Seitenelementen abgrenzt, vier verschiedene Grundeinstellungen möglich. Diese werden paarweise visualisiert und grenzen sich zusätzlich durch Ihre Darstellung als Checkboxen beziehungsweise Radiobuttons voneinander ab. Die visuelle Trennung spiegelt dabei die beiden Ebenen ihrer Funktionsweise wieder. Im Folgenden soll diese näher beschrieben werden.

Die linken beiden Einstellungsoptionen, die in Form von Checkboxen implementiert werden, ermöglichen verschiedene visuelle Anpassungen des Artikels. Der Button „Black Background“ färbt den Hintergrund des Artikels schwarz. Die Schaltfläche „Text Borders“ bewirkt, dass die einzelnen, farblich voneinander abgegrenzten Textelemente einen zusätzlichen Rahmen erhalten und somit visuell noch deutlicher voneinander abgegrenzt werden. Beide Einstellungsoptionen repräsentieren damit Hilfsvisualisierungen, die je nach den individuellen Bedürfnissen des Nutzers aktiviert werden können.

Die rechten beiden Einstellungsmöglichkeiten, die als Radiobuttons visualisiert werden, definieren die Art der Information, die dem Nutzer angezeigt werden soll. Je nach Bedarf kann hiermit bestimmt werden, welche Elemente in der Darstellung berücksichtigt werden sollen. Die Schaltfläche „Consider only text history“ bewirkt, dass nur inhaltliche Änderungen und Verlaufsinformationen im Artikel visualisiert werden. Klickt der Nutzer hingegen auf den Button „Consider also formatting history“, werden zusätzlich auch Änderungen in der Formatierung angezeigt. Hierzu ein kurzes Beispiel:

Den weiter unten abgebildeten Artikel „Rainbow Coffee House“ hat der Autor „Mona“ verfasst. Der Autor „Claudia“ hat im Anschluss daran bestimmte Passagen im Text fett markiert. Aktiviert der Nutzer nun die Checkbox „Consider only text history“, wird der Text lediglich in der Farbe des Autors „Mona“ eingefärbt. Klickt der Nutzer wiederum die Option „Consider also formatting history“ an, ändern die nachformatierten Textpassagen ihre Farbe und erscheinen in dem Blauton, der dem Autor „Claudia“ zugeordnet ist.

Betrachtet man die untenstehende Abbildung erneut, kann man neben den vier basalen Einstellungsmöglichkeiten und dem fein gerahmten Artikel, dessen Textbausteine je nach Wahl der Autorenfarben und der oben beschriebenen Einstellungsoptionen eingefärbt sind, noch weitere Elemente erkennen. Rechts am Rahmen des Artikels befindet sich ein farblich hervorgehobener und verdickter Scroll-Balken, der je nach vertikaler Länge Aufschluss über die restliche Textmenge gibt, die sich im Moment der Bildaufnahme außerhalb des Artikelfensters befand.

Ebenfalls rechts, aber außerhalb des Artikel-Segments befinden sich außerdem zwei Schaltflächen, mit deren Hilfe der Text vergrößert beziehungsweise verkleinert werden kann. Der Button zur Vergrößerung des

Textes ist mit einem Plus visualisiert, der zur Verkleinerung mit einem Minus. Oberhalb der beiden Schaltflächen der Zoom-Funktion ist außerdem noch eine Prozentzahl abgebildet, die zusätzlich die momentane Textgröße anzeigt.

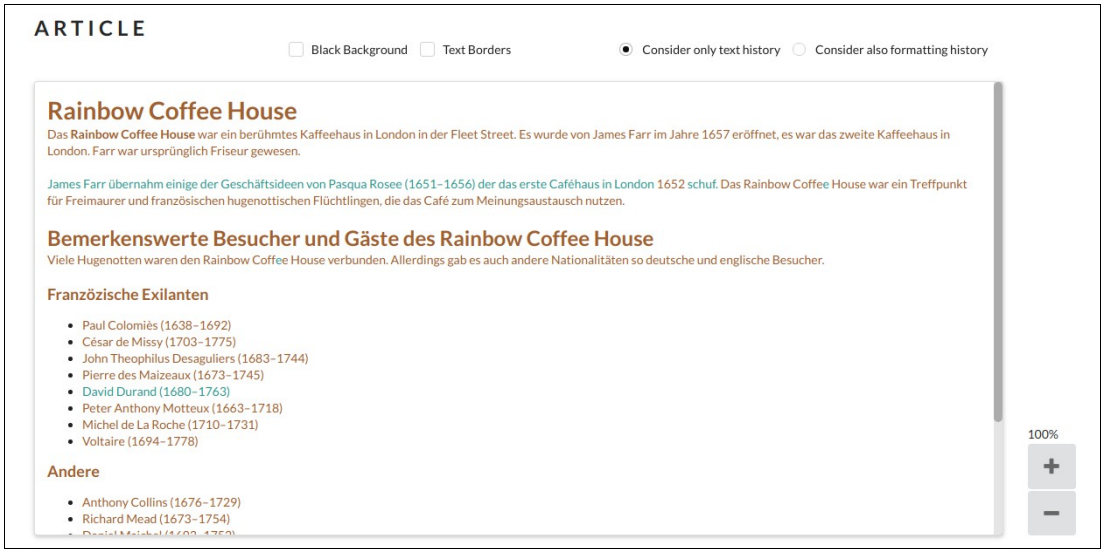

Abbildung 3.3: Artikel-Segment und Einstellungsmöglichkeiten

Die Textelemente selbst bieten weiteren Raum für Interaktion. Fährt man mit der Maus über eine Textstelle, wird zunächst der zugehörige Autoren-Button markiert und hervorgehoben. Klickt der Nutzer letztlich auch auf die Textstelle, öffnet sich, wie in Abbildung 3.4 zu sehen ist, ein weiteres Popup - die Commit-Historie. Auch hier zeigt das Popup durch einen kleinen Pfeil die Zugehörigkeit zur Textstelle an. Zusätzlich wird die angeklickte Textstelle vergrößert und in der jeweiligen Farbe als Basiselement des Popups und stetige Referenz der darüber liegenden Commit-Historie angezeigt.

Die Commit-Historie selbst visualisiert einzelne Einträge durch einen feinen Rahmen und deutet durch ein Trennzeichen zwischen den Einträgen die Verlaufsgeschichte an. Dabei steht der aktuellste, der Textstelle zugehörige Commit immer an unterster Stelle. Um vorherige Commits einsehen zu können, muss der Nutzer in der Historie zurückgehen und dazu nach oben scrollen. Innerhalb eines Eintrags sind folgende Elemente und Informationen enthalten:

Wie nachfolgende Abbildung zeigt, befindet sich links oben erneut der jeweilige Autoren-Button in der ihm zugewiesenen Farbe. Der Nutzer kann

durch Klick auf den Button jederzeit zur Profilseite des Autors navigieren und sich dort individuelle Informationen zum Verfasser einholen.

Den mittleren und größten Bereich füllt die Commit-Nachricht selbst aus. Sie gibt Aufschluss über die jeweilige Handlung, die am Text vorgenommen wurde. Die Commit-Nachricht wird mit einem feinen Rahmen hervorgehoben und von allen anderen Informationen abgegrenzt. Oberhalb des Rahmens wird die Zeitangabe der Nachricht vermerkt. Diese wird sowohl in relativer als auch in absoluter Darstellung angezeigt.

Jeder Eintrag wird durch den unten rechts platzierten Button „Commit Page“ vervollständigt. Wird die Schaltfläche angeklickt, gelangt der Nutzer zur entsprechenden Commit-Seite.

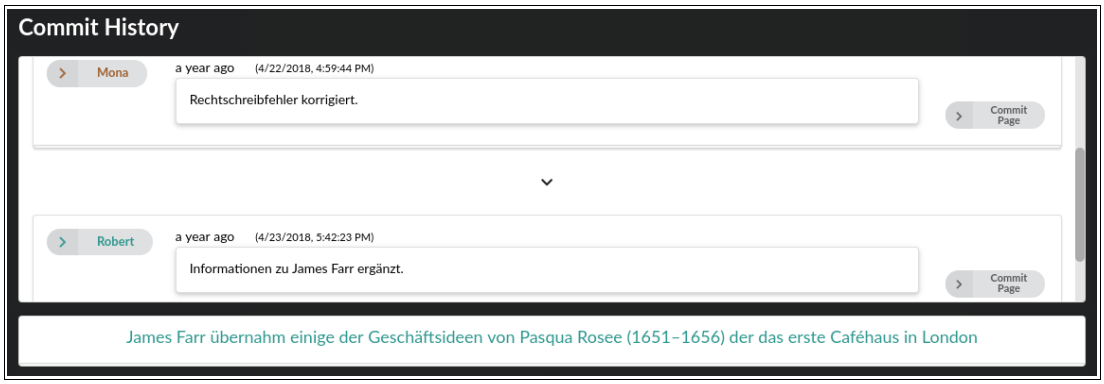

Abbildung 3.4: Popup einer Textstelle

Kontext Heatmap

Abbildung 3.5 zeigt ein weiteres Element des *Document History Overlay* UI - das *Heatmap Overlay*. Die Heatmap dient als zusätzliche Statistik und gibt dem Nutzer Aufschluss über die Änderungshäufigkeit und den Änderungszeitpunkt einer Textpassage. Letzteres ist eine zusätzliche Funktion und wird in Kapitel 3.4.2 noch einmal ausführlicher beschrieben.

Mithilfe eines Schiebereglers im oberen Bereich kann die Heatmap aktiviert werden. Im Grundzustand ist der Regler, sowie die Heatmap selbst, ausgegraut. Aktiviert der Nutzer die Heatmap, wird der Schiebereglers blau eingefärbt und auch die restlichen Bausteine werden entsprechend farbig. Im unten angefügten Bild, sieht man die visuelle Ausgestaltung der aktivierten Heatmap.

Im Wesentlichen gibt es neben dem Regler zwei relevante Bausteine. Innerhalb des Segments befindet sich die Legende der Heatmap, die eine Farbpalette zur Visualisierung der Änderungshäufigkeit bedient. Wie die untenstehende Abbildung zeigt, färbt sich die Heatmap mit steigender Anzahl der Commits dunkler.

Unterhalb der Legende befinden sich zwei weitere Radiobuttons: „Use frequency based mapping“ und „Use time based mapping“. Je nachdem welche Option aktiviert wird, wird eine frequenzbasierte oder zeitbasierte Heatmap angezeigt.

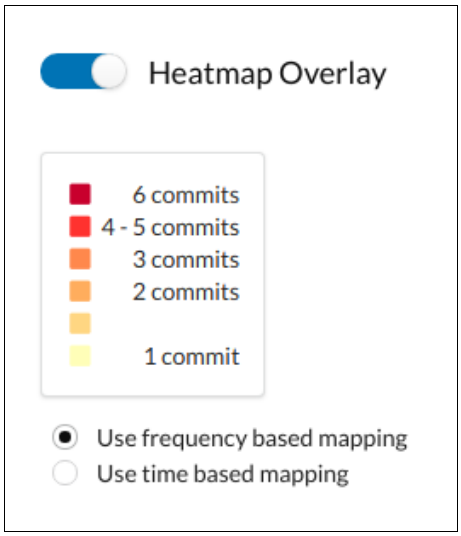

Abbildung 3.5: Elemente des *Heatmap Overlay*

Wie in Abbildung 3.6 zu sehen ist, wird der Text des Artikels nach Aktivierung der Heatmap nicht mehr in den jeweiligen Autoren-Farben dargestellt, sondern in denen der Heatmap. Die Information, die visualisiert wird, hat sich demnach geändert. Wo die Farbgebung zuvor Aufschluss über den jeweiligen Autor gegeben hat, wird nun je nach Einstellung die Änderungsfrequenz bzw. der zeitliche Änderungsverlauf angezeigt. Alle anderen Funktionalitäten bleiben erhalten und ändern sich nicht.

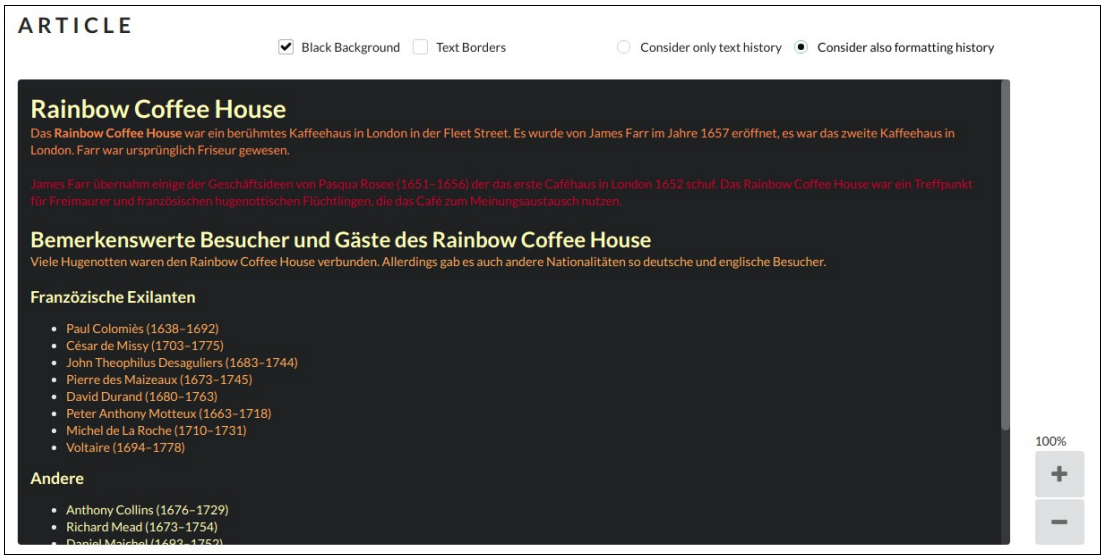

Abbildung 3.6: Artikel bei aktiver Heatmap

3.4.2 Zusätzliche Funktionalitäten

Im Folgenden sollen noch einmal all solche Funktionalitäten des *Document History Overlay* UI erläutert werden, die zwar nicht unmittelbar zur Umsetzung der spezifischen Anforderungen notwendig waren, aber dennoch eine Bereicherung der Benutzerschnittstelle darstellen. Sie dienen dem Nutzer als weitere Hilfestellung oder Zusatzfunktion zu den unerlässlichen Kernelementen.

Berücksichtigung der Formatierung

Wie bereits im vorherigen Kapitel beschrieben, bietet das *Document History Overlay* UI die Möglichkeit zur Auswahl der Art der angezeigten Informationen. Der Nutzer kann sich entweder ausschließlich die Historie eines Artikels auf inhaltlicher Ebene oder zusätzlich auch die Nachformatierung der Textelemente anzeigen lassen. Letztere ist im Sinne der Anforderung nicht zwingend notwendig, bietet aber interessante Zusatzinformationen. Beide Optionen stehen sowohl für das Autoren-Mapping als auch für die Heatmap zur Verfügung und erweitern dadurch die Informationsdarstellung.

