

TECHNISCHE HOCHSCHULE LÜBECK

Technische Hochschule Lübeck
Fachbereich Elektrotechnik und Informatik

Live-Kommentare und Änderungsvorschläge für Rohtextdokumente: Konzeption und prototypische Umsetzung einer Softwarebibliothek

im Studiengang
Informationstechnologie und Design, B.Sc.

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	Live-Kommentare und Änderungsvorschläge für Rohtextdokumente: Konzeption und prototypische Umsetzung einer Softwarebibliothek
Autor:	Sören Fischer
Ausgabedatum:	15.05.2025
Abgabedatum:	21.08.2025
1. Betreuer:	Prof. Dr. Sebastian Berndt
2. Betreuer:	Prof. Dr. Monique Janneck

Aufgabenstellung

In verteilten Entwicklungs- und Arbeitsumgebungen sind effiziente Werkzeuge zur gemeinsamen Textbearbeitung unerlässlich. Insbesondere bei der Arbeit mit Rohtext- und Markup Sprachen wie Latex, Markdown oder Typst fehlen jedoch häufig kollaborative Funktionen, die man aus modernen Rich-Text-Editoren kennt.

Ziel dieser Bachelorarbeit ist daher die Konzeption und prototypische Implementierung einer wiederverwendbaren Bibliothek, die den gleichzeitigen Zugriff mehrerer Nutzer:innen auf Rohtextdokumente ermöglicht und eine Live-Kommentar- und Änderungsfunktion bereitstellt – vergleichbar mit dem Vorschlagsmodus und der Änderungsverfolgung etablierter Office-Lösungen. Eine grafische Benutzeroberfläche ist nicht Teil dieser Arbeit.

Es sollen:

- verbreitete Office-Lösungen im Hinblick auf Vorschlags- oder Änderungsverfolgungen untersucht und daraus Anforderung abgeleitet werden
- geeignete Schnittstellen zur Integration in Drittprogramme definiert werden
- bestehende Konzepte und deren Implementierungen zur Datenreplikation und Versionskontrolle analysiert und auf ihre Eignung zur Erfüllung der formulierten Anforderungen geprüft werden
- eine geeignete Auswahl aus den betrachteten Implementierungen und Konzepten getroffen werden, die für die Umsetzung herangezogen wird
- eine prototypische Umsetzung dieser Auswahl durchgeführt werden
- eine Untersuchung zur Verbesserung der nichtfunktionalen Anforderungen vorgenommen werden, wenn die Zeit es erlaubt

TECHNISCHE HOCHSCHULE LÜBECK

Eigenständigkeitserklärung

Declaration of Originality

Fischer, Sören

Name, Vorname
Last name, first name

350695

Matrikelnummer
Matriculation number

Ich versichere hiermit, dass ich die vorliegende
I hereby declare that this

☐ **Hausarbeit** term paper ☒ **Bachelorarbeit** bachelor's thesis ☐ **Masterarbeit** master's thesis

mit dem Titel
with the title

Live-Kommentare und Änderungsvorschläge für Rohtextdokumente: Konzeption und prototypische Umsetzung einer Softwarebibliothek

eigenständig und ohne unerlaubte fremde Hilfe angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und Entlehnungen aus anderen Arbeiten kenntlich gemacht. Für den Fall, dass die Arbeit zusätzlich elektronisch und/ oder digital eingereicht wird, erkläre ich, dass die schriftliche und die elektronische und/ oder digitale Form identisch sind. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

is my own original work and any assistance from third parties has been acknowledged. I have clearly indicated and acknowledged all sources and resources as well as any borrowings from other works. In case of an additional electronic and/or digital submission of this work, I declare that the written form and the electronic and/or digital form are identical. This work has not previously been submitted either in the same or in a similar form to another examination office.

Ich bin damit einverstanden, dass die vorliegende Hausarbeit/ Bachelorarbeit/ Masterarbeit für Veröffentlichungen, Ausstellungen und Wettbewerbe des Fachbereiches verwendet und Dritten zur Einsichtnahme vorgelegt werden kann.

I agree that this work can be used for publishing, exhibition or competition purposes and can be inspected by third parties.

☒ **ja** Yes ☐ **nein** no ☐ **es liegt ein Sperrvermerk bis _____vor** there is an embargo period until

Ort, Datum
Place, Date

Unterschrift
Signature

Inhaltsverzeichnis

1 Einleitung 1
2 Marktanalyse 3
2.1 Ziel der Analyse 3
2.2 Auswahl der zu untersuchenden Programme 3
2.3 Funktionskatalog 5
2.3.1 Übersicht der implementierten Funktionen 5
2.3.2 Nutzer- und Rechteverwaltung 7
2.4 Funktionsweise des Vorschlagsmodus 8
2.4.1 Vorschläge erstellen 9
2.4.2 Arbeiten an Vorschlägen anderer Nutzender 13
2.4.3 Löschung über Vorschlagsgrenzen hinweg 14
2.4.4 Kommentare erstellen 15
3 Anforderungsdefinition 17
3.1 Funktionale Anforderungen 17
3.2 Nichtfunktionale Anforderungen 18
4 Technische Konzepte verteilter Echtzeit-Kollaboration 21
4.1 Ursprünge verteilter Kollaboration 21
4.2 Probleme verteilter Kollaboration 23
4.2.1 Das Modell 23
4.2.2 Divergenz 25
4.2.3 Kausalitätsverletzung 25
4.2.4 Intentionsverletzung 25
4.2.5 Konsistenzmodell 26
4.3 Operational Transformation 26
4.4 Conflict-Free Replicated Data Types 28
4.5 Konsistenz bei CRDTs und OT 30
4.6 Entscheidung für ein Konzept der verteilten Echtzeit-Kollaboration 31
4.6.1 Komplexität 31
4.6.2 Latenz 31
4.6.3 Praktische Umsetzung und Ökosystem 32
4.6.4 Fazit 32
5 Vorschlagsmodus 33
5.1 Markup-Zeichen 33
5.2 JSON-Dokument 34
5.3 Der CRDT mit Vorschlagsmodus 35
5.3.1 Nur-Text-CRDTs 35
5.3.2 Einfüge-Vorschlag hinzufügen 37
5.3.3 Kommentar hinzufügen 39
5.3.4 Annotationen aktualisieren 39
5.3.5 Annotationen entfernen 41
5.3.6 Die lokale Darstellung 41
5.3.7 Verarbeitung eingehender Operationen 43
6 Implementierung 45

6.1 Technologien ... 45
6.1.1 Auswahl der Plattform ... 45
6.1.2 Auswahl der CRDT-Bibliothek ... 46
6.1.3 Collabs ... 48
6.2 Architektur ... 49
6.2.1 Kernbibliothek: Verwaltung von Text und Annotationen ... 50
6.2.2 Erweiterungsbibliothek: Dokumenten- und Präsenzverwaltung ... 56
6.3 Schnittstellen zur Anwendungsentwicklung ... 57
6.3.1 Die TrackChanges-Klasse ... 57
6.3.2 Die TrackChangesApplication-Klasse ... 59
6.3.3 Die TrackChangesDocument-Klasse ... 59
6.4 Implementierungen über die Anforderungen hinaus ... 60
6.4.1 Prototypisches UI ... 60
6.4.2 Wiederverwendbare CodeMirror-Anbindung ... 60
6.4.3 Automatisch generierte API-Dokumentation ... 61
7 Evaluation ... 62
7.1 Funktionale Anforderungen ... 62
7.2 Nichtfunktionale Anforderungen ... 64
7.2.1 Latenz (NFA 1, NFA 2, NFA 3) ... 64
7.2.2 Konsistenz bei Nebenläufigkeit (NFA 4) ... 64
7.2.3 Entwicklererfahrung und Integration (NFA 5, NFA 6, NFA 7) ... 65
7.2.4 Geringer Entwicklungsaufwand (NFA 8) ... 65
7.3 Textanforderungen ... 65
7.3.1 TA 1: Bearbeitung des eigenen Vorschlags ... 65
7.3.2 TA 2 & TA 3: Wachsen von eigenen Einfüge-Vorschlägen ... 65
7.3.3 TA 4 & TA 5: Wachsen von Lösch-Vorschlägen ... 65
7.3.4 TA 6: Umgang mit Löschungen bei Lösch-Vorschlägen ... 66
7.3.5 TA 7: Bearbeitung von Vorschlägen anderer Nutzer im Editierungsmodus ... 66
7.3.6 TA 8: Wachsen von Vorschlägen anderer Nutzer ... 66
7.3.7 TA 9: Bearbeitung von Vorschlägen anderer Nutzer im Vorschlagsmodus ... 66
7.3.8 TA 10: Löschung über Vorschlagsgrenzen hinweg ... 66
7.3.9 TA 11: Wachsen von Kommentaren ... 66
8 Fazit ... 67
Literaturverzeichnis ... 69
Abbildungsverzeichnis ... 72
Tabellenverzeichnis ... 72
Codeblockverzeichnis ... 72
Glossar ... 73
Anhang: Grenzfallbetrachtungen ... 75

1 Einleitung

„Requirement 4: Support the […] primary writing activities: […] writing, editing, reviewing."

„Requirement 9: Support concurrent and sequential document access."

„Requirement 10: Support several document access methods: write, comment, read."

„Requirement 13: Support synchronous and asynchronous writing."

Schon vor über 30 Jahren definierten Posner und Baecker (1992) diese Anforderungen an Software, die das Schreiben von Texten in Teams ermöglichen soll. Software, die diese Anforderungen erfüllt, erfreut sich heute großer Beliebtheit: Microsoft Word oder Google Docs haben hunderte Millionen Nutzende (Google Inc. 2025; Izatt 2022).

Doch eines haben diese und alle im Rahmen der Arbeit gefundenen Programme, die diese Anforderungen unterstützen, gemeinsam: Sie fokussieren sich auf einen einzelnen Dateityp und das ist in den allermeisten Fällen ein Richtextformat.

Schon 2008 waren jedoch 45% aller wissenschaftlichen Publikationen im Bereich Informatik mit LaTeX verfasst (Brischoux und Legagneux 2009). Markdown findet eine große Verbreitung beim Schreiben von technischer Dokumentation im Softwarebereich. Und mit beispielsweise Typst, mit dem diese Arbeit verfasst ist, entstehen auch immer wieder neue Auszeichnungssprachen, die eben *keine* Richtextformate sind. Um diese von den gängigen Richtextformaten abzugrenzen, werden in dieser Arbeit die Begriffe Rohtext- oder Nur-Text-Formate genutzt.

Ein Programm, das das Schreiben all dieser Auszeichnungssprachen unterstützt und gleichzeitig die Anforderungen an moderne kollaborative Textarbeit erfüllt, konnte bei den Recherchen für diese Arbeit nicht gefunden werden.

Ziel dieser Arbeit ist daher die prototypische Entwicklung einer Softwarebibliothek, die den Grundstein für einen solchen Texteditor legt.

Zwei Themenfelder werden dabei besonders vertieft: Das ist zum einen die Echtzeit-Kollaboration, bei der mehrere Personen gleichzeitig an einem Dokument arbeiten können und die Änderungen in Echtzeit zwischen allen Beteiligten synchronisiert werden. Zum anderen ist das ein komfortabler Reviewprozess – in dieser Arbeit meist als Vorschlagsmodus bezeichnet. Zentral dabei sind Kommentare und Änderungsvorschläge. Kommentare können an bestimmten Textstellen angebracht werden, um Anmerkungen, Fragen oder Verbesserungshinweise zu hinterlassen, ohne den eigentlichen Inhalt zu verändern. Soll der Inhalt hingegen verändert werden, kann ein Änderungsvorschlag erstellt werden. Änderungen werden dabei nicht direkt übernommen, sondern speziell hervorgehoben, sodass sie später einzeln angeschaut und angenommen oder verworfen werden können.

Diese Arbeit ist dabei wie folgt strukturiert: Kapitel 3 beginnt mit einer Marktanalyse bestehender Kollaborationswerkzeuge, aus der in Kapitel 4 die Anforderungen an die zu entwickelnde Bibliothek abgeleitet werden. Kapitel 5 widmet sich den theoretischen Grundlagen und stellt die zentralen technischen Konzepte für verteilte Echtzeit-Kollaboration vor. In Kapitel 6 wird ein detailliertes Konzept für einen Conflict-Free Replicated Data Types (CRDT) mit integriertem Vorschlagsmodus entwickelt. Kapitel 7 beschreibt die Architektur und die wesentlichen Aspekte der prototypischen Implementierung. Die Ergebnisse der

Implementierung werden in Kapitel 8 evaluiert, indem sie den zuvor definierten Anforderungen gegenübergestellt werden. Die Arbeit schließt mit einem Fazit und einem Ausblick auf mögliche weiterführende Entwicklungen.

2 Marktanalyse

Am Beginn dieser Arbeit steht die Analyse bestehender Softwarelösungen, die sowohl einen Vorschlagsmodus, als auch Echtzeit-Kollaboration anbieten. Aus dieser Analyse werden im nächsten Schritt die notwendigen funktionalen Anforderungen an die zu entwickelnde Softwarebibliothek abgeleitet. Daher fokussiert sich diese Untersuchung zunächst darauf, welche Funktionen von etablierten Anbietern unterstützt werden. Neben diesem Überblick ist auch eine detaillierte Betrachtung einiger spezifischer Anwendungsfälle Teil dieses Abschnittes.

2.1 Ziel der Analyse

Vor dem Beginn der Untersuchung formulieren wir einige Forschungsfragen (FF), die mit der Marktanalyse beantwortet werden sollen:

Tabelle 1: Die in diesem Kapitel zu beantwortenden Forschungsfragen.

FF 1.1	Welche kollaborativen Textbearbeitungsprogramme existieren aktuell auf dem Markt?
FF 1.2	Wie werden Benutzerauthentifizierung und Rechteverwaltung gehandhabt?
FF 1.3	Welche Usability-Features bieten die Editoren?
FF 1.4	Wie funktioniert die Vorschlags- und Kommentarfunktion bei etablierten Editoren konkret, wie wird mit Grenzfällen umgegangen und welche Anforderungen können wir daraus für unsere Implementierung ableiten?

2.2 Auswahl der zu untersuchenden Programme

Zunächst soll FF 1.1 beantwortet werden: Welche kollaborativen Textbearbeitungsprogramme existieren aktuell auf dem Markt? Um das zu beantworten, wird eine Marktrecherche durchgeführt.

Die Suche beschränken wir auf Lösungen, die mindestens zwei der folgenden Kriterien erfüllen:

- Echtzeit-Kollaboration: Es muss möglich sein, mit mehreren Nutzenden am gleichen Dokument zu arbeiten und die Änderungen anderer Nutzender in Echtzeit angezeigt zu bekommen.
- Vorschlagsmodus: Es muss möglich sein, dass Nutzende Änderungen vorschlagen können, die andere Nutzende dann nachvollziehen und ablehnen oder annehmen können.
- Ausrichtung auf Endnutzende: Das Programm muss von Endnutzenden ohne besondere Fachkenntnisse genutzt werden können.

Um aus der anschließenden Anforderungsanalyse sinnvoll Anforderungen ableiten zu können, müssen alle Programme die Marktreife erreicht haben. Nicht relevant für diese Suche ist, ob Nur-Text oder Richtextformate bearbeitet werden, da die grundlegenden Prinzipien der Textbearbeitung die gleichen sind. Um welchen Texttyp es sich handelt, wird der Vollständigkeit halber in der Ergebnistabelle (Tabelle 2) dennoch aufgeführt.

Tabelle 2: Kollaborative Textbearbeitungsprogramme, die aktuell auf dem Markt sind

Software	Echtzeit-Kollaboration	Vorschlagsmodus	Ausrichtung auf Endnutzende	Texttyp	Url
Dropbox Paper	✓	×	✓	rich	https://paper.dropbox.com/
CKEditor	✓	✓	×	rich	https://ckeditor.com
Google Docs	✓	✓	✓	rich	https://docs.google.com
Microsoft Word online	✓	✓	✓	rich	https://word.cloud.microsoft/de-de/
ClickUp	✓	×	✓	rich	https://clickup.com/
Zoho Writer	✓	✓	✓	rich	https://www.zoho.com/de/writer/
Notion	✓	×	✓	rich	https://notion.so
Apple pages	✓	✓	✓	rich	https://www.apple.com/de/pages/
LibreOffice Writer	×	✓	✓	rich	https://www.libreoffice.org/
coda	✓	×	✓	rich	https://coda.io/
bit.ai	✓	×	✓	rich	https://bit.ai/
softmaker	×	✓	✓	rich	https://softmaker.com
Collabora Online	✓	✓	✓	rich	https://www.collaboraonline.com
Only Office	✓	✓	✓	rich	https://www.onlyoffice.com
WPS Writer	✓	✓	✓	rich	https://de.wps.com/office/writer/

Software	Echtzeit-Kollaboration	Vorschlagsmodus	Ausrichtung auf Endnutzende	Texttyp	Url
HedgeDoc	✓	✕	✓	markdown	https://hedgedoc.org/
HackMD	✓	(✓)[1]	✓	markdown	https://hackmd.io/
Overleaf	✓	✓	✓	latex	https://www.overleaf.com/

2.3 Funktionskatalog

Aus diesen Ergebnissen erstellen wir im nächsten Schritt eine Übersicht der implementierten Funktionen zur Beantwortung von FF 1.2 und 1.3. Dabei fokussieren wir uns nur auf die sieben Softwarelösungen, die alle drei definierten Kriterien erfüllen.

2.3.1 Übersicht der implementierten Funktionen

Zunächst führen wir zu jedem der Editoren eine kurze Recherche durch, um den ungefähren Funktionsumfang festzustellen. Die gefundenen Funktionen tragen wir als Vergleichskriterien zusammen und selektieren sie nach ihrer Relevanz für diese Arbeit. Relevant sind dabei Funktionen, die potentiell auch von einer im Hintergrund funktionierenden Bibliothek unterstützt werden müssen. Eine Funktionalität wie verschiedenfarbige Cursor, die nur auf Ebene des User Interfaces (UI) implementiert wird, fällt so beispielsweise heraus. Die ausgewählten Kriterien sind in Tabelle 3 aufgeführt.

Tabelle 3: Die ausgewählten Vergleichskriterien

Kürzel	Name	Beschreibung
VK 1	Echtzeitcursor	Die aktuelle Position des Cursors wird an die anderen Nutzenden übertragen.
VK 2	Präsenzindikator	Es wird angezeigt, welche Nutzenden das Dokument aktuell geöffnet haben.
VK 3	Textauswahl	Wenn Text ausgewählt wird, wird die markierte Stelle auch bei anderen Nutzenden hervorgehoben.
VK 4	Kommentare	Es können spezifische Textstellen ausgewählt und mit Kommentaren in Textform versehen werden.
VK 5	Chat	Es kann innerhalb des Editors mit anderen Nutzenden, die das Dokument aktuell geöffnet haben, über Kurznachrichten kommuniziert werden.

[1] Der Vorschlagsmodus von HackMD befindet sich noch in einer Beta-Phase.

Kürzel	Name	Beschreibung
VK 6	Anonyme Editierung	Ein Dokument kann angesehen und bearbeitet werden, ohne dass ein Account beim Anbieter notwendig ist.
VK 7	Versionshistorie	Änderungen am Dokument werden gespeichert und sind in einer Historie einsehbar.
VK 8	Änderung per Nutzer verfolgen	Es gibt eine Versionshistorie, die sich per Nutzer filtern lässt. Auch Änderungen, die nicht im Vorschlagsmodus gemacht werden, sind dort aufgeführt.
VK 9	Vorschau in Echtzeit	Wenn das Dokument geändert wird, aktualisiert sich die Ansicht des finalen Dokuments. Nur für Nur-Text relevant.
VK 10	Linkfreigabe	Die Freigabe eines Dokuments für andere Nutzende kann per Link erfolgen.
VK 11	Nutzerfreigabe	Es ist eine Anmeldung auf der Plattform möglich. Die Freigabe kann auf spezifische Accounts beschränkt werden.
VK 12	Passwortschutz	Zusätzlich zu einer Link- oder Nutzerfreigabe kann das Dokument außerdem mit einem Passwort geschützt werden.
VK 13	Ablaufdatum von Freigabelinks	Freigabelinks können mit einem Ablaufdatum versehen werden.
VK 14	Accountverwaltung	Es können Nutzeraccounts angelegt werden, denen Dokumente zugeordnet werden.
VK 15	Rechteverwaltung	Granularität der Rechte für den Dokumentenzugriff.

Im Hinblick auf diese Vergleichskriterien (VK) untersuchen wir die sieben Editoren. Die Ergebnisse dieser Untersuchung sind in Tabelle 4 gegenübergestellt. Das VK 15 wird in Tabelle 5 gesondert dargestellt.

Tabelle 4: Übersicht der implementierten Funktionen

Editor/ Feature	Google Docs	Microsoft Word online	Collabora Online	Only Office	Hack MD	Overleaf	Apple Pages
VK 1	✓	✓	✓	✓	✓	✓	✓
VK 2	✓	✓	✓	✓	✓	✓	✓
VK 3	✓	✕	✓	✕	✓	✕	✓
VK 4	✓	✓	✓	✓	✓	✓	✓
VK 5	✓	✕	✕	✓	✕	✓	✕
VK 6	✓	✓	✓	✓	✓	✕	✕
VK 7	✓	✓	✓	✓	✓	✓	✓
VK 8	✓	✕	✕	✕	✕	✕	✓
VK 9	✓	✓	✓	✓	✓	✕	✓
VK 10	✓	✓	✓	✓	✓	✓	✓
VK 11	✓	✓	✓	✕	✓	✓	✓
VK 12	✕	✓	✓	✕	✕	✕	✕
VK 13	✕	✓	✓	✓	✕	✕	✕
VK 14	✓	✓	✓	✓	✓	✓	✓

2.3.2 Nutzer- und Rechteverwaltung

In Tabelle 4 wird FF 1.2 schon teilweise beantwortet: Neben einer Linkfreigabe erlauben alle untersuchten Plattformen auch das Anlegen von Nutzeraccounts. Die Authentifizierung erfolgt dabei auf allen Plattformen mit E-Mail und Passwort, von einigen wird zusätzlich das Anmelden über Dienste von Drittanbietern angeboten.

Die Rechteverwaltung auf Dokumentenebene basiert bei allen Editoren auf der gleichen Grundlage: Bei der Freigabe können Rechte festgelegt werden, die berechtigte Nutzende beim Zugriff haben. Welche Rechte erteilt werden können, ist dabei von Software zu Software unterschiedlich (siehe Tabelle 5).

Tabelle 5: Rechteverwaltung auf Dokumentebene im Vergleich (VK 15)

Google Docs	Bearbeiten/Überprüfen/Lesen
Microsoft Word online	Bearbeiten/Lesen
Collabora Online	Bearbeiten/Lesen/Löschen
Only Office	Bearbeiten/Überprüfen/Kommentieren/Lesen
Hack MD	Bearbeiten/Lesen/Ansehen
Overleaf	Bearbeiten/Überprüfen/Lesen
Apple Pages	Bearbeiten/Lesen

2.4 Funktionsweise des Vorschlagsmodus

In FF 1.4 geht es um die konkrete Funktionsweise der Vorschlagsfunktion, insbesondere im Hinblick auf verschiedene Grenzfälle. Dafür schauen wir uns die Details der Vorschlagsfunktion an, die in der praktischen Nutzung oft übersehen werden: „Wie ‚wächst' ein Vorschlag?", „Wie hört er auf zu ‚wachsen'?" oder „Was passiert, wenn ein Nutzer den Vorschlag eines anderen Nutzers bearbeitet?". Neben dieser grundsätzlichen Funktionsweise gibt es oft Grenzfälle, bei denen es keine eindeutig richtige Auflösung gibt. Wenn beispielsweise Nutzerin A vorgeschlagen hat, das Wort ‚Stadt' durch ‚Ort' zu ersetzen und Nutzer B diesen Vorschlag wiederum durch ‚Dorf' ersetzt, dann gibt es mehrere Wege, damit umzugehen. Es ist jedoch wichtig für eine gute Gebrauchstauglichkeit, dass die Handhabung sowohl der allgemeinen Funktionen als auch der Grenzfälle den Nutzererwartungen entspricht.

Es würde den zeitlichen Rahmen dieser Arbeit übersteigen, Nutzertests durchzuführen, um die Nutzererwartungen festzustellen. Daher untersuchen wir zur Beantwortung von FF 1.4 die in Tabelle 2 aufgeführten Programme. Auch hier wird die Auswahl auf Software eingegrenzt, die alle drei Anforderungen (Echtzeit-Kollaboration, Vorschlagsmodus, Ausrichtung auf Endnutzende) erfüllt. Zudem wird die Software Apple Pages nicht näher betrachtet, da sie nicht plattform-agnostisch ist. Die Software HackMD untersuchen wir ebenfalls nicht, da deren Vorschlagssystem sich noch in einer Beta-Phase befindet.

Aus dieser Untersuchung leiten wir dann konkrete Textanforderungen (TA) ab, die mit der eigenen Implementierung erfüllt werden sollen.

Um auch Grenzfälle in der Bearbeitung zu erfassen, die in einer normalen Nutzung selten auftreten, gehen wir in Anlehnung an die Pairwise-Methode[2] vor, die bei Softwaretests verwendet wird. Dafür wird das Nutzungskonzept ähnlich zur Funktionalen Zerlegung[3] in seine elementaren Bestandteile und Zustände zerlegt (siehe Tabelle 6) und diese paarweise kombiniert betrachtet. Die ausgewählten Grenzfälle sind in Tabelle 7 aufgeführt.

Tabelle 6: Die grundsätzlichen Funktionen und Modi aus denen die zu betrachtenden Grenzfälle bestimmt werden

Vorschläge	1, >= 2 (gleicher Autor), >= 2 (unterschiedliche Autoren)
Aktion	Einfügen, Ersetzen, Löschen
Modus	Vorschlagsmodus, Bearbeitungsmodus

[2] https://de.wikipedia.org/wiki/Pairwise-Methode

[3] https://www.baeldung.com/cs/functional-decomposition

Tabelle 7: Die nach der Pairwise-Methode kombinierten Grenzfälle

Kürzel	Modus	Aktion	Vorschläge
GF 1	Vorschlagsmodus	Ersetzen	1
GF 2	Bearbeitungsmodus	Einfügen	1
GF 3	Bearbeitungsmodus	Ersetzen	>= 2 (gleicher Autor)
GF 4	Vorschlagsmodus	Einfügen	>= 2 (gleicher Autor)
GF 5	Vorschlagsmodus	Löschen	>= 2 (unterschiedliche Autoren)
GF 6	Bearbeitungsmodus	Ersetzen	>= 2 (unterschiedliche Autoren)
GF 7	Vorschlagsmodus	Löschen	1
GF 8	Vorschlagsmodus	Löschen	>= 2 (gleicher Autor)
GF 9	Vorschlagsmodus	Einfügen	>= 2 (unterschiedliche Autoren)

Für die Untersuchung legen wir zunächst einige Rahmenbedingungen fest: Wir gehen davon aus, dass zwei Nutzende Alice (A) und Bob (B) ein bereits angelegtes Dokument gleichzeitig bearbeiten. Als Text dient uns hierbei ein Ausschnitt des Wikipedia Artikel zu Wombats[4].

Zu Beginn untersuchen wir die allgemeine Funktionsweise des Vorschlagsmodus. Dabei arbeiten wir in dem Dokument und beobachten, wie sich Vorschläge und Kommentare verhalten: beispielsweise wie sie erstellt werden oder wie sie wachsen und aufhören zu wachsen.

Anschließend wenden wir uns den in Tabelle 7 ausgewählten Grenzfällen zu. Angelehnt an das „Architectural Approach Documentation Template“ (Kazman, Klein, und Clements 2000) definieren wir nun zu den Grenzfällen Testfall-Szenarien und legen für jeden Testfall eine Umgebung und einen konkreten Stimulus fest. Diese Szenarien reproduzieren wir dann in jedem der ausgewählten Programme und dokumentieren deren Antworten. Die Ergebnisse dieser Untersuchung sind im Anhang aufgeführt.

Der folgende Abschnitt erklärt nun die grundsätzliche Funktionsweise des Vorschlagsmodus, sowie einige ausgewählte Regeln für das Verhalten von Vorschlägen und Kommentaren, die sich aus den Beobachtungen der Grenzfallanalyse ergeben haben. Daraus formulieren wir konkrete Textanforderungen an die Implementierung unseres Vorschlagsmodus. Aufgrund der hohen Anzahl an möglichen Grenzfällen kann die Auswahl der Textanforderungen allerdings nicht als vollständig betrachtet werden, soll aber die Grundlage für ein gebrauchstaugliches Vorschlagssystem schaffen.

2.4.1 Vorschläge erstellen

Betrachten wir zunächst das grundsätzliche Erstellen von Einfüge- und Lösch-Vorschlägen. Dabei gehen wir hier von nur einer Person aus, die Vorschläge erstellt. Wie es sich verhält, wenn Nutzende Vorschläge anderer Nutzender bearbeiten, betrachten wir im nächsten Abschnitt.

Vorschläge zu erstellen funktioniert in allen untersuchten Programmen gleich. Es gibt einen globalen Schalter, mit dem festgelegt werden kann, ob Änderungen, die am Dokument vorgenommen werden, als Vorschlag oder als normaler Text eingefügt werden sollen.

[4] https://de.wikipedia.org/w/index.php?title=Wombats

2.4.1.1 Einfüge-Vorschlag erstellen

Angenommen, Alice liest die Bachelorarbeit von Bob. Sie ist gerade beim folgenden Abschnitt:

> Der Name Wombat stammt aus der ausgestorbenen Sprache der Darug.

Nun weiß Alice jedoch, dass es einige wenige Menschen gibt, die diese Sprache noch beherrschen. Den Hinweis möchte sie gerne im Text unterbringen, ist sich jedoch nicht sicher, ob Bob damit einverstanden ist. Daher schaltet sie nun den Vorschlagsmodus ein. Sie ergänzt das Wort „nähezu“:

> Der Name Wombat stammt aus der nähezu ausgestorbenen Sprache der Darug.

Das Wort wird als ein zusammenhängender Vorschlag erkannt und kann nun als solcher von Bob angenommen oder abgelehnt werden. Alice fällt allerdings auf, dass sie sich vertippt hat. Sie wollte eigentlich „nahezu“ statt „nähezu“ schreiben. Deshalb setzt sie ihren Cursor hinter das „ä“, löscht es mit der Rücktaste und fügt stattdessen ein „a“ ein. In allen untersuchten Programmen sieht der Satz nun wie folgt aus:

> Der Name Wombat stammt aus der nahezu ausgestorbenen Sprache der Darug.

Es wurde kein neuer Vorschlag erzeugt, sondern das Löschen und Einfügen von Alice innerhalb ihres eigenen Vorschlags ist in allen Programmen einfach möglich. Das ist die erste Textanforderung, die wir formulieren können.

TA 1: Bearbeitung des eigenen Vorschlags

BESCHREIBUNG

Schreibt ein Nutzer einen Einfüge-Vorschlag, kann er innerhalb dieses Vorschlags Löschungen und Einfügungen durchführen, ohne dass dafür eigene Vorschläge erstellt werden. Das funktioniert sowohl im Vorschlags- als auch im Editierungsmodus.

Aus diesem Beispiel können wir auch eine weitere Anforderung ableiten: Alice konnte das Wort „nahezu“ am Stück als Vorschlag schreiben, weil Einfüge-Vorschläge wachsen, wenn an ihrem Rand geschrieben wird. So hat Alice als Erstes das Leerzeichen nach dem Wort „der“ ergänzt und dadurch einen neuen Vorschlag erzeugt. Daraufhin hat sich ihr Cursor an der Stelle *nach* dem Leerzeichen befunden. Trotzdem wurden die folgenden von ihr ergänzten Buchstaben Teil des Vorschlags. Die Untersuchungen zeigen, dass dieses Verhalten in allen Programmen sowohl am Ende als auch am Anfang eines Vorschlags implementiert ist.

TA 2: Wachsen von eigenen Einfüge-Vorschlägen im Vorschlagsmodus

BESCHREIBUNG
Arbeitet ein Nutzer im Vorschlagsmodus am Rand eines eigenen Einfüge-Vorschlags, werden die Zeichen, die er schreibt, in diesen Vorschlag übernommen. Dabei ist es egal, ob er am Anfang oder am Ende des Vorschlags schreibt.

Aus dieser Beobachtung ergibt sich allerdings die Frage, wie man aufhören kann, in einem Vorschlag zu schreiben. Angenommen, Alice möchte vor ihrem Vorschlag noch das Wort „wirklich“ ergänzen. Das möchte sie aber nicht als Vorschlag ergänzen, da das ja auch passen würde, wenn Bob ihren „nahezu“ Vorschlag ablehnt. Die Lösung besteht in allen untersuchten Programmen darin, den Vorschlagsmodus auszuschalten. Dann hören Vorschläge auf zu wachsen, auch wenn man an ihrem Rand schreibt.

TA 3: Wachsen von eigenen Einfüge-Vorschlägen im Editierungsmodus

BESCHREIBUNG
Arbeitet ein Nutzer im Editierungsmodus am Rand eines eigenen Einfüge-Vorschlags, werden die Zeichen, die er schreibt, nicht in diesen Vorschlag übernommen. Dabei ist es egal, ob er am Anfang oder am Ende des Vorschlags schreibt.

2.4.1.2 Lösch-Vorschlag erstellen

Bisher haben wir nur Vorschläge betrachtet, bei denen Alice etwas eingefügt hat. Doch wie funktionieren Lösch-Vorschläge?

Nehmen wir uns dazu wieder einen Beispielsatz vor, den Alice gerade liest:

> Wombats können über kurze Strecken krasse Geschwindigkeiten von bis zu 40 km/h erreichen.

Alice findet, dass das Wort „krass“ hier unangemessen ist und möchte daher vorschlagen, es zu löschen. Dafür schaltet sie wieder den Vorschlagsmodus an und hat dann für die Löschung des Textes – wie in modernen Textbearbeitungsprogrammen üblich – mehrere Möglichkeiten: Sie könnte das gesamte Wort auswählen und mit der Rück- oder Entfernen-Taste löschen. Sie könnte auch ihren Cursor am Ende des Wortes positionieren und jeden Buchstaben einzeln mit der Rücktaste entfernen, oder sie macht es andersherum, positioniert also ihren Cursor am Anfang des Wortes und entfernt die Buchstaben mit der Entfernen-Taste.

Alle drei Wege das Wort zu löschen führen in allen untersuchten Programmen zu dem Ergebnis, dass ein einzelner, gesammelter Lösch-Vorschlag erstellt wird. Daraus können wir also ableiten, dass auch Lösch-Vorschläge wachsen können, wenn an ihrem Rand gelöscht wird. Analog zu Einfüge-Vorschlägen passiert das allerdings nur, wenn Nutzende sich weiterhin im Vorschlagsmodus befinden. Ein weiterer Test zeigt zudem, dass *nur* Löschungen zum Wachsen führen. Wird am Rand eines Lösch-Vorschlags ein Buchstabe eingefügt, wird dieser nicht Teil des Lösch-Vorschlags. Daraus können wir die folgenden Textanforderungen formulieren:

TA 4: Wachsen von Lösch-Vorschlägen im Vorschlagsmodus

BESCHREIBUNG
Löscht ein Nutzer im Vorschlagsmodus am Rand eines Lösch-Vorschlags weitere Buchstaben, so werden diese Löschungen Teil des ursprünglichen Lösch-Vorschlags. Dabei ist es egal, ob er am Anfang oder am Ende des Vorschlags löscht.

TA 5: Wachsen von Lösch-Vorschlägen im Editierungsmodus

BESCHREIBUNG
Löscht ein Nutzer im Editierungsmodus am Rand eines Lösch-Vorschlags weitere Buchstaben, so werden diese direkt entfernt und werden nicht Teil des Lösch-Vorschlags. Dabei ist es egal, ob er am Anfang oder am Ende des Vorschlags löscht.

Während das oben beschriebene Verhalten bei allen Programmen gleich ist, können bei der Umsetzung der eigentlichen Löschung zwei unterschiedliche Vorgehensweisen festgestellt werden:

Option 1:

...Strecken ~~krasse~~ Geschwindigkeiten...

Option 2:

...Strecken ⊥ Geschwindigkeiten...

Die meisten der untersuchten Programme setzen die Löschung wie in Option 1 dargestellt um: Der Text, der gelöscht wird, bleibt stehen, wird jedoch durchgestrichen. Nur Overleaf, ein Editor für LaTeX, entfernt die vorgeschlagene Löschung, wie in Option 2 dargestellt, direkt und kennzeichnet dies nur durch eine kleine Markierung im Text sowie einen seitlich stehenden Hinweis. Wird der Vorschlag abgelehnt, erfolgt eine Wiederherstellung des Originaltextes.

Für die meisten Anwendungsfälle erscheint die erste Option als die bessere, da so beispielsweise Bob, der die Änderungen von Alice liest, schneller nachvollziehen kann, was sie geändert hat.

TA 6: Umgang mit Löschungen bei Lösch-Vorschlägen

BESCHREIBUNG
Buchstaben, die ein Nutzer im Vorschlagsmodus zur Löschung vorschlägt, werden nicht direkt aus dem Dokument entfernt, sondern mit einer entsprechenden Annotation versehen.

2.4.2 Arbeiten an Vorschlägen anderer Nutzender

Nun betrachten wir, was passiert, wenn Nutzende mit den Vorschlägen anderer Nutzender interagieren. Dazu gehen wir davon aus, dass Alice weiterhin im Vorschlagsmodus schreibt. Sie schreibt den folgenden Satz:

> Wombats leben in Australien

Bob liest den Satz und möchte gerne ergänzen, dass Wombats *nur* in Australien leben. Dabei könnte er entweder im Editierungsmodus oder selber im Vorschlagsmodus schreiben.

2.4.2.1 Bob schreibt im Editierungsmodus

Bob ergänzt zunächst das Wort " nur" im Editierungsmodus. Die untersuchten Programme weisen zwei verschiedene Umsetzungsarten auf.

Option 1:

Wombats leben nur in Australien

Option 2:

Wombats leben nur in Australien

Option 1 wird von den meisten Programmen wie Microsoft Word Online oder OnlyOffice umgesetzt. Dabei wird der Vorschlag in zwei einzelne Vorschläge aufgeteilt und der von Bob eingefügte Text als normaler Text übernommen. Die Option 2 setzt von den untersuchten Programmen nur Google Docs um. Hier wird der eingefügte Text als Teil des Vorschlags übernommen und es ist visuell nicht mehr erkennbar, dass der Text von Bob geschrieben wurde.

Obwohl bei Option 2 die Information verloren geht, dass der Vorschlag von zwei verschiedenen Personen verfasst wurde, erscheint sie als die gebrauchstauglichere Variante. Denn angenommen, Bob schreibt seine Bachelorarbeit über Wombats und möchte sie nun von Alice korrekturlesen lassen. Alice ergänzt die Arbeit um den oben stehenden Satz. Wenn Bob nun alle Vorschläge durchgeht, kann er diesen Vorschlag selber bearbeiten und ihn danach als einen zusammenhängenden Vorschlag annehmen oder ablehnen. Im anderen Fall müsste er, wenn er sich entscheidet, den Vorschlag nicht anzunehmen, zwei Vorschläge ablehnen und dann auch noch sein eingefügtes " nur" selber löschen.

TA 7: Bearbeitung von Vorschlägen anderer Nutzender im Editierungsmodus

BESCHREIBUNG

Wird ein Einfüge-Vorschlag eines anderen Nutzers im Editierungsmodus bearbeitet, so wird die Bearbeitung als Teil des Vorschlags übernommen.

Dieses Verhalten ändert sich bei Google Docs, wenn am Rand eines Vorschlags geschrieben wird. Dann wird die Einfügung als normaler Text übernommen. Möchte Bob den Vorschlag von Alice also mit einem Ausrufezeichen abschließen, dann würde der Text anschließend so aussehen:

> Wombats leben nur in Australien!

Das erscheint als ein konsistentes Verhalten, da wir das bei Bearbeitungen an eigenen Vorschlägen in TA 3 schon so festgestellt haben.

TA 8: Wachsen von Vorschlägen anderer Nutzender

BESCHREIBUNG
Arbeitet ein Nutzer am Rand eines Vorschlags eines anderen Nutzers, werden die Zeichen, die er schreibt, nicht in diesen Vorschlag übernommen. Dabei ist es egal, ob er am Anfang oder am Ende des Vorschlags schreibt und ob er sich im Vorschlags- oder Editierungsmodus befindet.

2.4.2.2 Bob schreibt im Vorschlagsmodus

Im zweiten Fall schreibt Bob selber im Vorschlagsmodus. Auch hier gibt es zwei Optionen, wie die untersuchten Programme dies umsetzen.

Option 1:

Wombats leben nur in Australien

Option 2:

Wombats leben nur in Australien

Diese beiden Optionen ähneln stark denjenigen aus dem ersten Fall: Bei Option 1 wird der ursprüngliche Vorschlag in drei separate Untervorschläge aufgeteilt, die jeweils unabhängig voneinander angenommen oder abgelehnt werden können. In Option 2 wird der Vorschlag von Bob ebenfalls als eigenständiger Vorschlag angelegt, bleibt jedoch inhaltlich mit dem ursprünglichen Vorschlag verknüpft. Das bedeutet, dass im Falle einer Ablehnung des Vorschlags von Alice auch der davon abhängige Vorschlag von Bob automatisch entfällt. Diese Vorgehensweise erscheint als die logischste Umsetzung.

TA 9: Bearbeitung von Vorschlägen anderer Nutzenden im Vorschlagsmodus

BESCHREIBUNG
Wird ein Vorschlag eines anderen Nutzers im Vorschlagsmodus bearbeitet, so wird für die Ergänzung ein eigener Vorschlag angelegt, der jedoch ebenfalls entfernt wird, wenn der ihn umgebende Vorschlag entfernt wird. Der ursprüngliche Vorschlag bleibt unverändert bestehen.

2.4.3 Löschung über Vorschlagsgrenzen hinweg

Für andere Nutzende ist es also möglich, einen fremden Vorschlag zu bearbeiten, indem sie selber in den Editierungsmodus wechseln. Wenn sie stattdessen im Vorschlagsmodus schrei-

ben oder löschen, wird ein neuer Vorschlag im Vorschlag erstellt. Aber was passiert, wenn sie sich entscheiden von ihrem eigenen in einen fremden Vorschlag hinein zu löschen? Das ist sicherlich kein sehr häufiger Anwendungsfall, dennoch ist auch das ein Fall, der von einer Softwarebibliothek für einen Vorschlagsmodus behandelt werden muss. Gehen wir also von dem folgenden Ausgangszustand aus:

Alice und Bob schreiben beide zusammen im Vorschlagsmodus an einem Satz. Alice beginnt und schreibt „Wombats sind wirklich“, was Bob anschließend um „ sehr nachtaktiv“ ergänzt.

> Wombats sind wirklich sehr nachtaktiv.

Die Formulierung kommt Bob aber seltsam vor und er möchte die Wörter „wirklich sehr“ gerne löschen. Er arbeitet weiterhin im Vorschlagsmodus, markiert die beiden Wörter und löscht sie. Auch hier haben sich Unterschiede zwischen den untersuchten Programmen gezeigt:

Option 1:

Wombats sind ~~wirklich~~ nachtaktiv.

Option 2:

Wombats sind ~~wirklich sehr~~ nachtaktiv.

Die erste Option wird beispielsweise von Microsoft Word Online verwendet. Dabei wird das Wort „sehr“ aus dem Vorschlag von Bob unmittelbar entfernt. Für die Streichung des Wortes „wirklich“ aus dem Vorschlag von Alice hingegen wird ein eigener Löschvorschlag erzeugt, der – wie in TA 9 beschrieben – innerhalb des ursprünglichen Vorschlags liegt. Interessanterweise setzt Microsoft Word Online TA 9 bei regulären Bearbeitungen jedoch nicht um.

Die zweite Option entspricht der Vorgehensweise von Google Docs. Hier wird ein Löschvorschlag erstellt, der sowohl „wirklich“ als auch „sehr“ umfasst. Dies erscheint als die gebrauchstauglichere Variante, da sie die tatsächliche Aktion von Bob widerspiegelt: beide Wörter in einem Schritt auswählen und zur Löschung vorschlagen. Wir formulieren also die folgende etwas verallgemeinerte Textanforderung:

TA 10: Löschung über Vorschlagsgrenzen hinweg

BESCHREIBUNG
Umfasst eine Löschung im Vorschlagsmodus Text außerhalb eines eigenen Vorschlags, wird für den gesamten gelöschten Bereich ein neuer Lösch-Vorschlag erstellt.

2.4.4 Kommentare erstellen

Ein bislang nicht betrachteter Anwendungsfall ist das Hinzufügen von Kommentaren. Diese Funktion wird von allen untersuchten Programmen in sehr ähnlicher Form unterstützt.

Angenommen, Alice liest in der Arbeit von Bob folgenden Satz:

> Wombats zählen zu den grabend lebenden Säugetieren.

Sie fragt sich, was der Ausdruck „grabend lebend“ genau bedeutet. Um dies zu klären, möchte sie einen Kommentar verfassen. Dazu markiert sie die Wörter „grabend lebend“ und wählt im UI die entsprechende Option aus. In einem separaten Textfeld kann sie nun ihre Frage als Freitext eingeben und den Kommentar anschließend bestätigen. Der markierte Text wird hervorgehoben, während der Kommentartext selbst – bei allen getesteten Programmen – in einem separaten UI-Element seitlich angezeigt wird.

Wombats zählen zu den grabend lebenden Säugetieren.	Heißt das, sie leben in Höhlen?

Doch was geschieht, wenn Bob den Satz nachträglich ändert – und zwar vor, nach oder sogar innerhalb der markierten Stelle? Fügt er zum Beispiel direkt vor dem markierten Ausdruck das Wort „größten“ ein und ergänzt innerhalb der Markierung zwischen „grabend“ und „lebend“ die Worte „und in Höhlen“, ergibt sich in allen Programmen dasselbe Bild:

Wombats zählen zu den größten grabend und in Höhlen lebenden Säugetieren.	Heißt das, sie leben in Höhlen?

- Die innerhalb der Markierung eingefügten Wörter („und in Höhlen“) werden in den Kommentarbereich übernommen.
- Die außerhalb der Markierung eingefügte Ergänzung („größten“) bleibt dagegen unmarkiert.

Das zeigt: Kommentare wachsen nicht automatisch, wie es etwa bei Einfüge-Vorschlägen der Fall wäre. Diese Verhaltensweise ist sinnvoll, da sie der ursprünglichen Nutzerabsicht entspricht – die Frage von Alice bezog sich ausschließlich auf den markierten Ausdruck „grabend lebend“, nicht auf den gesamten Satz oder spätere Ergänzungen.

Wir können also die folgenden Textanforderungen definieren:

TA 11: Wachsen von Kommentaren

BESCHREIBUNG

Ergänzt ein Nutzer Text am Rand eines Kommentars, wird dieser Text nicht als Teil des Kommentars gespeichert. Wird Text innerhalb des Kommentars ergänzt, gehört er hingegen zum Kommentar.

3 Anforderungsdefinition

Mit der Marktanalyse wurde die Grundlage für die Anforderungsdefinition geschaffen. Neben den funktionalen Anforderungen, die aus Tabelle 4 und Abschnitt 2.4 abgeleitet werden, definieren wir auch einige nichtfunktionale Anforderungen, die uns später bei Technologieentscheidungen und Bewertungen unterschiedlicher Umsetzungsansätze helfen sollen.

3.1 Funktionale Anforderungen

Aus den in Abschnitt 2 untersuchten Funktionen von bereits auf dem Markt befindlicher Software leiten wir nun funktionale Anforderungen an unsere Softwarebibliothek ab. Diese unterteilen wir in „Muss-", „Soll-" und „Kann-"Anforderungen. Ob es sich um eine „Muss-", „Soll-" oder „Kann-"Anforderung handelt, wird neben dem geschätzten Implementierungsaufwand anhand der Zahl der Editoren, die diese Funktion implementieren, festgelegt.

Außen vor gelassen werden Funktionen, die eine serverseitige Implementierung benötigen, wie etwa die Account- oder Zugriffsverwaltung. Diese sind zwar idealerweise Bestandteil einer ausgereiften Bibliothek, doch beschäftigt sich diese Arbeit vorrangig mit der Umsetzung eines Vorschlagsmodus.

FA 1: Nur-Text-Bearbeitung | Muss

BESCHREIBUNG
Nutzende müssen in der Lage sein, Nur-Text-Dokumente (z. B. Markdown oder Typst) einzusehen und zu bearbeiten.

FA 2: Echtzeit-Kollaboration | Muss

BESCHREIBUNG
Änderungen von Nutzenden müssen in nahezu Echtzeit an die anderen Nutzenden übertragen werden.

FA 3: Vorschlagsmodus | Muss

BESCHREIBUNG
Es muss ein Vorschlagsmodus implementiert sein. Wenn dieser aktiviert ist, werden Modifikationen protokolliert und können nachträglich akzeptiert und in das Dokument übernommen oder abgelehnt und gelöscht werden. Grenzfälle werden, wie in den Textanforderungen beschrieben, den Nutzererwartungen entsprechend behandelt.

FA 4: Cursorposition | Muss

BESCHREIBUNG
Die jeweils aktuelle Cursorposition aller Nutzenden eines Dokuments muss in Echtzeit an alle anderen Nutzenden synchronisiert werden.

FA 5: Präsenzindikator | Muss

BESCHREIBUNG
Der Bearbeitungsstatus von allen Nutzenden muss in Echtzeit an die anderen Nutzenden übermittelt werden.

FA 6: Textauswahl | Soll

BESCHREIBUNG
Alle Nutzenden sollen in Echtzeit sehen können, wenn eine Textpassage von einem anderen Nutzenden ausgewählt wird.

FA 7: Kommentare | Muss

BESCHREIBUNG
Nutzende müssen die Möglichkeit haben, Kommentare an spezifischen Textstellen anzufügen.

FA 8: Chat | Kann

BESCHREIBUNG
Es kann eine Chatfunktion bereitgestellt werden, über die Nutzende miteinander kommunizieren können.

FA 9: Anonyme Editierung | Soll

BESCHREIBUNG
Nutzende sollen ein Dokument ansehen und bearbeiten können, ohne ein Benutzerkonto beim Anbieter zu besitzen.

FA 10: Versionshistorie | Kann

BESCHREIBUNG
Es kann eine Versionshistorie implementiert werden, sodass alle Änderungen am Dokument gespeichert und nachvollziehbar sind.

FA 11: Änderungen nach Nutzenden filtern | Kann

BESCHREIBUNG
Es kann die Versionshistorie nach Nutzenden gefiltert werden, um Änderungen spezifischer Nutzender anzuzeigen.

3.2 Nichtfunktionale Anforderungen

Für die Definition von nichtfunktionalen Anforderungen orientieren wir uns an den nach ISO/IEC 25010:2023 (ISO/IEC JTC 1/SC 7 2023) vorgeschlagenen Qualitätsmerkmalen. Nicht

relevant für diese Arbeit sind dabei die Kriterien *Security*, da diese erst im Rahmen einer zukünftigen Implementierung mit Account- und Nutzerverwaltung berücksichtigt wird, sowie *Safety*, deren Betrachtung den zeitlichen Rahmen dieser Arbeit überschreiten würde.

Da die nichtfunktionalen Anforderungen primär als Selektionshilfe und Richtlinien bei der Entwicklung dienen sollen, verzichten wir zudem auf messbare Metriken.

NFA 1: Lokale Eingabelatenz

BESCHREIBUNG
Bei der Texteingabe muss eine geringe Latenz gegeben sein. Diese ist gerade bei Korrekturaufgaben, für die diese Softwarebibliothek ausgelegt sein soll, für eine gute Gebrauchstauglichkeit notwendig (Schmid u. a. 2023).

NFA 2: Latenz kollaborativer Änderungen

BESCHREIBUNG
Änderungen anderer Nutzender müssen zeitnah für alle Nutzenden sichtbar sein.

NFA 3: Latenz bei Operationen

BESCHREIBUNG
Operationen, die keine Texteingaben sind, wie etwa das Annehmen oder Ablehnen von Änderungsvorschlägen oder das Verschieben von Text, müssen zeitnah ausgeführt werden.

NFA 4: Konsistenz bei Nebenläufigkeit

BESCHREIBUNG
Gleichzeitige Änderungen müssen den Nutzererwartungen entsprechend aufgelöst werden.

NFA 5: Verständlichkeit für Entwickelnde

BESCHREIBUNG
Entwickelnde sollen die Schnittstellen gut verstehen und die Bibliothek dadurch schnell produktiv nutzen können.

NFA 6: Installierbarkeit der Bibliothek

BESCHREIBUNG
Die Bibliothek soll einfach in Anwendungen integrierbar sein.

NFA 7: Anbindung an Editor-Frameworks

BESCHREIBUNG

Die Bibliothek soll flexibel mit verschiedenen Bibliotheken wie UI-Komponenten nutzbar sein.

NFA 8: Geringer Entwicklungsaufwand

BESCHREIBUNG

Aufgrund des begrenzten Zeitrahmens soll ein geringer initialer Entwicklungsaufwand angestrebt werden.

4 Technische Konzepte verteilter Echtzeit-Kollaboration

Ziel dieser Arbeit ist es, eine Softwarebibliothek umzusetzen, die es erlaubt, an geteilten Nur-Text-Dokumenten gemeinsam Änderungsvorschläge zu machen oder Kommentare zu schreiben. Die technische Grundlage dafür bilden Echtzeit-Kollaborations-Protokolle, die eine gleichzeitige Bearbeitung von geteilten Datenstrukturen ermöglichen und dabei Konflikte auflösen oder sogar ganz vermeiden.

Im folgenden Kapitel werden die wichtigsten Konzepte solcher Protokolle – insbesondere Operational Transformation (OT) und Conflict-Free Replicated Data Types (CRDT) – vorgestellt und auf ihre Eignung für diese Arbeit hin untersucht. Ziel ist es zunächst, anhand einer Literaturrecherche die Forschungsfragen in Tabelle 8 zu beantworten.

Tabelle 8: Forschungsfragen zur verteilten Echtzeit-Kollaboration.

FF 2.1	Welche technischen Konzepte existieren für verteilte Echtzeit-Kollaboration?
FF 2.2	Inwiefern unterstützen die gefundenen Konzepte die Erfüllung der in Abschnitt 3 definierten Anforderungen?

4.1 Ursprünge verteilter Kollaboration

Die späten 1980er Jahre haben mit der breiter werdenden Verfügbarkeit des Internets auch eine Zunahme sogenannter Computer Supported Collaborative Work (CSCW) und Groupware Systeme gesehen. Während klassische Multi-User-Systeme wie Datenbanken oder Betriebssysteme lediglich mehrere Accounts unterstützen, definieren Ellis und Gibbs (1989) Groupware Systeme als Software, die die unmittelbare und spontane Zusammenarbeit mehrerer Personen ermöglicht – oft in verteilten Netzwerken, mit wechselnden Teilnehmenden, ohne vorab festgelegten Ablauf und unter Nutzung zusätzlicher Kommunikationskanäle wie Audio- oder Videoverbindungen.

Diese Art von Software bedingt in der praktischen Implementierung eine Architektur, bei der der geteilte Inhalt bei allen Teilnehmenden repliziert wird. Nur so können laut Ellis und Gibbs (1989) in verteilten Netzwerken bei nichtdeterministischen Netzwerkbedingungen eine hohe Verfügbarkeit und zufriedenstellende Reaktionszeiten sichergestellt werden.

Die durch diese Replikation mögliche Nebenläufigkeit führt jedoch auch zu Problemen. Bei einem kollaborativen Texteditor kommt es beispielsweise zu einem klaren Konflikt, wenn ein Satz gelöscht wird, während jemand anderes gleichzeitig ein Wort einfügt.

Um eine Nebenläufigkeitskontrolle zu implementieren, die solche Konflikte auflöst oder gänzlich vermeidet, können Ansätze wie etwa Transaktionen oder *Single Active Participant* verfolgt werden. Diese Ansätze benötigen jedoch alle eine Art von *Locking*, bei dem bei einer Bearbeitung die jeweilige Ressource für die Bearbeitung durch andere Nutzende gesperrt wird. Neben mehreren Problemen technischer Natur, die Ellis und Gibbs (1989) identifizieren, ist Locking auch durch Wartezeiten der allgemeinen Gebrauchstauglichkeit abträglich.