Betrachten wir hierzu noch einmal das Beispiel des Artikels „Rainbow Coffee House“. Der Artikel wurde wie oben erläutert von Autor „Mona“ verfasst und von Autor „Claudia“ anschließend nachformatiert. Entscheidet sich der Nutzer für die Option „Consider only text history“ wird der Text ausschließlich in der Farbe des Autors „Mona“ eingefärbt. Ändert der Nutzer die Einstellung auf „Consider also formatting history“ wird die nachformatierte Textstelle in der Farbe des Autors „Claudia“ angezeigt.

Im Kontext der Heatmap verhält sich dies wie folgt: Stellt der Nutzer auf die Einstellung „Consider also formatting history“ um, so wird die nachformatierte Textstelle in einer dunkleren Heatmap-Farbe dagestellt, da jede Formatierung bei der frequenzbasierten Heatmap mitgezählt wird. Ist die zeitbasierte Heatmap aktiv, würde die nachformatierte Textstelle ebenfalls dunkler eingefärbt werden, da nach dem eigentlichen Verfassen der Passage eine weitere, zeitlich später vorgenommene Handlung erfasst wurde.

Zeitbasierte Heatmap

Abbildung 3.7 zeigt noch einmal die Heatmap und ihre Einstellungsoptionen. Die Anforderungen des *Document History Overlay* UI verlangen grundsätzlich nur nach der Implementierung einer Heatmap, die frequenzbasierte Informationen abbilden kann. Je häufiger also ein Commit stattgefunden hat, desto dunkler färbt sich die Heatmap. Wie das untenstehende Bild zeigt, existiert nun aber noch eine zweite Einstellungsoption: „Use time based mapping“.

Aktiviert der Nutzer diese Option, wird der zeitliche Verlauf visualisiert. Das bedeutet, dass diejenigen Einträge, die zuletzt bearbeitet worden sind, dunkler eingefärbt werden. Im Kontext der kooperativen Arbeit an Artikeln, kann auch diese Information von großem Mehrwert sein.

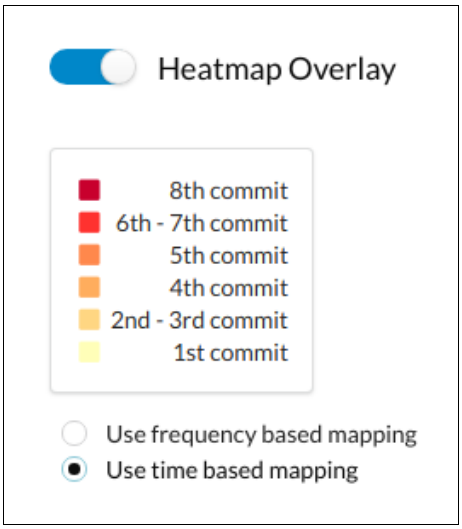

Abbildung 3.7: Legende bei zeitbasierter Heatmap

3.4.3 UX-Design

Neben der allgemeinen technischen Implementierung der Funktionalitäten des *Document History Overlay* UI, ist auch ihre benutzerfreundliche Umsetzung als eine der Hauptanforderungen dieser Arbeit zu betrachten. Die Benutzeroberfläche soll für alle möglichen Nutzergruppen intuitiv bedienbar sein, weshalb neben der Architektur auch das endgültige Design der Benutzerschnittstelle eine entscheidende Rolle spielt. Vor diesem Hintergrund wird im Folgenden zuerst auf einige grundlegende Design-Entscheidungen eingegangen. Anschließend werden die einzelnen Elemente des *Document History Overlay* UI noch einmal betrachtet und ihr Design begründet.

Layout

Zunächst fällt eine grundsätzliche Dreiteilung des Layouts der Benutzeroberfläche ins Auge, die keinesfalls willkürlich gewählt ist. Im oberen Bereich befindet sich die Leiste zur Bedienung der Autoren-Buttons, mittig ist das separierte Artikel-Segment mit seinen basalen Einstellungsmöglichkeiten zu finden und die Heatmap füllt den Platz rechts neben dem Artikelfenster aus.

Damit alle drei Bereiche übersichtlich und visuell hinreichend voneinander abgegrenzt werden konnten, galt es, eine gewisse räumliche Distanz zwischen den Elementen herzustellen. Das Artikel-Segment selbst nimmt dabei den zentralen und größten Bereich der Seite ein, weil der Nutzer hauptsächlich an und mit ihm arbeiten muss. Neben dem Artikelfenster ist auch der Bereich zur Bedienung der Autoren-Buttons relativ groß gehalten, womit die Hauptfunktionalitäten des *Document History Overlay* UI betont und in den Vordergrund gerückt werden. Da der Artikel mit der Bedienung der beiden anderen Bereiche korrespondiert, erschien es sinnvoll, diesen zentral zwischen die Elemente Heatmap und Autorenleiste zu setzen.

Damit auch beim Lesen längerer Artikel die Gesamtübersicht nicht unter einer allgemein scrollbaren Seite leidet, wurde der Artikel in ein separates Fenster implementiert, das in seiner Position zwischen Autorenbuttons und Heatmap statisch bleibt. Scrollbar ist letztlich nur der Text innerhalb des Artikelfensters. Damit bleibt das Grundlayout auch während des Scrollens vorhanden und garantiert dem Nutzer eine konstante Orientierung im Gesamtgeschehen. Zusätzlich wird durch die Separierung des scrollbaren Bereichs möglichem "Ruckeln" vorgebeugt.

Das gesamte Layout ist responsiv und somit auch auf beliebigen Mobilgeräten ohne Einschränkungen verwendbar.

Bedienbare Elemente

Damit der Nutzer durch die Vielfalt der Effekte und Optionen nicht irritiert wird und auf einen Blick seine aktuelle Konfiguration nachvollziehen kann, werden alle inaktiven Buttons und Bedienelemente automatisch ausgegraut. Das erhöht die Übersichtlichkeit vor allem auch deshalb, weil die Vielzahl der Autorenfarben ausgeblendet wird, sobald das *Heatmap Overlay* in Benutzung ist und umgekehrt.

Bezüglich der Ausgestaltung aller bedienbaren Elemente wurde darauf geachtet, dass all jene Schaltflächen und Buttons, von denen angenommen wird, dass sie am häufigsten genutzt werden, größer dargestellt wurden als jene, dessen Verwendung eher seltener eingeschätzt wird. Das hat den Vorteil, dass der Nutzer die relevanten Schaltflächen schneller findet und nicht von den seltener zu bedienenden Elementen abgelenkt oder irritiert wird. Das betrifft sowohl die Schaltflächen und Buttons auf der Hauptseite, als auch jene, die innerhalb der Popups zu finden sind.

Um die Bedienung so produktiv und schnell wie möglich zu gestalten, wurden alle Elemente mit eindeutigen Icons visualisiert. So werden alle Autoren mit einem kleinen Männchen visualisiert, die Zoom-Funktion besteht aus einem Plus- und einem Minus-Button und die Schaltflächen "Reassign colors" und "Reset colors" wurden durch entsprechende Pfeil-Icons ergänzt.

Darüber hinaus ist auch die Positionierung der Schaltflächen und Buttons nicht zufällig gewählt sondern an ihre jeweilige Funktion gekoppelt. Dies kann, wie im vorherigen Kapitel bereits erläutert wurde, bei der Anordnung der vier generellen Einstellungsoptionen ("Black Backgournd", "Text Borders", "Consider only text history" und "Consider also formatting history") des Artikels beobachtet werden.

Um das intuitive Verständnis der Buttons weiter zu verstärken, ist auch die Wahl von Checkboxen, Radiobuttons und Schiebereglern an die jeweilige Funktion des Bedienelements geknüpft. Der Schieberegler wird, wie im Kontext des *Heatmap Overlay*, zur Aktivierung einer neuen Funktion mit weiteren Einstellungsoptionen verwendet. Aktiviert der Nutzer die Heatmap, wird der vormals graue Regler blau und auch die zuvor ausgegraute Heatmap wird mit Farbe gefüllt. Somit ist auf einen Blick erkennbar, dass hier eine gänzlich neue Funktion aktiviert worden ist.

Checkboxen wurden verwendet, um Stil-Operationen zu visualisieren. Die Checkbox kann aktiviert oder deaktiviert sein und mit anderen Optionen kombiniert werden. Eine aktivierte Checkbox wird durch ein kleines Häkchen visualisiert. Radiobuttons hingegen liefern dem Nutzer ein "Entweder-Oder"-Verhalten.

Damit sich auch Popups in ihrer Weise von den übrigen Elementen abheben, wurde hier der Kontrast durch einen schwarzen Rahmen vor weißem Hintergrund verstärkt.

Kontext Autoren

Alle Autoren, die vom System registriert werden, erhalten einen eigenen Button und werden innerhalb einer Leiste am oberen Bildschirmrand gelistet. Die Leiste ist zur besseren Abgrenzung fein gerahmt und erstreckt sich über die gesamte Breite der Seite, um möglichst schmal und platzsparend dargestellt werden zu können. Höhe und Breite der Leiste sind unabhängig von der Anzahl der Autoren auf einen bestimmten Wert festgelegt und passen sich nicht an die Anzahl der Autoren an. Das hat den Vorteil, dass bei Auflistung sehr vieler Autoren das Gesamtlayout erhal-

ten bleibt und die restlichen Seitenelemente nicht weiter nach unten verschoben werden.

Im Extremfall sind damit nicht alle Autoren auf einen Blick sichtbar. Der Nutzer kann aber innerhalb der Leiste nach unten scrollen und so auch die anderen, weiter unten gelisteten Autoren in Augenschein nehmen. Damit dieses Verhalten keine Nachteile mit sich zieht, werden die Autoren ihrer Relevanz nach gelistet: Autoren mit einer sehr hohen Commit-Rate erscheinen dabei ganz oben und sind somit schneller aufrufbar. Durch die Scrollfunktion sowie die übrigen Funktionalitäten des Systems, kann der Nutzer aber auch auf weniger relevante Autoren ohne Probleme zugreifen.

Im rechten Bereich überhalb der Autorenleiste befinden sich zwei Buttons, mit deren Hilfe die Farbgebung aller Autoren auf einen Klick geändert werden kann. Die Schaltfläche “Reassign colors”, die mit zwei entgegengesetzt ausgerichteten Pfeilen symbolisiert ist, dient damit Personen als Schnellzugriff auf die Farbänderung. Der Button “Reset colors” ist mit einem Pfeil in rückwärtiger Kreisform visualisiert und bewirkt, dass alle Autoren-Farben auf die 12 Standardfarben zurückgesetzt werden. Beide Buttons befinden sich oberhalb des Autorenbereichs und wurden bewusst nicht in das Popup der Autorenbuttons selbst eingebaut. Die Aktivierung der Schaltflächen bewirkt nämlich eine Veränderung der Farbe aller Autoren-Buttons und nicht nur bei einzelnen. Individuelle Änderungen können über die Funktionen des Popups vorgenommen werden.

Möchte sich der Nutzer mit einem individuellen Autor beschäftigen, muss er mit dem einzelnen Autoren-Button selbst interagieren. Zunächst sei hier nochmal die Verknüpfung zum Artikeltext erläutert: Fährt der Nutzer mit der Maus über einen Autorenbutton, so wird der von ihm hinzugefügte oder bearbeitete Textbaustein visuell hervorgehoben. Umgekehrt wird der Autorenbutton eingefärbt, wenn man im Text auf den zugehörigen Textbaustein fährt. Dieses Verhalten dient Nutzern mit Farbenblindheit als Abhilfe, da sie sich am Kontrastverhalten der Elemente orientieren können.

Klickt der Nutzer auf einen Autoren-Button öffnet sich ein Popup. Im Popup werden dem Nutzer drei Bereiche zur Interaktion präsentiert, die zur besseren Abgrenzung durch eine feine Linie voneinander getrennt sind. Im obersten Bereich befindet sich eine grau hinterlegte Schaltfläche, die “Profile Page”. Ihre Ausgestaltung symbolisiert, dass man auf eine andere Seite weitergeleitet wird, sobald man sie anklickt. Darunter werden alle 12 Standardfarben in Form von Kreisen gelistet. Ganz unten

befindet eine weitere Schaltfläche, die eine Farbpalette öffnet, aus der bei Bedarf eine gänzlich neue Farbe für den Autor gewählt werden kann. Da davon ausgegangen wird, dass der Großteil der Nutzer eine der Standardfarben verwendet, ist der *Color Picker* nicht von vorneherein nutzbar, sondern muss bei Bedarf erst ausgeklappt werden.

Alle hier beschriebenen Änderungen werden nur auf den jeweiligen Autor angewendet, zu dem das Popup gehört. Seine Zugehörigkeit wird durch einen kleinen Pfeil im Rahmen des Popups visualisiert, der auf den Autor zeigt.

Kontext Artikel

Neben dem generellen strukturellen Aufbau des Artikel-Segments, der bereits unter dem Punkt "Layout" beschrieben wurde, soll hier nur noch auf einige spezielle Aspekte seiner Funktionen und dem Zusammenspiel eingegangen werden.

Insgesamt wurde beim Design und Verhalten des Textes darauf geachtet, dass gerade so viele Effekte und Funktionen eingebaut sind wie nötig. Der Nutzer soll optimal mit den Funktionalitäten umgehen können aber nicht von einer Masse an unterschiedlichen Effekten irritiert werden.

Fährt der Nutzer mit der Maus über einen Textbaustein, wird nicht nur der zugehörige Autor hervorgehoben, sondern auch der Baustein selbst. Beides unterstützt die Barrierefreiheit der Benutzeroberfläche wesentlich und gibt dem Nutzer eine umfassende Orientierung über den Commit und den dazugehörigen Autor.

Die beiden Einstellungsoptionen "Black Background" und "Text Borders" können im normalen Modus zwar als Hilfsvisualisierung genutzt werden, entfalten ihren wahren Mehrwert aber vor allem dann, wenn die Heatmap aktiv ist. Die invertierte Farbgebung des Hintergrunds sowie die zusätzliche Rahmung der Textbausteine helfen, die verschiedenen Stufen der Heatmap besser voneinander abgrenzen zu können.

Beim Klick auf eine Textstelle öffnet sich ein Popup, mit weiteren Informationen zur Commit-Historie. Das Popup erscheint oberhalb der angeklickten Textstelle, damit der weitere Lesefluss nicht behindert wird. Der aktuellste Commit wird folglich immer an unterster Stelle und damit direkt über der angeklickten Textstelle angezeigt. Möchte sich ein Nutzer über ältere Commits informieren, muss er in der Ordnung weiter zurück und damit wieder nach oben scrollen. Scrollbar ist darüber hinaus nur die

Commit-Historie und nicht das Popup selbst, damit der Nutzer nicht die Orientierung verliert und die Referenz zur Textstelle in jedem Fall erhalten bleibt. Zusätzliche Orientierung im zeitlichen Geflecht gibt die Zeitangabe oberhalb der Commit-Nachricht.

Farblich passt sich das Popup immer an die jeweils aktivierte Hintergrundfarbe an. Ist der Hintergrund weiß, so färbt sich das Popup schwarz und umgekehrt. Dadurch wird immer ein optimaler Kontrast gewährleistet.

Bezüglich der Zoom-Buttons sei hier lediglich darauf hingewiesen, dass Überschriften vom Zoom-Effekt ausgeschlossen sind, damit die Orientierung im Gesamtkontext erhalten bleibt. Besonders interessant gestaltet sich dadurch ein starkes Herauszommen bei aktivierter Heatmap, da so die Änderungshäufigkeit einzelner Abschnitte optimal sichtbar wird.

Kontext Heatmap

Die generelle Gestaltung der Heatmap wurde ebenfalls bereits erläutert. Im folgenden soll deshalb nur noch auf einige Details näher eingegangen werden, die bisher noch nicht beschrieben wurden.

Je nachdem welche Art der Heatmap gerade aktiv ist, ändert sich die Grammatik der Legende, so dass der Nutzer sofort einen Hinweis darauf erhält, welche Variante aktiv ist, ohne vorher noch einmal die Konfiguration zu prüfen. Ist die zeitbasierte Heatmap aktiviert, so heißt es in der Legende: “1st commit”, “2nd commit”, etc. Damit wird der Fokus auf die zeitliche Relevanz der Commits gelegt. Ist die frequenzbasierte Heatmap aktiviert, heißt es hingegen: “1 commit”, “2 commits”, etc. Somit ist für den Nutzer beim Lesen der Legende unmissverständlich klar, dass die Anzahl der Commits betrachtet wird.

Abschließend sei noch angemerkt, dass die Farben bei der zeitbasierten Heatmap mithilfe einer linearen Funktion zugewiesen werden, während dies bei der frequenzbasierten Heatmap über eine logarithmische Funktion geregelt wird. Je nach Variante ist die eine oder andere Darstellung sinnvoller. Die Logarithmische Darstellung beispielsweise eignet sich besser für die frequenzbasierte Heatmap, da sie schneller auf Änderungen anspricht, aber sehr viele Commits notwendig sind, ehe die dunkelste Farbvariante der Heatmap erreicht wird.

4 Implementierung

In diesem Kapitel wird zunächst ein grober Überblick über die verwendeten Technologien gegeben. Anschließend wird die Implementierung des *Document History Overlay* betrachtet. Der Großteil der für die Umsetzung notwendigen und auch relevanten Schritte wird anhand des Quellcodes gezeigt und im Detail beschrieben. Dieser ist in Ausschnitte gegliedert und unter Umständen gekürzt dargestellt, um so die wesentlichen Merkmale, Struktur und Semantik, in den Vordergrund zu stellen. Die syntaktische Korrektheit des Codes ist daher nicht immer gewährleistet und somit als unmaßgeblich zu betrachten. Als Basis für die Umsetzung dienen vorangegange Ergebnisse zu Architektur und Design. Es wird dabei zwischen Backend und Frontend unterschieden.

4.1 Technologien

Das Backend wurde in der Programmiersprache Java geschrieben. Für die Erstellung der Software wurde die integrierte Entwicklungsumgebung IntelliJ IDEA[3] von JetBrains genutzt. Der Build-Prozess wird durch das Build-Management-Tool Maven[4] unterstützt. Das Frontend wurde in der Programmiersprache TypeScript geschrieben und nutzt die JavaScript-Softwarebibliothek React[5]. Als Framework für die UI-Komponenten wurde Semantic UI React[6] verwedet. Für die Entwicklung wurde hier der Quelltext-Editor Visual Studio Code[7] von Microsoft genutzt.

[3] https://www.jetbrains.com/idea/
[4] https://maven.apache.org/
[5] https://reactjs.org/
[6] https://react.semantic-ui.com/
[7] https://code.visualstudio.com/

4.2 Backend

Zunächst wird ein Überblick über die notwendigen Schritte der Implementierung des Backends gegeben. Die Programmierung setzt hier stark auf den vorhandenen Code auf und erfordert daher lediglich Anpassungen an bereits bestehende Klassen, um die erarbeitete Architektur umsetzen zu können. Zum einen muss daher die *Vertex* Klasse um Variablen für *Commit ID* und *Project ID* erweitert werden. Damit das *Vertex* Objekt anschließend auch mit den IDs in der Datenbank persistiert werden kann, muss diese um Attribute in der entsprechenden Tabelle erweitert werden, um so die neuen Daten sowohl zu speichern als auch abrufen zu können. Zum anderen muss das eigentliche Speichern eines *Vertex* Graphen erweitert werden, indem *Commit ID* und *Project ID* der entsprechenden Revision mit übergeben werden und somit die Instanzen auch richtig erzeugt werden können. Da beim Speichern zwischen erstmaligem und nachfolgendem Speichern eines Graphen - bedingt durch die Existenz einer vorherigen Revision - unterschieden wird, müssen außerdem sogenannte Änderungsoperationen angepasst werden. Ebenso muss das Abrufen des Graphen erweitert werden, damit der *Resource Tree* richtig konstruiert werden kann und somit schließlich eine WOM-Resource erzeugt werden kann, die für jeden ihrer Knoten *Commit ID* und *Project ID* beinhaltet.

4.2.1 Vertex und Graphdatenbank

Die Revisionsinformationen einer WOM-Ressource soll zunächst auf die Graphdatenbank abgebildet werden. Eine WOM-Ressource besteht aus einer Menge von Knoten, die in der Datenbank als *Vertices* abgebildet werden. Die zugehörige Klasse muss zunächst aktualisiert werden. Listing 4.1 zeigt die Anpassung der *Vertex* Klasse. Diese wird um die Deklarierung der privaten Klassenvariablen *projectId* und *commitId* vom Datentyp *long* erweitert. Der gewählte Datentyp eignet sich gut für IDs und wird für diese auch an anderen Stellen im Code benutzt. Auch dem Konstruktor werden beide IDs übergeben, damit die *Vertices* später korrekt instanziiert werden können. Typische Getter- und Setter-Methoden werden für die neuen Variablen ebenso implementiert.

```java
public class Vertex {
  private static final VertexId PLACEHOLDER = new VertexId(Long.MIN_VALUE, Integer.MIN_VALUE);
  private VertexId vertexId;
  private VertexFlag flag;
  private Optional<Node> node;
  private Optional<Vertex> parentVertex;
  private List<Vertex> childVertices;
  private boolean isPersistent;
  private long projectId;
  private long commitId;

  public Vertex(..., long projectId, long commitId) {
    ...
    this.projectId = projectId;
    this.commitId = commitId;
  }
  ...
}
```

Listing 4.1: Implementierung der Vertex Klasse

Die *Vertices* Tabelle speichert *Vertices*, die genau einem Graph zugeordnet sind. Um hier *Commit ID* und *Project ID* eines *Vertex* speichern zu können muss die Datenbank dementsprechend erweitert werden. Listing 4.2 zeigt die Erweiterung um neue Attribute für beide IDs. Die zugehörigen Werte werden später im Datentyp *bigint* abgespeichert.

```java
public abstract class CassandraSchema {
  ...
  public static abstract class Tables {
    public static final String VERTICES = "vertices";
    public static final class Vertices {
      public static final int FLAG_NONE = 0;
      public static final int FLAG_DEEP = 1;
      public static final int FLAG_BELOW_DEEP = 2;
      public static final String GRAPH_ID = "graph_id";
      public static final String VERTEX_INDEX = "vertex_index";
      public static final String PARENT_VERTEX = "parent_vertex";
      public static final String CHILD_VERTICES = "child_vertices";
      public static final String FLAG = "flag";
      public static final String PAYLOAD = "payload";
      public static final String PROJECT_ID = "project_id";
      public static final String COMMIT_ID = "commit_id";
    }
    ...
  }
}
```

Listing 4.2: *Vertices* Tabelle mit Attributen für *Commit/Project ID*

Die Klasse *CassandraVertexDAO* wird zum Speichern eines *Vertex* in die Tabelle, als auch zum Abrufen eines *Vertex* aus dieser genutzt. Hierzu werden die Funktionen aus Listing 4.3 angepasst. Die Funktion insertVertexStatement greift mithilfe der Getter-Methoden auf *Commit ID* und *Project ID* des übergebenen *Vertex* zu und speichert den Wert in den zugehörigen Zellen ab. Die zweite Funktion rekonstruiert einen *Vertex*, indem sie aus der übergebenen Zeile der *Vertices* Tabelle unter anderem die Werte für beide IDs extrahiert. Schließlich wird der *Vertex* zurückgegeben und kann somit aus der Datenbank abgerufen werden.

```
public class CassandraVertexDAO implements VertexDAO {
  ...
  private String insertVertexStatement(final Vertex vertex) {
    return QueryBuilder
        .insertInto(CassandraSchema.Tables.VERTICES)
        .value(CassandraSchema.Tables.Vertices.GRAPH_ID, vertex.getVertexId().getGraphId())
        .value(CassandraSchema.Tables.Vertices.VERTEX_INDEX, vertex.getVertexId().getIndex())
        .value(CassandraSchema.Tables.Vertices.FLAG, mapVertexFlag(vertex.getFlag()))
        .value(CassandraSchema.Tables.Vertices.PARENT_VERTEX, QueryBuilder.bindMarker())
        .value(CassandraSchema.Tables.Vertices.CHILD_VERTICES, QueryBuilder.bindMarker())
        .value(CassandraSchema.Tables.Vertices.PAYLOAD, QueryBuilder.bindMarker())
        .value(CassandraSchema.Tables.Vertices.PROJECT_ID, vertex.getProjectId())
        .value(CassandraSchema.Tables.Vertices.COMMIT_ID, vertex.getCommitId())
        .getQueryString();
  }
  ...
  private static Vertex convertResultRowToVertex(final Row row, final Document ownerDocument) {
    ...
    final long projectId = row.getLong(CassandraSchema.Tables.Vertices.PROJECT_ID);
    final long commitId = row.getLong(CassandraSchema.Tables.Vertices.COMMIT_ID);

    return new Vertex(vertexId, flag, parentVertexId, childVertexIds, payload, true, projectId,
    commitId);
  }
}
```

Listing 4.3: Speichern und Abrufen der neuen Attribute

4.2.2 Graphspeicherung und -operationen

Nun soll eine WOM-Resource inklusive der Angabe von *Commit ID* und *Project ID* als Graph abgespeichert werden können. In der Klasse *ResourceStorageAPIImpl* wird das in Abschnitt 4.2.1 beschriebene Speichern und Abrufen von *Vertices* angestoßen. Listing 4.4 zeigt die verantwortlichen Funktionen. Wichtig bei dieser Variante ist, dass ein vollständiger Graph ohne Verwendung einer vorherigen Revision gespeichert wird. Es handelt sich daher um die erstmalige Speicherung. Die Funktion *storeGraph* wird unter anderen mit den Parametern *Commit ID* und *Project ID* aufgerufen. Diese werden lediglich weitergereicht, bis sie in *domToVertexGraph* genutzt werden, um einen *Vertex* korrekt zu instanziieren. Wie bereits beschrieben kann dieser später durch die *insertVertexStatement* Methode in die Datenbank geschrieben werden.

```
@Override
public void storeGraph(..., long projectId, long commitId) {
  doStoreGraph(..., projectId, commitId);
  ...
}
...
protected void doStoreGraph(..., long projectId, long commitId) {
  ...
  domToVertexGraph(..., projectId, commitId);
  ...
}

private int domToVertexGraph(..., long projectId, long commitId) {
  Vertex vertex = new Vertex(graphId, index, resource);
  vertex.setProjectId(projectId);
  vertex.setCommitId(commitId);
  ...
}
```

Listing 4.4: Die *storeGraph* Methode bei erstmaligem Speichern

Wird hingegen die *storeGraph* Methode zusätzlich mit der *Graph ID* der vorherigen Revision aufgerufen wird durch Überladen der Code aus Listing 4.5 aufgerufen. Diesmal wird allerdings ein sogenanntes *Change Tree Node* Objekt zurückgegeben, welches den Änderungsbaum darstellt und sämtliche Änderungsoperationen beinhaltet. Hier werden *Commit ID* und *Project ID* Parameter ebenfalls weitergereicht und schließlich von der statischen *apply* Funktion der *ChangeTreeApplyOTron* Klasse verwertet.

```
@Override
public CTNNode storeGraph(..., final long projectId, final long commitId) {
  final CTNNode changeTree = doStoreGraph(..., projectId, commitId);
  storeCustomEdges(edges);

  return changeTree;
}
...
protected CTNNode doStoreGraph(..., final long projectId, final long commitId) {
  ...
  ChangeTreeApplyOTron.apply(..., projectId, commitId);
  ...
}
```

Listing 4.5: Die *storeGraph* Methode bei nachfolgendem Speichern

Die Funktion aus Listing 4.6 führt alle Änderungsoperationen durch und übergibt den jeweiligen *Handlern* die beiden IDs. Es werden hier lediglich die Operationen *NODE*, *INSERT*, *UPDATE* und *SPLIT* betrachtet. *DELETE* und *MOVE* sind weitere Operationen, allerdings sind diese im weiteren Verlauf irrelevant bezüglich der Anpassung an *Commit ID* und *Project ID*.

```
public class ChangeTreeApplyOTron {
  ...
  private static void apply(..., final long projectId, final long commitId) {
    for (ChangeTreeNode change : changes) {
      switch (change.getOp()) {
        case NODE:
          handleNode((CTNNode) change, parentsChildVertices, graphId, moveMap, projectId, commitId);
          break;
        case INSERT:
          handleInsert((CTNInsert) change, parentsChildVertices, graphId, projectId, commitId);
          break;
        case UPDATE:
          handleUpdate((CTNUpdate) change, parentsChildVertices, graphId, projectId, commitId);
          break;
        ...
        case SPLIT:
          handleSplit((CTNSplit) change, parentsChildVertices, graphId);
          break;
        ...
      }
    }
  }
  ...
}
```

Listing 4.6: Weiterreichen der IDs an relevante Änderungsoperationen

Die Anpassungen der *Handler* wird in Listing 4.7 im Detail gezeigt. Die Funktion *handleNode* ruft hier wiederum *apply* mit den selben Parametern auf, damit stets zu allen Operationen navigiert werden kann. Die nächsten beiden *Handler* sind für das Einfügen und das Ändern eines Knoten zuständig. In beiden Fällen werden die durchgereichten IDs genutzt, um sie beim erzeugten *Vertex* zu setzen. Dadurch werden sowohl bei einem neu eingefügten Knoten als auch bei einem bereits existierenden Knoten stets die aktuellen Commit-Daten an der richtigen Stelle in der Ressource zugeordnet. Die Methode *handleSplit* erzeugt aus einem *Vertex* zwei neue *Vertices* und fügt diese in den Baum ein. Beide Knoten müssen daher die *Commit ID* und *Project ID* des ursprünglichen Knotens erhalten.

```
private static void handleNode(..., final long projectId, final long commitId) {
  ...
  apply(graphId, vertex.getChildVertices(), node.getChanges(), moveMap, projectId, commitId);
}

private static void handleInsert(..., final long projectId, final long commitId) {
  ...
  insertTree(graphId, tree, projectId, commitId);
  ...
}

private static Vertex insertTree(..., long projectId, long commitId) {
  final Vertex vertex = new Vertex(graphId, treeNode.cloneNode(false));
  vertex.setProjectId(projectId);
  vertex.setCommitId(commitId);
  recreateVertexChildrenFromDom(graphId, treeNode, vertex);
  return vertex;
}
...
private static void handleUpdate(..., final long projectId, final long commitId) {
  final Vertex vertex = vertices.get(update.getPos());
  vertex.setProjectId(projectId);
  vertex.setCommitId(commitId);
  ...
}
...
private static void handleSplit(...) {
  ...
  final Vertex originalVertex = vertices.get(splitNodeIndex);
  ...
  final long projectId = originalVertex.getProjectId();
  final long commitId = originalVertex.getCommitId();

  final Vertex splitNodeA = insertTree(graphId, originalNode,  projectId, commitId);
  final Vertex splitNodeB = insertTree(graphId, nodeB, projectId, commitId);

  vertices.remove(splitNodeIndex);
  vertices.add(splitNodeIndex, splitNodeB);
  vertices.add(splitNodeIndex, splitNodeA);
}
```