Daher entwickelten Ellis und Gibbs (1989) als eine der Ersten, eine Art von Nebenläufigkeitskontrolle, die Konflikte auflöst, ohne Locking zu benötigen. Dafür nutzen sie einen Algorithmus, der widersprüchliche Nutzeroperationen so transformiert, dass sie trotz ihres Konfliktes angewendet werden können.

Auf diesem Prinzip (der Operational Transformation (OT)) basieren etwa Google Docs und der Kooperationsmodus von Microsoft Word (Xu, Sun, und Li 2014). Allerdings handelt es sich bei den in der Industrie verwendeten OT Systemen nur um Abwandlungen des Algorithmus von Ellis und Gibbs (1989), denn die volle Dezentralisierung, die ihr Ansatz vorsieht, hat sich als sehr komplex herausgestellt. Stattdessen wird OT in der Praxis in fast allen Fällen in einem Server-Client Kontext eingesetzt, wie er von Nichols u. a. (1995) für das Jupiter System entwickelt wurde (siehe Abschnitt 4.3).

Um eine einfachere Dezentralisierung z. B. in Peer-to-Peer (P2P) Netzwerken zu ermöglichen, haben Oster u. a. (2006) eine Art von Algorithmus entwickelt, die später unter dem Namen CRDT formalisiert wurde. Ihr Ansatz *Without OT* (WOOT) unterscheidet sich vor allem dadurch, dass die voneinander unabhängigen Operationen eines Systems so entworfen werden, dass sie kommutativ sind, was die Dezentralisierung des Systems erheblich vereinfacht. Durch die Kommutativität der unabhängigen Operationen werden Konflikte nicht aufgelöst, sondern gänzlich vermieden, weshalb man von konfliktfreien Datentypen spricht.

CRDT werden bis heute in verschiedenen Publikationen weiterentwickelt (Weiss, Urso, und Molli 2008; Preguica u. a. 2009; Roh u. a. 2011; Lloyd u. a. 2011; Nicolaescu u. a. 2015; Briot, Urso, und Shapiro 2016; Weidner u. a. 2023) und es gibt einige ausgereifte Implementierungen (siehe Abschnitt 4.4).

Neben OT und CRDT, die in der Literatur gut behandelt sind, gibt es auch einige spezifische Implementierungen für verteilte Kollaboration an Textdateien, die davon abweichen. So hat Haverbeke (2015) beispielsweise für das Editorframework Prosemirror[5] einen eigenen Ansatz entwickelt, der sich grob an dem Client-Server Modell von OT orientiert. Der größte Unterschied besteht jedoch darin, dass eingehende Änderungen nicht transformiert werden. Stattdessen muss ein Client auftretende Konflikte zunächst lokal auflösen, indem die eigene Änderung angepasst wird.

Aufbauend darauf hat auch die Codehosting-Plattform GitHub einen auf der Versionsverwaltungssoftware Git basierenden Ansatz für Echtzeit-Kollaboration entwickelt (Donham und Rifkin 2023). Als Datenstruktur kommen dabei sogenannte Git-Object-Graphs zum Einsatz. Ein zentraler Server hält die aktuelle Version der gemeinsam bearbeiteten Struktur als Hash-Baum vor, während Clients kontinuierlich Änderungen *pullen* und eigene Änderungen lokal einfügen, bevor sie sie an den Server senden. Abweichende Zustände erfordern, wie bei Prosemirror oder eben Git, eine lokale Zusammenführung mit der aktuellen Serverversion.

Einen anderen Ansatz verfolgt die kollaborative Office-Suite CryptPad[6]. Sie verwendet eine Patch-Kette zur Darstellung eines Dokuments, die sich konzeptionell an der von Bitcoin bekannten Nakamoto-Chain orientiert (CryptPad 2013). Jeder Patch verweist dabei auf einen vorherigen Dokument-Zustand. Alle Clients halten lokal ein gemeinsames *Authorative Document* vor, während lokale Änderungen als *Uncomitted Work* zwischengespeichert und regelmäßig an einen zentralen Server gesendet werden. Eingehende Patches werden nur angewendet, wenn sie auf dem aktuellen lokalen Zustand des Authorative Document basieren;

[5] https://prosemirror.net/
[6] https://cryptpad.org/

bei Konflikten wird das Dokument auf den letzten gemeinsamen Zustand zurückgesetzt und dann mit den Änderungen der längsten Patch-Kette neu aufgebaut.

Im Folgenden werden die beiden am weitesten verbreiteten und wissenschaftlich am besten beschriebenen Ansätze für konfliktfreie Replikation in verteilten Editoren, OT und CRDT, genauer erläutert. Dafür werden zunächst die Probleme, die Nebenläufigkeit insbesondere bei der kollaborativen Arbeit an Textdokumenten mit sich bringt, definiert und dann erklärt, wie diese von OT und CRDT behandelt werden.

4.2 Probleme verteilter Kollaboration

Die bereits in Abschnitt 4.1 beschriebene Definition von Groupware-Systemen durch Ellis und Gibbs (1989) wurde von Sun u. a. (1998) auf die folgenden Charakteristika heruntergebrochen:

1. Echtzeit: Das System reagiert schnell auf lokale Eingaben. Die Latenz zur Anwendung von Änderungen anderer Nutzender hängt nur von der Latenz der externen Kommunikation zwischen den Geräten ab.
2. Verteilt: Mehrere Nutzende können von unterschiedlichen Geräten aus zusammenarbeiten, auch wenn sie über verschiedene Netzwerke mit unvorhersehbaren Latenzen verbunden sind.
3. Unbeschränkt: Alle Nutzenden dürfen gleichzeitig und uneingeschränkt an beliebigen Stellen des Dokuments arbeiten.

Auf Basis dieser Definition ist nur eine replizierte Architektur denkbar, um die Echtzeit-Anforderung zu erfüllen und schnell auf lokale Eingaben reagieren zu können.

Das heißt, auf den Geräten aller teilnehmenden Nutzenden ist eine Kopie des Dokuments vorhanden. In Abschnitt 4.1 wurde bereits zusammengefasst, dass dies aufgrund der dadurch möglichen Nebenläufigkeit zu Problemen führen kann. Diese Probleme sollen folgend eingehender betrachtet werden.

4.2.1 Das Modell

Um die Probleme aufzuzeigen, gehen wir hier von einem P2P System ohne eine Nebenläufigkeitskontrolle aus, wie es von Ellis und Gibbs (1989) definiert wird.

Ein solches Groupwaresystem G besteht aus einer endlichen Menge von Softwareinstanzen oder kurz *Instanzen*. Eine Instanz läuft auf einem Endgerät, wobei es auch denkbar wäre, dass ein Gerät mehrere Instanzen bereitstellt.
Eine Instanz hält jeweils ein Replikat der zu bearbeitenden Datenstruktur – das *Instanzobjekt* – und kann diese über nicht näher definierte Schnittstellen anzeigen. Die Nutzenden können diese Daten jederzeit bearbeiten. Theoretisch sind verschiedene Arten von Datenstrukturen möglich; der Anschaulichkeit halber gehen wir hier von einer Zeichenkette aus. Wir gehen zudem davon aus, dass alle Instanzen mit allen anderen Instanzen kommunizieren können. Formal kann so ein System wie folgt modelliert werden:

$G = (I, O)$, wobei
I eine Menge von Instanzen und
O eine Menge von möglichen *Operatoren* ist.

Die Menge O sind Operatoren die auf dem Instanzobjekt definiert sind. Wir definieren O beispielsweise als $\{O_1, O_2\}$, wobei gilt:

$O_1 = \text{einfügen}[X; P]$ Buchstaben X an Position P einfügen
$O_2 = \text{löschen}[P]$ Buchstaben an Position P löschen

Eine konkrete Ausprägung eines Operators ist eine Operation o, die auf ein Instanzobjekt angewendet werden kann; beispielsweise $o = O_1[u; 2]$. Wird o auf das Instanzobjekt *abc* angewendet, ändert sich dessen Zustand zu $o('\text{abc}') = '\text{abuc}'$.[7]

Eine Instanz führt drei Aktionen aus:

1. Operationsgenerierung: Eine Operation wird vom Nutzer einer Instanz spezifiziert. Die Instanz verpackt diese Operation in eine *Operationsanfrage*, die sie an alle Instanzen im System sendet.
2. Operationsempfang: Eine Operationsanfrage wird von einem anderen System empfangen.
3. Operationsausführung: Eine Operationsanfrage wird ausgeführt.

Nun geht es darum, bei der Ausführung von Operationen die Konsistenz des Systems zu wahren. Allgemein verstehen wir unter Konsistenz in diesem Kontext, dass nach Ausführung aller Operationen auf allen Instanzen die gleiche und vom Nutzer so beabsichtigte Datenstruktur entsteht – d. h. die Replikate aller Instanzobjekte sind gleich.

Zunächst soll anhand einiger Beispiele illustriert werden, zu welchen Arten von Inkonsistenzen es in einem System ohne Nebenläufigkeitskontrolle kommen kann. Dabei orientieren wir uns an Sun u. a. (1998). Stellen wir uns dafür ein in Abbildung 1 dargestelltes Groupwaresystem mit drei teilnehmenden Instanzen A, B und C vor. Jede generierte Operation wird unmittelbar auf der jeweiligen Instanz ausgeführt, an die anderen Instanzen gesendet und dort ebenfalls unverändert ausgeführt.

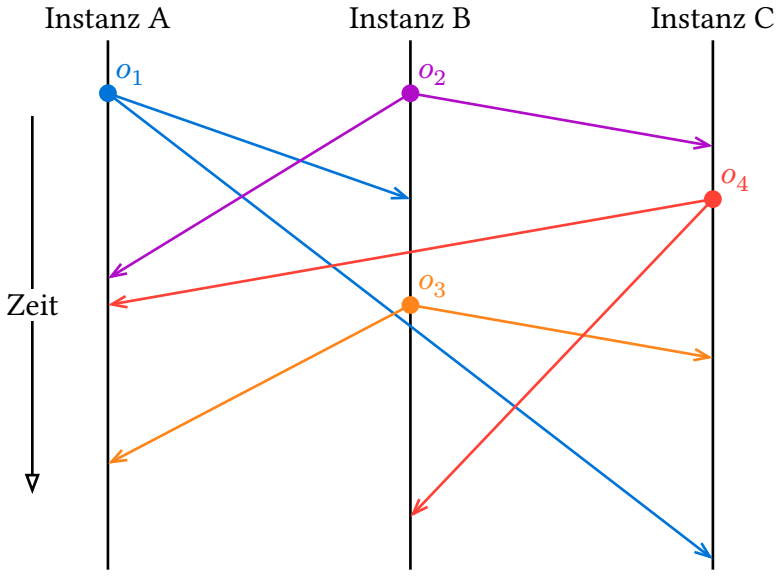


Abbildung 1: Ein beispielhafter Ablauf einer Echtzeit-Kollaborations-Sitzung (Sun u. a. 1998)

[7] Wir nutzen eine Nullindizierung, also z. B. $a_0 b_1 c_2$.

4.2.2 Divergenz

Aufgrund der nicht vorhersehbaren Netzwerklatenzen kann es dazu kommen, dass Operationen auf verschiedenen Instanzen in unterschiedlicher Reihenfolge ausgeführt werden. Betrachten wir Abbildung 1, dann werden die Operationen zwar in der zeitlichen Reihenfolge o_1, o_2, o_4, o_3 generiert, aber nur auf der *Instanz A* auch in der Reihenfolge o_1, o_2, o_4, o_3 ausgeführt, auf *Instanz B* hingegen in der Reihenfolge o_2, o_1, o_3, o_4 und auf *Instanz C* in der Reihenfolge o_2, o_4, o_3, o_1.

Da die Operationen, so wie sie in Abschnitt 4.2.1 definiert sind, nicht kommutativ sind, ist das Ergebnis der Ausführung aller Operationen auf allen Instanzen unterschiedlich.

4.2.3 Kausalitätsverletzung

Diese Divergenz kann zudem dazu führen, dass Operationen nicht in der richtigen kausalen Reihenfolge ausgeführt werden. So wird o_3 auf *Instanz B* generiert, nachdem o_1 dort ausgeführt wurde. Es ist also möglich, dass o_3 von o_1 kausal *abhängig* ist. Auf *Instanz C* trifft o_3 jedoch vor o_1 ein. Wenn allerdings eine Abhängigkeit zwischen o_1 und o_3 besteht, kann das im System oder beim Nutzer von *Instanz C* zu Verwirrung führen.

Gehen wir ganz konkret davon aus, dass o_1 den String “Halllo Welt” in das geteilte Dokument einfügt und o_3 das überflüssige “l” löscht, dann würde die Ausführung von o_3 vor o_1 dazu führen, dass o_3 auf nicht existenten Inhalt verweist.

Ein anderes Beispiel lässt sich anhand eines Quizzes demonstrieren: Stellt Alice mit o_1 die Frage „Wann findet der inoffizielle Wombat-Tag statt?“ und in o_3 beantwortet Bob als Nutzer von *Instanz B* die Frage mit „Am 22. Oktober“, dann wäre der Nutzer von *Instanz C* sehr verwundert, die Antwort „Am 22. Oktober“ vor der Frage zu sehen.

Es ist also wichtig, die kausale Reihenfolge von Änderungen zu bewahren. Dafür muss zumindest eine partielle Ordnung festgelegt werden.

4.2.4 Intentionsverletzung

Durch gleichzeitige Operationsgenerierung kann der *tatsächliche Effekt* einer Operation von ihrem *beabsichtigten Effekt* abweichen. Die Operationen o_1 und o_2 werden gleichzeitig generiert. Sie sind also kausal unabhängig voneinander. Dennoch kann es bei ihrer Ausführung zu einer Verletzung der Konsistenz kommen.

Das demonstriert ein Beispiel von Sun u. a. (1998): Das geteilte Dokument enthält die Zeichenkette „ABCDE“. Die Operation o_1 ist definiert als $o_1 = \text{einfügen}['12', 1]$, also die Zahl „12“ zwischen „A“ und „B“ einzufügen und o_2 ist definiert als $o_2 = \text{löschen}[2, 2]$, also zwei Buchstaben ab Stelle zwei zu löschen (hier „CD“). Werden beide Operationen ausgeführt, dann ist die *Intention* der Operationen „A12BE“. Der *tatsächliche Effekt* auf *Instanz A* ist jedoch „A1CDE“. Denn hier wurde o_1 zuerst ausgeführt und die Zahl „12“ eingefügt. Dadurch ändern sich die Indizes und durch o_2 wird ab der zweiten Stelle nicht mehr „CD“, sondern „2B“ gelöscht.

Von Weidner und Kleppmann (2023) wird dieser Effekt im Kontext von CRDTs auch als *Interleaving* bezeichnet. Kleppmann (2020) beschreibt es als eines der schwer zu lösenden Probleme bei der Implementierung von CRDTs.

Wir können die Intention einer Operation wie folgt definieren (Sun u. a. 1998):

Definition 1: Intention einer Operation

Die Intention einer Operation O ist der Ausführungseffekt, der durch die Anwendung von O auf den Dokumentenzustand, aus dem O erzeugt wurde, erreicht werden kann.

4.2.5 Konsistenzmodell

Unter Beachtung der in den Abschnitten Abschnitt 4.2.2, Abschnitt 4.2.3 und Abschnitt 4.2.4 beschriebenen Probleme können wir nun ein Konsistenzmodell erstellen. Dazu übernehmen wir das von Sun u. a. (1998) definierte Modell, verallgemeinern es aber soweit, dass es auch für CRDTs Gültigkeit besitzt.

Definition 2: Konsistenzmodell

Ein kooperatives Bearbeitungssystem gilt als konsistent, wenn es stets die folgenden Eigenschaften erfüllt:

1. **Konvergenz**: Wenn die gleiche Menge von Operationen auf allen Seiten ausgeführt wurde, sind die Instanzobjekte aller Instanzen identisch.
2. **Kausalitätswahrung**: Für jedes Paar von Operationen o_a und o_b gilt: Wenn o_b von o_a kausal abhängig ist, so wird o_a auf allen Instanzen vor o_b ausgeführt.
3. **Intentionserhaltung**: Für jede Operation o gilt: Die Wirkung von o auf allen Instanzen entspricht der Intention von o und die Ausführung von o verändert nicht die Ausführung kausal unabhängiger Operationen.

4.3 Operational Transformation

Wie in Abschnitt 4.1 schon beschrieben, entstand der erste Algorithmus zur OT von Ellis und Gibbs (1989) als ein P2P System. Diese Dezentralisierung ging allerdings einher mit einem hohen Implementierungsaufwand und hoher Komplexität (Nichols u. a. 1995).

Daher nutzen alle bekannten in der Praxis eingesetzten OT Systeme einen zentralen Server als Koordinator (Nicolaescu u. a. 2015). So kann die Kommunikation zwischen dem Server und den Clients als synchron und damit stark vereinfacht betrachtet werden. Die Instanzen müssen nicht mehr zu jeder Zeit Nachrichten von allen anderen Instanzen verarbeiten können.

Entwickelt wurde der erste Client-Server Ansatz auf Basis von *dOTP* von Nichols u. a. (1995) für die Software Jupiter. Diese Arbeit ist auch die Basis für Google OT, das für Google Wave entwickelt wurde und heute bei Google Docs im Einsatz ist (Xu, Sun, und Li 2014).

Aufgrund der geringen praktischen Relevanz von dezentralisierten OT Systemen erläutern wir OT im Folgenden anhand des Jupiter Client-Server-Ansatzes.

Das Modell aus Abschnitt 4.2.1 ändert sich also insofern, als dass nicht länger alle Instanzen direkt miteinander kommunizieren. Stattdessen synchronisiert sich jede Instanz mit einem zentralen Server, der selbst wie eine weitere Instanz agiert. Der Server hält den aktuellen Zustand des geteilten Dokuments vor, wendet eingehende Änderungen direkt auf diesen zentralen Zustand an und verteilt sie anschließend an alle übrigen Instanzen.

Zur Synchronisation kommt ein optimistischer Zweiparteien-Algorithmus zum Einsatz, der die tatsächliche Asynchronität der Netzwerkkommunikation ausblendet. Für den Server scheint es stets so, als hätte der Client alle bis dato empfangenen Nachrichten bereits abgearbeitet, bevor er eine neue Änderung übermittelt – und umgekehrt. Das ist wichtig, denn nur wenn die Kommunikation synchron ist oder zumindest zu sein scheint, funktioniert der Ansatz, einfach alle Änderungen eines Clients an die anderen weiterzuleiten.

In der Praxis kann es durch Netzwerklatenzen natürlich dennoch zu Abweichungen im Zustand zwischen Client und Server kommen. Der Algorithmus sorgt nun dafür, dass eingehende, widersprüchliche Änderungen so *transformiert* werden, dass diese relativ zum aktuellen Zustand des lokalen Dokuments Sinn ergeben.

Um festzustellen, ob zwei Operationen in Konflikt zueinander stehen können, führen sie jeweils den Zustand der Instanz mit, auf der sie generiert wurden. Dieser Zustand wird als Tupel (n_c, n_s) dargestellt aus der Zahl n_c der Operationen, die der Client generiert und angewendet hat und der Zahl n_s der Operationen, die er vom Server bekommen und angewendet hat. Anhand dieser Zahlen kann die Ordnung der Operationen abgeleitet werden. Diese zeitliche Ordnung stellt gleichzeitig die kausale Ordnung dar: o_4 ist von o_1 abhängig, weil o_1 vor o_4 generiert wurde, oder formal:

Definition 3: Kausale Ordnung bei Operational Transformation

Gegeben seien zwei Operationen o_1 und o_2, die auf den Instanzen A bzw. B erzeugt wurden. Dann gilt $o_1 \prec o_2$ genau dann, wenn eine der folgenden Bedingungen erfüllt ist:

1. $A{=}B$ und die Erzeugung von o_1 erfolgte vor der Erzeugung von o_2
2. $A{\neq}B$ und die Ausführung von o_1 auf der Instanz A erfolgte vor der Erzeugung von o_2 auf Instanz B
3. Es existiert eine Operation o_x, sodass $o_1 \prec o_x$ und $o_x \prec o_2$.

Ein Konflikt liegt nun beispielsweise vor, wenn beide Seiten unabhängig voneinander und zur gleichen Zeit Operationen generieren und so zunächst keine zeitliche und damit kausale Ordnung bestimmt werden kann. Denken wir an das Szenario aus Abschnitt 4.2.4:

Client und Server starten im Zustand $(0, 0)$ mit der Zeichenkette „ABCDE“. Auf diesen Zustand wendet der Client die Operation $o_1 = \text{einfügen}['12', 1]$ an und wechselt in den Zustand $(1, 0)$. Gleichzeitig wendet der Server die Operation $o_2 = \text{löschen}[2, 2]$ an und wechselt in den Zustand $(0, 1)$. Der hier auftretende Konflikt zwischen o_1 und o_2 kann auf dem Client aufgelöst werden, indem der Index von o_2 um die Einfügung von o_1 verschoben wird – in diesem Fall also um 2 Stellen. Damit befindet sich der Client im Zustand $(1, 1)$. Der Server kann o_1 unverändert anwenden, da die Löschoperation o_2 hinter der Einfügung liegt. Damit wechselt auch er in den Zustand $(1, 1)$ und der Konflikt ist aufgelöst.

Um solche Transformationen anzuwenden, wird eine allgemeine Funktion

$$\text{xform}(c, s) = \{c', s'\} \tag{1}$$

definiert, wo c und s die originalen Operationen von Server und Client (im Beispiel o_1 und o_2) sind. Die resultierenden c' und s' sind die transformierten Operationen für die gilt: Wenn

der Client s' anwendet und der Server c', dann sind Server und Client wieder im gleichen Zustand.[8]

Die Komplexität des Vorgehens bei OT besteht nun darin, die Transformationen so zu wählen, dass die in Definition 2 definierten Konsistenzkriterien eingehalten werden. Dabei können die notwendigen Transformationen als eine $m \times m$ Matrix betrachtet werden, wobei m die Mächtigkeit der Menge O, also die Zahl der möglichen Operationen, ist. D. h. für vier mögliche Operationen müssten bereits 16 Transformationen implementiert werden (Ellis und Gibbs 1989).

4.4 Conflict-Free Replicated Data Types

Der erste Ansatz zu CRDTs wurde von Oster u. a. (2006) unter dem Namen *WOOT* (Without OT) vorgeschlagen. Heute gibt es mit etwa RGA (Roh u. a. 2011) oder YATA (Nicolaescu u. a. 2015) Weiterentwicklungen dieses Algorithmus, die deutlich performanter sind und weitere Datentypen unterstützen. WOOT ist ohne solche Optimierungen jedoch einfacher zu verstehen und als direkte Nachfolge zu OT gut mit dieser zu vergleichen, weshalb wir das Prinzip von CRDTs im Folgenden anhand dieses Algorithmus erklären werden. Hier geht es um die Funktionsweise von CRDTs im Vergleich zu OT auf konzeptioneller Ebene. In einem späteren Kapitel (siehe Abschnitt 5) werden wir diese Prinzipien nochmals in einer vereinfachten, operationenbasierten Form betrachten, die als Grundlage für eine konkrete softwaretechnische Erweiterung dient.

Die meisten OT-Algorithmen setzen für das Kausalitätskriterium aus Definition 2 auf eine zeitliche Beziehung (siehe Definition 3). Eine Operation ist dabei von allen Operationen, die vor ihrer Erzeugung ausgeführt wurden, abhängig. Damit werden selbst eigentlich semantisch unabhängige Operationen voneinander abhängig. Bei WOOT hingegen leitet sich die Kausalität direkt aus der semantischen Abhängigkeit ab: So ist zum Beispiel die Löschung eines Buchstabens nur von dessen vorheriger Einfügung abhängig.

Das funktioniert, indem die Identifizierung von Elementen der Datenstruktur nicht mehr anhand des Indexes erfolgt, sondern indem sie eine eindeutige Kennung erhalten. Die in Abschnitt 4.2.1 definierten Operatoren $O_1 = \text{einfügen}[X; P]$ und $O_2 = \text{löschen}[P]$ müssen wir dazu anpassen:

$O_1 = \text{einfügen}[A \prec X \prec B]$	Buchstaben X zwischen den Buchstaben A und B einfügen
$O_2 = \text{löschen}[X]$	Buchstaben X löschen

Eine Einfügen-Operation ist so nur von der Präsenz der Buchstaben A und B abhängig, eine Löschung sogar nur von der Existenz des zu löschenden Buchstaben. Sind die entsprechenden vorausgehenden Operationen noch nicht vorhanden, kann einfach gewartet werden. Dadurch ist es egal, in welcher Reihenfolge Operationen auf einer Instanz eintreffen. Solange alle Operationen auf allen Instanzen verarbeitet werden, ist das System *irgendwann konsistent*.

Betrachten wir noch einmal das Beispiel aus Abschnitt 4.2.4: Zunächst müssen wir hier die Operationen in unser neues Format überführen. Aus $o_1 = \text{einfügen}['12', 1]$ wird $o_1 = \text{einfügen}["A" \prec "12" \prec "B"]$ und aus $o_2 = \text{löschen}[2, 2]$ wird $o_2 = \text{löschen}["\text{CD}"]$.

[8] Weichen die Zustände von Client und Server um mehr als eine Operation ab, muss dieser Ansatz erweitert werden. Das soll aber aus Zeitgründen hier nicht betrachtet werden.

Wenden wir diese Operationen an, ist es nun egal, ob zuerst o_1 und dann o_2 ausgeführt wird oder erst o_2 und dann o_1 – das Ergebnis wird in beiden Fällen das gleiche sein, da o_2 nicht kausal von o_1 abhängig ist und andersherum.

Einen Sonderfall bildet hier die Löschung: Angenommen in Operation o_4 entfernt Nutzer B das Zeichen b, während Nutzerin A in Operation o_3 (die die Löschung noch nicht erhalten hat) genau vor b ein X einfügen möchte ($o_3 = \text{einfügen}(X \prec b)$). Dann entsteht ein Konflikt, weil das X von Nutzerin A auf ein Zeichen referenziert, das Nutzer B bereits gelöscht hat. WOOT löst dieses Problem, indem es das gelöschte b intern als unsichtbaren *Grabstein* behält.

Ein Buchstabe besteht also bei WOOT nicht mehr nur aus einem alphabetischen Wert, der sich an einem Index befindet, sondern ist wie folgt aufgebaut:

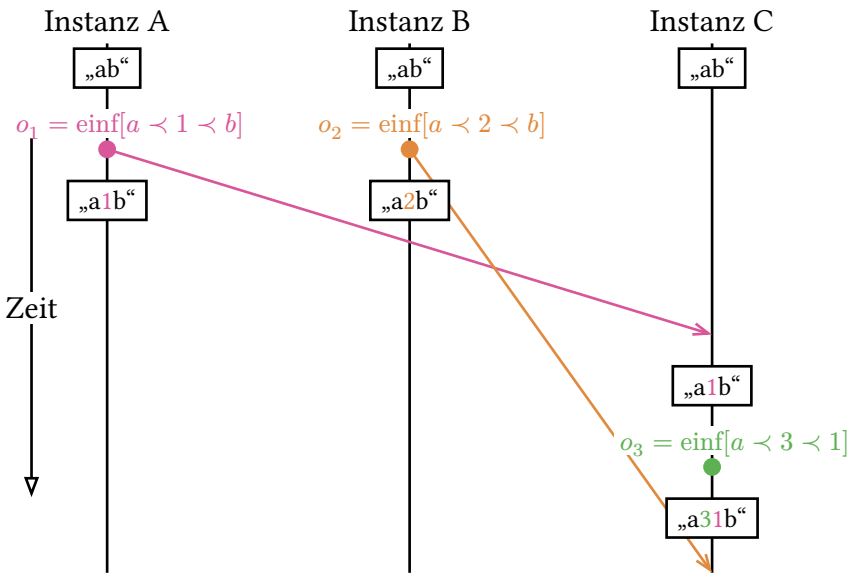
Ein W-Buchstabe ist ein fünf-Tupel $\left(\text{id}, \alpha, v, \text{id}_{\text{cp}}, \text{id}_{\text{cn}}\right)$, wo
- id die eindeutige Kennung des Buchstaben und
- α der alphabetische Wert des Buchstaben ist,
- $v \in \{\text{True}, \text{False}\}$ angibt, ob der Buchstabe sichtbar ist,
- id_{cp} die Kennung des vorherigen Buchstaben und
- id_{cn} die Kennung des nächsten Buchstaben ist.

Aus den Verweisen der einzelnen Buchstaben auf vorhergehende oder nächste Buchstaben entsteht nun eine Ordnung. Dabei handelt es sich jedoch nur um eine partielle und keine *totale Ordnung*. Das heißt, nicht alle Elemente verweisen in einem eindeutigen Sinne aufeinander. Eine totale Ordnung, also eine immer gleiche Reihenfolge, wird jedoch benötigt, um Konvergenz zu erreichen und allen Nutzenden das gleiche Dokument anzeigen zu können.

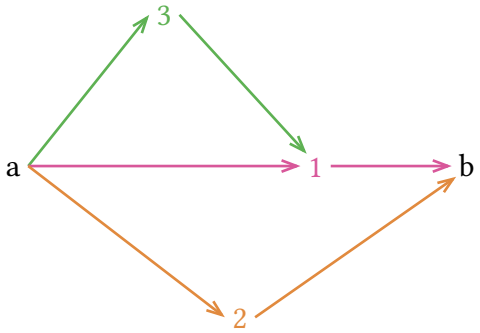
Gehen wir von drei Instanzen aus, die Operationen generieren, wie in Abbildung 2 dargestellt, stellen wir fest, dass es mehrere valide Reihenfolgen gibt, die sowohl die Intention, als auch die partielle Ordnung bewahren: „a312b“, „a321b“, „a231b“.
WOOT löst dieses Problem, indem eine monotone lineare Erweiterungsfunktion genutzt wird: Zunächst wird ein Buchstabe c mit den Kennungen $x \prec c \prec y$ zwischen x und y platziert. Gibt es jedoch mehrere Buchstaben, die zwischen x und y liegen, dann muss es eine monotone Berechnung geben, mit der die Reihenfolge auf allen Instanzen festgelegt wird. WOOT beachtet dabei zunächst, ob schon Abhängigkeiten zwischen diesen Buchstaben bestehen und zieht dann die Kennungen zur Festlegung der Reihenfolge der unabhängigen Buchstaben heran.

Sind in unserem Beispiel die Kennungen in der Reihenfolge ‚1‘ < ‚2‘ < ‚3‘, könnte das System die auf Instanz C eintreffende ‚2‘ aus o_2 entweder vor ‚3‘ oder nach ‚1‘ platzieren. Da jedoch bereits eine Abhängigkeit von ‚3‘ auf ‚1‘ besteht – das heißt, dass o_3 erst nach o_1 erzeugt wurde und es folglich Instanzen geben kann, auf denen ‚3‘ noch nicht existiert – muss sich die neue Einfügung an der unabhängigen ‚1‘ orientieren. Die daraus resultierende totale Ordnung lautet daher: „a312b“.

Die Methode, ein neues Element zu platzieren, muss also unabhängig von der Reihenfolge sein, in der die Elemente empfangen werden, um Konvergenz zu erreichen.



(a) Die beispielhafte Entstehung einer partiellen Ordnung



(b) Die Ordnungsbeziehungen der partiellen Ordnung

Abbildung 2: Partielle Ordnung (Oster u. a. 2006)

4.5 Konsistenz bei CRDTs und OT

Im Folgenden soll eine knappe Übersicht der wesentlichen Unterschiede zwischen CRDTs und OT gegeben werden, und zwar hinsichtlich der Art und Weise, wie sie die in Definition 2 definierten Konsistenzkriterien erfüllen.

Tabelle 9: Die Erfüllung der Konsistenzkriterien bei CRDTs und OT im Vergleich

	CRDTs	**OT**
Konvergenz	Unabhängige Operationen sind durch eindeutige Kennungen der Elemente kommutativ gestaltet, sodass es nicht auf die Reihenfolge der Anwendung ankommt, um Konvergenz zu erreichen.	Durch fest definierte Transformationsfunktionen werden widersprüchliche Operationen so angepasst, dass alle Instanzen bei Ausführung derselben Operationsmenge am Ende denselben Zustand erreichen.
Kausalitätswahrung	CRDTs leiten Kausalität rein aus semantischer Abhängigkeit ab. Eine Operation wird angewendet,	Die kausale Reihenfolge wird zeitlich definiert: Jede Operation wird erst nach allen kausal

	CRDTs	OT
	sobald ihre direkten Abhängigkeiten erfüllt sind.	(=zeitlich) vorausgehenden Operationen transformiert und angewendet.
Intentionserhaltung	Da jede Operation ein eindeutig identifizierbares Element adressiert, wirkt sie genau so, wie beabsichtigt, ohne dass spätere Änderungen die ursprüngliche Absicht verfälschen.	Durch die Transformationsregeln bleibt die ursprüngliche Intention jeder Operation erhalten (z. B. „füge X an Position p ein“ bleibt auch nach Verschiebungen semantisch gleich).

4.6 Entscheidung für ein Konzept der verteilten Echtzeit-Kollaboration

Nach der Vorstellung der grundlegenden Konzepte von OT und CRDTs sowie der spezifischen Probleme bei der verteilten Textkollaboration, soll in diesem Kapitel zur Beantwortung der FF 2.2 eine Entscheidung für eine der beiden Technologien getroffen werden. Diese Wahl ist zudem für die nachfolgende Konzeption und Implementierung der Softwarebibliothek für den Vorschlagsmodus wichtig. Als Entscheidungsgrundlage dienen die im Anforderungskatalog definierten nichtfunktionalen Anforderungen (NFAs), insbesondere im Hinblick auf Komplexität und Latenz sowie den pragmatischen Entwicklungsaufwand.

4.6.1 Komplexität

Wie in Abschnitt 4.3 dargelegt, werden OT-Systeme in der Praxis fast ausschließlich in einer Client-Server-Architektur umgesetzt. Dies vereinfacht zwar die Logik auf den Clients, verlagert jedoch die Komplexität der Zustandssynchronisation und der Operationstransformation auf einen zentralen Server. Insbesondere die Notwendigkeit, für jede mögliche Operationspaarung eine Transformationsfunktion zu definieren, führt zu einer hohen Implementierungskomplexität, die fehleranfällig ist und die Erweiterbarkeit des Systems erschwert (Ellis und Gibbs 1989).

CRDTs hingegen sind für dezentrale Architekturen wie Peer-to-Peer-Netzwerke konzipiert. Zwar erfordert das Design von CRDTs, wie in Abschnitt 4.4 am Beispiel WOOT gezeigt, ebenfalls einen durchdachten Aufbau (z. B. durch eindeutige IDs, Grabsteine und totale Ordnung), doch diese Komplexität steigt nicht quadratisch mit der Zahl der möglichen Operationen.

4.6.2 Latenz

Beide Ansätze sind darauf ausgelegt, eine niedrige lokale Eingabelatenz (NFA 1) zu ermöglichen, indem Änderungen sofort lokal angewendet werden. Bei CRDTs werden Operationen – egal ob lokal oder remote – auf ein abstraktes Zustandsmodell angewendet. Da die Logik zur Konfliktauflösung hier auf der lokalen Instanz implementiert ist, kann es dabei zu etwas höheren Latenzen kommen, als bei OT, wo lokale Änderungen sofort auf den Text angewendet werden können. Bei gut implementierten CRDTs sind diese Latenzen in der Praxis jedoch vernachlässigbar gering (Gentle 2021).

Die Latenz für kollaborative Änderungen (NFA 2) hängt bei beiden Konzepten stark von der Netzwerklatenz ab. Moderne CRDT-Implementierungen wie Yjs haben jedoch gezeigt, dass

sie in Benchmarks eine sehr hohe Performanz erreichen, die mit optimierten OT-Systemen vergleichbar ist (Weidner u. a. 2023).

4.6.3 Praktische Umsetzung und Ökosystem

Ein entscheidender Faktor für die Auswahl einer Technologie im Rahmen eines zeitlich begrenzten Projekts ist das verfügbare Ökosystem aus Bibliotheken, Werkzeugen und Dokumentationen (NFA 8). Hier zeigen sich deutliche Unterschiede zwischen OT und CRDT.

Das Ökosystem für OT ist stark fragmentiert. Während große Unternehmen wie Google und Microsoft robuste, aber proprietäre Implementierungen nutzen, ist die Zahl ausgereifter und aktiv gepflegter Open-Source-Bibliotheken gering. Die Einarbeitung in und die Anpassung von bestehenden OT-Frameworks ist oft aufwendig.

Im Gegensatz dazu hat sich in den letzten Jahren ein aktives Ökosystem um CRDTs gebildet. Projekte wie Yjs[9] oder Automerge[10] bieten gut dokumentierte und praxiserprobte Implementierungen. Zudem ermöglichen Projekte wie Collabs[11] (Weidner u. a. 2023) eine flexible Erweiterung bestehender und eine einfache Entwicklung neuer CRDT-Algorithmen. Community-Sammlungen wie crdt.tech[12] helfen beim Einarbeiten und bei der Recherche.

4.6.4 Fazit

Beide Technologien sind in der Lage, die grundlegenden Anforderungen an ein kollaboratives System zu erfüllen. Die Latenzanforderungen (NFA 1, NFA 2) werden durch die replizierte Architektur, die beiden Ansätzen zugrunde liegt, prinzipiell erfüllt. Auch die Konsistenz (NFA 4) wird durch beide Paradigmen gewährleistet, wenn auch auf unterschiedliche Weise. Während OT die Intention einer Operation durch Transformationen wahrt, vermeiden CRDTs Konflikte durch ihr Design. In seltenen Fällen kann es auch bei modernen CRDT-Algorithmen, wie in Abschnitt Abschnitt 4.2.4 beschrieben, zu Interleaving-Effekten kommen, die jedoch für die meisten Anwendungsfälle akzeptabel sind (Kleppmann 2020).

Der entscheidende Vorteil von CRDTs für diese Arbeit liegt jedoch in ihrer praktischen Zugänglichkeit und dem geringeren Implementierungsaufwand (NFA 8). Die Möglichkeit, auf eine stabile, performante und gut dokumentierte Grundlage zurückzugreifen, minimiert die Entwicklungsrisiken und ermöglicht die Konzentration auf die eigentliche Fachlogik des Vorschlagsmodus.

Aufgrund dieser Abwägung wird für die in dieser Arbeit entwickelte Softwarebibliothek auf die Technologie der CRDTs gesetzt.

[9] https://yjs.dev/
[10] https://automerge.org/
[11] https://collabs.readthedocs.io
[12] https://crdt.tech/

5 Vorschlagsmodus

Für die Entwicklung eines Vorschlagsmodus auf Basis der CRDT-Technologie liegt es nahe, zunächst keinen komplett neuen CRDT zu entwickeln, sondern diese Funktion mithilfe existierender CRDTs umzusetzen. Im Folgenden werden wir anhand von zwei Ansätzen jedoch zeigen, dass die Verwendung existierender CRDTs für unsere Zwecke nicht ohne weiteres möglich ist. Daher wird sich Abschnitt 5.3 mit der Konzeption eines eigenen CRDT mit integriertem Vorschlagsmodus beschäftigen.

Diesen Überlegungen zugrunde liegt der von Litt u. a. (2022) entwickelte Peritext CRDT für die Synchronisierung von Richtext-Formatierungen. Bei der Recherche zeigte sich nämlich, dass Richtext-Formatierungen und Vorschlagsannotationen relativ ähnlich sind und sich einige grundlegende Konzepte übernehmen lassen. Auch die anschauliche Art der Grafiken ist von Litt u. a. (2022) inspiriert.

5.1 Markup-Zeichen

Eine zunächst vielversprechend erscheinende Möglichkeit für die Entwicklung eines Vorschlagsmodus ist die Nutzung eines etablierten Markup-Systems, ähnlich wie es CriticMarkup[13] für Markdown-Dokumente tut. Die Grundidee dieses Ansatzes ist einfach: Änderungsvorschläge werden nicht in einer separaten Datenstruktur, sondern direkt im Text durch spezielle Steuerzeichen markiert. Löschungen, Einfügungen oder Ersetzungen werden so in den normalen Text eingebettet und können später vom UI speziell hervorgehoben werden.

Der Vorschlag einer Löschung, mit der das doppelte “ist” entfernt werden soll, würde mit CriticMarkup etwa so aussehen:

```
1 Dies ist {--ist --}eine Löschung.
```
MD

Während dieser Ansatz für einen einzelnen Autor oder für eine rein sequentielle Bearbeitung gut funktioniert, stößt er in einem kollaborativen Szenario mit gleichzeitig arbeitenden Nutzenden an seine Grenzen. Betrachten wir hierzu folgendes Beispiel:

Alice und Bob bearbeiten zur gleichen Zeit und ohne Kenntnis der Aktionen der jeweils anderen Person den Satz „Wombats sind flauschige Tiere!“.

Bob möchte das Wort „flauschige“ löschen:

```
1 Wombats sind {--flauschige --}Tiere!
```
MD

Während Alice den Vorschlag macht, “sind flau” zu löschen:

```
1 Wombats {--sind flau--}schige Tiere!
```
MD

Ein Standard-CRDT für Text würde beide Änderungen nun zusammenführen. Da er die spezielle Bedeutung der „`{--`“ und „`--}`“ Zeichen nicht kennt, behandelt er sie wie gewöhnliche Buchstaben oder Satzzeichen. Das Ergebnis der Zusammenführung könnte dann wie folgt aussehen:

[13] http://criticmarkup.com/

```
1 Wombats {--sind {--flau--}schige --}Tiere!
```
MD

An diesem Punkt ist die ursprüngliche Intention der Nutzenden nicht mehr zweifelsfrei nachvollziehbar. Durch die Verschachtelung der Markup-Zeichen ist es unmöglich geworden, die ursprünglichen Operationen – „lösche ‚flauschige'" von Bob und „lösche ‚sind flau'" von Alice – zu rekonstruieren. Man könnte das Ergebnis stattdessen auch so interpretieren, dass ein Nutzer die Löschung von „sind flauschige" vorgeschlagen hat, während ein anderer die Löschung von „flau" vorschlug.

Über dieses grundlegende Problem der Zusammenführung hinaus gibt es weitere Nachteile:

- Fehlende Zuordnung zum Autor: Ein reines Markup-System wie CriticMarkup enthält standardmäßig keine Informationen darüber, wer einen Vorschlag gemacht hat. Diese Information ist aber entscheidend, um Änderungen zu diskutieren und diese gezielt anzunehmen oder abzulehnen.
- Konflikte mit der Zielsprache: Die für das Markup verwendeten Zeichen könnten im bearbeiteten Textformat bereits eine andere Bedeutung haben. Stellt man sich vor, es würde nicht Markdown, sondern beispielsweise Programmcode bearbeitet, könnten Zeichen wie `{` und `}` die Syntax des Codes ungültig machen.

5.2 JSON-Dokument

Neben Nur-Text-CRDTs existieren etwa mit Automerge (Automerge 2025) auch Ansätze für kollaborative Systeme, die JSON-ähnliche Datenstrukturen nutzen.

Denkbar wäre, den Text als eine Liste von JSON Objekten zu speichern, die zusätzliche Meta-Informationen zu den jeweiligen Abschnitten enthalten. Das könnte z. B. wie folgt aussehen:

```
1 [
2   { "text": "Wombats sind ", "state": "accepted" },
3   { "text": "flauschige", "state": "proposed_deletion", "author": "Bob" },
4   { "text": " Tiere!", "state": "accepted" }
5 ]
```
JSON

Diese Struktur repräsentiert den Satz „Wombats sind flauschige Tiere!", wobei das Wort „flauschige" von Bob zur Löschung vorgeschlagen wurde.

Diese Repräsentation erweist sich jedoch als problematisch, wenn man die Zusammenführungs-Semantik von allgemeinen JSON-CRDTs anwendet. Wie bereits Litt u. a. (2022) für die Anwendung von Richtext-Formatierungen aufzeigen, kann dieser Ansatz die Absicht der Nutzenden bei konkurrierenden Änderungen verfälschen. Um dies für unseren Anwendungsfall zu demonstrieren, betrachten wir folgendes Szenario:

Der Ausgangszustand ist ein einfacher Satz, der als einzelnes Segment in unserer Liste gespeichert ist:

```
1 [{ "text": "Wombats sind Tiere!", "state": "accepted" }]
```
JSON

Alice möchte nun vorschlagen, das Wort „Tiere!“ zu löschen. Um dies im Datenmodell abzubilden, muss das ursprüngliche Segment aufgeteilt werden. Ihre Aktion wird in zwei Operationen übersetzt:

1. Der Text des ersten Segments wird auf „Wombats sind“ gekürzt.
2. Ein neues Segment `{ "text": "Tiere!", "state": "proposed_deletion", "author": "Alice" }` wird an Position 1 in die Liste eingefügt.

Bob möchte währenddessen vorschlagen, „Tiere!“ durch „Säugetiere!“ zu ersetzen. Dies ist eine kombinierte Aktion aus Löschung und Einfügung, die sein System ebenfalls durch Aufspaltung des Segments umsetzt:

1. Der Text des ersten Segments wird identisch zu Alices auf „Wombats sind“ gekürzt.
2. Ein Segment für die Löschung wird eingefügt: `{ "text": "Tiere!", "state": "proposed_deletion", "author": "Bob" }`.
3. Ein Segment für die Einfügung wird direkt dahinter eingefügt: `{ "text": "Säugetiere!", "state": "proposed_insertion", "author": "Bob" }`.

Wenn diese Änderungen zusammengeführt werden, kommt die Schwäche des Modells zum Vorschein. Ein CRDT wie Automerge, der Listen zusammenführt, betrachtet konkurrierende Einfügungen an derselben Position nicht als Konflikt, sondern erhält beide. Das Ergebnis der Zusammenführung konvergiert zu folgendem Zustand:

```JSON
[
  { "text": "Wombats sind ", "state": "accepted" },
  { "text": "Tiere!", "state": "proposed_deletion", "author": "Alice" },
  { "text": "Tiere!", "state": "proposed_deletion", "author": "Bob" },
  { "text": "Säugetiere!", "state": "proposed_insertion", "author": "Bob" }
]
```

Das Segment mit dem Text „Tiere!“ wurde im Datenmodell also dupliziert. Litt u. a. (2022) schreiben zwar, dass auch andere Merging-Algorithmen denkbar wären, aber sie haben keinen gefunden, der in allen Fällen das gewünschte Ergebnis liefert.

5.3 Der CRDT mit Vorschlagsmodus

Wie also können wir die Information, ob es sich bei einem Textabschnitt um einen Vorschlag handelt, effizient speichern? Die Lösung besteht in der Erweiterung eines existierenden Nur-Text-CRDTs. Im Folgenden wiederholen wir zunächst die zentralen Eigenschaften dieser Art von CRDT, bevor wir zeigen, wie Vorschläge und Kommentare erzeugt werden und wie diese Operationen in einen effizienten lokalen Zustand überführt werden können.

5.3.1 Nur-Text-CRDTs

Die grundlegenden Prinzipien von Nur-Text-CRDTs wurden bereits in Abschnitt 4.4 im Vergleich zu OT ausführlich dargestellt. In diesem Abschnitt steht nun ein anderer Aspekt im Vordergrund: Statt der konzeptionellen Abgrenzung geht es hier um eine softwaretechnisch orientierte Perspektive. Diese Darstellung dient als Grundlage für die im nächsten Abschnitt eingeführte Erweiterung um Vorschlags- und Kommentarfunktionalitäten.

Das CRDT besteht aus mehreren Instanzen, die jeweils ein eigenes Replikat des Dokuments führen. Alle Änderungen an einem Replikat erfolgen ausschließlich durch Operationen. Die Instanzen tauschen diese Operationen untereinander aus, um langfristig denselben Dokumentenzustand herzustellen.

Jede Operation erhält dabei eine eindeutige Kennung, bestehend aus einem lokalen Zähler, der bei jeder auf einer Instanz verarbeiteten Operation hochgezählt wird (ein sogenannter *Lamport-Zeitstempel*), sowie der eindeutigen Kennung der Instanz auf der die Operation generiert wird. Eine Operations-Kennung könnte also beispielsweise `32@node1` lauten, wobei `node1` die eindeutige Kennung der lokalen Instanz ist und `32` die Zahl der bisher auf `node1` verarbeiteten Operationen. Diese Operations-Kennung ist so über den gesamten CRDT eindeutig.

Mithilfe dieser Kennungen kann die für die Konvergenz erforderliche totale Ordnung abgeleitet werden: Dabei gilt `counter1@node1 < counter2@node2`, wenn `counter1 < counter2`. Wenn `counter1 = counter2`, dann kann ein alphabetischer Vergleich der Instanz-Kennungen den Konflikt auflösen. Die Kennungen dienen bei der Berechnung der totalen Ordnung in erster Linie der Konfliktlösung bei unabhängigen Änderungen. Das genaue Vorgehen ist in Abschnitt 4.4 beschrieben.

In unserer vereinfachten Form eines Nur-Text-CRDTs besteht das zu bearbeitende Dokument aus einer Datenstruktur, die die Abfolge von Buchstaben enthält[14]. Diese Buchstaben haben alle eine eindeutige Kennung, in unserem Fall die Kennung der Operation, die sie erzeugt hat.

Um einen Buchstaben einzufügen, erzeugen wir nun eine Operation nach dem folgenden Schema:

```JSON
{
  action: "insert",
  id: "2@A",
  char: "a",
  id_before: "1@A"
}
```

Das Zeichen „a“ wird hier nach dem Zeichen mit ID `1@A` eingefügt. Anders als im WOOT-Algorithmus wird nur der vorhergehende Buchstabe angegeben, nicht auch der nachfolgende. Diese Vereinfachung reicht für die meisten Anwendungen aus.

Die Löschung eines Buchstaben kann beispielsweise durch die folgende Operation vorgenommen werden:

```JSON
{
  action: "delete",
  id: "8@B",
```

[14]Vereinfacht deshalb, weil moderne und performante CRDT-Implementierungen wie etwa im Framework Yjs in der Regel nicht auf der Ebene einzelner Zeichen, sondern auf der Ebene größerer Textsegmente – etwa Wörter, Sätze oder Absätze – operieren. Dadurch sinkt die Anzahl der im CRDT verwalteten Einheiten, was Insert- und Delete-Operationen beschleunigt und zugleich den Speicherbedarf reduziert, da weniger Metadaten pro Zeichen anfallen.

```
4    id_remove: "1@A"
5  }
```

Statt das Zeichen dauerhaft zu entfernen, wird es lediglich als unsichtbar markiert. So bleibt seine Referenzierbarkeit für spätere Operationen erhalten.

Den Zustand unseres Dokuments können wir uns also wie in Abbildung 3 vorstellen.



Abbildung 3: Der Zustand eines lokalen Dokuments. Jeder Buchstabe hat eine eindeutige Kennung. Ein Boolean Wert markiert, ob der Buchstabe angezeigt wird oder nicht.

Entscheidend für den Vorschlagsmodus ist nun die Erweiterung eines solchen Nur-Text-CRDTs um eine zweite Datenstruktur, die Annotationen für Textabschnitte vorhält.

5.3.2 Einfüge-Vorschlag hinzufügen

Die zweite Datenstruktur ist eine Folge von Objekten, die Informationen zur Art der Annotation, dem Start und dem Ende des Textabschnitts sowie verschiedene Metainformationen enthalten. Da die Folge Teil eines CRDTs ist, kann sie ebenfalls nur über Operationen verändert werden.

Arbeitet nun also Alice im Vorschlagsmodus und möchte einen Buchstaben einfügen, dann werden zwei Operationen generiert. Mit der ersten Operation wird, wie bereits beschrieben, zunächst der Text eingefügt:

```JSON
1  {
2    action: "insert",
3    id: "1@A",
4    char: "L",
5    id_before: "start"
6  }
```

Die zweite Operation betrifft den Vorschlagsmodus und hat das folgende Schema:

```JSON
1   {
2     action: "addition"
3     description: "insert_suggestion",
4     id: "9@A"
5     start_range: null,
6     open_start: true,
7     end_range: "1@B",
8     open_end: true
9     //... einige Meta-Informationen
10  }
```

Zentral dabei ist das Feld `action`, das die Art der Operation beschreibt. Hier handelt es sich um eine `addition`-Operation, da eine Annotation hinzugefügt wird. Außerdem gibt es `update`- und `removal`-Operationen, dazu später mehr.

Auch diese Operation besitzt eine eindeutige Kennung. Zusätzlich enthält sie ein Beschreibungsfeld, das die Art der Operation beschreibt – in diesem Fall handelt es sich um einen Einfüge-Vorschlag.

Wird diese Operation angewendet, ergibt sich der in Abbildung 4 dargestellte Zustand. Eine Annotation, die einen Vorschlag markiert, reicht vom Startpunkt über den eingefügten Buchstaben bis zum darauffolgenden Buchstaben.

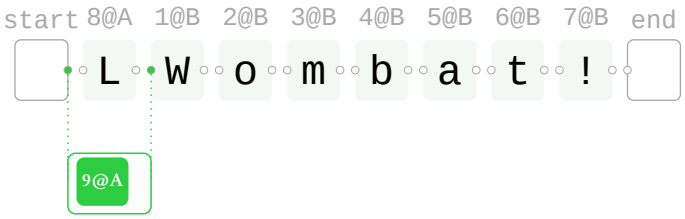


Abbildung 4: Fügt Alice einen Buchstaben im Vorschlagsmodus hinzu, wird neben dem Buchstaben auch eine Annotation eingefügt, die vom Startpunkt über den eingefügten Buchstaben bis zum darauffolgenden Buchstaben reicht.

Besonders wichtig sind die Felder `start_range` und `end_range`: Sie geben an, von welchem bis zu welchem Zeichen die einzufügende Annotation reicht. Da hier nur ein einzelner Buchstabe (`8@A`) eingefügt wird, umschließt die Range zunächst genau diesen einen Buchstaben.

Damit jedoch nicht für jeden neu eingefügten Buchstaben sofort eine eigene Annotation entsteht – was sowohl unpraktisch für Nutzende als auch schlecht für die Performance wäre – kommen die `open_end`- und `open_start`-Markierungen ins Spiel, die über den eingefügten Buchstaben hinaus reichen.

Zur Veranschaulichung: Man kann sich vorstellen, dass jeder Buchstabe zwei Ankerpunkte besitzt – einen davor und einen danach. Endet eine Range am ersten Ankerpunkt, gehört der Buchstabe nicht mehr zur Range. Endet sie am zweiten Ankerpunkt, ist er weiterhin Teil der Range. Beim Start einer Range ist es genau umgekehrt: Startet die Range am ersten Ankerpunkt, ist der Buchstabe Teil der Range, startet sie am zweiten Ankerpunkt, ist er es nicht.

Das Feld `end_range` in unserer Operation beschreibt, dass das Ende der Range bei Buchstabe `1@B` liegt. Der Wert `open_end = true` gibt an, dass der erste Ankerpunkt verwendet wird – der Buchstabe ist also nicht mehr in der Range enthalten. Der Wert `open_start = true` gibt Analog dazu an, dass der zweite Ankerpunkt des Startpunkt genutzt wird. Dass das `start_range` Feld dabei auf `null` gesetzt ist, bedeutet, dass der Start des Dokuments als Startpunkt genutzt wird.

Wenn Alice nun, wie in Abbildung 5 dargestellt, direkt hinter `8@A` weitere Buchstaben einfügt, liegen diese weiterhin vor `1@B` und bleiben damit Teil der Range. Das gleiche gilt auch für Buchstaben, die sie vor `8@A` einfügt.

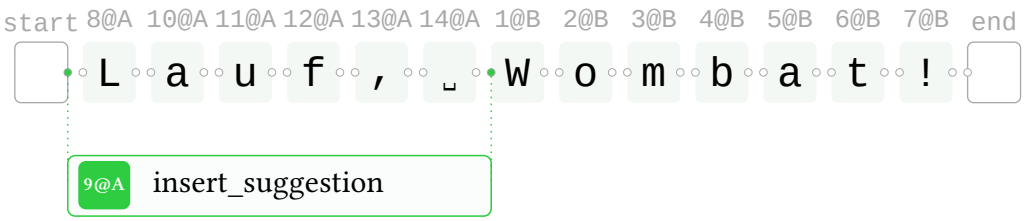


Abbildung 5: Werden innerhalb der Range Buchstaben eingefügt, so wächst die Range, ohne dass eine neue Range erstellt werden muss.