Listing 4.7: Anpassung relevanter *Handler* an die neue *Vertex* Struktur

In einem letzen Schritt wird das Abrufen des Graphen anhand der Funktion *retrieveGraph* realisiert. Diese stößt schließlich die *assembleResourceTree* Methode an, welche in Listing 4.8 gezeigt wird. Hier werden die *Resource Nodes*, die mit jedem *Vertex* asoziiert werden, verbunden und zu einem *Resource Tree* zusammengesetzt. Bevor dies geschieht werden *Commit ID* und *Project ID* als Attribute des *HIST* Namensraums auf die jeweiligen *Resource Nodes* geschrieben. Der Namensraum wurde dabei genau für diesen Zweck definiert.

```
private Node assembleResourceTree(final Vertex vertex, final Map<VertexId, Vertex> vertexMap) {
  ...
  histNode = (Element) vertex.getNode().get();
  long projectId = vertex.getProjectId();
  long commitId = vertex.getCommitId();

  if (projectId != Long.MIN_VALUE)
    histNode.setAttributeNS("http://sweble.org/schema/hist30", "hist:projectId",
    String.valueOf(projectId));

  if (commitId != Long.MIN_VALUE)
    histNode.setAttributeNS("http://sweble.org/schema/hist30", "hist:commitId",
    String.valueOf(commitId));
  ...
}
```

Listing 4.8: Setzen der Attribute im *Resource Tree*

So kann schließlich eine WOM-Ressource abgerufen werden, die *Commit ID* und *Project ID* für jeden ihrer Knoten enthält. Das Backend stellt nun ein passendes Format bereit, das vom Frontend in angemessener Form genutzt werden kann.

4.3 Frontend

Um die erarbeiteten Anforderungen an das Frontend erfüllen zu können, müssen geeignete Klassen implementiert werden. Zunächst wird der bereits vorhandene *Article Renderer so* angepasst, dass er später von den Komponenten sinnvoll genutzt werden kann. Anschließend werden die neuen Klassen implementiert. Diese lassen sich in *DocumentHistoryOverlay*, *DocumentHistoryDecorator* und *HistoryWrapper* gliedern. Die Darstellung und Struktur der Implementierung richtet sich dabei grob den Aufbau und Arbeitsablauf, wie er im vorherigen Kapitel beschrieben wurde. Abschließend wird noch auf einige Algorithmen eingegangen, die für die Implementierung eine besondere Bedeutung haben.

4.3.1 Article Renderer

Der sogenannte *Article Renderer* ist für die Visualisierung eines Artikels zuständig. Hier werden lediglich die *Props* der Komponenten *DisplayArticle* und *DisplayArticleSimple* um eine optionale *Decorator* Liste erweitert. Diese wird unter anderem genutzt, um eine *ReactElementFactory* zu instanziieren. Es werden React-Elemente für jedes WOM-Element des Artikels erzeugt. Dadurch findet der Dekorierer direkte Anwendung beim Anzeigen eines Artikels. Die Nutzung der angepassten Komponenten wird im nächsten Abschnitt im Kontext des Artikels dargestellt.

4.3.2 DocumentHistoryOverlay

Zunächst wird die Klasse *DocumentHistoryOverlay* betrachtet. Es handelt sich hier um die Seite, auf deren Basis alle weiteren Implementierungen aufbauen. Zuerst wird auf zugehörige *Props* und den *State* eingegangen, anschließend auf fundamentale Funktionen, die nach dem Mounten der Komponente ausgeführt werden. Erst im Anschluss wird dann auf den eigentlichen Aufbau dieser eingegangen und die *render* Methode anhand zugehöriger Elemente und Funktionen beschrieben.

Der eigentliche Artikel, auf den alle Funktionalitäten des *Document History Overlay* angewendet werden sollen, wird der Komponente über die *Props* übergeben. Listing 4.9 zeigt das zugehörige Interface. Der Artikel ist vom Typ *WomDocument*, eine Format, welches aus einem XML-Dokument geparst werden kann. Dieses Dokument wird vom Backend über die zugehörige WOM-Ressource bereitgestellt und enthält eine für das Frontend relevante Struktur, welche bereits in Abschnitt 4.2 behandelt wurde.

```
export interface HistoryOverlayProps {
  article?: WomDocument;
  onSelectArticle?: any;
}
```

Listing 4.9: Implementierung des *HistoryOverlayProps* Interface

Der *State* der Komponente setzt sich aus den Zustandsvariablen in Listing 4.10 zusammen und wird im Konstruktor der Klasse initialisiert. Dabei wird jeder Variable zunächst entweder ein leerer *String*, eine leere

Liste oder der Booleschen Wert *false* zugewiesen. Der *State* wurden so gestaltet, dass jederzeit ermittelt werden kann, welche Features bezüglich Stil und Berechnung des *Document History Overlay* aktiv sind. Es werden hier Informationen sowohl über Autoren- und Farbenliste, als auch über die Auswirkungen der Maus-Ereignisse *mouseover* und *click* im jeweiligen Feature gespeichert. Die einzelnen Zustandsvariablen und deren Bedeutung werden später im Kontext der jeweiligen Funktionen detaillierter betrachtet.

```
export interface HistoryOverlayState {
  article?: WomDocument;

  isHeatmapActive: boolean;
  authorColors: string[];
  authorNames: string[];
  isTextBordersActive: boolean;
  isBlackBackgroundActive: boolean;
  isFormattingHistoryActive: boolean;
  isTimeBasedMappingActive: boolean;
  heatmapLegendBoundary: number;
  articleStyleId: string;
  isAuthorButtonHovered: boolean;

  hoveredAuthorButtonColor: string;
  hoveredAuthorButtonName: string;

  clickedAuthorButtonName: string;
  clickedAuthorButtonColor: string;
  isSketchPickerVisible: boolean;
  articleZoom: number;
}
```

Listing 4.10: Implementierung des *HistoryOverlayState* Interface

Die *Lifecycle* Methode *componentDidMount* wird in Listing 4.11 gezeigt. Nach dem Mounten der Komponente werden zum einen die Farben, die den Autoren zugewiesen werden sollen, in der entsprechenden Liste auf ihren Standardwert gesetzt. Zum anderen wird die *setArticle* Methode aufgerufen, die anhand des übergebenen Artikels die wichtigsten Zustandswerte berechnet und speichert. Auf diese Weise wird der Initialzustand des *DocumentHistory*Overlay berechnet, nachdem ein (neuer) Artikel der Komponente übergeben wurde. Die bereits beschriebene Methode *resetColors* ist daher außerhalb dieser Methode angesiedelt, da nicht bei jedem neuen Artikel, der geladen wird, die Farbenliste zurückgesetzt werden soll.

```
componentDidMount() {
  this.resetColors();
  if (this.state.article) {
    this.setArticle(this.state.article);
  }
}
```

Listing 4.11: Implementierung der *componentDidMount* Methode

Listing 4.12 zeigt die die Methode *setArticle* im Detail. Es werden die Namen aller Autoren, die zu dem Artikel beigetragen haben, in der Liste *authorNames* gespeichert. Die Methode, die diese Namen aus dem Artikel extrahiert, wird in Listing 4.13 gezeigt. Des weiteren wird der aktuell ausgewählte Stil für die Anzeige des Artikels im UI berechnet. Abschließend wird der obere Grenzwert für die Zuweisung von Farben bezüglich des *Heatmap Overlay* berechnet. Auf die Berechnung wird später in diesem Abschnitt eingegangen.

```
setArticle = (
  article: WomDocument,
) => {
  const authorNames = this.getAuthorsOfArticle(article);
  const isFirstArticle = this.state.article ? this.state.isBlackBackgroundActive : true;
  const isBlackBackgroundActive = isFirstArticle;

  const articleStyleId = getArticleStyleId(
    false,
    isBlackBackgroundActive,
    this.state.isTextBordersActive,
  );

  const heatmapLegendBoundary = this.state.isTimeBasedMappingActive ?
  this.getTimeBasedHeatmapBoundary(article) :
  this.getHeatmapBoundary(article);

  this.setState({
    article,
    authorNames,
    isBlackBackgroundActive,
    articleStyleId,
    heatmapLegendBoundary,
  });
};
```

Listing 4.12: Initiales Setzen von Artikelinformationen

Im folgenden wird die bereits erwähnte *getAuthorsOfArticle* Funktion betrachtet, welche eine Liste von Autorennamen zurückgibt. Hier werden alle Commits des übergebenen Artikels in einer Liste gespeichert. Aufgrund der Tatsache, dass der Autorenname Bestandteil eines Commits ist, lassen sich so alle zugehörigen Namen ermitteln. Die Liste wird durchlaufen und alle Autorennamen werden in eine neue Liste gespeichert. Diese enthält neben dem jeweiligen Namen auch die Anzahl der Commits, bei denen der Name zugewiesen ist. Abschließend können so die Autoren in ihrer Liste nach absteigender Commit-Häufigkeit sortiert werden. Der Autor mit den meisten Commits steht somit an erster Stelle.

```
getAuthorsOfArticle = (
  article?: WomDocument,
): string[] => {
  if (!article) {
    return [];
  }

  let authors: string[] = [];
  const authorsWithFrequencies: {
    name: string;
    frequency: number;
  }[] = [];

  const commits: CommitType[] = this.getAllCommitsOfArticle(article, true) as CommitType[];
  commits.forEach((commit) => {
    if (authors.includes(commit.author)) {
      const i = authors.indexOf(commit.author);
      authorsWithFrequencies[i].frequency = authorsWithFrequencies[i].frequency + 1;
    } else {
      authors.push(commit.author);
      authorsWithFrequencies.push({ name: commit.author, frequency: 1 });
    }
  });

  authorsWithFrequencies.sort((a, b) => {
    return b.frequency - a.frequency;
  });

  authors = [];
  authorsWithFrequencies.forEach((author) => {
    authors.push(author.name);
  });

  return authors;
};
```

Listing 4.13: Ermitteln aller Autoren eines Artikels

Um die Commits, die im vorherigen Schritt genutzt wurden, zu erhalten, wird die rekursive Methode aus Listing 4.14 betrachtet. Diese durchläuft alle *Child Nodes* des übergebenen Artikels. Die Attributwerte für *commitId* und *projectId* aus dem *HIST* Namensraum werden nun aus jedem Kindknoten gelesen und für die Initialisierung der Konstanten *commitId* und *projectId* genutzt. Mittels dieser Daten kann nun ein Konstrukt vom Typ *CommitType* erhalten werden, der sogenannte Commit. Dieser wird anschließend in einer Liste gespeichert, wobei doppelte Einträge grundsätzlich verhindert werden. Anschließend wird die Methode für jedes der Kindknoten rekursiv aufgerufen und die Liste wird weitergereicht. So können alle Commits in der Liste gespeichert werden bis schließlich keine weiteren *Child Nodes* vorhanden sind. Dies stellt die Abbruchbedingung dar. Abschließend wird die Liste aller Commits zurückgegeben.

```
getAllCommitsOfArticle = (
  article: WomDocument|WomElement,
  asConst?: boolean,
  currentCount?: CommitType[]|string[],
): CommitType[]|string[] => {

  let commits: any[] = [];

  if (currentCount) {
    commits = currentCount;
  }

  article.childNodes.forEach((child) => {
    if (child instanceof WomElement) {
      const commitId = child.getAttributeNS(HIST_NS, 'commitId');
      const projectId = child.getAttributeNS(HIST_NS, 'projectId');

      const commit: CommitType|null = (commitId && projectId) ?
        receiveCommit(commitId, projectId) :
        null;

      if (commit !== null) {
        const data = (asConst && asConst === true) ? commit : commit.name;
        if (!commits.includes(data)) {
          commits.push(data);
        }
      }
      commits = this.getAllCommitsOfArticle(child as WomElement, asConst, commits);
    }
  });

  return commits;
};
```

Listing 4.14: Ermitteln aller Commits eines Artikels

Wie bereits beschrieben, wurde das Interface namens *CommitType* definiert, um das Konstrukt namens Commit darstellen zu können. Listing 4.15 zeigt die zugehörigen *Properties*. So sind alle Daten vorhanden, die für die weitere Implementierung wichtig sind.

```
interface CommitType {
  author: string;
  timestamp: number;
  name: string;
  commitMessage: string;
  project: string;
}
```

Listing 4.15: Implementierung des *CommitType* Interface

Im nächsten Schritt wird auf die *render* Methode der Seite eingegangen, welche die Elemente in Javascript Syntax zurückgibt. Listing 4.16 gibt dabei einen groben Überblick über die wichtigsten Elemente. Neben einem *Button* für „Reset colors“ gibt es auch einen für „Reassing colors“. Darauf folgen die sogenannten *Author Buttons* und auch einige Kontrollkästchen, welche für Stil und Darstellung des gezeigten Artikels zuständig sind. Anschließend wird der Artikel dargestellt. Ebenso wird jeweils ein *Button* für das Vergrößern und Verkleinern des Textes eingefügt. Abschließend werden Elemente bezüglich des *Heatmap Overlay* zurückgege-

ben. Diese beinhalten ein Kontrollkästchen zum Ein- und Ausschalten der Heatmap, die zugehörige Legende, und noch einmal Kontrollkästchen für die Art der Berechnungs. Alle Elemente nutzen den *State* um in bestimmten Situationen deaktiviert werden zu können. So ist beispielsweise der *Button* für „Reset colors“ deaktiviert, sobald das *Heatmap Overlay* aktiv ist, oder noch kein Artikel in die Seite geladen wurde.

```
return(
  ...
  <Button
    content={'Reset colors'}
    disabled={this.state.isHeatmapActive || !this.state.article}
  />
  ...
  {renderedAuthorButtons}
  ...
  <Checkbox
    checked={this.state.isBlackBackgroundActive}
    label="Black Background"
  />
  ...
  <Segment inverted={this.state.isBlackBackgroundActive}>
    <span style={{ fontSize: `${this.state.articleZoom}%` }}>
      {renderedArticle}
    </span>
  </Segment>
  ...
  <Button disabled={this.state.articleZoom === ZOOM_MAX}/>
  ...
  <Segment disabled={!this.state.isHeatmapActive}>
    {renderedHeatmapLegend}
  </Segment>
  ...
  <Checkbox
    checked={!this.state.isTimeBasedMappingActive}
    disabled={!this.state.isHeatmapActive}
    label="Use frequency based mapping"
  />
  ...
);
```

Listing 4.16: Überblick über die Elemente des *DocumentHistoryOverlay*

Im folgenden wird die Implementierung aller Elemente der Seite in drei Abschnitte unterteilt, um eine bessere Sicht auf die Strukturierung bieten zu können. Diese lassen sich den Kontexten „Autoren“, „Artikel“ und „Heatmap“ gliedern.