5.3.3 Kommentar hinzufügen

Während wir bei einem Einfüge-Vorschlag möchten, dass er automatisch wächst, sobald dahinter oder davor getippt wird, ist das bei einem Kommentar nicht so (siehe TA 11). Hier soll ein konkreter Textabschnitt markiert und mit ein paar Worten versehen werden können. Würde der Kommentar automatisch wachsen, sobald dahinter Buchstaben eingefügt werden, könnte das die Absicht des Autors verfälschen. Daher setzen wir hier den `open_end` und `open_start` Wert auf `false` und geben das inklusive Ende und den inklusiven Start der Range an. Eine Operation würde dann beispielsweise so aussehen:

```JSON
{
  action: "addition",
  description: "add_comment",
  id: "15@B"
  start_range: "8@A",
  open_start: false,
  end_range: "12@A",
  open_end: false,
  value: "Wohin läuft er denn?"
  //... einige Meta-Informationen
}
```

Wie eine solche geschlossene Annotation dann aussieht, sehen wir in Abbildung 6.

Den Inhalt des Kommentars speichern wir hier der Einfachheit halber direkt in der Operation, denkbar wäre es allerdings auch, den Kommentar in einer externen Datenbank abzulegen und hier nur mit einer entsprechenden Kennung darauf zu verweisen.

5.3.4 Annotationen aktualisieren

Während Einfüge-Vorschläge im Laufe der Bearbeitung wachsen können, indem zwischen ihrem Endpunkt und ihrem Ankerpunkt neue Zeichen eingefügt werden, wachsen Lösch-Vorschläge ausschließlich dann, wenn Nutzende explizit weitere Zeichen entfernen. Wird ein Zeichen zur Löschung vorgeschlagen, so wird keine unmittelbare Textänderung vorgenommen. Stattdessen erzeugen wir eine Lösch-Vorschlag-Operation, deren Start- und Endpunkt auf genau dieses zu löschende Zeichen zeigen. Der Wert `open_end` ist dabei auf `false` gesetzt, um das Wachsen des Vorschlags durch Einfügungen am Rand zu vermeiden.

Ein Problem ergibt sich, wenn ein Nutzer unmittelbar neben einem bestehenden Lösch-Vorschlag ein weiteres Zeichen entfernt. Nach dem bisherigen Modell würde dies zu einem

separaten Lösch-Vorschlag führen. Dieses Verhalten entspricht jedoch nicht den Erwartungen der meisten Nutzenden: Sie würden intuitiv davon ausgehen, dass sich der bestehende Vorschlag einfach „ausweitet“, um das neu gelöschte Zeichen mit einzuschließen (siehe TA 4).

Ein ähnliches Problem tritt auf, wenn ein Nutzer im Editierungsmodus am Rand eines bestehenden Vorschlags schreibt (siehe TA 3). Er würde erwarten, dass die Annotation nicht mehr mitwächst, da er ja nicht mehr im Vorschlagsmodus schreibt. Mit dem bestehenden Modell ist das allerdings nicht möglich, da die Ankerpunkte der Annotation auf die vorherigen bzw. nächsten Buchstaben verweisen.

Abhilfe kann bei diesen Problemen geschaffen werden, indem wir einen Mechanismus zum Aktualisieren von Annotationen schaffen. Dafür führen wir eine zweite Art von Operationen ein: Die `update`-Operation. Gehen wir beispielsweise von einem Zustand wie in Abbildung 6 dargestellt aus, in dem die Zeichen `13@A` und `14@A` zur Löschung vorgeschlagen wurden.

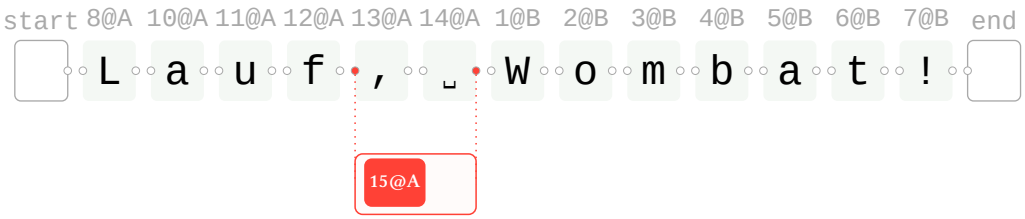


Abbildung 6: Ein beispielhafter Lösch-Vorschlag im Vorschlagsmodus. Die Zeichen `13@A` und `14@A` wurden zur Löschung vorgeschlagen.

Angenommen, Alice, die diese Löschung vorgeschlagen hat, möchte nun auch den Buchstaben `12@A` löschen. Statt nun einen neuen Lösch-Vorschlag zu erstellen, können wir mit der folgenden `update`-Operation den aktuellen Vorschlag erweitern.

```json
{
  action: "update",
  id: "16@A",
  dependent_on: "15@A"
  updated_properties: {
    start_range: "12@A"
  }
}
```

Hierzu verweist das Feld `dependent_on` eindeutig auf den zu aktualisierenden Vorschlag. In `updated_properties` werden ausschließlich die zu ändernden Werte angegeben. Durch diesen eindeutigen Bezug bleibt die Kommutativität gewahrt: Trifft die referenzierte Operation später ein, wird die `update`-Operation erst dann angewendet. Eine ausführlichere Darstellung dieses Vorgehens findet sich in Abschnitt 6.2.

Das Wachsen von Vorschlägen beim Wechsel in den Editierungsmodus kann nun verhindert werden, indem der schließende Ankerpunkt des Einfüge-Vorschlags, an dessen Rand im Editierungsmodus geschrieben wird, auf den neu eingefügten Buchstaben verschoben wird. Da – wie bei allen Einfüge-Vorschlägen – der vordere Ankerpunkt des Buchstabens verwendet

wird, ist der neue Buchstabe kein Teil der Annotation. Alle weiteren Buchstaben werden *hinter* diesem eingefügt und liegen damit ebenfalls nicht mehr innerhalb der Annotation.

5.3.5 Annotationen entfernen

Um Vorschläge zu entfernen, können sie entweder angenommen oder abgelehnt werden, während Kommentare als gelöst markiert werden können. Das geschieht durch eine dritte Art von Operationen: die `removal`-Operation.

Wird beispielsweise wie in Abbildung 5 ein Vorschlag angezeigt und möchte Bob diesen annehmen, erstellen wir eine Operation wie die folgende:

```JSON
{
  action: "removal",
  description: "accept_suggestion",
  id: "15@B",
  dependent_on: "9@A"
  //... weitere Metainformationen
}
```

Neben der Annahme eines Vorschlags durch `accept_suggestion` ist auch die Ablehnung (`decline_suggestion`) sowie das Entfernen eines Kommentars durch `remove_comment` vorgesehen. Entscheidend ist dabei stets, dass jede dieser Operationen explizit angibt, auf welche Range sie sich bezieht (`dependent_on`).

5.3.6 Die lokale Darstellung

Der Zustand eines Dokuments ergibt sich theoretisch aus der Gesamtheit aller Operationen, die jemals von allen Instanzen generiert wurden. Eine bloße Liste dieser Operationen ist jedoch weder für Nutzende noch für Programme leicht interpretierbar. Daher stellt sich die Frage, wie eine lokale Repräsentation des bearbeiteten Dokuments gestaltet sein kann, sodass sowohl lesende Zugriffe als auch die Verarbeitung eingehender Operationen effizient möglich sind.

Das grundsätzliche Modell wurde in den Grafiken der vorhergehenden Kapitel schon verwendet: Dabei wird von zwei Datenstrukturen ausgegangen. Eine Datenstruktur beinhaltet alle Buchstaben mit ihrer Kennung und der Information, ob sie gelöscht sind oder nicht. Eine separate Datenstruktur wird dabei genutzt, um die Meta-Informationen, also die Annotationen, zu speichern. Diese Annotationsstruktur kann (aber muss nicht) für jeden Buchstaben des Rohtextes zusätzliche Informationen enthalten.

Beim Initialisieren unseres lokalen Datenmodells bestehen daher diese zwei leeren Datenstrukturen: eine für den Text und eine für die Annotationen. Treffen nun `insert`-Operationen für den Text ein, werden die entsprechenden Zeichen in die Textstruktur eingefügt. Woher die Operationen kommen, ist dabei irrelevant. Möglich ist z. B., dass sie aus dem Speicher geladen werden oder von einer Remote-Instanz eintreffen. Dadurch entsteht das uns bereits aus Abbildung 3 bekannte Modell.

Nehmen wir nun an, dass die Zeichenkette “`Lauf,␣`” wie in Abbildung 5 im Vorschlagsmodus geschrieben wurde. Es trifft also eine `addition`-Operation mit der Beschreibung

`insert_suggestion` ein, die den Bereich von `8@A` bis `1@B` umfasst. Daraus muss nun eine Annotation generiert und diese in die lokale Struktur eingefügt werden. Dazu werden in der Annotationsstruktur sogenannte Datenpunkte an den Positionen des Start- und Endzeichens eingefügt. Ein Datenpunkt ist ein Array von Objekten, die jeweils eine Annotation beschreiben – abgeleitet aus den Operationen, die die Annotation erstellen. Am Endpunkt des Bereichs wird das entsprechende Objekt zusätzlich mit der Information versehen, dass hier der Gültigkeitsbereich der Annotation endet.

Zu Problemen kann es nach diesem Modell nun kommen, sobald sich zwei Vorschläge überlappen wie in Abbildung 7.

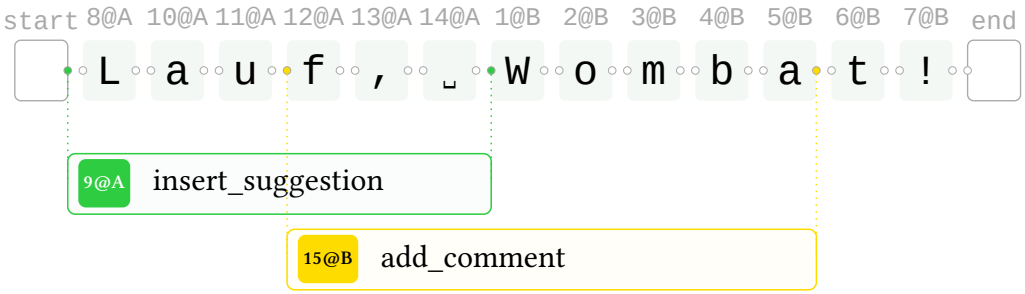


Abbildung 7: Eine Annotation reicht von `8@A` bis `1@B`, während eine zweite Annotation von `12@A` bis `5@B` reicht. Sie überlappen sich.

Angenommen, wir möchten wissen, welche Annotationen am Buchstaben `12@A` gültig sind. An dieser Position ist bereits ein Datenpunkt enthalten, der die Annotation `add_comment` enthält. Es bleibt jedoch unklar, ob davor schon ein Annotationsbereich – wie hier die `insert_suggestion` Annotation – begonnen hat, der bis hierher reicht. Um das herauszufinden, müsste das Dokument rückwärts bis zum Anfang durchsucht werden. Das müsste beispielsweise immer dann passieren, wenn im Vorschlagsmodus ein Buchstabe eingefügt wird, da es dann relevant ist, ob an dieser Stelle bereits eine Vorschlagsannotation existiert. In unserem kleinen Beispiel ist das noch möglich, stellen wir uns allerdings vor, dass das Dokument mehrere tausend Zeichen lang ist, so ist es höchst ineffizient, bei jeder Einfügung das komplette Dokument zu durchsuchen.

Um dieses Problem zu lösen, bedienen wir uns eines Tricks wie ihn auch Litt u. a. (2022) vorschlagen: In jedem Datenpunkt speichern wir alle ab diesem Punkt gültigen Annotationen.

Das bedeutet: Wird eine neue Annotation eingefügt und werden dafür an Start- und Endposition neue Datenpunkte erzeugt, werden beim Anlegen dieser Datenpunkte alle zu diesem Zeitpunkt gültigen Annotationen aus dem vorhergehenden Datenpunkt kopiert – allerdings nur jene, deren Gültigkeit nicht bereits dort endet. Wie das aussieht, wird in Abbildung 8 dargestellt.

Möchten wir nun die an `12@A` gültigen Annotationen ermitteln, genügt es, ab dieser Position rückwärts zu gehen, bis der erste Datenpunkt gefunden wird. In unserem Beispiel befindet sich dieser direkt an `12@A`. Alle dort aufgeführten Annotationen, die kein Endsymbol tragen, gelten als an dieser Stelle aktiv.

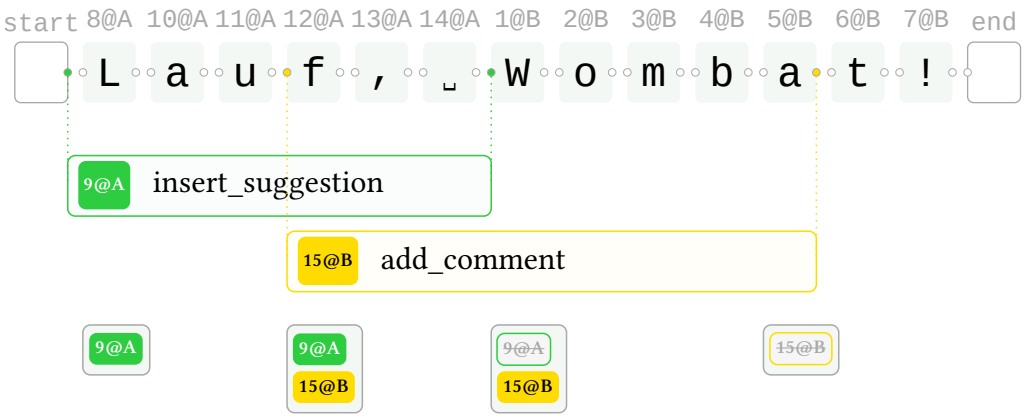


Abbildung 8: Die Datenpunkte enthalten alle ab ihrer Position gültigen Annotationen.

5.3.7 Verarbeitung eingehender Operationen

Die im vorangegangenen Abschnitt beschriebene lokale Darstellung der Annotationsstruktur enthält nur alle gültigen Annotationen, die durch `addition`-Operationen erstellt werden. Diese Annotationen können jedoch wie beschrieben durch andere Operationen bearbeitet und entfernt werden. Wie also wird sichergestellt, dass die lokale Annotationsstruktur zu jedem Zeitpunkt nur die gültigen und aktuellen Annotationen enthält, egal in welcher Reihenfolge die Operationen eintreffen, die zu diesen Annotation führen?

Die Lösung liegt in einem zweistufigen Vorgehen: Während eine primäre Datenstruktur alle jemals eingetroffenen Operationen persistent und unveränderlich speichert (ein sogenanntes Append-only-Log), wird die in Abschnitt 5.3.6 beschriebene, für die Darstellung relevante lokale Annotationsstruktur aus diesen Operationen abgeleitet.

Entscheidend für diesen Prozess ist, dass für jede eingehende Operation zunächst ihre Historie aus dem Log extrahiert wird. Die Historie einer Annotation besteht aus der initialen `addition`-Operation sowie allen dazugehörigen `update`- und `removal`-Operationen, die sich auf dieselbe Kennung beziehen. Diese Historie wird anschließend anhand des Lamport-Zeitstempels sortiert, um eine auf allen Instanzen gleiche Reihenfolge der Ereignisse herzustellen. Je nachdem, um was für eine Art von Operation es sich handelt, wird die eintreffende Operation dann unterschiedlich behandelt:

5.3.7.1 Verarbeitung einer `addition`-Operation

Trifft eine `addition`-Operation ein, wird ihre sortierte Historie zunächst daraufhin überprüft, ob bereits eine `removal`-Operation mit einem späteren Zeitstempel existiert. Ist dies der Fall, so gilt die Annotation als bereits bei ihrer Ankunft widerrufen und wird ignoriert. Existiert keine solche `removal`-Operation, werden alle `update`-Operationen aus der Historie auf die `addition`-Operation angewendet. Diese finale Version wird schließlich in die lokale Annotationsstruktur eingefügt.

5.3.7.2 Verarbeitung einer `update`-Operation

Eine eintreffende `update`-Operation kann nur verarbeitet werden, wenn die zugehörige `addition`-Operation bereits bekannt ist. Ist das nicht der Fall, wird die `update`-Operation zunächst ignoriert und erst beim Eintreffen der `addition`-Operation angewendet.

Ist diese allerdings schon vorhanden, wird auch hier zuerst die Historie überprüft: Existiert eine `removal`-Operation mit einem späteren Zeitstempel, ist die gesamte Annotation bereits ungültig, und die `update`-Operation wird verworfen. Andernfalls wird, wie bei der `addition`-Verarbeitung beschrieben, der aktuelle Zustand der Annotation durch die Anwendung aller vorangegangenen `update`-Operationen aktualisiert. Anschließend wird die (möglicherweise bereits in der lokalen Struktur vorhandene) veraltete Version der Annotation entfernt und durch die neu berechnete, aktuellere Version ersetzt.

5.3.7.3 Verarbeitung einer `removal`-Operation

Die Verarbeitung einer `removal`-Operation ist geradlinig. Nach der Sortierung der Historie wird geprüft, ob diese `removal`-Operation die Operation mit dem jüngsten Zeitstempel ist. Wenn ja, wird die entsprechende Annotation – falls vorhanden – endgültig aus der lokalen Annotationsstruktur entfernt. Sollte in der Historie jedoch eine andere Operation mit einem noch späteren Zeitstempel existieren (z. B. eine spätere `update`-Operation), wäre diese `removal`-Operation bereits überholt und würde ignoriert werden.

6 Implementierung

Auf Basis der zuvor vorgestellten Konzepte widmen wir uns nun der prototypischen Umsetzung einer Softwarebibliothek. Ziel dieser Implementierung ist es, zu überprüfen, ob das entwickelte Verfahren zur Verarbeitung und Speicherung von Vorschlägen und Kommentaren an einem Nur-Text-CRDT wie erwartet funktioniert. Dazu werden zunächst geeignete Technologien ausgewählt. Anschließend wird die Architektur erläutert und es werden zentrale Aspekte der Implementierung beleuchtet. Abschließend wird aufgezeigt, wie mithilfe der bereitgestellten Schnittstellen eine mögliche Integration der Softwarebibliothek in ein UI realisiert werden kann.

6.1 Technologien

Am Anfang der Implementierung steht die Auswahl eines geeigneten Technologie-Stacks. Zentral ist dabei die Entscheidung über die Plattform, auf der die Bibliothek realisiert werden soll. Auch die Wahl einer passenden Echtzeit-Kollaborations-Bibliothek spielt eine wichtige Rolle, da im Rahmen dieser Bachelorarbeit nicht genügend Zeit zur Verfügung steht, um eine gesamte moderne CRDT-Infrastruktur – wie etwa die Kommunikation zwischen Instanzen oder die Serialisierung von Operationen in Binärformate – vollständig selbst zu entwickeln.

6.1.1 Auswahl der Plattform

Im Folgenden wird der Begriff Plattform in Bezug auf die Gesamtheit der technologischen Grundlagen, auf denen eine Software implementiert wird, verwendet. Dazu zählen unter anderem die verwendeten Programmiersprachen, Laufzeitumgebungen, verfügbare Bibliotheken sowie die unterstützenden Entwicklungswerkzeuge.

Typische Beispiele sind etwa Plattformen wie die Java Virtual Machine (JVM), auf der Sprachen wie Java, Kotlin oder Scala ausgeführt werden, oder native Umgebungen wie C/C++ mit entsprechender Compiler-Toolchain. In diesem Projekt fiel die Entscheidung jedoch zugunsten der Webplattform, konkret unter Verwendung von TypeScript als Programmiersprache sowie dem Browser als Laufzeitumgebung.

Diese Plattform hat sich in den letzten Jahren besonders im Kontext kollaborativer Anwendungen als attraktiv erwiesen. Anwendungen, die direkt im Browser ausgeführt werden und einfach über einen Link zugänglich sind, bieten eine niedrige Einstiegshürde im Gegensatz zu klassischen Desktop-Anwendungen, da die Notwendigkeit einer Installation entfällt. Das erleichtert die Verbreitung und Nutzung wesentlich. Darüber hinaus ermöglicht die Webplattform eine vom Betriebssystem unabhängige Nutzung, da moderne Browser auf nahezu allen Geräten verfügbar sind.

Ein weiterer Vorteil ist das umfangreiche Ökosystem an CRDT-Bibliotheken, das auf der Webplattform zur Verfügung steht. Viele der etablierten Bibliotheken, etwa Automerge oder Yjs, wurden ursprünglich für den Einsatz im Browser entwickelt. Zwar existieren inzwischen auch CRDT-Implementierungen für andere Sprachen wie Rust[15] oder Python[16], die bekanntesten Bibliotheken sind jedoch die für das Webumfeld (Yjs 2025; Automerge 2025).

[15] Etwa https://github.com/rust-crdt/rust-crdt

[16] Etwa https://pypi.org/project/python3-crdt/

Neben den technischen Argumenten spielt auch ein praktischer Aspekt bei der Wahl der Plattform eine Rolle: die vorhandene persönliche Erfahrung im Umgang mit Webtechnologien. Bereits bestehende Kenntnisse in JavaScript, TypeScript sowie im Umgang mit Browser-APIs reduzieren den Entwicklungsaufwand erheblich.

6.1.2 Auswahl der CRDT-Bibliothek

Auf Basis dieser grundlegenden Entscheidung für die Webplattform kann nun die Auswahl der CRDT-Bibliothek getroffen werden. Sie wird die Grundlage der Implementierung des in dieser Arbeit entworfenen CRDTs sein. Das Ziel ist demnach die Erweiterung einer existierenden CRDT-Bibliothek um die neue Datenstruktur.

In diesem Kapitel wird das Vorgehen zur Auswahl der passenden CRDT-Bibliothek beschrieben. Zunächst werden die Bewertungskriterien definiert, anschließend werden relevante Kandidaten analysiert und verglichen.

6.1.2.1 Bewertungskriterien

Die wichtigste Anforderung ist die Fähigkeit der CRDT-Bibliothek, die Definition und Integration benutzerdefinierter CRDTs zu ermöglichen. Eine hohe Erweiterbarkeit reduziert den Entwicklungsaufwand (NFA 8) stark. Folgende Fragen sind hierbei leitend:

- Stellt die CRDT-Bibliothek eine explizite und stabile Programmierschnittstelle zur Verfügung, um benutzerdefinierte CRDTs zu implementieren?
- Ist der Prozess zur Erstellung benutzerdefinierter Typen klar dokumentiert und gibt es Code-Beispiele oder Vorlagen?
- Ermöglicht die Architektur die Komposition bestehender CRDTs zu neuen, komplexeren Strukturen?

Eine CRDT-Bibliothek, die dieses Kriterium der Erweiterbarkeit nicht oder nur unzureichend erfüllt, ist für den Zweck dieser Arbeit ungeeignet.

Neben der Erweiterbarkeit werden weitere Kriterien zur Bewertung herangezogen. Orientierungshilfe bieten dabei die in Abschnitt 3 definierten nichtfunktionalen Anforderungen:

- Stabilität und Korrektheit (NFA 4): Die CRDT-Bibliothek muss eine CRDT-Implementierung bereitstellen, die tatsächlich Konsistenz bei nebenläufigen Änderungen herstellt. Ein wissenschaftlicher Hintergrund oder eine breite Nutzung in Produktivsystemen sind Zeichen dafür, dass das der Fall ist.
- Performance und Latenz (NFA 1, NFA 2, NFA 3): Die CRDT-Bibliothek sollte eine Architektur unterstützen, die geringe Latenzen sowohl bei lokalen Eingaben als auch bei der Synchronisation von Änderungen anderer Teilnehmender ermöglicht. Da aus Zeitgründen keine eigenen Benchmarks durchgeführt werden, wird die Performance anhand von Benchmarks Dritter und der grundlegenden Architektur bewertet.
- Entwicklererfahrung und Integration (NFA 5, NFA 6, NFA 7): Die CRDT-Bibliothek sollte eine modulare Architektur aufweisen, die eine wartbare und für Dritte verständliche Implementierung ermöglicht. Sie muss sich ohne großen Aufwand in bestehende Systeme integrieren lassen und eine unkomplizierte Installation sowie Konfiguration gewährleisten.

6.1.2.2 Analyse der Kandidaten

Auf Basis einer Marktrecherche wurden die folgenden vier prominenten CRDT-Implementierungen als Kandidaten identifiziert: Yjs, Automerge, Collabs und json-joy. Bibliotheken wie kappa-db, gun oder Logux, die eher als komplette Datenbank- oder Synchronisationslösungen zu verstehen sind und dabei teilweise auf CRDTs setzen, sind aufgrund ihrer Komplexität nicht Teil des Vergleichs.

- Yjs: Yjs gilt als eine der performantesten CRDT-Bibliotheken, insbesondere im Bereich kollaborativer Textbearbeitung (Gentle 2021). Die Architektur von Yjs ist auf die Nutzung und Kombination eines festen Satzes optimierter „Shared Types“ ausgelegt. Die Erstellung fundamental neuer CRDT-Typen ist zwar nicht unmöglich, aber sie ist nicht das primäre Designziel und wird in der Dokumentation nicht prominent behandelt. Yjs ist eine recht weit verbreitete CRDT-Bibliothek. (Yjs 2025)
- Automerge: Diese Bibliothek bietet ein JSON-ähnliches Datenmodell, das aufgrund seiner Vertrautheit die Einarbeitung (NFA 5) erleichtern kann. JSON-Datentypen sind allerdings – wie in Abschnitt 5.2 beschrieben – nicht für die Implementierung des vorgestellten CRDTs geeignet. Wie Yjs ist Automerge ebenfalls weit verbreitet und gut dokumentiert. Ein größerer Nachteil von Automerge ist jedoch eine schlechte Performance und hoher Speicherverbrauch, insbesondere bei großen Dokumenten oder einer langen Änderungshistorie (Gentle 2021). Dies steht im direkten Widerspruch zu den Anforderungen an eine geringe Latenz (NFA 1, NFA 3). Ähnlich wie bei Yjs ist auch hier die Architektur auf die Nutzung des bereitgestellten Datenmodells fokussiert, nicht auf dessen Erweiterung um neue CRDT-Typen. (Automerge 2025)
- json-joy: Diese Bibliothek implementiert wie Automerge ein JSON-ähnliches Datenmodell, ist allerdings weniger weit verbreitet. Eine Erweiterung ist möglich, aber nicht sehr ausführlich dokumentiert (Streamich 2025). Aufgrund des JSON-Datentyps fällt auch diese Bibliothek aus der engeren Auswahl, obwohl die Performance sich in Benchmarks als sehr gut herausgestellt hat (Daleckis 2023).
- Collabs: Diese Bibliothek positioniert sich als ein Framework, dessen zentrales Designziel die Flexibilität und Erweiterbarkeit ist. Sie soll es ermöglichen, eigene CRDTs zu implementieren oder bestehende Bausteine neu zu kombinieren. Die Entwickler stellen eine API und Code-Vorlagen bereit. Die wissenschaftliche Fundierung durch ein begleitendes Paper schafft zudem Vertrauen in die Korrektheit und Stabilität des Ansatzes (NFA 4). Die Performance ist in Benchmarks etwa auf dem Niveau von Yjs (Weidner u. a. 2023).

6.1.2.3 Vergleich der CRDT-Bibliotheken anhand der abgeleiteten Kriterien

Die Gegenüberstellung in Tabelle 10 macht die Entscheidung deutlich. Während Yjs durch gute Performance und ein etabliertes Ökosystem besticht, scheitert es am zentralen Kriterium der Erweiterbarkeit. Automerge und json-joy sind aufgrund ihrer Fokussierung auf JSON-Datentypen ungeeignet.

Tabelle 10: Die Gegenüberstellung der vier ausgewählten CRDT-Bibliotheken anhand der ausgewählten Kriterien.

Kriterium	Yjs	Automerge	json-joy	Collabs
Erweiterbarkeit	-	-	+	++
Stabilität und Korrektheit	++	++	+	+
Performance und Latenz	++	--	++	++
Entwicklererfahrung und Integration	++	o	o	+

Collabs erfüllt das primäre Kriterium der Erweiterbarkeit, das aus der Anforderung NFA 8 abgeleitet wurde, am besten. Gleichzeitig bietet die Bibliothek eine solide Grundlage in den sekundären Kriterien. Die Performance wird als gut bewertet, und die wissenschaftliche Fundierung erlaubt Vertrauen in die Korrektheit der Algorithmen (NFA 4). Dass bereits Werkzeuge für die Erweiterung bereitgestellt werden, senkt die Hürden für die Implementierung.

Mit diesem Ergebnis wird Collabs als Basis für die prototypische Umsetzung einer Softwarebibliothek im Rahmen dieser Bachelorarbeit ausgewählt. Sie bietet den besten Kompromiss für die spezifischen Anforderungen dieses Projekts.

6.1.3 Collabs

Bevor wir uns der Implementierung zuwenden, soll hier zunächst nochmal ein genaueres Bild der von Weidner u. a. (2023) vorgestellten und für diese Arbeit gewählten CRDT-Bibliothek Collabs vermittelt werden.

Collabs stellt eine Art Baukastensystem zur Verfügung, mit dem eigene CRDTs von Grund auf neu oder durch die Komposition bestehender Bausteine implementiert werden können. Jede CRDT-Implementierung in Collabs ist dabei eine Subklasse der zentralen `Collab`-Klasse. Diese Klasse definiert eine Schnittstelle, die eine CRDT-Instanz kapselt: Sie empfängt und sendet Operationen und kann ihren Zustand speichern und laden. Soll ein eigenes CRDT implementiert werden, müssen im Wesentlichen die Methoden dieser Schnittstelle implementiert werden. Dabei gibt es zwei Arten, wie Daten geladen werden:

- Operationsbasierte Interaktion: Methoden zum Senden (`send`) und Empfangen (`receive`) von einzelnen Operationen ermöglichen die Live-Kollaboration. Auf Basis der eintreffenden Operationen kann dann beispielsweise ein lokales Modell aufgebaut werden wie in Abschnitt 5.3.6 beschrieben.
- Zustandsbasierte Interaktion: Methoden zum Speichern (`save`) und Laden (`load`) eines Zustands ermöglichen die Persistenz und die Synchronisation durch das Zusammenführen ganzer Zustände. So können beispielsweise auch neue Teilnehmende schnell auf den aktuellen Stand gebracht werden. Ein Zustand ist im Prinzip die Sammlung aller bisher erzeugten Operationen des Systems.

In der Praxis besteht eine kollaborative Anwendung jedoch selten aus nur einem einzigen CRDT. Collabs organisiert daher alle `Collab`-Instanzen eines Dokuments zur Laufzeit in einer hierarchischen Baumstruktur. Diese Struktur spiegelt typischerweise das Datenmodell der Anwendung wider. Beispielsweise könnte ein Textdokument an der Wurzel stehen und als Kinder `Collab`-Instanzen für den Textinhalt, Metadaten oder, wie in dieser Arbeit relevant, eine Liste von Vorschlägen enthalten.

Diese Baumstruktur regelt die Kommunikation: Sendet eine Kind-Komponente eine Operation, wird diese Nachricht entlang des Baumpfades nach oben gereicht, wobei jeder Elternknoten die Nachricht mit einer Kennung des Kindes versieht. Empfängt ein Elternknoten eine Nachricht, leitet er sie anhand dieser Kennung an das zuständige Kind weiter. Dieses Multiplexing ermöglicht es, komplexe Datenmodelle aus vielen einzelnen CRDT-Instanzen aufzubauen, deren Korrektheitsgarantien sich auf die Gesamtstruktur vererben.

Ein Elternknoten im Baum hat dabei die vollständige Kontrolle über seine Kinder. Er ist nicht nur für deren Erzeugung und Zerstörung verantwortlich, sondern kann auch in deren Nachrichtenfluss eingreifen, um die Konsistenz des Gesamtsystems zu gewährleisten. So kann ein Listen-CRDT beispielsweise Nachrichten, die für ein bereits gelöschtes Element bestimmt sind, verwerfen oder umleiten.

6.2 Architektur

Nach diesen grundlegenden Entscheidungen kann mit der eigentlichen Implementierung des Prototyps begonnen werden. Das bereits beschriebene „Baukastensystem“ von Collabs erlaubt es, eine modulare Architektur zu wählen. Diese besteht aus zwei zentralen Bibliotheken: einer Kernbibliothek, die die grundlegende Logik für Text und Annotationen implementiert, und einer darauf aufbauenden Erweiterungsbibliothek, die eine vollständige Anwendungsstruktur mit Dokumenten- und Präsenzverwaltung bereitstellt.

- Die Kernbibliothek (`track-changes-crdt`) ist der zentrale Teil der Implementierung. Sie enthält die `TrackChanges`-Klasse, die den in Abschnitt 5.3.6 beschriebenen Mechanismus zur Verwaltung von Text und Annotationen umsetzt. Dies schließt die Verarbeitung von `addition`, `removal`- und `update`-Operationen sowie die Auflösung von Konflikten ein.
- Die Erweiterungsbibliothek (`track-changes-application`) demonstriert die Nutzung der Kernbibliothek in einem größeren Kontext. Sie implementiert eine `TrackChangesApplication`-Klasse, die mehrere Dokumente verwalten kann und zusätzlich Funktionen wie die Präsenzanzeige von Nutzenden integriert.

Wie sich die Architektur der beiden Bibliotheken zusammensetzt, ist auch in Abbildung 9 dargestellt.

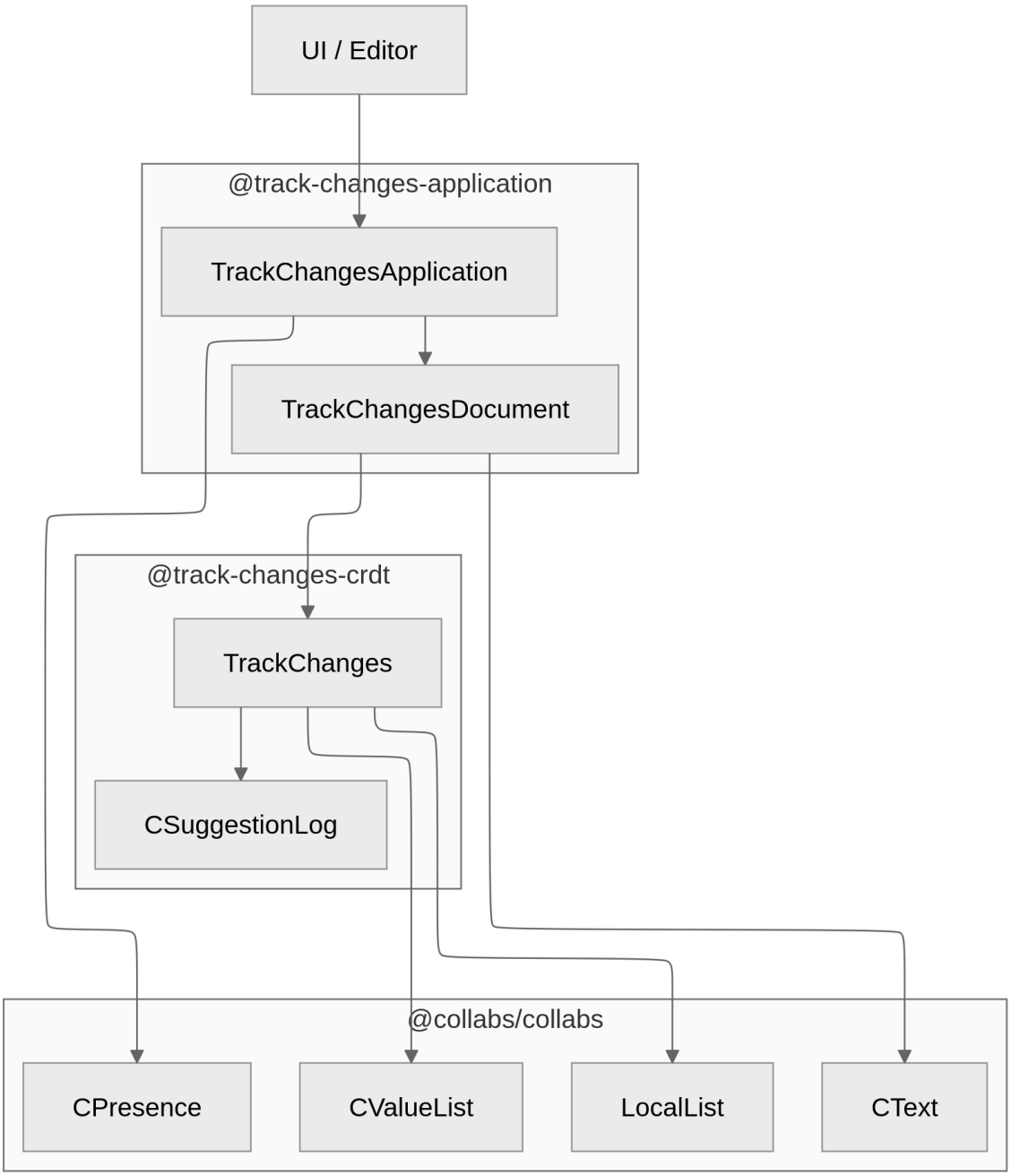


Abbildung 9: Die Architektur der Track-Changes-Bibliotheken

6.2.1 Kernbibliothek: Verwaltung von Text und Annotationen

Die Kernbibliothek setzt die in Abschnitt 5 vorgestellten Konzepte in TypeScript um. Die zentrale Klasse ist `TrackChanges`, die von `CObject` aus dem Collabs-Framework erbt und mehrere untergeordnete CRDTs nutzt.

6.2.1.1 Interner Zustand

Der Zustand eines Dokuments in der `TrackChanges`-Klasse wird durch drei wesentliche Collabs-Instanzen verwaltet (siehe Codeblock 1), die eine direkte Entsprechung im theoretischen Modell haben:

- text: `CustomCValueList<string>`: Diese Instanz repräsentiert den Nur-Text-CRDT. Sie speichert die Zeichen des Dokuments in einer geordneten Liste und stellt sicher, dass Einfüge-

und Löschoperationen auch bei nebenläufiger Ausführung zu einem konsistenten Ergebnis führen. Die Implementierung der CValueList basiert auf dem Fugue Algorithmus (Weidner und Kleppmann 2023) und wird vom Collabs-Projekt bereitgestellt. Diese Basis wurde für diese Arbeit angepasst, um zusätzlich zur `delete` Methode, die einen Startindex und die Zahl der zu löschenden Buchstaben entgegen nimmt, auch eine `deleteRange` Methode zu implementieren, die alle Buchstaben von einer Startposition bis zu einer Endposition entfernt.

- annotationLog: `CAnnotationLog`: Dies ist eine benutzerdefinierte CRDT-Implementierung, die als sogenanntes Append-only-Log für alle Annotationsoperationen dient (`insert_suggestion`, `add_comment` etc.). D. h. jede eintreffende oder lokale Operation, die Annotationen betrifft, wird hier verarbeitet, im Fall von lokalen Operationen an alle anderen teilnehmenden Instanzen gesendet und im Hauptspeicher vorgehalten. Diese Klasse ist zudem für die Serialisierung und Deserialisierung des Append-only-Logs aller verarbeiteten Operationen zuständig, um sie persistieren zu können. Das Log ist dabei als eine `Map` angelegt, die von einer `addition`-Operations-Kennung auf alle Operationen verweist, die sich auf diese `addition` beziehen. So kann effizient auf die Historie einzelner `addition`-Operationen zugegriffen werden, was für das Verfahren zur Verarbeitung von Operationen relevant ist (siehe Abschnitt 5.3.7).
- annotationList: `LocalList<AnnotationDataPoint>`: Diese Datenstruktur ist die praktische Umsetzung des in Abschnitt 5.3.6 beschriebenen Konzepts der Datenpunkte. Es handelt sich um eine nicht-replizierte, lokale Sicht auf die Annotationen. `LocalList` mappt die Positionen aus dem text-CRDT auf `AnnotationDataPoint`-Objekte, die alle an dieser Stelle gültigen Annotationen enthalten, wie in Abbildung 8 dargestellt.

```ts
export class TrackChanges
  extends CObject<TrackChangesEventsRecord>
  implements ICursorList {
  // Der Nur-Text-CRDT für die Zeichen des Dokuments
  private readonly text: CustomCValueList<string>;
  // Das replizierte, Append-only-Log für Annotations-Operationen
  private readonly annotationLog: CAnnotationLog;
  // Die lokale, nicht-replizierte Sicht mit den Datenpunkten
  private readonly annotationList: LocalList<AnnotationDataPoint>;

  constructor(init: InitToken, options: { userId: string }) {
    super(init);
    // ... Initialisierung der Collabs und Event-Listener
  }
}
```

Codeblock 1: Der Beginn der Track-Changes-Klasse.

6.2.1.2 Die Annotation

Der zentrale Baustein des implementierten Vorschlagsmodus ist der Datentyp Annotation (siehe Codeblock 2). Er beschreibt die Struktur von Operationen, im Fall von `addition`-Annotationen wird dieser Datentyp aber auch für die Repräsentation der Annotationen in der lokalen Annotationsstruktur und in ausgehenden Events verwendet. Aufgrund dieser komplexen Doppelrolle ist die Annotation als ein sogenannter Union-Typ realisiert.

Jede Annotation, unabhängig von ihrer spezifischen Aufgabe, hat eine Reihe von grundlegenden Eigenschaften, die in der Basisschnittstelle `AnnotationBase` zusammengefasst sind. Dazu gehören die für den CRDT essenziellen Metadaten: ein lokaler Zähler (`lamport`) sowie eine Instanzkennung (`senderID`), aus denen sich, wie in Abschnitt 4.4 beschrieben, eine eindeutige Operationskennung zur Herstellung einer totalen Ordnung ableiten lässt. Zusätzlich wird eine globale Kennung in Form einer UUID (`id`) verwendet. Dies erwies sich als notwendig, da es in der zugrundeliegenden Bibliothek Collabs vereinzelt zu doppelten Lamport-Zeitstempeln kam, wenn Operationen in extrem kurzen Zeitabständen erzeugt wurden.

Weiterhin spezifiziert `type`, ob es sich um einen Kommentar (`COMMENT`) oder einen Vorschlag (`SUGGESTION`) handelt. Die userId zeigt an, von welchem Nutzer die Annotation erstellt wurde. Das ist unabhängig von der senderID, da es so möglich wird, dass ein Nutzer etwa auf mehreren Geräten eingeloggt ist und das gleiche Dokument bearbeitet.

Auf dieser Basis werden mit drei spezialisierten Schnittstellen die konkreten Operationen definiert:

- `AdditionAnnotation`: Diese Annotation repräsentiert die Erzeugung einer neuen Markierung im Text. Das Feld `action` hat hier immer den Wert `ADDITION`. Über das `description`-Feld wird die Art der Hinzufügung genauer beschrieben, etwa als `ADD_COMMENT` oder `INSERT_SUGGESTION`. Die Funktionsweise der Felder `startPosition`, `endPosition`, `openStart` und `openEnd` wurde bereits in Abschnitt 5 erläutert. Sie definieren den exakten Textbereich der Annotation. Für Kommentare enthält das optionale `value`-Feld den eigentlichen Text des Kommentars.
- `UpdateAnnotation`: Mit dieser Operation werden die Eigenschaften einer bereits existierenden `AdditionAnnotation` modifiziert. Hier ist der `action`-Typ immer `UPDATE`. Wichtig ist dabei das Feld `dependentOn`, das die `id` der zu aktualisierenden Annotation enthält. Die zu ändernden Eigenschaften selbst werden in `updatedProperties` in einem Objekt übergeben, das die meisten Eigenschaften einer `AdditionAnnotation` enthalten kann. Dabei werden immer nur die tatsächlichen Änderungen übertragen, anstatt die gesamte Annotation neu zu senden.
- `RemovalAnnotation`: Diese Annotation dient dazu, eine bestehende `AdditionAnnotation` zu entfernen oder aufzulösen (z.B. durch Annehmen oder Ablehnen eines Vorschlags). Die `action` ist hier `REMOVAL`. Wie bei der `UpdateAnnotation` verweist das Feld `dependentOn` auf die ursprüngliche Annotation, die als entfernt oder erledigt markiert werden soll. Das `description`-Feld präzisiert die Art der Entfernung, beispielsweise als `ACCEPT_SUGGESTION`.

```ts
// Gemeinsame Basis-Eigenschaften aller Annotationen.
interface AnnotationBase {
  readonly id: string;
  readonly type: AnnotationType;
  readonly lamport: number;
  readonly senderID: string;
  readonly userId: string;
}

// Repräsentiert das Hinzufügen einer neuen Markierung.
export interface AdditionAnnotation extends AnnotationBase {
  readonly action: AnnotationAction.ADDITION;
  // Felder zur Definition des Textbereichs (startPosition, endPosition, etc.)
}

// Repräsentiert das Entfernen oder Auflösen einer Markierung.
export interface RemovalAnnotation extends AnnotationBase {
  readonly action: AnnotationAction.REMOVAL;
  readonly dependentOn: string;
}

// Repräsentiert die Aktualisierung einer bestehenden Markierung.
export interface UpdateAnnotation extends AnnotationBase {
  readonly action: AnnotationAction.UPDATE;
  readonly dependentOn: string;
  // Objekt, das nur die geänderten Eigenschaften der Original-Annotation enthält.
  readonly updatedProperties: Partial<AdditionAnnotation>;
}

// Zusammengefasster Union-Typ für allgemeine Annotations-Aktionen.
export type Annotation =
  | AdditionAnnotation
  | RemovalAnnotation
  | UpdateAnnotation;
```

Codeblock 2: Verkürztes Datenmodell der Annotationen. Der `Annotation`-Typ fasst die drei spezialisierten Operationen zusammen, die sich eine `AnnotationBase` teilen und durch das `action`-Feld unterschieden werden.

6.2.1.3 Verarbeitung von Annotationen

Die theoretischen Überlegungen aus Abschnitt 5.3.7 werden in der Methode `processAnnotation` in die Praxis umgesetzt. Das Prinzip der Verarbeitung ist in diesem Abschnitt schon beschrieben, im Folgenden sind daher nur einige Implementierungsdetails konkretisiert.

6.2.1.3.1 Das Annotations-Log als Map

Die Grundlage für die Verarbeitung ist der schnelle Zugriff auf die komplette Historie einer Annotation. Wie beschrieben, wird hierfür das `annotationLog` genutzt – eine Map, die von der Kennung einer `addition`-Operation auf eine Liste aller zugehörigen Operationen verweist. Diese Liste wird dabei von unserem Annotation-CRDT stets nach Lamport-Timestamp sortiert. Das alles ist für die Performanz wichtig: Statt bei jeder eingehenden Operation das gesamte Log aller Annotationen durchsuchen zu müssen, kann so mit einem einzigen Map-Zugriff die gesamte, bereits sortierte Historie einer spezifischen Annotation abgerufen werden.

6.2.1.3.2 Atomare Operationen zur Aktualisierung

Wenn dann nach einer eintreffenden `update`-Operation eine Aktualisierung einer bestehenden Annotation in der lokalen Datenpunkt-Struktur notwendig wird, wird das als atomare Operation aus Entfernung und anschließender Hinzufügung umgesetzt. Anstatt zu versuchen, eine bereits existierende Annotation in der `annotationList` zu verändern, wird der Zustand der Annotation vor dem Update mithilfe ihrer Historie rekonstruiert und entfernt. Anschließend wird der neue Zustand inklusive des Updates berechnet und als komplett neue Annotation hinzugefügt. Dieser Ansatz vermeidet fehleranfällige In-Place-Modifikationen der Datenpunkt-Struktur und ermöglicht auch eine konsistente Generierung von Events für die Anwendungsschicht, die so immer nur aus den atomaren Operationen der Entfernung und Hinzufügung bestehen.

6.2.1.3.3 Rekonstruktion des aktuellen Zustands

Vor jeder Modifikation der `annotationList` wird der relevante Zustand einer Annotation durch die Hilfsmethode `applyUpdateOperations` aus ihrer Historie im `annotationLog` rekonstruiert.

- Bei einer `addition`-Operation wird deren finaler Zustand durch die Anwendung aller bereits bekannten `update`-Operationen ermittelt, bevor sie der `annotationList` hinzugefügt wird.
- Bei einer `removal`-Operation wird der letzte gültige Zustand der Annotation rekonstruiert, um die korrekten Start- und Endpunkte für die Entfernung aus den Datenpunkten der `annotationList` zu kennen.

6.2.1.4 Umsetzung der Nutzeraktionen

Die öffentlichen Methoden der `TrackChanges`-Klasse kapseln die Erzeugung der entsprechenden Operationen.

6.2.1.4.1 Einfügen

Das Einfügen von Text erfolgt über die Methode `insert`, die drei Parameter entgegennimmt: den Startindex der Einfügestelle, die einzufügenden Zeichen sowie einen Boolean-Wert, der

angibt, ob es sich um einen Vorschlag handelt (`isSuggestion`). Findet die Einfügung im Vorschlagsmodus statt und liegt an der betreffenden Position noch kein Einfüge-Vorschlag desselben Nutzers vor, wird eine `insert_suggestion`-Operation erzeugt.

Gemäß dem theoretischen Konzept wird dabei als `endPosition` das erste Zeichen nach der Einfügung und als `startPosition` das Zeichen vor der Einfügung gesetzt. Sowohl `openEnd` als auch `openStart` werden auf `true` gesetzt. Dadurch entsteht eine „wachsende“ Range, wie in Abbildung 5 dargestellt.

Eine Besonderheit ergibt sich in Bezug auf TA 3: Erfolgt die Einfügung nicht im Vorschlagsmodus, prüft das System, ob sie am Rand einer bestehenden wachsenden Annotation liegt. In diesem Fall wird eine `update`-Operation erzeugt, die die Start- oder Endposition der bestehenden Annotation – je nachdem, ob am Anfang oder Ende der Range eingefügt wird – auf den ersten eingefügten Buchstaben setzt. Dadurch wird verhindert, dass die neue Einfügung Teil der bestehenden Range wird.

6.2.1.4.2 Löschen

Die Methode `delete` implementiert die Logik für das Löschen von Zeichen im Text. Ihr werden drei Parameter übergeben: der Startindex des zu löschenden Bereichs, die Anzahl der zu entfernenden Zeichen sowie ein Flag, das angibt, ob es sich um einen Vorschlag handelt (`isSuggestion`).

Handelt es sich nicht um einen Vorschlag, werden die Zeichen unmittelbar aus dem Text-CRDT entfernt. Ist hingegen ein Lösch-Vorschlag gemeint, wird zunächst geprüft, ob der gesamte zu löschende Bereich innerhalb eines bestehenden Einfüge-Vorschlags desselben Nutzers liegt. Falls dies zutrifft, erfolgt ebenfalls eine direkte Löschung.

In allen anderen Fällen wird eine `delete_suggestion`-Operation erzeugt. Dabei kommt die Methode `createDeleteSuggestion` zum Einsatz, die das in der Theorie beschriebene Zusammenführen benachbarter Lösch-Vorschläge umsetzt: Es wird überprüft, ob unmittelbar angrenzend an den neuen Löschbereich bereits ein Lösch-Vorschlag desselben Nutzers existiert. Ist das der Fall, wird eine `update`-Operation generiert, um den bestehenden Lösch-Vorschlag zu vergrößern.

6.2.1.4.3 Kommentieren

Kommentare können mithilfe der Methode `addComment` hinzugefügt werden. Dazu werden der Startindex, der Endindex und der Text des Kommentars übergeben. Die Methode überprüft die Gültigkeit des Start- und Endindex und erzeugt, wenn sie gültig sind, eine `addComment`-Operation.

Die Methode `removeComment` erstellt die entsprechend gegensätzliche `removeComment`-Operation. Dieser Methode muss lediglich die Kennung der Kommentar-Annotation übergeben werden, die entfernt werden soll.

6.2.1.4.4 Entfernen von Annotationen

Die Methoden `acceptSuggestion`, `declineSuggestion` und `removeComment` erzeugen jeweils die dazugehörigen `removal`-Operationen. Dafür muss ihnen die eindeutige Kennung der zu entfernenden Annotation übergeben werden.

Für die Erstellung der `removal`-Operationen ist es essenziell, dass die Kennung des ursprünglichen Vorschlags im Feld `dependentOn` referenziert wird. Diese explizite Verknüpfung – wie im theoretischen Teil gefordert – stellt sicher, dass die Beziehung zwischen Operation und Ursprungsvorschlag eindeutig bleibt. Dadurch wird die korrekte Verarbeitung der Operationen sichergestellt, selbst wenn diese in einer nicht-chronologischen Reihenfolge eintreffen.

Die Funktionen sind außerdem dafür verantwortlich, etwaigen Text zu entfernen. Wird ein Einfüge-Vorschlag abgelehnt, oder ein Lösch-Vorschlag angenommen, muss der in dieser Range liegende Text entfernt werden. Damit hier immer der gesamte Text innerhalb der Range entfernt wird, selbst wenn gleichzeitig ein anderer Nutzer noch Text dort einfügt, findet hier die von uns implementierte `deleteRange` Methode der `CustomCValueList` Anwendung.

6.2.2 Erweiterungsbibliothek: Dokumenten- und Präsenzverwaltung

Während die Kernbibliothek die Datenlogik eines einzelnen Dokuments kapselt, demonstriert die Erweiterungsbibliothek deren Einbettung in eine vollständige Anwendung. Die `TrackChangesApplication` fungiert hier als Wurzel des Collabs-Baums.

Sie verwendet zwei zentrale Collabs-Typen:

- documents: `CMap<DocumentID, TrackChangesDocument>`: Eine `CMap` ist eine replizierte Map. Hier wird sie genutzt, um eine Sammlung von Dokumenten zu verwalten, wobei jede `DocumentID` auf eine Instanz von `TrackChangesDocument` verweist. Dies ermöglicht das Erstellen und Löschen von Dokumenten während einer kollaborativen Sitzung. Neue `TrackChangesDocument` werden dabei von der `CMap` selbst instanziiert, um die Korrektheit der Dokumente zu gewährleisten (siehe Codeblock 3-13)
- presence: `CPresence<PresenceState>`: Diese Instanz dient der Verwaltung von Präsenzinformationen. Jede Instanz kann hier ihren Status (z. B. `viewing: true`) und ihre aktuelle Cursor-Position inklusive einer etwaigen Selektion teilen. Diese Informationen werden nicht dauerhaft im Dokumentenzustand gespeichert, sondern dienen der Anzeige von „Live-Cursorn“ und Anwesenheitslisten.

Die `TrackChangesDocument`-Klasse selbst ist wiederum eine eigene Klasse, die die Kernbibliothek für den Inhalt mit einem `CText` für den Dateinamen kombiniert.

```ts
export class TrackChangesApplication extends CObject {
  readonly presence: CPresence<PresenceState>;
  readonly documents: CMap<DocumentID, TrackChangesDocument, []>;

  constructor(init: InitToken, userId: string) {
    super(init);

    this.presence = super.registerCollab(/* ... */);
    this.documents = super.registerCollab(
      "documents",
      // Eine Callback-Funktion, um ein neues TrackChangesDokument zu
      instanziieren.
```

```
12        (init) => new CMap(init, (init, key) => {
13          return new TrackChangesDocument(init, key, this.userId);
14        })
15      );
16    }
17  }
```

Codeblock 3: Der Beginn der `TrackChangesApplication`-Klasse. Im Constructor wird deutlich, dass die `CMap` für die Instanziierung neuer `TrackChangesDocument` verantwortlich ist.

6.3 Schnittstellen zur Anwendungsentwicklung

Teil der nichtfunktionalen Anforderungen ist die einfache Integration in ein UI (NFA 7). Um dies zu gewährleisten, stellt der Prototyp eine öffentliche Programmierschnittstelle (API) zur Verfügung.

Im Folgenden werden die öffentlichen Schnittstellen der beiden im vorherigen Kapitel vorgestellten Bibliotheken – der Kernbibliothek `track-changes-crdt` und der Erweiterungsbibliothek `track-changes-application` – detailliert beschrieben. Die Darstellung orientiert sich an der Perspektive von Entwickelnden, die diese Bibliotheken zur Erstellung einer kollaborativen Anwendung nutzen möchten.

6.3.1 Die TrackChanges-Klasse

Diese Klasse wird in der Kernbibliothek `track-changes-crdt` bereitgestellt.

Sie repräsentiert ein einzelnes kollaboratives Dokument mit Vorschlags- und Kommentarfunktionalität und ist die zentrale Klasse für die Interaktion mit dem Dokumenteninhalt.