Kontext Autoren

Grundsätzlich ist zu beachten, dass der Name und die zugewiesene Farbe des Autors in jeweils unterschiedlichen Listen verwaltet werden, um so simple und unterschiedliche Operationen auf diese zu ermöglichen. Dennoch teilen sich beide Listen sozusagen den Index. Daher gilt bei der Zuordnung stest folgendes: Dem Autoren an Stelle i in der Autorenliste wird die Farbe an Stelle i in der Liste der Autoren-Farben zugeordnet.

Zunächst werden die Schaltflächen namens „Reassign colors“ und „Reset colors“ betrachtet. Die zugehörigen Funktionen aus Listing 4.17 werden aufgerufen, sobald auf die Schaltflächen geklickt wird. Beim Klick auf erstgenannten *Button* wird die Liste der Farben, die auf die Autorenliste zugeordnet wird, aus dem *State* gelesen. Dabei wird die Farbe an erster Stelle temporär gespeichert, aus der Liste entfernt und schließlich durch die *push* Funktion hinten wieder angehängt. Beim Zurücksetzen der Farben hingegen wird eine leere Liste erzeugt und zunächst mit den Standardfarben für Autoren gefüllt. Anschließend wird die Funktion *fillWithRandomColours* aufgerufen. Hier wird überprüft, ob es mehr Autoren als die zwölf Standardfarben gibt. Falls dem so ist, wird der Liste für jeden weiteren Autor eine zufällige Farbe hinzugefügt. Daher werden bei jedem Klick auf die Schaltfläche alle Farben ab der 13. Stelle zufällig erzeugt.

```
onClickButtonReassignColors = () => {
  const authorColors = this.state.authorColors;
  const removedColor = authorColors[0];
  authorColors.shift();
  authorColors.push(removedColor);
  this.setState({ authorColors });
};

resetColors = () => {
  let authorColors: string[] = [];
  authorColors = this.fillWithDefaultColors(authorColors);
  authorColors = this.fillWithRandomColours(authorColors);
  this.setState({ authorColors });
};

fillWithDefaultColors = (
  array: string[],
) => {
  const result = array;
  return result.concat(DEFAULT_AUTHOR_COLORS);
}

fillWithRandomColours = (
  array: string[],
) => {
  const result = array;

  while (result.length < this.state.authorNames.length) {
    let newColor = getRandomColor();
    while (result.includes(newColor)) {
      newColor = getRandomColor();
    }
    result.push(newColor);
  }

  return result;
};
```

Listing 4.17: *Handler* für die Änderung der Farbenliste

Die grobe Struktur der *Author Buttons* ist in Listing 4.18 zu sehen. Hier wird eine Liste erzeugt, die die Schaltflächen beinhalten soll. So wird für jeden Eintrag der Autorenliste auss dem S*tate* ein *Popup* in die Liste ein-

gefügt. Die wichtigsten Bestandteile dieses Elements sind Auslöser und Inhalt. Beim Klick auf den *Author Button* (Auslöser) wird also das *Popup* mitsamt Inhalt angezeigt. Im folgenden wird auf beide Bestandteile eingegangen.

```
renderAuthorButtons = () => {
  const authorButtons = [];

  for (let i = 0; i < this.state.authorNames.length; i = i + 1) {
    const authorName = this.state.authorNames[i];
    const authorColor = this.state.authorColors[i];

    ...

    authorButtons.push(
      <Popup
        content={authorButtonPopupContent}
        inverted
        key={`authorButton_${i}`}
        on="click"
        position="bottom left"
        trigger={authorButton}
      />,
    );
  }

  return authorButtons;
};
```

Listing 4.18: Implementierung der Autoren-Leiste

Der eigentlich *Author Button* wird in Listing 4.19 gezeigt. Er beinhaltet sowohl den Namen, als auch die zugeordnete Farbe des zu repräsentierenden Autors. Falls das *Heatmap Overlay* aktiv ist wird keine Farbe gesetzt. Das User-Icon innerhalb der Schaltfläche wird standardmäßig nur als Umriss gezeichnet. Falls aber der Name des zu repräsentierenden Autors in der Zustandsvariable *hoveredAuthorButtonName* gesetzt ist, wird dieser Umriss ausgefüllt dargestellt. Dadurch wird eine Animation auf den *Button* ermöglicht. Für das Setzen der Zustandsvariable sind die *Handler* zuständig, die bei *Mouse Events* auf dem *Button* ausgeführt werden. Sobald die Maus über der Schaltfläche liegt, wird der zugehörige Autorenname gespeichert. Andernfalls wird ein leerer *String* gespeichert.

```
const authorButton = (
  <span
    onMouseEnter={onMouseEnterHandler}
    onMouseLeave={onMouseLeaveHandler}
  >
    <Button
      content={authorName}
      icon={this.state.hoveredAuthorButtonName === authorName ? 'user' : 'user outline'}
      onClick={onClickAuthorButtonHandler}
      style={{ color: !this.state.isHeatmapActive ? this.state.authorColors[i] : ''  }}
    />
  </span>
);
```

Listing 4.19: Implementierung eines *Author Button*

Der Inhalt des *Popup* selbst wird in Listing 4.20 dargestellt. Die wichtigsten Elemente sind ein einfacher *Button*, der als Link zur Profilseite des Autors fungiert, und sogenannte *Color Picker*, welche im Falle eines aktiven *Heatmap Overlay* ausgeblendet werden. Beim ersten handelt es sich um eine Reihe von farbigen Schaltflächen. Er wird daher mit der Liste der Standardfarben initialisiert und dient zur schnellen Zuordnung des gewählten Autoren zu einer dieser Farben. Der zweite wird mit der aktuellen Farbe des Autors initialisiert und wird so als klassischer *Color Picker* genutzt.

```
const authorButtonPopupContent = (
  ...
  <Button
    as={Link}
    content={'Profile Page'}
    to={`/authors/${this.state.clickedAuthorButtonName}`}
  />
  <span hidden={this.state.isHeatmapActive}>
    <CirclePicker
      colors={DEFAULT_AUTHOR_COLORS}
      onChange={handleColorChange}
    />
    ...
    <Accordion.Content active={this.state.isSketchPickerVisible}>
      <SketchPicker
        color={this.state.clickedAuthorButtonColor}
        onChange={handleColorChange}
      />
    </Accordion.Content>
    ...
  </span>
  ...
);
```

Listing 4.20: Überblick über die Elemente des *Popup* eines *Author Button*

Über die zugehörige Funktion *handleColorChange* wird so die vom Benutzer gewählte Farbe an der entsprechenden Stelle in der Liste für Autoren-Farben gespeichert. Die Funktion ist in Listing 4.21 gezeigt.

```
const handleColorChange = (
  color: any,
) => {
  const clickedAuthorButtonColor = color.hex;
  authorColors[this.state.authorNames.indexOf(this.state.clickedAuthorButtonName)] =
    color.hex;

  this.setState({ clickedAuthorButtonColor, authorColors });
};
```

Listing 4.21: *Handler* für das Auswählen einer Farbe

Kontext Artikel

Im nächsten Schritt wird die Umsetzung der Kontrollkästchen für den Artikel gezeigt. Diese lassen sich in zwei unterschiedliche Arten unterteilen, nämlich diejenigen, die den allgemeinen Stil des Artikels betreffen, und diejenigen, die diverse Berechnungen auf Basis der Knoten des Artikels nach sich ziehen und so unter Umständen andere Farbenzuweisungen bewirken. Listing 4.22 zeigt die Funktionen, die beim Klicken auf die entsprechende *Checkbox* aufgerufen werden. So kann zum einen die Hintergrundfarbe des Artikel invertiert werden und zum anderen kann der Text mit Umrahmungen dargestellt werden. Die jeweiligen Zustandsvariablen werden hier angepasst und anschließend genutzt, um so eine neue *articleStyleId* zu berechnen und in den *State* zu schreiben. Diese ist wichtig für den Stil der Texthervorhebung bei einem *Mouseover Event* und wird später vom *DocumentHistoryDecorator* genutzt. Die Funktion *onClickRadioFormattingElements* schaltet lediglich ein Feature ein, dessen Umsetzung in Abschnitt 4.3.5 genauer beschrieben wird.

```
onClickCheckboxBlackBackground = () => {
  const isBlackBackgroundActive = !this.state.isBlackBackgroundActive;
  const articleStyleId = getArticleStyleId(
    false,
    isBlackBackgroundActive,
    this.state.isTextBordersActive,
  );

  this.setState({ isBlackBackgroundActive, articleStyleId });
};

onClickCheckboxBorder = () => {
  const isTextBordersActive = !this.state.isTextBordersActive;
  const articleStyleId = getArticleStyleId(
    false,
    this.state.isBlackBackgroundActive,
    isTextBordersActive,
  );

  this.setState({ isTextBordersActive, articleStyleId });
};

onClickRadioFormattingElements = () => {
  const isFormattingHistoryActive = !this.state.isFormattingHistoryActive;
  this.setState({ isFormattingHistoryActive });
};
```

Listing 4.22: *Handler* für die Änderung des Artikel-Stils

Die Funktion *renderArticle* aus Listing 4.23 stellt sozusagen das Herz des *DocumentHistoryOverlay* dar. Falls ein Artikel existiert, wird ein neuer *DocumentHistoryDecorator* mit dem aktuellen *State* instanziiert. So kann dieser stets auf den aktuellen *State* der Seite zugreifen. Der *Decorator* wird nun zusammen mit dem Artikel über die *Props* der Komponente

DisplayArticleSimple übergeben, deren Ursprung im *Article Renderer* liegt. Dies hat die Anwendung des Dekorierers auf den Artikel zur Folge.

```
renderArticle = () => {
  if (this.state.article) {
    const decorator: Decorator = new DocumentHistoryDecorator(this.state);

    return (
      <DisplayArticleSimple
        article={this.state.article}
        decorator={[decorator]}
      />
    );
  }
};
```

Listing 4.23: Erzeugen des Dekorierers und Anzeigen des Artikels

Für das Vergrößern und Verkleinern des Textes sind die Funktionen aus Listing 4.24 zuständig. Beim Klick auf den *Button* zum Vergrößern werden der aktuelle Wert der Zustandsvariable *articleZoom* und der Wert für das Intervall der Vergrößerung addiert und das Ergebnis in den *State* geschrieben. Falls das Ergebnis die Grenze für die maximale Vergrößerung überschreitet, wird stattdessen der Grenzwert ind den *State* geschrieben. So wird sichergestellt, dass die Grenzen unabhängig vom Intervall eingehalten werden. Die Verkleinerung des Textes erfolgt analog.

```
onClickButtonZoomIn = () => {
  const value = this.state.articleZoom + ZOOM_INTERVALL;
  const articleZoom = value <= ZOOM_MAX ? value : ZOOM_MAX;
  this.setState({ articleZoom });
};

onClickButtonZoomOut = () => {
  const value = this.state.articleZoom - ZOOM_INTERVALL;
  const articleZoom = value >= ZOOM_MIN ? value : ZOOM_MIN;
  this.setState({ articleZoom });
};
```

Listing 4.24: *Handler* für die Änderung der Textgröße

Kontext Heatmap

Abschließend wird die Umsetzung derjenigen Elemente betrachtet, die dem *Heatmap Overlay* zugeordnet sind. Listing 4.25 zeigt die Funktion, die beim Klicken auf das entsprechende Kontrollkästchen ausgeführt wird und die Heatmap so aktiviert. Die *articleStyleId* muss ebenfalls neu berechnet werden, da sich der Stil der Texthervorhebung bei einem *Mouseover Event* ändern kann.

```
onClickToggleHeatmapOverlay = () => {
  const isHeatmapActive = !this.state.isHeatmapActive;
  const articleStyleId = getArticleStyleId(
    false,
    this.state.isBlackBackgroundActive,
    this.state.isTextBordersActive,
  );

  this.setState({ isHeatmapActive, articleStyleId });
};
```

Listing 4.25 : *Handler* für die Aktivierung des *Heatmap Overlay*

Im Anschluss wird die Legende der Heatmap erzeugt. Listing 4.26 zeigt die grobe Struktur und Umsetzung. Für jede Farbe in der Liste der Heatmap-Farben wird eine Zeile erzeugt. Jede Zeile der Legende besteht dabei zum einen aus einem farbigen Icon, das den entsprechend gefärbten Text des Artikels repräsentiert und zum anderen aus einer zugehörigen Beschreibung beziehungsweise Bedingung für die Zuordnung. Diese wird anhand eines Index und der oberen Grenze beziehungsweise maximalen Anzahl an Commits im Artikel berechnet. Die Berechnung des Index soll erst in Abschitt 4.3.5 im Kontext der Zuweisung von Farben näher betrachtet werden.

```
renderHeatmapLegend = () => {
  const colorsCount: number = HEATMAP_COLORS.length;
  const commitsCount: number = this.state.heatmapLegendBoundary;

  ...

  for (let i = 1; i <= commitsCount; i = i + 1) {
    const index: number = !this.state.isTimeBasedMappingActive ?
    computeHeatmapColorIndexLog(i, colorsCount, commitsCount) :
    computeHeatmapColorIndexLin(i, colorsCount, commitsCount);

    ...
  }

  const heatmapLegend = [];

  for (let i = colorsCount - 1; i >= 0; i = i - 1) {
    ...

    heatmapLegend.push(
      <div key={`heatmapLegend_${i}`}>
        <Icon
          name="square"
          style={{ color: this.state.isHeatmapActive ? HEATMAP_COLORS[i] : '' }}
        />
        <span className="heatmapLegendDescription">
          {range}
          {description}
        </span>
      </div>,
    );
  }

  return heatmapLegend;
};
```