6.3.1.1 Eigenschaften

`length: number`
Gibt die Anzahl der aktuell sichtbaren Zeichen im Dokument zurück. Gelöschte Zeichen werden nicht mitgezählt.

`totalOrder: CTotalOrder`
Ermöglicht den Zugriff auf die zugrundeliegende totale Ordnung des Text-CRDTs. Diese Eigenschaft ist für fortgeschrittene Anwendungsfälle gedacht, etwa um eigene lokale Datenstrukturen auf Basis der Zeichenreihenfolge zu erstellen.

6.3.1.2 Methoden zur Textmanipulation

`insert(index: number, values: string, isSuggestion: boolean): void`
Fügt den Zeichen-String `values` am `index` ein. Der Parameter `isSuggestion` steuert das Verhalten:

- `false`: Der Text wird direkt in das Dokument eingefügt.
- `true`: Der Text wird als Einfüge-Vorschlag (`insert_suggestion`) markiert. Besteht an der Einfügestelle bereits ein Einfüge-Vorschlag des gleichen Nutzers, wird der Text an diesen angehängt.

`delete(index: number, count: number, isSuggestion: boolean): void`
Entfernt `count` Zeichen ab dem `index`. Auch hier steuert `isSuggestion` das Verhalten:

- `false`: Die Zeichen werden direkt entfernt.

- `true`: Die Zeichen werden nicht entfernt, sondern es wird ein Lösch-Vorschlag (`delete_suggestion`) über den Bereich gelegt. Liegt der zu löschende Bereich vollständig innerhalb eines Einfüge-Vorschlags desselben Nutzers, werden die Zeichen stattdessen direkt gelöscht.

6.3.1.3 Methoden zur Verwaltung von Annotationen

`acceptSuggestion(id: SuggestionId): void`
Akzeptiert einen Vorschlag. Der Vorschlag wird durch seine `id` identifiziert. Handelt es sich um einen Einfüge-Vorschlag, wird die Markierung entfernt und der Text wird zu normalem Inhalt. Handelt es sich um einen Lösch-Vorschlag, werden die markierten Zeichen tatsächlich gelöscht.

`declineSuggestion(id: SuggestionId): void`
Lehnt einen Vorschlag ab. Handelt es sich um einen Einfüge-Vorschlag, werden die vorgeschlagenen Zeichen gelöscht. Handelt es sich um einen Lösch-Vorschlag, wird lediglich die Markierung entfernt.

`addComment(startIndex: number, endIndex: number, comment: string): void`
Fügt einen Kommentar mit dem Text `comment` über den Zeichenbereich von `startIndex` (inklusiv) bis `endIndex` (inklusiv) hinzu.

`removeComment(id: SuggestionId): void`
Entfernt einen Kommentar, der durch seine `id` identifiziert wird.

6.3.1.4 Methoden zum Abfragen des Zustands

`getActiveSuggestions(): Suggestion[]`
Gibt ein Array aller aktuell aktiven Vorschläge und Kommentare zurück.

Die Klasse implementiert außerdem das `ICursorList`-Interface von Collabs und bietet eine Reihe von Methoden, die eine string- oder array-ähnliche Interaktion mit dem Text ermöglichen. Dazu gehören `slice()`, `getPosition()`, `indexOfPosition()`, `hasPosition()`, `getByPosition()`, `positions()`, `toString()`, `charAt()` und `clear()`.

6.3.1.5 Events

Die `TrackChanges`-Klasse emittiert Events, um die anbindende Anwendung über Zustandsänderungen zu informieren.

`on("Insert", (event: TrackChangesTextInsertEvent) => void)`
Wird ausgelöst, wenn Text eingefügt wird. Das `event`-Objekt enthält den `index`, die eingefügten Zeichen (`values`) und `annotations`, eine Liste aller auf den neuen Text zutreffenden Annotationen.

`on("Delete", (event: TextEvent) => void)`
Wird ausgelöst, wenn Text permanent gelöscht wird. Das `event`-Objekt enthält den `index` und die entfernten Zeichen (`values`).

`on("AnnotationAdded", (event: TrackChangesAnnotationAddedEvent) => void)`
Wird ausgelöst, wenn ein neuer Vorschlag oder Kommentar dem Dokument hinzugefügt wird. Das Event enthält den `startIndex`, `endIndex` und die `annotation`-Daten.

`on("AnnotationRemoved", (event: TrackChangesAnnotationRemovedEvent) => void)`
Wird ausgelöst, wenn ein Vorschlag oder Kommentar entfernt wird. Das Event enthält neben den Bereichsinformationen auch den Grund für die Entfernung (`reason`, z. B. `ACCEPTED` oder `DECLINED`) und die Daten der entfernten Annotation (`annotation`).

6.3.2 Die TrackChangesApplication-Klasse

Diese Klasse wird in der Erweiterungsbibliothek `track-changes-application` bereitgestellt. Dies ist die Wurzelklasse der gesamten Anwendung. Eine Anwendung instanziiert typischerweise genau eine `TrackChangesApplication`.

6.3.2.1 Eigenschaften

`userId: string`
Die eindeutige Kennung des aktuellen Nutzers.

`presence: CPresence<PresenceState>`
Ermöglicht die Verwaltung von Präsenzinformationen. Über diese Eigenschaft kann der eigene Status (aktuelle Cursor-Position, Anwesenheit) gesetzt und die Statusinformationen anderer Nutzender abonniert werden (z. B. via `presence.on("Change", ...)`).

`documents: CMap<DocumentID, TrackChangesDocument, []>`
Eine kollaborative Map, die alle Dokumente der Sitzung enthält. Die anbindende Anwendung kann diese Map nutzen, um auf Dokumente zuzugreifen und auf Änderungen an der Dokumentenliste zu lauschen.

6.3.2.2 Methoden

`createDocument(fileName: string): DocumentID`
Erstellt ein neues, leeres `TrackChangesDocument` mit dem gegebenen `fileName`, fügt es der documents-Map hinzu und gibt dessen eindeutige `DocumentID` zurück.

`removeDocument(id: DocumentID): void`
Entfernt das Dokument mit der angegebenen `DocumentID` aus der `documents`-Map.

6.3.3 Die TrackChangesDocument-Klasse

Diese Klasse wird ebenfalls in der Erweiterungsbibliothek `track-changes-application` bereitgestellt. Sie repräsentiert ein einzelnes Dokument innerhalb der `TrackChangesApplication`. Instanzen dieser Klasse werden nicht direkt erstellt, sondern über die `documents`-Map der Anwendung verwaltet.

6.3.3.1 Eigenschaften

`id: DocumentID`
Die eindeutige Kennung des Dokuments.

`fileName: CText`
Ein kollaboratives Textobjekt (`CText`), das den Dateinamen des Dokuments enthält. Änderungen an diesem Feld sind ebenfalls kollaborativ und können über Events (`fileName.on("Insert", ...)` etc.) beobachtet werden.

`content: TrackChanges`
Eine Instanz der `TrackChanges`-Klasse aus der Kernbibliothek. Über diese Eigenschaft erhält

die Anwendung Zugriff auf den gesamten Inhalt des Dokuments und alle Methoden und Events zur Manipulation von Text und Annotationen, wie sie im vorherigen Abschnitt beschrieben wurden.

6.3.3.2 Methoden

Diese Klasse hat keine eigenen Methoden.

6.4 Implementierungen über die Anforderungen hinaus

Die Aufgabenstellung dieser Arbeit sieht die prototypische Umsetzung einer Kernbibliothek mit Änderungsvorschlägen und Kommentaren vor. Die Entwicklung eines grafischen UI war nicht Teil der Aufgabenstellung. Um jedoch einen Beitrag zur Vervollständigung einer Gesamtlösung zu leisten, wurden sowohl ein grafisches UI als auch weitere Komponenten zumindest in einem experimentellen Umfang im Rahmen der Implementierung realisiert.

6.4.1 Prototypisches UI

Die konkrete Implementierung der erarbeiteten theoretischen Konzepte erwies sich als anspruchsvoll. Dies lag insbesondere daran, dass viele verschiedene Grenzfälle beachtet werden mussten. Fehler sind dabei oft subtil und lassen sich durch automatisierte Tests nur eingeschränkt erkennen – beispielsweise dann, wenn dieselben falschen Annahmen sowohl in der Implementierung als auch in den Tests wiederholt werden.

Das prototypische UI diente daher als Prüfwerkzeug: Es machte es möglich, die Bibliotheken unter realistischen Anwendungsbedingungen zu nutzen. Durch die direkte visuelle Rückmeldung konnten sowohl die korrekte Umsetzung der analysierten Textanforderungen überprüft als auch schwer nachvollziehbare Fehler identifiziert werden.

Das UI wurde mit Vue.js in TypeScript umgesetzt, für das Styling kam TailwindCSS zum Einsatz. Als Editor-Komponente wird CodeMirror verwendet.

6.4.2 Wiederverwendbare CodeMirror-Anbindung

Im Rahmen der Entwicklung des UI wurde der Zustand des CRDTs mit einem Nur-Text-Editor synchronisiert. Dabei wird CodeMirror genutzt. Um diese Logik wiederverwendbar zu machen und die Integration der Bibliothek in andere Anwendungen zu vereinfachen, wurde die Synchronisation in eine eigenständige Anbindung gekapselt.

Diese Anbindung umfasst die gesamte Logik, die für die bidirektionale Synchronisation notwendig ist:

- Sie lauscht auf Events der `TrackChanges`-Instanz (z. B. `Insert`, `Delete`, `AnnotationAdded`) und übersetzt diese in entsprechende Aktualisierungen der Editor-Ansicht, inklusive der visuellen Hervorhebung von Vorschlägen und Kommentaren.
- Umgekehrt fängt sie Nutzereingaben im CodeMirror-Editor ab und ruft die entsprechenden Methoden der `TrackChanges`-Klasse auf (z. B. `insert` oder `delete`).

Die Anbindung ist als eigenständiges, von der UI unabhängiges npm-Modul implementiert.

6.4.3 Automatisch generierte API-Dokumentation

Um die nichtfunktionale Anforderung einer kurzen Einarbeitungszeit für Entwickelnde (NFA 5) zu erfüllen, ist eine technische Dokumentation der öffentlichen Schnittstellen hilfreich.

Während der Entwicklung der Kern- und Erweiterungsbibliotheken wurde daher auf eine ausführliche Kommentierung aller öffentlichen Klassen, Methoden, Typen und Schnittstellen im JSDoc-Format geachtet. Diese Kommentare dienen nicht nur der Lesbarkeit des Quellcodes, sondern können auch in einem Build-Schritt mithilfe des Werkzeugs `TypeDoc` verarbeitet werden. Das Ergebnis ist eine statische HTML-Dokumentationswebsite, die eine durchsuchbare Referenz der gesamten API bereitstellt.

7 Evaluation

Im vorangegangenen Kapitel wurde die prototypische Implementierung der Softwarebibliothek für Live-Kommentare und Änderungsvorschläge vorgestellt. Das Ziel des vorliegenden Kapitels ist es nun, den entwickelten Prototyp zu evaluieren und zu überprüfen, inwieweit er die zuvor definierten Anforderungen erfüllt.

Zunächst wird die Umsetzung der funktionalen Anforderungen (FA) aus Abschnitt 3.1 bewertet. Anschließend erfolgt eine qualitative Beurteilung der nichtfunktionalen Anforderungen (NFA) aus Abschnitt 3.2. Den Abschluss bildet die Untersuchung der in Abschnitt 2.4 formulierten Textanforderungen (TA), die das korrekte Verhalten des Systems in spezifischen Grenzfällen des Vorschlagsmodus beschreiben.

7.1 Funktionale Anforderungen

In Abschnitt 3.1 wurden funktionale Anforderungen aus der Marktanalyse abgeleitet. Die folgende Tabelle gibt einen Überblick über die Erfüllung der einzelnen Anforderungen.

Tabelle 11: Übersicht und Bewertung der Erfüllung der funktionalen Anforderungen.

Kürzel	**Titel**	**Priorität**	**Status**	**Begründung der Bewertung**
FA 1	Nur-Text-Bearbeitung	Muss	Vollständig erfüllt	Die Kernbibliothek `track-changes-crdt` nutzt eine Datenstruktur `CValueList<string>` des Collabs-Frameworks, die auf die Bearbeitung von rohem Text ausgelegt ist.
FA 2	Echtzeit-Kollaboration	Muss	Vollständig erfüllt	Durch die Verwendung der CRDT-Bibliothek Collabs werden alle Operationen zwischen den Instanzen in Echtzeit synchronisiert.
FA 3	Vorschlagsmodus	Muss	Vollständig erfüllt	Die Implementierung der Kernbibliothek `track-changes-crdt` setzt das in Abschnitt 5 entworfene Konzept für Vorschläge und Kommentare vollständig um. Zur Erfüllung der Textanforderungen siehe Abschnitt 7.3.
FA 4	Cursorposition	Muss	Vollständig erfüllt	Die `CPresence`-Komponente in der Erweiterungsbibliothek `track-changes-application` ermöglicht die Übermittlung von Cursorpo-

Kürzel	Titel	Priorität	Status	Begründung der Bewertung
				sitionen und Selektionen in Echtzeit.
FA 5	Präsenzindikator	Muss	Vollständig erfüllt	Die `CPresence`-Komponente wird zur Übermittlung des Anwesenheitsstatus genutzt und erfüllt somit diese Anforderung.
FA 6	Textauswahl	Soll	Vollständig erfüllt	Die von `CPresence` übermittelten Daten umfassen auch Selektionsbereiche, die von dem anbindenden UI zur Hervorhebung genutzt werden können.
FA 7	Kommentare	Muss	Vollständig erfüllt	Die `TrackChanges`-Klasse bietet dedizierte Methoden (`addComment`, `removeComment`) und verarbeitet Kommentare als eigenen Annotationstyp.
FA 8	Chat	Kann	Nicht umgesetzt	Eine Chat-Funktion wurde im Rahmen des Prototyps nicht implementiert, da der Fokus auf der Kernfunktionalität des Vorschlagsmodus lag.
FA 9	Anonyme Editierung	Soll	Vollständig erfüllt	Das CRDT-Modell erfordert keine Authentifizierung. Die `TrackChangesApplication` erwartet lediglich eine `userId`, die auch clientseitig generiert werden kann. Eine weitergehende Authentifizierung etwa mit Serverkomponenten wurde aus Zeitgründen nicht umgesetzt.
FA 10	Versionshistorie	Kann	Teilweise erfüllt	Das Konzept des `CAnnotationLog` als unveränderliches Append-only-Log speichert die gesamte Historie aller Operationen.

Kürzel	Titel	Priorität	Status	Begründung der Bewertung
				Eine Auswertung des Logs in Bezug auf die Historie der Änderungen erfolgte jedoch nicht.
FA 11	Änderungen nach Nutzenden filtern	Kann	Nicht umgesetzt	Aufbauend auf FA 10 wurde auch diese Funktion im Prototyp nicht umgesetzt. Die in jeder Annotation gespeicherte `userId` liefert jedoch die konzeptionelle Grundlage dafür.

Alle „Muss"- und „Soll"-Anforderungen werden schon durch diese frühe Implementierung des Prototyps erfüllt. Die als optional („Kann") klassifizierten Anforderungen wurden zugunsten der Fokussierung auf die zentralen Forschungsfragen dieser Arbeit nur teilweise oder gar nicht implementiert. Die Architektur der Bibliothek erlaubt es jedoch, dass diese Funktionen in zukünftigen Versionen ergänzt werden können.

7.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen aus Abschnitt 3.2 beschreiben die Qualitätsmerkmale der Softwarebibliothek. Auch wenn sie primär als Entscheidungshilfe der Architektur- und Technologieentscheidungen dienen sollten, erfolgt hier eine kurze Einschätzung ihrer Erfüllung.

7.2.1 Latenz (NFA 1, NFA 2, NFA 3)

Die Anforderungen an eine geringe Latenz werden durch den Einsatz eines CRDT-basierten Ansatzes adressiert, da hier lokale Änderungen unmittelbar im lokalen Replikat des Dokuments ausgeführt werden. Dadurch ist die Eingabelatenz (NFA 1) für Nutzende minimal und nicht von der Netzwerkverbindung abhängig. Die Wahl der Collabs-Bibliothek sorgt zudem für eine schnelle Verarbeitung eingehender Operationen, was die Latenz bei kollaborativen Änderungen (NFA 2) gering hält. Auch Operationen wie das Annehmen oder Ablehnen von Vorschlägen (NFA 3) bedeuten aufgrund ihrer separaten Verarbeitung als einfache Annotationsbereiche keinen hohen Rechenaufwand und sind daher auch ohne größere Eingabeverzögerung ausführbar.

7.2.2 Konsistenz bei Nebenläufigkeit (NFA 4)

Die Konsistenz des Dokumentenzustands ist die zentrale Eigenschaft von CRDTs. Durch die Nutzung des wissenschaftlich fundierten Collabs-Frameworks (siehe Abschnitt 6.1.3) wird die Konvergenz der Dokumente auf allen Instanzen sichergestellt. Das in Abschnitt 5 entworfene Modell für Annotationen stellt durch eindeutige Kennungen, eine Darstellung der direkten Abhängigkeiten und die Verarbeitung im Append-only-Log sicher, dass auch bei nebenläufigen Änderungen an Vorschlägen oder Kommentaren ein konsistenter Zustand erreicht wird.

7.2.3 Entwicklererfahrung und Integration (NFA 5, NFA 6, NFA 7)

Die Anforderungen werden durch eine modulare Architektur und eine klar definierte öffentliche Schnittstelle erfüllt. Die Aufteilung in eine Kernbibliothek (track-changes-crdt) und eine Erweiterungsbibliothek (`track-changes-application`), kombiniert mit einer konsequenten Typisierung in TypeScript, steigert die Verständlichkeit (NFA 5). Ergänzt wird dies durch eine ausführliche Schnittstellenbeschreibung, aus der automatisch eine vollständige Dokumentation generiert wird.

Als npm-Paket konzipiert, lässt sich die Bibliothek problemlos in modernen Webprojekten installieren (NFA 6). Ihre API ist bewusst UI-Framework-agnostisch gestaltet und ermöglicht so eine flexible Integration in unterschiedliche Editor-Frameworks (NFA 7). Die Praxistauglichkeit wurde erfolgreich in einer Implementierung mit dem CodeMirror-Editor unter Beweis gestellt.

7.2.4 Geringer Entwicklungsaufwand (NFA 8)

Wie in Abschnitt 6.1.3 dargelegt, wurde mit Collabs ein Framework gewählt, das die Erweiterung um benutzerdefinierte CRDTs vorsieht. Durch die Wiederverwendung bestehender Bausteine (z. B. für den Text-CRDT und die Kommunikationsschicht) konnte der Entwicklungsaufwand reduziert und der Fokus auf die Konzeption und Implementierung der Annotationslogik gelegt werden.

7.3 Textanforderungen

Die in Abschnitt 2.4 formulierten Textanforderungen (TA) sind das Ergebnis der Analyse von Grenzfällen in etablierten Editoren. Im Folgenden wird für jede Anforderung erläutert, wie sie im vorgestellten CRDT-Konzept und dessen Implementierung umgesetzt wurde.

7.3.1 TA 1: Bearbeitung des eigenen Vorschlags

Ein Nutzer kann einen eigenen Einfüge-Vorschlag bearbeiten, ohne dass neue Vorschläge entstehen. Dies wird dadurch erreicht, dass die `insert`- und `delete`-Methoden der `TrackChanges`-Klasse prüfen, ob die Operation innerhalb einer bestehenden Annotation desselben Nutzers stattfindet. Ist dies der Fall, wird die Textänderung direkt im zugrundeliegenden Text-CRDT ausgeführt, ohne eine neue Annotation zu erzeugen.

7.3.2 TA 2 & TA 3: Wachsen von eigenen Einfüge-Vorschlägen

Das automatische Wachsen von Einfüge-Vorschlägen im Vorschlagsmodus (TA 2) wird durch das in Abschnitt 5 beschriebene Konzept der „offenen Ankerpunkte“ realisiert. Bei der Erstellung einer `insert_suggestion`-Operation werden die Flags `open_start` und `open_end` gesetzt, um eine Range zu definieren, die über die initial eingefügten Zeichen hinausgeht. Schreibt der Nutzer im Editierungsmodus (TA 3), wird dieses Wachsen unterbunden, indem die `insert`-Methode eine `update`-Operation erzeugt, welche die offene Range des Vorschlags an den neu eingefügten Buchstaben heftet und so von der neuen Einfügung abgrenzt.

7.3.3 TA 4 & TA 5: Wachsen von Lösch-Vorschlägen

Analog zu Einfüge-Vorschlägen wachsen auch Lösch-Vorschläge, wenn im Vorschlagsmodus am Rand der Range weiter gelöscht wird (TA 4). Die `delete`-Methode prüft, ob angrenzend an den neuen Löschbereich bereits ein Lösch-Vorschlag desselben Nutzers existiert. Ist dies der Fall, wird der bestehende Vorschlag durch eine `update`-Operation erweitert, statt einen

neuen zu erstellen. Im Editierungsmodus (TA 5) wird stattdessen eine direkte Löschung im Text-CRDT vorgenommen, wodurch der bestehende Vorschlag unberührt bleibt.

7.3.4 TA 6: Umgang mit Löschungen bei Lösch-Vorschlägen

Diese Anforderung wird dadurch erfüllt, dass bei einem Lösch-Vorschlag die Zeichen nicht tatsächlich aus dem Text-CRDT entfernt werden. Stattdessen wird lediglich eine `delete_suggestion`-Annotation über den entsprechenden Bereich gelegt. Die UI ist dann dafür verantwortlich, diesen Text als durchgestrichen zu rendern. Erst beim Akzeptieren des Vorschlags wird die eigentliche Löschoperation im Text-CRDT ausgeführt.

7.3.5 TA 7: Bearbeitung von Vorschlägen anderer Nutzer im Editierungsmodus

Wird ein Vorschlag eines anderen Nutzers im Editierungsmodus bearbeitet, wird die Änderung Teil des ursprünglichen Vorschlags. Die Implementierung der `insert`-Methode erkennt diesen Fall und fügt den Text in den Text-CRDT ein, ohne die bestehende Annotation zu verändern. Da die Annotation auf den Positionen der äußersten Buchstaben basiert, wird die neue Einfügung automatisch von ihr abgedeckt.

7.3.6 TA 8: Wachsen von Vorschlägen anderer Nutzer

Ein fremder Vorschlag wächst nicht, wenn an seinem Rand geschrieben wird. Dies wird durch die Logik in den `insert`- und `delete`-Methoden sichergestellt, die bei Operationen am Rande einer fremden Annotation regulären Text einfügen bzw. löschen, anstatt die fremde Annotation zu modifizieren.

7.3.7 TA 9: Bearbeitung von Vorschlägen anderer Nutzer im Vorschlagsmodus

Wird ein fremder Vorschlag im Vorschlagsmodus bearbeitet, wird ein neuer, verschachtelter Vorschlag erstellt. Dies wird umgesetzt, indem die `insert`-Methode eine neue, eigenständige `addition`-Operation erzeugt. Die Abhängigkeit, dass der innere Vorschlag bei Ablehnung des äußeren ebenfalls entfernt wird, ist noch nicht komplett umgesetzt. Da die Buchstaben, an die die innere Annotation gebunden ist, auch nach einer Löschung noch als Grabsteine vorhanden sind, ist eine manuelle Logik nötig, um die Annotation tatsächlich zu entfernen. Diese ist noch nicht implementiert.

7.3.8 TA 10: Löschung über Vorschlagsgrenzen hinweg

Umfasst eine Löschung im Vorschlagsmodus Text, der sich über mehrere Vorschläge oder normalen Text erstreckt, wird ein einziger, übergreifender Lösch-Vorschlag erstellt. Die `delete`-Methode erkennt, wenn sich eine Löschung nicht ausschließlich in einem Vorschlag des gleichen Nutzers befindet und erzeugt in diesem Fall eine einzelne `delete_suggestion`-Operation, die den gesamten markierten Bereich abdeckt, anstatt zu versuchen, die bestehenden Annotationen zu modifizieren.

7.3.9 TA 11: Wachsen von Kommentaren

Kommentare wachsen nicht automatisch. Dies wird bei der Erstellung einer `add_comment`-Operation sichergestellt, indem die `open_start`- und `open_end`-Flags auf `false` gesetzt werden. Dadurch wird die Annotation fest an die initial markierten Zeichen gebunden. Änderungen innerhalb der markierten Zeichen werden Teil des Kommentars, während Änderungen außerhalb keine Auswirkung auf die Annotation haben.

8 Fazit

Komfortable Reviewprozesse, nahtlose Zusammenarbeit, Zugriff von überall, Flexibilität, Einfachheit - seit über 30 Jahren sind das die Kriterien für eine ideale kollaborative Textarbeit. Doch während Richtext-Editoren heute fast alle dieser Anforderungen erfüllen, werden die meisten Nur-Text-Dokumente noch immer bearbeitet wie vor – ebendiesen – 30 Jahren.

Das Ziel dieser Arbeit war es, den Grundstein für kollaborative Texteditoren zu legen, die den Komfort moderner Richtext-Anwendungen auf die Bearbeitung von Nur-Text-Formaten übertragen.

Dafür wurde zunächst eine Marktanalyse durchgeführt. Aus der Untersuchung etablierter Programme wie Google Docs und Microsoft Word Online konnten wir funktionale und nichtfunktionale Anforderungen für die Implementierung einer Softwarebibliothek sowie spezifische Textanforderungen für die intuitive Handhabung von Vorschlägen und Kommentaren ableiten. Diese Analyse schuf eine Basis für die nachfolgende technische Konzeption eines Vorschlagsmodus. Die Auseinandersetzung mit den theoretischen Grundlagen von Echtzeit-Kollaboration führte zur Entscheidung, bei der Umsetzung auf CRDTs zu setzen.

Der Fokus liegt dabei auf der Entwicklung eines Konzepts für einen CRDT, der einen Vorschlagsmodus nativ integriert. Es wurde gezeigt, dass naive Ansätze wie die Verwendung von Markup-Zeichen oder generischen JSON-Strukturen bei nebenläufigen Bearbeitungen zu Inkonsistenzen und dem Verlust der Nutzerintention führen. Das vorgestellte Konzept löst diese Probleme, indem es Annotationen als eigene Datenstruktur behandelt, die über kommutativ gestaltete Operationen verwaltet wird und deren Annotationen über eindeutige Kennungen mit einem grundlegenden Text-CRDT verknüpft sind. Eine daraus abgeleitete effiziente lokale Darstellung vereinfacht die Anbindung an ein UI.

Die prototypische Umsetzung dieser Konzepte in Form einer wiederverwendbaren Softwarebibliothek unter Verwendung des Collabs-Frameworks hat die Praxistauglichkeit des entworfenen Modells bestätigt. Die Evaluation zeigte, dass alle als kritisch eingestuften funktionalen und nichtfunktionalen Anforderungen erfüllt wurden. Insbesondere konnten auch nahezu alle zuvor erarbeiteten Textanforderungen mithilfe dieses Modells umgesetzt werden, sodass die Bibliothek das erwartete, gebrauchstaugliche Verhalten des Vorschlagsmodus korrekt abbildet.

Aufbauend auf dieser prototypischen Entwicklung eröffnen sich zahlreiche Möglichkeiten, um das System weiterzuentwickeln.

Ein unmittelbarer nächster Schritt wäre die Vervollständigung der im Prototyp nur teilweise oder nicht umgesetzten Anforderungen. Dazu zählt die Implementierung der optionalen „Kann“-Anforderungen wie einer filterbaren Versionshistorie (FA 10, FA 11). Die hierfür notwendigen Daten, wie die `userId` in jeder Annotation oder ein Operationenprotokoll, sind in der Architektur bereits vorhanden und müssen lediglich für das UI aufbereitet werden. Ebenso sollte die Handhabung entfernter Vorschläge (TA 9) implementiert werden. Ein weiterer Schritt könnte dann die Entwicklung einer umfassenden Test-Suite sein, um sicherzustellen, dass tatsächlich alle Grenzfälle korrekt gehandhabt werden. Damit könnte der Prototyp zu einer robusten Softwarelösung weiterentwickelt werden.

Diese könnte dann in eine vollständige Anwendung zum kollaborativen Schreiben von Nur-Text-Dokumenten eingebettet werden. Dabei wären insbesondere Herausforderungen wie die Vorschau verschiedener Auszeichnungssprachen und die Integration in ein Dateisystem, das etwa auch die Einbettung von Bildern oder sogar eine Git-Integration ermöglicht, zu lösen. Auch die Ergänzung um serverseitige Komponenten, die über die reine Synchronisation hinausgehen, wäre an dieser Stelle sinnvoll. Eine Serverinfrastruktur könnte für die persistente Speicherung von Dokumenten, eine zentrale Benutzerauthentifizierung sowie eine differenzierte Rechteverwaltung sorgen. Alternativ könnte der völlig dezentrale Ansatz von CRDTs weiterverfolgt und untersucht werden, wie etwa Authentifizierung und persistente Speicherung in verteilten Systemen ohne zentralen Server gehandhabt wird.

Weitere Forschung könnte sich zudem auf die Optimierung der CRDT-Algorithmen konzentrieren. So könnten etwa Operationen auf Ebene von Textsegmenten statt einzelner Zeichen zu Effizienzsteigerungen bei sehr großen Dokumenten führen.

Mit der Konzeption eines CRDTs, der Kommentare und Änderungsvorschläge unterstützt, konnte gezeigt werden, dass ein Vorschlagsmodus für CRDTs ein zwar komplexes aber lösbares Problem ist. In weiterer Arbeit kann darauf aufgebaut werden, um moderne Arbeitsabläufe auch bei der Bearbeitung von Markdown, Typst, AsciiDoc und Co zu ermöglichen.

Literaturverzeichnis

Automerge. 2025. „GitHub - automerge/automerge: A JSON-like data structure (a CRDT) that can be modified concurrently by different users, and merged again automatically.“. 2025. https://github.com/automerge/automerge.

Briot, Loïck, Pascal Urso, und Marc Shapiro. 2016. „High Responsiveness for Group Editing CRDTs“. In *Proceedings of the 2016 ACM International Conference on Supporting Group Work*, 51–60. GROUP '16. Sanibel Island, Florida, USA: Association for Computing Machinery. https://doi.org/10.1145/2957276.2957300.

Brischoux, Francois, und Pierre Legagneux. 2009. „Don't format manuscripts“. *The Scientist* 23 (7): 24. https://www.the-scientist.com/dont-format-manuscripts-44040.

CryptPad. 2013. „GitHub - cryptpad/chainpad: Realtime Collaborative Editor Algorithm“. 29. November 2013. https://github.com/cryptpad/chainpad.

Daleckis, Vadims. 2023. „json-joy“. 2023. https://jsonjoy.com/blog/list-crdt-benchmarks.

Donham, Jake, und Devon Rifkin. 2023. „GitHub Next: Realtime GitHub“. GitHub Next. 9. November 2023. https://githubnext.com/projects/rtgh/.

Ellis, C. A., und S. J. Gibbs. 1989. „Concurrency control in groupware systems“. *SIGMOD Rec.* 18 (2): 399–407. https://doi.org/10.1145/66926.66963.

Gentle, Seph. 2021. „CRDTs go brrr“. 2021. https://josephg.com/blog/crdts-go-brrr/.

Google Inc. 2025. „Microsoft Word: Edit Documents - Apps on Google Play“. 11. August 2025. https://play.google.com/store/apps/details?id=com.microsoft.office.word.

Haverbeke, Martin. 2015. „Collaborative editing in ProseMirror“. 28. Juli 2015. https://marijnhaverbeke.nl/blog/collaborative-editing.html.

ISO/IEC JTC 1/SC 7. 2023. „Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model“. https://www.iso.org/standard/78176.html.

Izatt, Matthew. 2022. „Year in Review: 12 awesome ways for developers to learn, build, and grow with Google Workspace“. 2022. https://developers.googleblog.com/en/year-in-review-12-awesome-ways-for-developers-to-learn-build-and-grow-with-google-workspace.

Kazman, Rick, Mark Klein, und Paul Clements. 2000. „ATAM: Method for Architecture Evaluation“. https://doi.org/10.21236/ada382629.

Kleppmann, Martin. 2020. „CRDTs: the hard parts“. 6. Juli 2020. https://www.youtube.com/watch?v=x7drE24geUw.

Litt, Geoffrey, Sarah Lim, Martin Kleppmann, und Peter van Hardenberg. 2022. „Peritext: A CRDT for Collaborative Rich Text Editing“. *Proc. ACM Hum.-Comput. Interact.* 6 (CSCW2). https://doi.org/10.1145/3555644.

Lloyd, Wyatt, Michael J. Freedman, Michael Kaminsky, und David G. Andersen. 2011. „Don't settle for eventual: scalable causal consistency for wide-area storage with COPS“. In

Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 401–16. SOSP '11. Cascais, Portugal: Association for Computing Machinery. https://doi.org/10.1145/2043556.2043593.

Nichols, David A., Pavel Curtis, Michael Dixon, und John Lamping. 1995. „High-latency, low-bandwidth windowing in the Jupiter collaboration system“. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, 111–20. UIST '95. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery. https://doi.org/10.1145/215585.215706.

Nicolaescu, Petru, Kevin Jahns, Michael Derntl, und Ralf Klamma. 2015. „Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types“. In . https://doi.org/10.1007/978-3-319-19890-3_55.

Oster, Gérald, Pascal Urso, Pascal Molli, und Abdessamad Imine. 2006. „Data consistency for P2P collaborative editing“. In *Conference on Computer Supported Cooperative Work*. https://api.semanticscholar.org/CorpusID:14596943.

Posner, Ilona R., und Ronald M. Baecker. 1992. „How People Write Together“. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, 127–38.

Preguica, Nuno, Joan Manuel Marques, Marc Shapiro, und Mihai Letia. 2009. „A Commutative Replicated Data Type for Cooperative Editing“. In *2009 29th IEEE International Conference on Distributed Computing Systems*, 0:395–403. https://doi.org/10.1109/ICDCS.2009.20.

Roh, Hyun-Gul, Myeongjae Jeon, Jin-Soo Kim, und Joonwon Lee. 2011. „Replicated abstract data types: Building blocks for collaborative applications“. *J. Parallel Distrib. Comput.* 71 (3): 354–68. https://doi.org/10.1016/j.jpdc.2010.12.006.

Schmid, Andreas, Michael Bierschneider, Johannes Hoffmann, Yu Liu, und Raphael Wimmer. 2023. „Effects of Text Input Latency on Performance and Task Load“. *MUM '23: Proceedings of the 22nd International Conference on Mobile and Ubiquitous Multimedia*, Dezember, 99–107. https://doi.org/10.1145/3626705.3627784.

Streamich. 2025. „GitHub - streamich/json-joy: json-joy is a library that implements cutting-edge real-time and collaborative editing algorithms and utilities for JSON data models, with a focus on developing the JSON CRDT (Conflict-free Replicated Data Type) specification and implementation.“. 2025. https://github.com/streamich/json-joy.

Sun, Chengzheng, Xiaohua Jia, Yanchun Zhang, Yun Yang, und David Chen. 1998. „Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems“. *ACM Trans. Comput.-Hum. Interact.* 5 (1): 63–108. https://doi.org/10.1145/274444.274447.

Weidner, Matthew, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, Yicheng Zhang, Gregory Schare, Xuan Tang, Sicheng Xing, und Heather Miller. 2023. „Collabs: A Flexible and Performant CRDT Collaboration Framework“. 2023. https://arxiv.org/abs/2212.02618.

Weidner, Matthew, und Martin Kleppmann. 2023. „The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing“. 2023. https://arxiv.org/abs/2305.00583.

Weiss, Stéphane, Pascal Urso, und Pascal Molli. 2008. „Logoot: a P2P collaborative editing system“. In . https://api.semanticscholar.org/CorpusID:13892890.

Xu, Yi, Chengzheng Sun, und Mo Li. 2014. „Achieving convergence in operational transformation: conditions, mechanisms and systems“. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, 505–18. CSCW '14. Baltimore, Maryland, USA: Association for Computing Machinery. https://doi.org/10.1145/2531602.2531629.

Yjs. 2025. „GitHub - yjs/yjs: Shared data types for building collaborative software“. 2025. https://github.com/yjs/yjs.

Abbildungsverzeichnis

Abbildung 1 Ein beispielhafter Ablauf einer Echtzeit-Kollaborations-Sitzung (Sun u. a. 1998) 24
Abbildung 2 Partielle Ordnung (Oster u. a. 2006) 30
Abbildung 3 Der Zustand eines lokalen Dokuments. Jeder Buchstabe hat eine eindeutige Kennung. Ein Boolean Wert markiert, ob der Buchstabe angezeigt wird oder nicht. 37
Abbildung 4 Fügt Alice einen Buchstaben im Vorschlagsmodus hinzu, wird neben dem Buchstaben auch eine Annotation eingefügt, die vom Startpunkt über den eingefügten Buchstaben bis zum darauffolgenden Buchstaben reicht. 38
Abbildung 5 Werden innerhalb der Range Buchstaben eingefügt, so wächst die Range, ohne dass eine neue Range erstellt werden muss. 39
Abbildung 6 Ein beispielhafter Lösch-Vorschlag im Vorschlagsmodus. Die Zeichen `13@A` und `14@A` wurden zur Löschung vorgeschlagen. 40
Abbildung 7 Eine Annotation reicht von `8@A` bis `1@B`, während eine zweite Annotation von `12@A` bis `5@B` reicht. Sie überlappen sich. 42
Abbildung 8 Die Datenpunkte enthalten alle ab ihrer Position gültigen Annotationen. ... 43
Abbildung 9 Die Architektur der Track-Changes-Bibliotheken 49

Tabellenverzeichnis

Tabelle 1 Die in diesem Kapitel zu beantwortenden Forschungsfragen. 3
Tabelle 2 Kollaborative Textbearbeitungsprogramme, die aktuell auf dem Markt sind 3
Tabelle 3 Die ausgewählten Vergleichskriterien 5
Tabelle 4 Übersicht der implementierten Funktionen 7
Tabelle 5 Rechteverwaltung auf Dokumentebene im Vergleich (VK 15) 7
Tabelle 6 Die grundsätzlichen Funktionen und Modi aus denen die zu betrachtenden Grenzfälle bestimmt werden 8
Tabelle 7 Die nach der Pairwise-Methode kombinierten Grenzfälle 9
Tabelle 8 Forschungsfragen zur verteilten Echtzeit-Kollaboration. 21
Tabelle 9 Die Erfüllung der Konsistenzkriterien bei CRDTs und OT im Vergleich 30
Tabelle 10 Die Gegenüberstellung der vier ausgewählten CRDT-Bibliotheken anhand der ausgewählten Kriterien. 48
Tabelle 11 Übersicht und Bewertung der Erfüllung der funktionalen Anforderungen. 62

Codeblockverzeichnis

Codeblock 1 Der Beginn der Track-Changes-Klasse. 51
Codeblock 2 Verkürztes Datenmodell der Annotationen. Der `Annotation`-Typ fasst die drei spezialisierten Operationen zusammen, die sich eine `AnnotationBase` teilen und durch das `action`-Feld unterschieden werden. 53
Codeblock 3 Der Beginn der `TrackChangesApplication`-Klasse. Im Constructor wird deutlich, dass die `CMap` für die Instanziierung neuer `TrackChangesDocument` verantwortlich ist. 56

Glossar

CRDT, Conflict-Free Replicated Data Types: Eine Klasse von Datenstrukturen, die für die Echtzeit-Kollaboration in verteilten Systemen entwickelt wurde.

(pp. 4, 5, 1, 21, 22, 23, 25, 26, 28, 30, 31, 32, 33, 34, 35, 36, 37, 45, 46, 47, 48, 49, 50, 51, 52, 54, 55, 60, 62, 63, 64, 65, 66, 67, 68, 72)

CSCW, Computer Supported Collaborative Work: Ein Forschungsgebiet, das untersucht, wie Gruppen zusammenarbeiten und wie diese Zusammenarbeit durch Computertechnologie unterstützt und verbessert werden kann.

(pp. 21)

Echtzeit-Kollaboration: Bezeichnet Technologien und Plattformen, die es Nutzenden ermöglichen, quasi in Echtzeit zu kommunizieren und gemeinsam an Inhalten zu arbeiten. Der Begriff „Echtzeit“ wird hier trotz einer systembedingten Latenz (typischerweise unter 200 ms) verwendet. Er beschreibt die synchrone Natur des Systems, die – im Gegensatz zu asynchronen Systemen – eine unmittelbare Übertragung von Nutzeränderungen an alle Teilnehmenden ermöglicht.

(pp. 1, 3, 4, 5, 17, 21, 22, 24, 45, 62, 67, 72)

Nur-Text: Bezeichnet Textformate, die im Gegensatz zu Richtext keine Formatierungs- oder Layoutinformationen enthalten. Die Quelldateien bestehen aus reinem Text, sind daher menschenlesbar und können mit jedem einfachen Texteditor geöffnet und bearbeitet werden. Beispiele sind einfache Textdateien (.txt) sowie Auszeichnungssprachen wie Markdown (.md) oder HTML (.html).

(pp. 4, 1, 3, 6, 17, 21, 34, 35, 36, 37, 45, 50, 60, 62, 67, 68)

OT, Operational Transformation: Eine etablierte Kategorie von Algorithmen für die Echtzeit-Kollaboration. OT ist einer der am weitesten verbreiteten Ansätze in der Praxis.

(pp. 4, 21, 22, 23, 26, 28, 30, 31, 32, 35, 72)

P2P, Peer-to-Peer: Eine Netzwerktopologie, in der alle Teilnehmenden (Peers) gleichberechtigt sind und direkt miteinander kommunizieren, ohne dass ein zentraler Server für die Vermittlung erforderlich ist.

(pp. 22, 23, 26)

Richtext: Im Gegensatz zu Rohtext verknüpfen Richtextformate Text- mit Layout- und Formatierungsinformationen. Die Quelldateien sind nicht direkt menschenlesbar und erfordern spezielle Software (z. B. Textbearbeitungsprogramme) zur korrekten Darstellung und Bearbeitung. Typische Beispiele sind die Dateiformate von Microsoft Word (.docx) und LibreOffice Writer (.odt).

(pp. 1, 3, 33, 67)

UI, User Interface: Die Benutzerschnittstelle, über die ein Mensch mit einer Maschine oder einer Software interagiert. Sie umfasst alle Elemente, die der Eingabe (z. B. Tasten, Menüs) und Ausgabe (z. B. Bildschirmanzeigen) dienen.

(pp. 5, 16, 20, 33, 45, 57, 60, 63, 67)

Vorschlagsmodus: Bezeichnet eine Funktion in Textbearbeitungsprogrammen, bei der von Nutzenden vorgenommene Änderungen nicht direkt übernommen werden. Stattdessen werden sie als Vorschläge markiert und können zu einem späteren Zeitpunkt einzeln ange-

nommen oder abgelehnt werden. In dieser Arbeit umfasst der Begriff Vorschlagsmodus in den meisten Fällen auch das Erstellen von Kommentaren zu bestimmten Textabschnitten.

(pp. 4, 1, 3, 5, 6, 8, 9, 17, 31, 32, 33, 35, 37, 38, 52, 55, 62, 65, 67, 68, 72)

Anhang: Grenzfallbetrachtungen

Szenario	Im Vorschlagsmodus wird ein Vorschlag ersetzt.
Fall	GF 1
Umgebung	Beide arbeiten im Vorschlagsmodus. Nutzerin A hat vorgeschlagen, das Wort 'Stadt' durch 'Ort' zu ersetzen.
Stimulus	Nutzer B markiert den Vorschlag und ersetzt ihn durch 'Dorf'.
Antwort	**Google Docs**: Der komplette Vorschlag A wird durchgestrichen. Vorschlag B wird im Kommentar als ‚Ersetzen: „OrtStadt“ durch „Dorf“‘ angegeben. **Microsoft Word online**: Nur ‚Ort‘ wird durchgestrichen. Vorschlag B wird als ‚Ort wurde durch Dorf ersetzt‘ angegeben. **Collabora Online**: Jede Änderung wird als separater Vorschlag erfasst. Der finale Zustand ist: ‚Stadt‘ wird von Nutzer A gelöscht, ‚Ort‘ wird von Nutzer B gelöscht, ‚Dorf‘ wird von Nutzer B ergänzt. Die visuelle Reihenfolge bleibt bei ‚StadtOrtDorf‘. **OnlyOffice**: Jede Änderung wird als separater Vorschlag erfasst. Der finale Zustand ist: ‚Stadt‘ wird von Nutzer A gelöscht, ‚Ort‘ wird von Nutzer B gelöscht, ‚Dorf‘ wird von Nutzer B ergänzt. Die visuelle Reihenfolge ist ‚Stadt-DorfOrt‘. **Overleaf**: Vorschlag A wird gänzlich überschrieben. Vorschlag B wird im Kommentar als ‚Changed Stadt to Dorf‘ angegeben.

Szenario	Im Editierungsmodus wird am Rand eines Vorschlags etwas bearbeitet.
Fall	GF 2
Umgebung	Nutzerin A arbeitet im Vorschlagsmodus, Nutzer B im Editierungsmodus. Nutzerin A hat vorgeschlagen, das Wort Wombat durch 'flauschiges Tier' zu ersetzen.
Stimulus	Nutzer B ändert 'flauschig' zu 'borstig'.
Antwort	**Google Docs**: Der Vorschlag A wird gekürzt um ‚flauschig‘ und besteht nun noch aus ‚es Tier‘. ‚borstig‘ wird als normaler Text übernommen. **Microsoft Word online**: Der Vorschlag A wird gekürzt um ‚flausch‘ und besteht nun noch aus ‚iges Tier‘. ‚borst‘ wird als normaler Text übernommen. **Collabora Online**: Der Vorschlag A wird gekürzt um ‚flauschig‘ und besteht nun noch aus ‚es Tier‘. ‚borstig‘ wird als normaler Text übernommen. **OnlyOffice**: Der Vorschlag A wird gekürzt um ‚flauschig‘ und besteht nun noch aus ‚es Tier‘. ‚borstig‘ wird als normaler Text übernommen. **Overleaf**: Der Vorschlag A wird gekürzt um ‚flauschig‘ und besteht nun noch aus ‚es Tier‘. ‚borstig‘ wird als normaler Text übernommen.

Szenario	Im Editierungsmodus wird innerhalb eines Vorschlags etwas bearbeitet.
Fall	GF 2

Umgebung	Nutzerin A arbeitet im Vorschlagsmodus, Nutzer B im Editierungsmodus. Nutzerin A hat vorgeschlagen, das Wort 'vergleichsweise' durch 'einigmaßen' zu ersetzen.
Stimulus	Nutzer B ergänzt mit einem 'er' das 'einigmaßen' zu 'einigermaßen.
Antwort	**Google Docs**: Vorschlag A wird ergänzt um das ‚er', ohne, dass erkennbar ist, dass die Änderung von einem anderen Nutzer kommt. **Microsoft Word online**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als normaler Text übernommen. **Collabora Online**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als normaler Text übernommen. **OnlyOffice**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als normaler Text übernommen. **Overleaf**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als normaler Text übernommen.

Szenario	Im Vorschlagsmodus wird innerhalb eines Vorschlags etwas bearbeitet.
Fall	GF 2
Umgebung	Beide Nutzende arbeiten im Vorschlagsmodus. Nutzerin A hat vorgeschlagen, das Wort 'vergleichsweise' durch 'einigmaßen' zu ersetzen.
Stimulus	Nutzer B ergänzt mit einem 'er' das 'einigmaßen' zu 'einigermaßen.
Antwort	**Google Docs**: Vorschlag A bleibt bestehen wie er ist. Für die Ergänzung des ‚er' wird ein zweiter Vorschlag erstellt. Beide Vorschläge können separat angenommen oder abgelehnt werden. Wird allerdings der umgebene Vorschlag A abgelehnt, verschwindet auf Vorschlag B. **Microsoft Word online**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als zweiter Vorschlag übernommen. Alle drei Vorschläge können unabhängig voneinander angenommen werden. **Collabora Online**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als zweiter Vorschlag übernommen. Alle drei Vorschläge können unabhängig voneinander angenommen werden. **OnlyOffice**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als zweiter Vorschlag übernommen. Alle drei Vorschläge können unabhängig voneinander angenommen werden. **Overleaf**: Vorschlag A wird in zwei Vorschläge ‚einig' und ‚maßen' aufgeteilt. Das ‚er' wird als zweiter Vorschlag übernommen. Alle drei Vorschläge können unabhängig voneinander angenommen werden.

Szenario	Im Editierungsmodus werden zwei Vorschläge ersetzt.
Fall	GF 3 + 6
Umgebung	Nutzerin A arbeitet im Vorschlagsmodus, Nutzer B im Editierungsmodus. Nutzerin A bearbeitet den Satz 'Wombats leben im südlichen und östlichen Australien' und hat vorgeschlagen im Vorschlag A 'südlich' durch 'nördlich' und im Vorschlag B 'östlich' durch 'westlich' zu ersetzen.

Stimulus	Nutzer B markiert den ganzen Satz und ersetzt ihn durch 'Wombats leben irgendwo in Australien'.
Antwort	**Google Docs**: Beide Vorschläge werden entfernt. **Microsoft Word online**: Beide Vorschläge werden entfernt. **Collabora Online**: Beide Vorschläge werden entfernt. **OnlyOffice**: Beide Vorschläge werden entfernt. **Overleaf**: Die Vorschläge werden entfernt, es bleibt jedoch ein Änderungskommentar vorhanden, der die Änderung als ‚Nutzer A deleted südlichöstlich' beschreibt. Die Änderung kann weiterhin angenommen werden. Wird sie das, so verschwindet sie - im Text ändert sich jedoch nichts.

Szenario	Zwei Vorschläge werden hintereinander vom gleichen Autor eingefügt.
Fall	GF 4
Umgebung	Nutzerin A arbeitet im Vorschlagsmodus an den Worten 'breitsohligen Füße'.
Stimulus	Nutzerin A ändert zunächst als Vorschlag A 'breitsohlige ' zu 'große '. Dann ändert sie an anderer Stelle etwas und ersetzt dann als Vorschlag B 'Füße' mit 'Pfoten'.
Antwort	**Google Docs**: Wird ‚Füße' markiert und direkt ersetzt, wird die Ersetzung ‚Pfoten' mit Vorschlag A verbunden und als gelöscht markiert. Der Kommentar beschreibt die Änderung als ‚Ersetzen: „breitsohligen PfotenFüße" durch „großen"'. Wird hingegen ‚Füße' mit Backspace manuell gelöscht, werden Vorschlag A und Vorschlag B als separate Änderungen behandelt. **Microsoft Word online**: Beide Vorschläge werden als separate Änderungen behandelt. **Collabora Online**: Die Vorschläge werden in vier einzelne Änderungen aufgeteilt. ‚breitsohligen' und ‚Füßen' werden gelöscht und ‚große' und ‚Pfoten' eingefügt. **OnlyOffice**: Die Vorschläge werden in vier einzelne Änderungen aufgeteilt. ‚breitsohligen' und ‚Füßen' werden gelöscht und ‚große' und ‚Pfoten' eingefügt. **Overleaf**: Beide Vorschläge werden in einer Änderung zusammengefasst.

Szenario	Es wird im Vorschlagsmodus Text aus Vorschlägen unterschiedlicher Autoren gelöscht.
Fall	GF 5
Umgebung	Beide Nutzende arbeiten im Vorschlagmodus. Nutzerin A schreibt den Vorschlag 'Sie leben in' und Nutzer B schreibt direkt dahinter ' Höhlen'.
Stimulus	Nutzer B markiert den Text 'in Hö' und löscht ihn.
Antwort	**Google Docs**: Es werden insgesamt zwei Vorschläge generiert: Die Hinzufügung von Nutzerin A und eine Ersetzung durch Nutzer B. Letztere wird so beschrieben: ‚Ersetzen: „in H" durch „öhlen"'. Das gelöschte ‚H' wird als durchgestrichen angezeigt. **Microsoft Word online**: Es werden insgesamt zwei Vorschläge generiert: Die

	Hinzufügung von Nutzerin A und eine Ersetzung durch Nutzer B. Letztere wird so beschrieben: ‚in wurde durch öhlen ersetzt'. Das gelöschte ‚H' wird nicht mehr angezeigt. **Collabora Online**: Es werden insgesamt zwei Vorschläge generiert: Die Hinzufügung von Nutzerin A und eine Ersetzung durch Nutzer B. Das gelöschte ‚H' wird nicht mehr angezeigt. **OnlyOffice**: Es werden insgesamt zwei Vorschläge generiert: Die Hinzufügung von Nutzerin A und eine Ersetzung durch Nutzer B. Das gelöschte ‚H' wird nicht mehr angezeigt. **Overleaf**: Es werden insgesamt zwei Vorschläge generiert: Die Hinzufügung ‚Sie leben' von Nutzerin A und die Hinzufügung ‚öhlen' von Nutzer B. Das Löschen wird nicht abgebildet.

Szenario	Es wird im Vorschlagsmodus ein Vorschlag gelöscht.
Fall	GF 7
Umgebung	Beide Nutzende arbeiten im Vorschlagsmodus. Nutzerin A schreibt den Vorschlag 'Wombats sind Pflanzenfresser.'
Stimulus	Nutzer B löscht den Vorschlag.
Antwort	**Google Docs**: Es wird ein zweiter Vorschlag für die Löschung generiert. Vorschlag A wird durchgestrichen, Vorschlag B lautet 'Löschen: „Wombats sind Pflanzenfresser.'. **Microsoft Word online**: Es wird ein zweiter Vorschlag für die Löschung generiert. Vorschlag A wird durchgestrichen. **Collabora Online**: Es wird ein zweiter Vorschlag für die Löschung generiert. Vorschlag A wird durchgestrichen. **OnlyOffice**: Es wird ein zweiter Vorschlag für die Löschung generiert. Vorschlag A wird durchgestrichen. **Overleaf**: Der Vorschlag wird gelöscht.

Szenario	Es wird im Vorschlagsmodus Text zwischen zwei Vorschlägen des gleichen Autors gelöscht.
Fall	GF 8
Umgebung	Nutzerin A arbeitet im Vorschlagsmodus, Nutzer B im Editierungsmodus. Sie schreibt in dem Satz 'Wombats zählen zu den kleinsten grabend lebendigen Säugetieren' und korrigiert 'kleinsten' zu 'größten' und 'lebendigen' zu 'lebenden'.
Stimulus	Nutzer B markiert 'ten grabend lebend' und löscht den Text.
Antwort	**Google Docs**: Es werden zwei beschreibende Kommentare für die Ersetzungen von Nutzerin A generiert. Diese ändern sich auch nach der Löschung nicht. Im Text selbst wird nun Vorschlag A als ‚größ' angezeigt und Vorschlag B als ‚en' mit durchgestrichenem ‚lebendigen'. Es ist visuell nicht zu erkennen, dass es sich um separate Vorschläge handelt. **Microsoft Word online**: Es wird nur ein Vorschlag generiert: Eine Ersetzung

	von ‚kleinsten‘ durch ‚größen‘. **Collabora Online**: Es werden zwei Vorschläge generiert: Eine Löschung von ‚kleinsten‘ und eine Hinzufügung von ‚größen‘. **OnlyOffice**: Es werden zwei Vorschläge generiert: Eine Löschung von ‚kleinsten‘ und eine Hinzufügung von ‚größen‘. **Overleaf**: Es werden zwei Vorschläge generiert: Eine Ersetzung von ‚kleinsten‘ zu ‚größ‘ und eine Ersetzung von ‚lebendigen‘ zu ‚en‘.

Szenario	Es werden im Vorschlagsmodus zwei Vorschläge von unterschiedlichen Autoren eingefügt.
Fall	GF 9
Umgebung	Beide Nutzende arbeiten im Vorschlagsmodus. Nutzerin A schreibt Vorschlag A 'Wombats sind '.
Stimulus	Nutzer B ergänzt nach Vorschlag A den Vorschlag B 'Beuteltiere'.
Antwort	**Google Docs**: Beide Vorschläge werden als separate Vorschläge eingefügt und angezeigt. **Microsoft Word online**: Beide Vorschläge werden als separate Vorschläge eingefügt und angezeigt. **Collabora Online**: Beide Vorschläge werden als separate Vorschläge eingefügt und angezeigt. **OnlyOffice**: Beide Vorschläge werden als separate Vorschläge eingefügt und angezeigt. **Overleaf**: Beide Vorschläge werden als separate Vorschläge eingefügt und angezeigt.