Listing 4.26: Konstruieren der Heatmap-Legende

Um die obere Grenze der Heatmap berechnen zu können, dienen die Funktionen aus Listing 4.27. Da bei der Visualisierung des *Heatmap Overlay* zwischen frequenzbasierter und zeitbasierter Interpretation von Commits im Artikel unterschieden wird, werden die oberen Grenzen ebenso auf unterschiedliche Art und Weise berechnet. Die Funktion *getTimeBasedHeatmapBoundary* errechnet diese, indem sie alle unterschiedlichen *Commit IDs* zählt. Die Funktion *getHeatmapBoundary* hingegen wendet eine Heuristik an. Hier wird in einer Eltern-Kind-Beziehung die längste Sequenz von aufeinanderfolgenden Knoten mit gleichzeitig unterschiedlichen *Commit IDs* ermittelt und anschließend gezählt. Dies geschieht anhand einer rekursiven Funktion, in der jeder Knoten des Artikels durchlaufen wird und die Änderungshäufigkeiten jeder Knoten miteinander verglichen werden. Der zugrundeliegende Algorithmus wird in Abschnitt 4.3.5 näher betrachtet.

```
getTimeBasedHeatmapBoundary = (
  article?: WomDocument,
): number => {
  if (!article) {
    return 0;
  }
  const allCommits = this.getAllCommitsOfArticle(article);
  return allCommits.length;
};

getHeatmapBoundary = (
  article?: WomDocument,
): number => {
  if (!article) {
    return 0;
  }
  const sequence: string[] = this.getLongestDistinctiveCommitSequenceOfArticle(article);
  return sequence.length;
};

getLongestDistinctiveCommitSequenceOfArticle = (
  article: WomDocument|WomElement,
): string[] => {
  let commits: string[] = [];
  article.childNodes.forEach((child) => {
    if (child instanceof WomElement) {
      commits = getChangeFrequenciesOfSurroundingElements(
        child as WomElement,
        commits,
        this.state.isFormattingHistoryActive,
      );

      const compare = this.getLongestDistinctiveCommitSequenceOfArticle(child);
      commits = compare.length > commits.length ? compare : commits;
    }
  });

  return commits;
};
```

Listing 4.27: Ermitteln aller Grenzen der Heatmap

Die Umsetzung der Kontrollkästchen um die Interpretation von Commits im Artikel zu ändern wird in Listing 4.28 gezeigt. Diese machen Ge-

brauch von den Funktionen zur Berechnung der oberen Grenze der Heatmap und schreiben anschließend den neuen Wert in den *State*.

```
onClickRadioComputationMode = () => {
  const isTimeBasedMappingActive = !this.state.isTimeBasedMappingActive;
  const heatmapLegendBoundary = !this.state.isTimeBasedMappingActive ?
    this.getTimeBasedHeatmapBoundary(this.state.article) :
    this.getHeatmapBoundary(this.state.article);

  this.setState({ isTimeBasedMappingActive , heatmapLegendBoundary });
};
```

Listing 4.28: *Handler* für die Änderung der Heatmap-Art

4.3.3 DocumentHistoryDecorator

In diesem Abschnitt wird die Implementierung und Funktionsweise des *DocumentHistoryDecorator* beschrieben. Wie bereits erwähnt, wird der *Decorator* mit dem *State* des *DocumentHistoryOverlay* initialisiert. Listing zeigt den Konstruktor.

```
constructor(historyOverlayState: HistoryOverlayState) {
  super();
  state = historyOverlayState;
}
```

Listing 4.29: Implementierung des Konstruktors

Die Klasse ist von *Decorator* abgeleitet und implementiert deshalb die Funktion *applyTo*, welche in Listing 4.30 abgebildet ist. Diese wird für jedes einzelne Element des Artikels ausgeführt. Die Hauptaufgabe des *DocumentHistoryDecorator* ist recht simpel und besteht aus dem Überschreiben der *Props* dieser Elemente. Zum einen wird *style.color* und zum anderen wird *className* neu gesetzt. Hierfür muss die neue Farbe des Elements und auch die *articleStyleId* zuvor berechnet werden.

```
applyTo(
  props: ReactArticleElementProps,
  createElement: DecoratorReturnType,
): DecoratorReturnType {
  ...

  overrideReactProps(
    props, {
      style: {
        color: elementColor,
      },
      className: articleStyleId,
    },
  );

  return DocumentHistoryDecorator.wrap(createElement);
}
```

Listing 4.30: Überschreiben der *Props* von Artikel-Elementen

Listing 4.31 zeigt den zugehörigen Code der Berechnungen. Bei der Ermittlung der Farbe wird unterschieden, ob das *Heatmap Overlay* aktiv ist. So wird entweder *getAuthorColor* oder *getHeatColor* aufgerufen, um die Berechnung durchzuführen. Die beiden Funktionen werden in Abschnit 4.3.5 detaillierter betrachtet. Dennoch benötigen sie als Parameter *commitId* und *projectId* des Elements. Über die *Props* kann ein Konstrukt vom Typ *WomElement* erhalten werden, und falls es sich um ein dementsprechendes Textelement handelt, können so die benötigten Attributewerde aus dem *HIST* Namensraum gelesen und so genutzt werden. Die Attributwerte können hier nicht nur aus Textelementen, sondern auch aus formatierenden Elemente gelesen werden. In diesem Fall ist ein Feature aktiviert, das über die Zustandsvariable *isFormattingHistoryActive* eingeschaltet werden kann. Die Umsetzung dieses Features wird ebenso in einem späteren Abschnitt beschrieben.

Die *articleStyleId* wird als Wert für das Attribut *className* genutzt, um ein Element mittels des Klassen-Selektors konkreten Stylesheet-Regeln in der für sie vorgesehenen CSS-Klasse zuordnen zu können. Die Pseudoklasse *:hover* wird hier genutzt, um eine Markierung des aktuellen Textelements zu visualisieren. Dies geschieht beim Bewegen des Mauszeigers über den zugehörigen *Author Button,* der diejenige Farbe enthält, die im vorherigen Schritt berechnet wurde. Die Markierung selbst wird realisiert, indem eine Hintergrundfarbe für das Textelement gesetzt wird. Die Darstellung dieser Visualisierung ändert sich sobald die Hintergrundfarbe des Artikels geändert oder Textrahmen ein- und ausgeblendet werden. Aufgrund der vorherigen Erläuterungen macht die Berechnung der *articleStyleId* so Gebrauch von der Zustandsvariable *isAuthorButtonHovered* und der Methode *getArticleStyleId.* Diese Funktion erzeugt, basierend auf den Werten der Zustandsvariablen *isBlackBackgroundActive* und *isText-*

BordersActive, einen String, welcher in der CSS-Klasse als *ID* für den Stil des Textelements beziehungsweise der Markierung fungiert.

```
const womElem: WomElement = getWomElement(props);

if (isWomText(womElem)) {
  let commitId: string|undefined = womElem.getAttributeNS(HIST_NS, 'commitId');
  let projectId: string|undefined = womElem.getAttributeNS(HIST_NS, 'projectId');

  const commit = state.isFormattingHistoryActive ?
    getLatestFormattingParentCommit(womElem) :
    null;

  commitId = (commit === null) ? commitId : commit.name;
  projectId = (commit === null) ? projectId : commit.project;

  const authorName: string = (commitId && projectId) ? getAuthorName(commitId, projectId) : '';
  const authorColor = getAuthorColor(authorName, state.authorNames, state.authorColors);

  elementColor = (commitId && projectId) ? (
    !state.isHeatmapActive ?
      authorColor :
      getHeatColor(
        commitId,
        projectId,
        womElem,
        state.isTimeBasedMappingActive,
        state.heatmapLegendBoundary,
        state.isFormattingHistoryActive,
      )
  ) :
  '';

  articleStyleId =
    state.isAuthorButtonHovered ? (
      (state.hoveredAuthorButtonColor === authorColor) ?
        state.articleStyleId :
        getArticleStyleId(false, state.isBlackBackgroundActive, state.isTextBordersActive)
    ) :
    state.articleStyleId;
}
```

Listing 4.31: Ermitteln der Farbe und Style-ID eines Artikel-Elements

Sobald die *Props* mit den soeben berechnet Werten überschrieben wurden, wird das zu erzeugende Element der *wrap* Funktion aus Listing 4.32 übergeben. Die Komponente *HistoryWrapper* dient hier als Wrapper für das Element. Über die *Props* erhält sie ebenso den *historyOverlayState* und wird schlussendlich zurückgegeben.

```
static wrap(
  createElement: DecoratorReturnType,
) {
  return (props: ReactArticleElementProps) => (
    <HistoryWrapper historyOverlayState={state}>
      {createElement(props)}
    </HistoryWrapper>
  );
}
```

Listing 4.32: Implementierung der *wrap Methode*

4.3.4 HistoryWrapper

Im folgenden Abschnitt wird die Umsetzung des *HistoryWrapper* veranschaulicht. Dieser erhält über die *Props* zum einen Kindelemente beziehungsweise das eine Kindelement, für das es als Wrapper fungieren soll und und zum anderen den *historyOverlayState*. Zugehöriger *PropType* ist in Listing 4.33 dargestellt.

```
interface HistoryWrapperPropType {
  children: React.ReactElement<any>;
  historyOverlayState: HistoryOverlayState;
}
```

Listing 4.33: Implementierung des *HistoryWrapperPropType* Interface

Die Aufgabe Hauptaufgabe hier ist das Anzeigen eines *Popup* bei Klick auf das Kindelement des Wrappers (Text). Im folgenden wird auf die zugehörige Bestandteile, die in Listing 4.34 zu sehen sind, eingegangen.

```
render(): React.ReactChild|null {
  if (this.child == null) {
    return null;
  }

  ...

  return(
    <Popup
      content={renderedPopupContent}
      disabled={commitId == null}
      inverted={!this.props.historyOverlayState.isBlackBackgroundActive}
      on="click"
      onMount={onMountHandler}
      trigger={renderedPopupTrigger}
    />
  );
}
```

Listing 4.34: Implementierung des *Popup*

Der Auslöser für das *Popup* wird in der Funktion *renderPopupTrigger* implementiert. Wie in Listing 4.35 zu sehen ist, wird lediglich das Kindelement mit einem weiteren Element vom Typ *span* umspannt, welches zusätzliche *Event Handler Properties* enthält.

```
renderPopupTrigger = (
  textElement: React.ReactElement,
  commitId: string|undefined,
  projectId: string|undefined,
) => {
  const onMouseEnterHandler = () => this.onMouseEnterHandler(commitId, projectId);
  const onMouseLeaveHandler = () => this.onMouseLeaveHandler();

  return (
    <span
      onClick={this.onClickHandler}
      onMouseEnter={onMouseEnterHandler}
      onMouseLeave={onMouseLeaveHandler}
    >
      {textElement}
    </span>
  );
};
```

Listing 4.35: Implementierung des Auslösers für das *Popup*

Durch diese wird nun die Behandlung von *Mouse Events* ermöglicht, die in Listing 4.36 zu sehen ist. Wie in Abschnitt 4.3.2 bereits erwähnt, wird zum einen so die Animation des Icons des zugehörigen *Author* Buttons realisiert. Dies geschieht indem der Autorenname als Wert in die Variable *authorHovered* geschrieben wird und diese anschließend vom *DocumentHistoryOverlay* abgerufen wird. Der Autorenname wiederum kann mithilfe der übergebenen *commitId* und *projectId* ermittelt werden. Zum anderen wird die *preventDefault* Methode auf dem Event aufgerufen, die entsprechende Aktionen des Browsers verhindert. So hat beispielsweise der Klick auf einen Link oder auch ein Rechtsklick auf Text im Artikel keine Wirkung.

```
onMouseEnterHandler = (
  commitId: string|undefined,
  projectId: string|undefined,
) => {
  authorHovered = (commitId && projectId) ? getAuthorName(commitId, projectId) : '';
};

onMouseLeaveHandler = () => {
  authorHovered = '';
};

onClickHandler = (
  e: React.MouseEvent<HTMLSpanElement, MouseEvent>,
) => {
  e.preventDefault();
};
```

Listing 4.36: *Handler* für die *Mouse Events* auf den Artikel-Elementen

Der eigentliche Inhalt des *Popup* wird nun betrachtet. Wie in Listing 4.37 zu sehen ist, beinhaltet er neben einer Überschrift sowohl ein *Segment* für alle Commits der Historie, als auch eines für das referenzierte Textelement, auf das geklickt wurde. Um dieses *referencedElement* zu erhalten, wird ein neues *React Element* mithilfe der Methode *createElement* er-

zeugt. So wird als Typ des Elements *div* gewählt. Für die Textfarbe wird die *style.color* Eigenschaft, und für den Text selbst wird der *textContent* des aktuellen *WomElements* direkt übernommen.

```
renderPopupContent = (
  referencedElement: React.ReactElement,
  commitId: string|undefined,
  projectId: string|undefined,
) => {
  ...

  return (
    <Container>
      <Header content={'Commit History'}/>
      <Segment content={completeContent}/>
      <Segment
        content={referencedElement}
        inverted={this.props.historyOverlayState.isBlackBackgroundActive}
      />
    </Container>
  );
};
```

Listing 4.37: Überblick über die angezeigten Elemente des *Popup*

Listing 4.38 zeigt den Aufbau der angezeigten Commit-Historie. Dabei wird jeder Eintrag als eigene *Grid.Row* realisiert und erhält eine sogenannte Zeilen-ID, die die aktuelle *Commit ID* enthält. Innerhalb dieser Zeile wird zuerst ein *Button* mit der Farbe *authorColor* als Link zur *authorUri* (Profilseite) eingefügt. Des weiteren wird die Komponente *Moment* genutzt, um den Zeitstempel eines Commits benutzerfreundlich und in relativer Form anzuzeigen. Ebenso erfolgt über *localeCommitDate* die absolute Zeitangabe. Schließlich werden noch Commit-Nachricht in Form eines *Segment* und ein Link zur Commit-Seite in Form eines *Button* dargestellt. Eine zusätzliche visuelle Abgrenzung der einzelnen Einträge wird durch einen *Divider* inklusive eines Icons geschaffen.

```
completeContent.push(
  ...
  <Grid.Row id={`popupEntries_${commitHistory[i].name}`}>
    ...
    <Button
      as={Link}
      content={commitHistory[i].author}
      style={{ color: authorColor }}
      to={authorUri}
    />
    ...
    <Moment locale={LOCALE} fromNow>{commitDate}</Moment>
    {localeCommitDate}
    ...
    <Segment content={commitHistory[i].commitMessage}/>
    ...
    <Button
      as={Link}
      content={'Commit Page'}
      to={commitUri}
    />
    ...
  </Grid.Row>
  <Divider content={dividerIcon}/>
  ...
);
```

Listing 4.38: Implementierung eines Eintrags in der Commit-Historie

Die Ermittlung aller soeben erwähnten Variablen ist in Listing 4.39 abgebildet und erfolgt mithilfe der *commitHistory* Liste. Sie enthält alle Commits, die zeitlich bis zu dem aktuellen Commit erfolgt sind. Die Liste lässt sich - wie auch ein einzelnes Commit-Objekt beziehungsweise dessen Eigenschaften – über eine REST Schnittstelle ermitteln. Dabei werden *commitId* und *projectId* übergeben. Die Liste der Commits wird nun durchlaufen. Die *commitUri* beziehungsweise *authorUri* kann jeweils direkt aus der *name* beziehungsweise *author* Eigenschaften des Commits abgeleitet werden. Ebenso können *commitMessage* und *timestamp* direkt aus den Commit-Eigenschaften gelesen werden. Mittels *getAuthorColor* lässt sich wieder die Autoren-Farbe ermittlen, die dem ersten Button zugeordnet wird.

```
const commitHistory: CommitType[] = receiveCommitHistory(commitId, projectId);
const completeContent = [];

for (let i = 0; i < commitHistory.length; i = i + 1) {
  const commitDate: number = (commitHistory[i].timestamp * 1000);
  const localeCommitDate: string = new Date(commitDate).toLocaleString(LOCALE);

  const commitUri: string = `/commits/${commitHistory[i].name}`;

  const authorUri: string = `/authors/${commitHistory[i].author}`;
  const authorColor: string =
    !this.props.historyOverlayState.isHeatmapActive ?
    getAuthorColor(
      commitHistory[i].author,
      this.props.historyOverlayState.authorNames,
      this.props.historyOverlayState.authorColors,
    ) :
    '';

  const dividerIcon =
    (i + 1 !== commitHistory.length) ? <Icon name="chevron down"/> : null;

  completeContent.push(...);
}
```

Listing 4.39: Ermitteln aller für einen Eintrag relevanten Konstrukte

Der *onMountHandler* aus Listing 4.40 ist der letzte Bestandteil des *Popup* und wird beim Mounten aufgerufen. Mithilfe der zugeteilten Zeilen-ID kann der letzten Eintrag der Commit-Historie in Form des zugehörigen HTML-Elements ermittelt werden. Die Methode *scrollIntoView* scrollt dieses nun in den sichtbaren Bereich des *Popup*.

```
onMountHandler = (
  commitId: string|undefined,
) => {
  const latestCommit = commitId ? document.getElementById(`popupEntries_${commitId}`) : null;
  if (latestCommit != null) {
    latestCommit.scrollIntoView({ block: 'start', behavior: 'smooth' });
  }
};
```

Listing 4.40: *Handler* für das Scrollen zum letzten Eintrag

4.3.5 Algorithmen

Abschließend werden diejenigen Algorithmen betrachtet, die in vorherigen Ausführungen zwar erwähnt wurden, sich jedoch keiner bestimmten Klasse zuordnen lassen. Dabei werden diese hinsichtlich ihrer Funktionalitäten gruppiert, nämlich der Zuweisung von Farben und der Berücksichtigung der Formatierung.

Zuweisung von Farben

Die Textelemente des Artikels werden je nach Situation entweder der Liste für Autoren-Farben oder der für Heatmap-Farben zugewiesen. Die folgenden beiden Funktionen werden genutzt, um einen Farbwert in hexadezimaler Farbdefinition zurückzugeben. Listing 4.41 beinhaltet die Funktion zur Ermittlung der Autoren-Farbe. Hier werden die Autorenliste und die Farbenliste aufeinander abgebildet. Aufgrund des so geteilten Index kann anhand eines einzelnen Autorennamens die zugehörige Farbe ermittelt werden. Für die Zuordnung wurden bewusst zwei Listen statt einer einzigen gewählt, da auf die Farbenliste noch weitere Operationen ausgeführt werden können. So kann deren Komplexität so niedrig wie möglich gehalten werden.

```
export function getAuthorColor(
  authorName: string,
  authorNames: string[],
  authorColors: string[],
): string {
  const index: number|null = authorNames ? authorNames.indexOf(authorName) : null;
  const authorColor: string = index !== null ? authorColors[index] : 'black';
  return authorColor;
}
```

Listing 4.41: Ermitteln der Autoren-Farbe

Listing 4.42 zeigt die Funktion zum Ermitteln der Heatmap-Farbe eines Textelements. Dabei wird mittels eines Index die Farbe aus der Liste der Heatmap-Farben gelesen. Dieser Index wird mithilfe von drei Werten berechnet, nämlich der Anzahl aller unterschiedlichen Commits, der Commit-Häufigkeit und der Anzahl der verfügbaren Farben. Die Anzahl der Commits im Artikel entspricht der *heatmapLegendBoundary*, welche der Funktion bereits übergeben wurde. Auch die Anzahl verfügbarer Farben lässt sich relativ schnell ableiten, indem die Länge der Heatmap-Farbenliste genommen wird. Die Commit-Häufigkeit unterscheidet zwischen den Heatmap-Arten und ergibt sich wie folgt: Bei einer zeitbasierten Heatmap entspricht die Häufigkeit der Länge der Commit-Historie des aktuellen Commits. Bei der frequenzbasierten Heatmap hingegen wird eine Heuristik angewendet.

```
export function getHeatColor(
  commitId: string,
  projectId: string,
  womElem: WomElement,
  isTimeBasedMappingActive: boolean,
  heatmapLegendBoundary: number,
  isFormattingHistoryActive: boolean,
): string {
  const frequencies: string[] = [];
  frequencies.push(receiveCommit(commitId, projectId).name);

  const commitFrequency: number = isTimeBasedMappingActive ?
    receiveCommitHistory(commitId, projectId).length :
    getChangeFrequenciesOfSurroundingElements(
      womElem,
      frequencies,
      isFormattingHistoryActive,
    ).length;

  const colorsCount: number = HEATMAP_COLORS.length;
  const commitsCount: number = heatmapLegendBoundary;

  const heatColor: string = !isTimeBasedMappingActive ?
    HEATMAP_COLORS[computeHeatmapColorIndexLog(commitFrequency, colorsCount, commitsCount)] :
    HEATMAP_COLORS[computeHeatmapColorIndexLin(commitFrequency, colorsCount, commitsCount)];

  return heatColor;
}
```

Listing 4.42: Ermitteln der Heatmap-Farbe

Die Funktion aus Listing 4.43 implementiert diese Heuristik. Dabei handelt es sich um eine rekursive Funktion, die alle Nachbarn und Eltern eines Knoten durchläuft und sämtliche *Commit IDs* in eine Liste schreibt. Dabei wird der Wert von wiederholt auftretenden IDs nur ein einziges mal geschrieben. Mithilfe der Länge dieser Liste lässt sich so eine wahrscheinliche Aussage über die Commit-Häufigkeit treffen.

```
export function getChangeFrequenciesOfSurroundingElements(
  womElem: WomElement,
  commits: string[],
  isFormattingHistoryActive: boolean,
): string[] {
  const frequencies = commits;

  ...

  const commit = receiveCommit(commitId, projectId);
  if (!frequencies.includes(commit.name)) {
    if (formattingElementsAllowed.includes(womElem.localName) || isWomText(womElem)) {
      frequencies.push(commit.name);
    }
  }

  const parent: WomElement = womElem.parentNode as WomElement;

  if (!(parent instanceof WomElement)) {
    return frequencies;
  }

  const siblings: WomElement[] = parent.childNodes as WomElement[];

  ...

  return getChangeFrequenciesOfSurroundingElements(parent, frequencies, isFormattingHistoryActive);
}
```

Listing 4.43: Ermitteln der Commit-Häufigkeit

Mittels der drei ermittelten Werte kann nun der Index berechnet werden, der benötigt wird, um die entsprechende Heatmap-Farbe aus der Liste zu erhalten. Listing 4.44 zeigt die zugehörige logarithmische beziehungsweise lineare Funktion, welche im Kontext einer frequenzbasierten beziehungsweise zeitbasierten Heatmap aufgerufen wird. Die abgebildeten Formeln wurden so hergeleitet, dass jeder Häufigkeit beziehungsweise jedem unterschiedlichem Commit eine Farbe zugeordnet werden kann, und dabei auch alle vorhandenen Farben zu benutzt werden. Im Detail wird also bei beiden Funktionen an einen Grenzwert angenähert, der der Länge der Farbenliste minus eins entspricht, um so auf einen geeigneten Index abzubilden.

```
export function computeHeatmapColorIndexLog(
  commitFrequency: number,
  colorsCount: number,
  commitsCount: number,
): number {
  const index = (commitsCount !== 1) ?
    Math.round(Math.log2(commitFrequency) * (colorsCount - 1) / Math.log2(commitsCount)) :
    colorsCount - 1;

  return (index > 0) ? index : 0;
}

export function computeHeatmapColorIndexLin(
  commitFrequency: number,
  colorsCount: number,
  commitsCount: number,
): number {
  const index = (commitsCount !== 0) ?
    Math.round(commitFrequency / commitsCount * colorsCount - 1) :
    colorsCount - 1;

  return (index > 0) ? index : 0;
}
```

Listing 4.44: Ermitteln des Index zur Auswahl von Heatmap-Farben

Abbildung 4.1 zeigt die resultierenden Funktionsgraphen anhand eines Beispiels. Auf der X-Achse wird die Commit-Häufigkeit beziehungsweise der x-te Commit, und auf der Y-Achse der Index abgebildet. Es gibt sechs Heatmap-Farben und den Elementen des Artikels wurden insgesamt sechs unterschiedliche *Commit IDs* zugeordnet.

Die linke Abbildung zeigt den Funktionsgraphen der logarithmischen Funktion. Dieser repräsentiert daher die frequenzbasierte Heatmap. So ergibt sich bei einer Commit-Häufigkeit von sechs Commits ein Index mit dem Wert „5“. In der Farbenliste entspricht dies der letzte Stelle. An dieser ist die dunkelste Farbe der Heatmap gespeichert.

Bei der rechten Abbildung ist der Funktionsgraph der linearen Funktion zu sehen. Er repräsentiert somit die zeitbasierte Heatmap. Dem ersten Commit im Artikel wird der Index mit dem Wert „0“ zugewiesen. In der zugehörigen Farbenliste ist an dieser Stelle die hellste Farbe gespeichert.

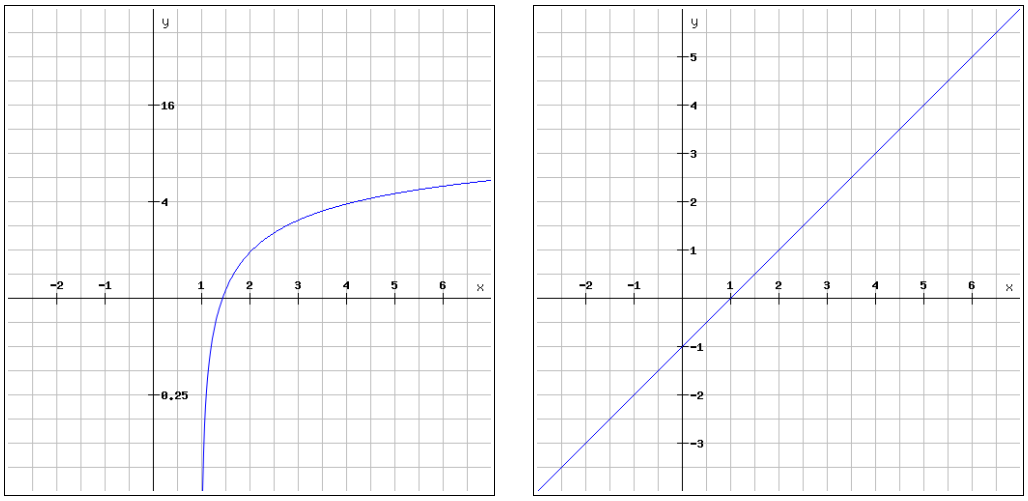

Abbildung 4.1: Resultierende Funktionsgraphen des Beispiels

Berücksichtigung der Formatierung

Um zuletzt auch textformatierende Elemente des Artikels für die Zuweisung von Farben berücksichtigen zu können, wird die rekursive Funktion aus Listing 4.45 benötigt. Hier werden alle textformatierenden Elternknoten eines Textelements durchlaufen und die Zeitstempel untereinander verglichen. Die c*ommitId* und *projectId* des textformatierenden Elements mit dem höchsten aller Zeitstempel wird anschließend für die Erzeugung eines Commit-Objekts genutzt. Für das initial übergebene Textelement wird anstatt des eigentlichen Commits nun der Commit des textformatierenden Elternknoten übergeben, der zu einem späteren Zeitpunkt gemacht wurde. Aufgrund der Tatsache, dass sich die Zuweisung von Farben auf Commit-Objekte stützt, wird dem Textelement später so eine andere Farbe zugewiesen. Das sogenannte textformatierendes Element wird definiert durch eine Liste, in welcher der jeweilige *localname* aller zu berücksichtigenden *Nodes* gespeichert ist. Wenn die Liste also den *String* „b“ enthält, so werden alle <*b*> Elemente berücksichtigt. Diese Liste besteht aktuell aus den gängigen HTML-Elementen, kann aber grundsätzlich um beliebige WOM-Elemente erweitert werden.

```
export function getLatestFormattingParentCommit(
  womElem: WomElement,
): CommitType|null {
  ...

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

  if (parentCommit == null) {
    return currentCommit;
  }

  const formattingElementsAllowed = getFormattingElementsAllowed(true);

  if (isWomText(womElem) || formattingElementsAllowed.includes(womElem.localName)) {
    if (formattingElementsAllowed.includes(parent.localName)) {
      const currentCommitTime = currentCommit.timestamp;
      const parentCommitTime = parentCommit.timestamp;
      currentCommit = (currentCommitTime > parentCommitTime) ? currentCommit : parentCommit;
    }

    parentCommit = getLatestFormattingParentCommit(parent);
    if (parentCommit !== null) {
      const currentCommitTime = currentCommit.timestamp;
      const parentCommitTime = parentCommit.timestamp;
      currentCommit = (currentCommitTime > parentCommitTime) ? currentCommit : parentCommit;
    }

    return currentCommit;
  }

  return getLatestFormattingParentCommit(parent);
}
```

Listing 4.45: Ermitteln des neusten textformatierenden Elements

5 Evaluation

Gegenstand dieses Kapitels ist die Evaluation aller Ergebnisse hinsichtlich der definierten Anforderungen aus Kapitel 2. Zu diesem Zweck werden die jeweiligen Abnahmekriterien noch einmal betrachtet.

Wie die erfolgreiche Implementierung des Prototypen in Kapitel 4 gezeigt hat, konnte die Visualisierung der Historie eines Dokuments realisiert werden. Durch das Zusammenspiel der entwickelten beziehungsweise angepassten Klassen können Commit-Informationen sinnvoll verarbeitet werden. So kann die Historie des Artikels nicht nur über die Farbzuweisung von Text und Autor dargestellt werden, sondern auch mittels einer zeitbasierten Heatmap.

Das Klicken von Text hat die Anzeige von zusätzlichen Informationen zur Folge. Dies wurde anhand eine Popups umgesetzt, das im *DocumentHistoryWrapper* implementiert wurde. Es enthält neben der eigentlichen Historie und den geforderten Informationen über Commit-Nachricht und -Zeitpunkt auch eine Zuweisung zu entsprechenden Autoren.

Zusätzlich Statistiken können in Form einer Heatmap dargestellt werden. Mithilfe der bereitgestellten dynamischen Legende können so Rückschlüsse auf Änderungshäufigkeiten und -zeitpunkte bestimmter Textabschnitte gezogen werden.

Durch die Platzierung von Links in die entsprechenden Autoren-Schaltflächen und Popups von Textelementen konnte eine schnelle und praktikable Navigation zu weiterführenden und relevanten Informationen realisiert werden.

Aufgrund der Nutzung des Frameworks Semantic UI React konnte der Prototyp responsiv implementiert werden. Das Aussehen und auch die Handhabung der Benutzeroberfläche wurde, wie in Kapitel 3.4 ausführlich beschrieben, einheitlich gestaltet.

Ebenso das Design der Benutzerschnittstelle ist nach besten Bemühungen gestaltet. So wurde ein ausführliches UX-Design entworfen und es wurden auch zahlreiche Aspekte berücksichtigt, um die Barrierefreiheit zu maximieren.

Auch eine leichte Anpassung des Designs ist gewährleistet indem sämtliche Darstellungen der Inhalte in eine CSS-Klasse ausgelagert wurden. Die Ausnahme bilden einige wenige Elemente, die bedingtes Rendern ausnutzen müssen, um ihre Funktionalität zu erfüllen. Nichtsdestotrotz ist der Stil der HTML-Elemente des User Interface leicht änderbar und kann leicht an neue Gegebenheiten angepasst werden.

Um alle Aussagen bezüglich der Implementierung sowohl des Backends als auch des Frontends zu stützen, kann versichert werden, dass alle bisherigen Tests erfolgreich durchgelaufen sind und alle Funktionalitäten ihren beabsichtigten Zweck auch erfüllen.

Zusammenfassend lässt sich nun sagen, dass alle Anforderungen an das *Document History Overlay* erfüllt wurden und dieses erfolgreich realisiert wurde.

6 Zusammenfassung und Ausblick

Ausgangsgegenstand dieser Arbeit war die webbasierte Kollaborationsplattform Sweble, die sich mitunter dadurch auszeichnet, dass sie Inhalte nicht in textueller Form ihrer jeweiligen Ausgangssprache, sondern in einer standardisierten Baumstruktur speichern kann. Als übergeordnetes Ziel wurde darauf aufbauend zunächst das Potential dieser graphbasierten Speicherung erforscht, um letztlich das gemeinsame Wissensmanagement in Sweble mithilfe neuer Funktionen zu optimieren. Damit kollaboratives Arbeiten an Artikeln ein Maximum an Produktivität erreichen kann, ist es essentiell, dass die genutzte Software über eine bloße Bereitstellung gemeinsam bearbeitbarer Dokumente hinausgeht und dem Nutzer eine intuitiv bedienbare Oberfläche zum Abrufen zusätzlicher Informationen über das jeweilige Dokument bietet.

Die generelle Architektur und das Design wurden deshalb im Rahmen dieser Arbeit an die speziellen Anforderungen angepasst und unterstützen nun die einfache Handhabung der Kollaborationsplattform. So wurde gezeigt, wie das Backend Informationen über die Historie eines Dokuments in einem geeigneten Format speichern muss, damit das Frontend diese in einem weiteren Schritt für den Nutzer sinnvoll verarbeiten und visuell darstellen kann.

Des Weiteren wurde ein Design-Prototyp entwickelt und anhand dessen näher auf Aspekte bezüglich UI/UX eingegangen. Im wesentlichen kann nun jeder Nutzer Informationen darüber einholen, wer das Dokument zu welchem Zeitpunkt bearbeitet hat und welche Änderungen dabei am Text vorgenommen wurden. Um dem Nutzer eine grundlegende Orientierung über alle Kollaborationspartner zu geben, wurde die Kennzeichnung aller Autoren mittels individueller Farben ermöglicht. Bei der Wahl der Farbpalette wurde darauf geachtet, dass der Kontrast von Vorder- und Hintergrund auf einem möglichst hohen Niveau bleibt, damit so die Verwen-

dung der Software möglichst barrierefrei ist und Probleme in der Farbwahrnehmung individueller Nutzer kompensiert werden. Autoren-Farben können entweder per Mausklick automatisch verteilt oder aber über den Aufruf der Farbpalette individuell zugeordnet und getauscht werden. Im Text werden die Passagen entsprechend der letzten Änderung eingefärbt, sodass der Nutzer auf einen Blick erkennen kann, wer die Passage zuletzt angepasst hat. Das ganze wird zusätzlich durch visuelle Effekte unterstützt, um die Barrierefreiheit zu maximieren. Die Benutzeroberfläche wurde dabei so gestaltet, dass der Nutzer neben dem Abrufen reiner Verlaufsinformationen per Mausklick auch zu anderen Elementen navigieren kann. So kann beispielsweise die Profilseite des jeweiligen Autors aufgerufen werden oder zur Commit-Seite navigiert werden. Darüberhinaus erlaubt es die Benutzeroberfläche Rückschlüsse über die Änderungshäufigkeit einer bestimmten Passage zu ziehen. Durch die Implementierung einer Heatmap können besonders intensiv bearbeitete Textstellen leicht ausfindig gemacht und bei Bedarf überprüft werden. Die Farbpaletten der Heatmap und der Autoren-Farben entsprechen einer hohen Barrierefreiheit und sind damit für beinahe alle individuellen Nutzer der Sweble-Software uneingeschränkt verwendbar.

Bei allen oben beschriebenen Funktionalitäten stand nicht nur die technische Implementierung selbst im Fokus dieser Arbeit, sondern auch ihre möglichst benutzerfreundliche Umsetzung. Durch die Erweiterungen soll Sweble allen Nutzern eine intuitive und möglichst simple Benutzeroberfläche bieten, die auch ohne spezifische Vorkenntnisse bedienbar sein soll. Durch einfache Interaktionen soll ein Maximum an Informationen zur Verfügung stehen.

Die Evaluation der Arbeit hat gezeigt, dass alle Erweiterungen erfolgreich in die Software implementiert werden konnten. Insgesamt sind keine Einschränkungen ihrer Funktionsweise zu dokumentieren und alle Tests verliefen positiv.

Für zukünftige Arbeiten könnte man sich ebenso einige sinnvolle Erweiterungen vorstellen. So könnte zum einen das *Document History Overlay* ohne zugehörige Seite und nur über einen Dekorierer realisiert werden. Im Detail könnte so die ganze Logik direkt auf den visualisierten Artikel aufsetzen und somit jederzeit, auch in anderen Kontexten, flexibel ein- und ausgeblendet werden. Die Anzeige der Autoren und auch die zahlreichen Konfigurationsmöglichkeiten könnten hier beispielsweise in ein weiteres Popup beziehungsweise Fenster verlagert werden, welches geöffnet bleibt und vom Nutzer auch positioniert werden kann. Zum anderen könnte man auch eine alternative Berechnung der Änderungshäufigkei-

ten der Textelemente in Betracht ziehen, welche von der frequenzbasierten Heatmap genutzt wird. Die hierfür verwendete Heuristik könnte beispielsweise durch eine Erweiterung des WOM-Formats ersetzt werden, bei dem der Pfad zu einem Textknoten im Baum immer mitgespeichert wird. Beim Vergleich mit vorheriger Revision könnte so die daraus resultierende Abweichung genutzt werden, um so die Änderungshäufigkeit der Knoten zuverlässig bestimmen zu können.

Nichtsdestotrotz ist es bereits in der jetzigen Ausgestaltung gelungen, kooperatives Arbeiten mit Sweble auf eine völlig neue Ebene zu heben.

Anhang A UI-Beispiele

Folgende Beispiele zeigen noch einmal die Benutzerschnittstelle des implementierten Prototypen in zwei unterschiedlichen Anwendungsszenarien. Dabei wird im Gegensatz zu Abschnitt 3.4 nun die vollständige Benutzeroberfläche mit all ihren Elementen betrachtet, um so alle Funktionalitäten im direkten Zusammenspiel darstellen zu können.

Abbildung 7.1 zeigt, wie der Nutzer mit der Maus über den Autoren-Button fährt und die entsprechenden Textstellen des Autors markiert werden.

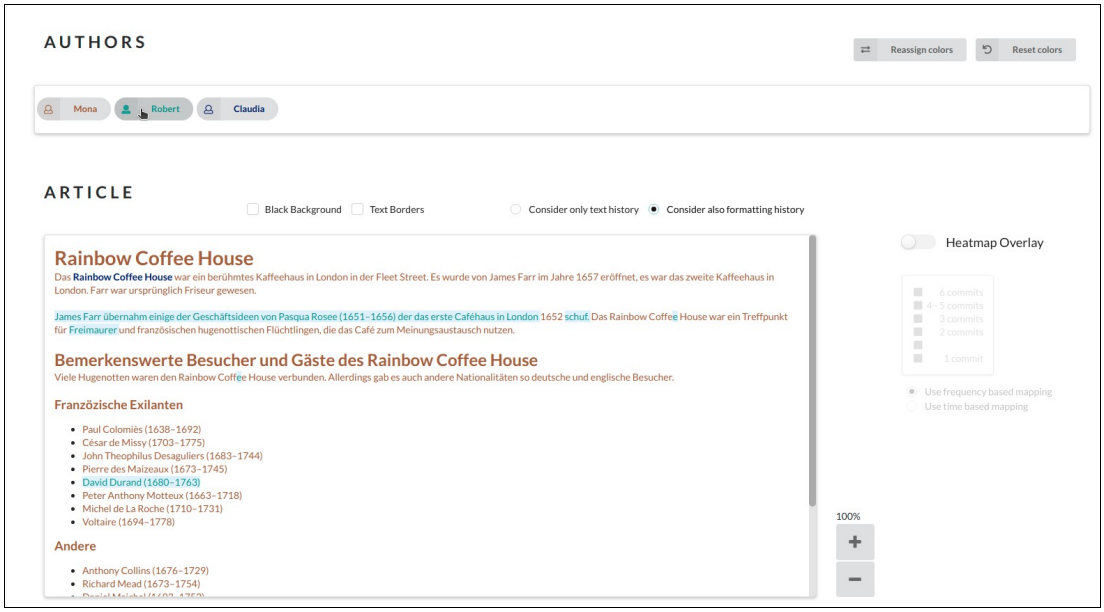

Abbildung 7.1: *Document History Overlay*

Abbildung 7.2 zeigt, wie der Nutzer bei aktiver Heatmap mit der Maus über eine Textstelle des Artikes fährt. Der Autoren-Button, der dieser Stelle zugewiesen ist, wird visuell hervorgehoben. Des Weiteren hat der Nutzer den Artikel verkleinert und den Stil angepasst, um einen besse-

ren Gesamteindruck der Änderungshäufigkeiten aller Textstellen zu erhalten.

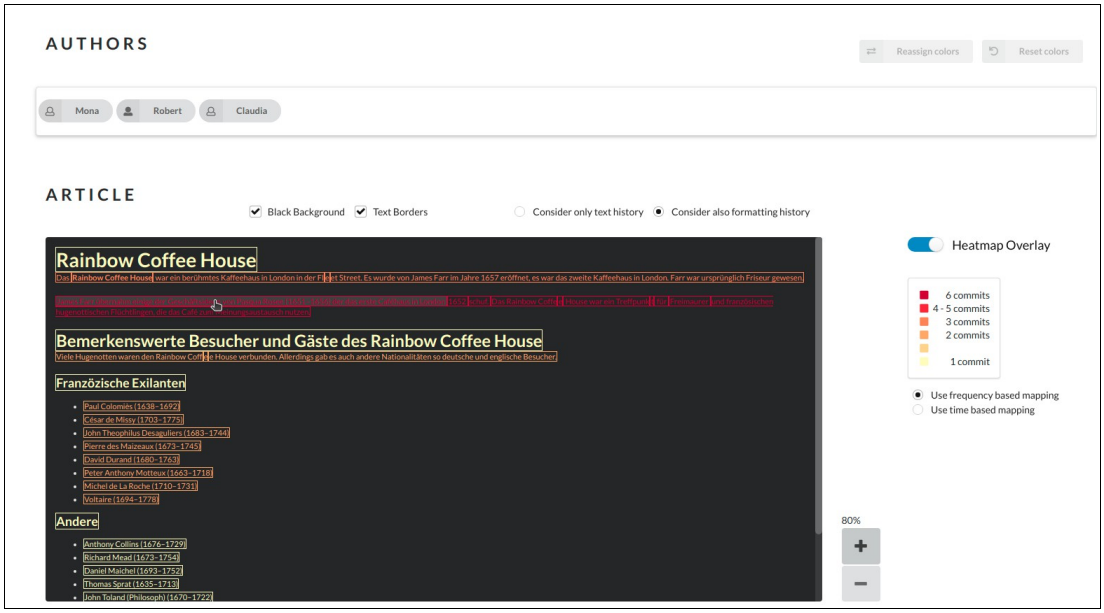

Abbildung 7.2: *Heatmap Overlay*

Abbildung 7.3 zeigt das zugehörige Popup nach einem Klick auf das markierte Textelement des vorherigen Beispiels.

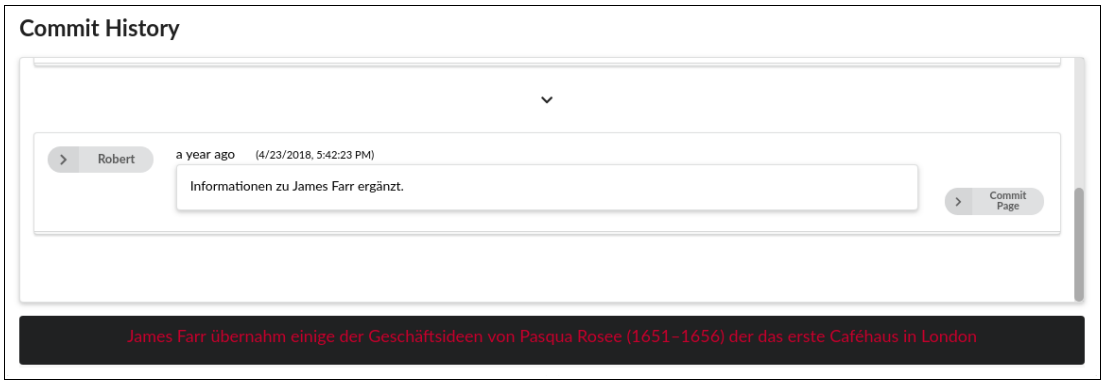

Abbildung 7.3: Popup eines Textelements bei aktiver Heatmap

Anhang B Bill of Materials

Die folgende Auflistung beinhaltet sämtliche Fremdsoftware, die von der Implementierung verwendet wird.

Name	**Version**	**Lizenz**	**URL**
react-color	2.17.3	MIT	https://www.npmjs.com/package/react-color
react-moment	0.9.2	MIT	https://www.npmjs.com/package/react-moment

Literaturverzeichnis

Dohrn, H. & Riehle, D. (2011). Design and Implementation of the Sweble Wikitext Parser: Unlocking the Structured Data of Wikipedia. In *Proceedings of the 7th International Symposium on Wikis and Open Collaboration* (S. 72– 81). WikiSym '11. Mountain View, California: ACM. Doi:10.1145/2038558.2038571

Dohrn, H. & Riehle, D. (2013). Design and Implementation of Wiki Content Transformations and Refactorings. In *Proceedings of the 9th In ternational Symposium on Open Collaboration* (2:1–2:10). WikiSym '13. Hong Kong, China: ACM. doi:10.1145/2491055.2491057

Dohrn, H. & Riehle, D. (2014). Fine-grained Change Detection in Structured Text Documents. In *Proceedings of the 2014 ACM Symposium on Document Engineering* (S. 87–96). DocEng '14. Fort Collins, Colorado, USA: ACM. doi:10.1145/2644866.2644880

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison- Wesley.

Haase, M. (2016). Integration und Erweiterung eines visuellen Editors in Sweble Hub (Masterarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg).

Knogl, D. (2016). Design and Implementation of Graph-DB based Storage for Wikipedia Articles (Masterarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg.