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

FELIX JALOWSKI
BACHELOR THESIS

MARKDOWN-UNTERSTÜTZUNG FÜR DIE SWEBLE HUB SOFTWARE

Eingereicht am 30. Juli 2018

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

Versicherung

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

Erlangen, 30. Juli 2018

License

Erlangen, 30. Juli 2018

Abstract

Sweble Hub is a wiki-like, web-based collaboration platform for managing knowledge. So far, users have been able to create and edit articles using either MediaWiki's markup language Wikitext or a visual editor.

To provide users with a better selection of markup languages, the goal of this thesis is to add support for Markdown to Sweble Hub. For this purpose, an overview of existing Markdown variants and parsers is given, needed components are identified and their requirements are specified.

Building onto this, the architecture and implementation of the following components is described: Firstly, a CommonMark-compliant Markdown parser is selected and a renderer, which handles the transformation of a Markdown document into Sweble's Wiki Object Model (WOM), is implemented on top of it. Additionally, a Pretty Printer, which handles a transformation of WOM documents to Markdown, is realized.

Zusammenfassung

Das Sweble Hub ist eine Wiki-ähnliche, webbasierte Kollaborationsplattform zum Wissensmanagement. Bisher konnten Nutzer zur Bearbeitung von Artikeln entweder MediaWikis Auszeichnungssprache Wikitext, oder einen visuellen Editor verwenden.

Um Nutzern eine bessere Auswahl an Auszeichnungssprachen anbieten zu können, ist das Ziel dieser Arbeit eine Unterstützung von Markdown für Sweble Hub. Dazu wird ein Überblick über existierende Markdown-Varianten und Parser gegeben, benötigte Einzelkomponenten identifiziert und Anforderungen an diese gestellt.

Daraufhin wird die Konzeption und Entwicklung der folgenden Komponenten beschrieben: Einerseits wird ein Markdown-Parser ausgewählt, der dem CommonMark-Standard folgt und basierend auf diesem ein Renderer umgesetzt, welcher eine Transformation eines Markdown-Dokumentes in das Wiki Object Model (WOM) von Sweble durchführt. Andererseits wird ein Pretty Printer entwickelt, welcher WOM-Dokumente nach Markdown transformiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Heutzutage wird eine große Menge an Wissen in einer Vielzahl von Wikis mit bis zu Millionen von Artikeln[1] verwaltet, oft mit variierender Syntax[2]. Mit dem Sweble Hub wird eine webbasierte Kollaborationsplattform zum Wissensmanagement geschaffen. Die Funktion und Zielsetzung von Sweble lässt sich mit Wikis im allgemeinen, sowie Wikipedia im speziellen, vergleichen. Ein zentraler Aspekt von Sweble ist dabei das *Wiki Object Model* (WOM) zur Speicherung von Artikeln. Im Gegensatz zu existierenden Wiki-Projekten, bei denen Dokumente in textueller Form ihrer jeweiligen Auszeichnungssprache gespeichert werden, sind WOM-Dokumente in einem standardisierten Format in einer Baumstruktur abgebildet. Dies bietet mehrere Vorteile: einerseits werden automatisierte Refactorings erleichtert (Dohrn und Riehle, 2013), andererseits ist das Format, in dem Dokumente gespeichert werden, damit unabhängig von der Bearbeitungsweise oder -sprache. Bislang unterstützt Sweble als Auszeichnungssprache zur Editierung von Artikeln MediaWiki Wikitext, welches in vielen Wikis, unter anderem der Wikipedia, zur Bearbeitung von Artikeln zum Einsatz kommt. Darüber hinaus wurde ein visueller Editor entwickelt, mit welchem WOM-Dokumente nach dem *What You See Is What You Get*-Prinzip erstellt und editiert werden können (Haase, 2016).

Um Nutzer des Sweble Hub zur Kollaboration anzuregen, muss die Lern- und Einstiegsschwelle so niedrig wie möglich sein. Daher sollte ihnen ermöglicht werden, Artikel auf die Art und Weise zu bearbeiten, die ihnen am komfortabelsten erscheint. Dies kann zum Beispiel dadurch erreicht werden, neben Wikitext auch andere Auszeichnungssprachen zur Bearbeitung anzubieten. Deshalb soll Sweble um die Möglichkeit erweitert werden, Dokumente mit Hilfe von Markdown zu bearbeiten. Die Entscheidung für Markdown ergibt sich aus der weiten Verbreitung: Varianten sind bei einigen großen Webseiten, beispielsweise Reddit[3] und GitHub[4] im Einsatz, um Nutzern die Formatierung ihrer Inhalte zu vereinfachen.

[1] https://wikiindex.org/List_of_wikis_by_number_of_pages
[2] https://www.wikimatrix.org/syntax.php
[3] https://github.com/reddit/snudown
[4] https://help.github.com/articles/about-writing-and-formatting-on-github/

Diese Arbeit ist wie folgt strukturiert: Zunächst wird in Kapitel 2 auf notwendige Grundlagen für die weiteren Abschnitte eingegangen. In Kapitel 3 werden die Anforderungen, welche zum Erreichen des Ziels der Markdown-Unterstützung für Sweble erfüllt sein müssen, detailliert beschrieben. Dabei werden auch die einzelnen dafür notwendigen Komponenten identifiziert. Kapitel 4 beschreibt die Architektur und das Design dieser identifizierten Komponenten. In Kapitel 5 wird daraufhin auf Details der Implementierung eingegangen, die basierend auf den Architekturentscheidungen entwickelt wurde. Zuletzt wird in Kapitel 6 evaluiert, inwiefern die gestellten Anforderungen erfüllt werden konnten, gefolgt von einer Zusammenfassung und einem Ausblick in Kapitel 7.

2 Grundlagen

Dieses Kapitel geht auf grundlegende Begriffe und Technologien ein, die für den weiteren Verlauf der Arbeit benötigt werden.

2.1 Wiki Object Model

Wie in Kapitel 1 bereits kurz beschrieben wurde, ist das WOM eines der Grundbausteine des Sweble Hub. Die Beschreibung des WOM (Dohrn und Riehle, 2011b) entstand aus der ursprünglichen Motivation, einen Wikitext-Parser zu entwickeln, der Zugriff auf den Inhalt von Artikeln in Form eines Abstrakten Syntaxbaumes erlaubt (Dohrn und Riehle, 2011a). Obwohl das WOM zunächst primär zur strukturierten Darstellung von Wikitext-Artikeln konzipiert wurde, erlaubt es letztendlich eine Abstraktion von der zugrundeliegenden Auszeichnungssprache. Artikel werden im WOM in einer Baumstruktur dargestellt, die dem *Document Object Model* (DOM), welches zur Darstellung von HTML-Dokumenten zum Einsatz kommt, ähnelt. Diese Struktur ermöglicht eine einfache automatisierte Transformation von Dokumenten. Die Knotentypen im WOM setzen sich dabei zusammen aus von XHTML übernommen, sowie aus WOM-spezifischen Knoten.

Zur Visualisierung dieser Struktur ist in Auflistung 2.1 ein beispielhafter minimaler Wikitext-Artikel abgebildet, Auflistung 2.2 zeigt die entsprechende WOM-Darstellung dieses Artikels.

```
1 = Heading =
2 Text
```

Auflistung 2.1: Wikitext-Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<article xmlns="http://sweble.org/schema/wom30"
    xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:mww="http://sweble.org/schema/mww30"
    xmlns:wom="http://sweble.org/schema/wom30" version="3.5">
   <body>
      <section level="1">
         <heading>
            <rtd>=</rtd>
               <text> Heading </text>
            <rtd>=</rtd>
         </heading>
         <body>
            <text>
</text>
            <html:p>
               <text>Text</text>
            </html:p>
            <text>
</text>
         </body>
      </section>
   <body>
</article>
```

Auflistung 2.2: WOM-Darstellung des gezeigten Wikitext-Beispiels

Hervorzuheben sind hierbei die *Round Trip Data* (RTD)-Knoten. In diesen ist die Formatierung des zugrundeliegenden Dokumentes in der entsprechenden Auszeichnungssprache festgehalten. Anhand dieser Elemente kann - gemeinsam mit den Text-Elementen des Baumes - eine verlustfreie Rücktransformation zum Originaldokument erfolgen, was in bestimmten Anwendungsfällen notwendig ist, zum Beispiel wenn eine fortgeführte Editierung des Dokumentes in der Auszeichnungssprache erfolgt, die Speicherung allerdings im WOM-Format.

2.2 Markdown

Markdown[5] ist eine von John Gruber entwickelte Auszeichnungssprache mit der Zielsetzung, sowohl leicht lesbar als auch einfach schreibbar zu sein.

Nach Leonard (2016) lassen sich Auszeichnungssprachen auf einem Spektrum zwischen formal und informal einsortieren (Abbildung 2.1). Für informales Markup wird hier auch der Begriff *Lightweight Markup Language* (vereinfachte Auszeichnungssprache) verwendet. Formales Markup bietet den Vorteil einer leichteren

[5] https://daringfireball.net/projects/markdown/

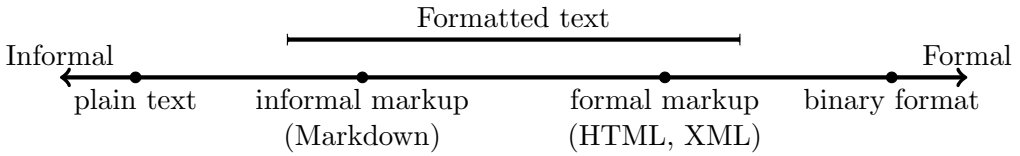

Abbildung 2.1: Formalitäts-Spektrum von Datenformaten für Text (basierend auf Leonard, 2016)

Interoperabilität zwischen verschiedenen Sprachen, da es besser formalisiert ist. Informales Markup hingegen wird als nutzerfreundlicher bezeichnet, da es leichter zu erlernen ist. XML und HTML sind hierbei am formalen Ende des Spektrums einsortiert, während Markdown auf der informalen Seite einzuordnen ist. Es wird hervorgehoben, dass die Zielgruppe für informales Markup im Allgemeinen und Markdown im Speziellen vor allem nicht-technische Nutzer sind, da die Lernkurve vergleichsweise gering ist.

Die ursprüngliche Veröffentlichung von Markdown besteht aus einer Spezifikation der Syntax sowie einem in Perl geschriebenen Parser, der aus Markdown-Dateien HTML-Dokumente erstellt. Markdown erlaubt Nutzern mithilfe seiner Syntax, diverse Formatierungen bei Texten anzuwenden, ohne dabei eine komplexere Auszeichnungssprache wie HTML lernen zu müssen.

2.3 CommonMark

CommonMark[6] ist eine Spezifikation für Markdown, deren Ziel es ist, eine einheitliche und klare Definition von Markdown zu beschreiben. Durch die Popularität von Markdown, verbunden mit der Tatsache, dass die letzte Aktualisierung des originalen Parsers im Dezember 2004 stattfand, ergeben sich mehrere Probleme, die in MacFarlane (2017) identifiziert werden. Einerseits existieren inzwischen viele verschiedene Implementierungen von Markdown. Da die originale Spezifikation in vielen Fällen keine eindeutige Antwort liefert, wie bestimmte Eingaben zu interpretieren sind, teils große Unterschiede zwischen den diversen Parsern. Andererseits folgt selbst die originale Implementierung teils nicht der festgeschriebenen Syntaxbeschreibung.

Mithilfe des Tools *babelmark*[7] lassen sich eine große Zahl an Parser-Implementierungen vergleichen und die Unterschiede zwischen diesen visualisieren. Die folgende Eingabe dient als Beispiel zur Verdeutlichung der Differenzen:

[6] http://commonmark.org/
[7] https://babelmark.github.io

```
1. fee
2. fie
-  foe
-  fum
```

Auflistung 2.3: Markdown-Beispiel: Listen

Der originalen Spezifikation folgend sollten hierbei zwei Listen entstehen, eine nummerierte Liste, sowie eine unsortierte Liste. Allerdings ergeben sich einige Unterschiede in der Darstellung zwischen verschiedenen Parsern:

```
<ol>
   <li>
      fee
   </li>
   <li>
      fie
   </li>
</ol>
<ul>
   <li>
      foe
   </li>
   <li>
      fum
   </li>
</ul>
```

```
<ol>
   <li>
      fee
   </li>
   <li>
      fie
   </li>
   <li>
      foe
   </li>
   <li>
      fum
   </li>
</ol>
```

```
<ol>
   <li>
      fee
   </li>
   <li>
      fie
      <ul>
         <li>
            foe
         </li>
         <li>
            fum
         </li>
      </ul>
   </li>
</ol>
```

cheapskate 0.10.5 Haskell
commonmark.js 0.28.1 Javascript M↓
DFM-Latest 2.37.0.0 C#
earmark 1.1.1 Elixir
GitHub Flavored Markdown 0.17.4 C M↓
league/commonmark 0.16.0 Php M↓
lunamark 0.4.0 Lua
markdig (advanced) 0.15.0.0 C# M↓
Markdig (DocFX) 2.37.0.0 C# M↓
markdig 0.15.0.0 C# M↓
markdown-it 8.4.0 JavaScript M↓
maruku-math 0.7.3.beta1 Ruby
maruku 0.7.3.beta1 Ruby
MD4C (strict) 0.2.7 C M↓
MD4C 0.2.7 C M↓
pandoc (strict) 2.2.1 Haskell

DFM 2.37.0.0 C#
markdown101 1.0.1 Perl
markdown102 1.0.2b8 Perl
multimarkdown 5.1.0 C
python-markdown 2.6.5 Python
rdiscount 2.1.8 Ruby
redcarpet 3.3.4 Ruby
s9e/TextFormatter (Fatdown/PHP) Php

cebe/extra 1.2.0 Php
cebe/gfm 1.2.0 Php
cebe/markdown 1.2.0 Php
kramdown 1.2.0 Ruby
parsedown 1.6.0 Php

Abbildung 2.2: Vergleich von Ergebnissen verschiedener Parser

Dabei werden bereits nur die drei häufigsten Ergebnisse gezeigt, es gibt sogar bei diesem noch recht simplen Besipiel noch weitere Varianten. Selbst die originale Implementierung folgt hier nicht der Spezifikation. Solche Unklarheiten kommen in anderen Situation noch deutlich häufiger vor; *babelmark* listet über 40 verschiedene Beispiele, bei denen Parser zu teils deutlich unterschiedlichen Ergebnissen kommen. CommonMark versucht, solche Situationen so umfassend wie möglich zu klären.

Grundlegende Syntax von CommonMark

Im folgenden Abschnitt wird grundlegend auf die Syntax von CommonMark eingegangen, wie sie in der Spezifikation beschrieben ist.

Es wird zunächst eine grundlegende Unterteilung zwischen Block-Elementen und Inline-Elementen getroffen. Block-Elemente werden darüber hinaus in *Container blocks* und *Leaf blocks* unterteilt, wobei erstere weitere Block-Elemente enthalten können, während letztere Inline-Elemente enthalten. Eine Übersicht über die verschiedenen Elemente ist in Tabelle 2.1 zu sehen.

Container blocks	Leaf blocks	Inlines
Block quotes	Thematic breaks	Backslash escapes
List items	ATX headings	Entity & numeric char refs
Lists	Setext headings	Code spans
	Fenced code blocks	(Strong) Emphasis
	Indented code blocks	Links
	HTML blocks	Images
	Link ref. definitions	Autolinks
	Paragraphs	Raw HTML
	Blank lines	Line breaks
		Textual content

Tabelle 2.1: Übersicht über die Elemente von CommonMark

In Abbildung 2.3 ist ein beispielhaftes CommonMark-Dokument seiner HTML-Darstellung gegenübergestellt. Zur Veranschaulichung der Struktur sind im Quelldokument die entsprechenden Elemente markiert: *Container blocks* sind rot eingerahmt, *Leaf blocks* in grün und Inline-Elemente sind in blau dargestellt.

```
> foo
>
> - bar
> - baz

# foo
```

```
<blockquote>
   <p>foo</p>
   <ul>
      <li>bar</li>
      <li>baz</li>
   </ul>
</blockquote>
<h1>foo</h1>
```

Abbildung 2.3: Beispielhaftes CommonMark-Dokument mit HTML-Darstellung

2.4 Konzepte

2.4.1 Visitor Pattern

Der Visitor (Besucher) ist ein Entwurfsmuster, das dem Zweck dient, Operationen auf Elemente einer Objektstruktur anzuwenden, ohne die Klassen der Elemente verändern zu müssen, mit denen es arbeitet (Gamma, Helm, Johnson und Vlissides, 1995). Das Entwurfsmuster bietet sich vor allem an, wenn viele verschiedene und unabhängige Operationen auf den Elementen angewandt werden sollen und im Laufe der Zeit häufig neue Operationen definiert werden.

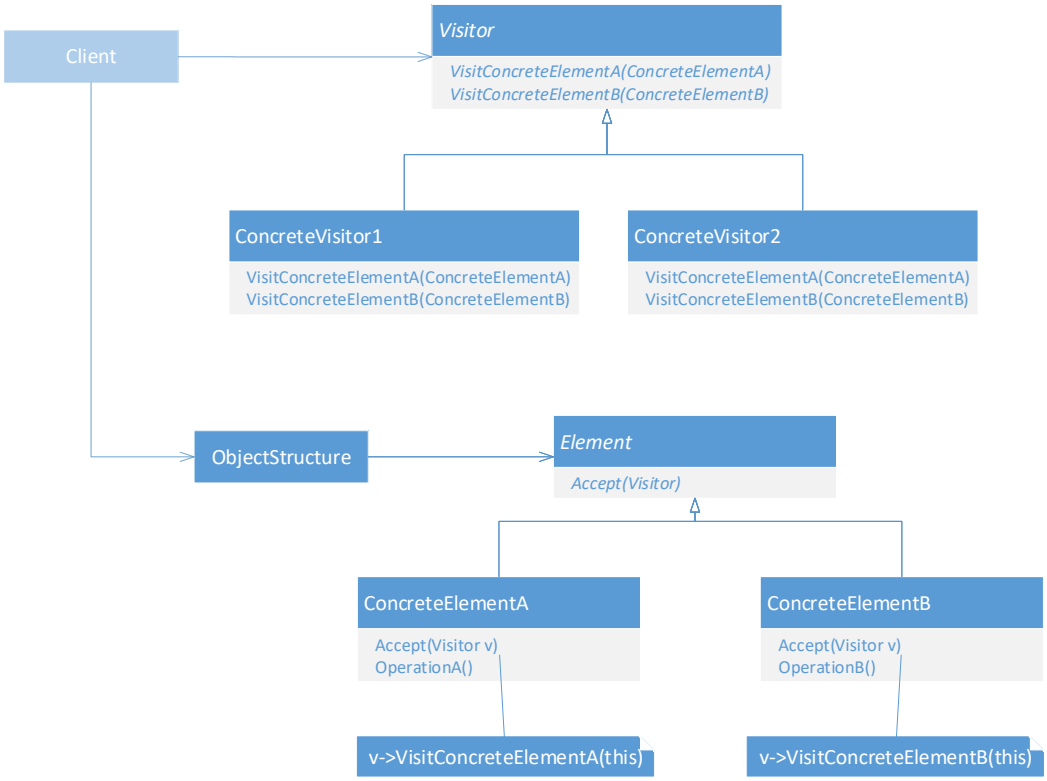

Abbildung 2.4: Struktur des Visitor-Entwurfsmusters (adaptiert von Gamma et al., 1995, S. 334)

Die Klasse *Visitor* deklariert Visit-Operationen für jede *ConcreteElement*-Klasse der Objektstruktur, die *ConcreteVisitor*-Klassen implementieren diese Operationen. *Element* deklariert eine `Accept`-Methode, die einen Visitor als Parameter nimmt. Diese wird wiederum von den einzelnen *ConcreteElements* implementiert.

Als Vorteile der Nutzung des Visitor werden unter anderem die einfache Hin-

zufügbarkeit neuer Operationen gelistet, sowie die zentrale Verwaltung verwandter Operationen (getrennt von unzusammenhängenden) und die Möglichkeit, mit Objekten aus nicht zusammengehörenden Klassenhierarchien zu arbeiten. Als Nachteil ergibt sich eine erschwerte Erweiterbarkeit der Klassen der konkreten Elemente, weil mit dem Hinzufügen neuer Elemente auch die Notwendigkeit der Implementierung von Besuchern für diese einhergeht.

2.4.2 Builder Pattern

Das Entwurfsmuster Builder (Erbauer) dient laut Gamma et al. (1995) dem Ziel, die Konstruktion eines komplexen Objektes von seiner Repräsentation zu trennen, sodass mit dem gleichen Konstruktionsprozess verschiedene Repräsentationen erstellt werden können. Es bietet sich dann an, wenn die Erstellung der Einzelteile eines komplexen Objektes unabhängig von der Erstellung des Objektes sein soll.

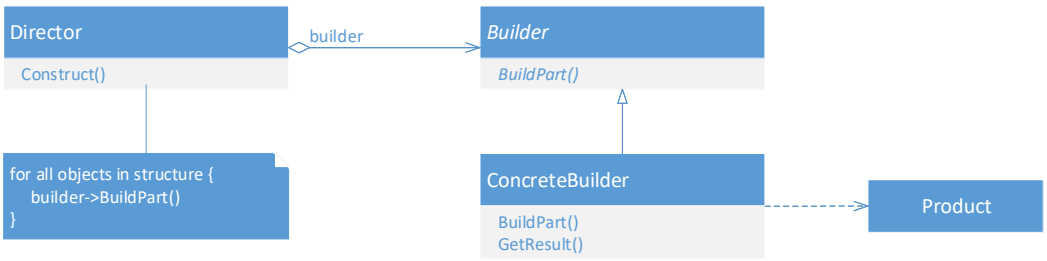

Abbildung 2.5: Struktur des Builder-Entwurfsmusters (adaptiert von Gamma et al., 1995, S. 98)

Der *Builder* spezifiziert ein abstraktes Interface zur Erstellung der Teile eines komplexen *Product*-Objektes, der *ConcreteBuilder* implementiert das Interface; er erstellt die Einzelteile des Objektes, verwaltet die von ihm erstellte Repräsentation und bietet ein Interface um das von ihm erstellte Objekt entgegenzunehmen. Der *Director* nutzt dieses Interface um ein Objekt zu konstruieren.

Die Konsequenzen der Verwendung des Entwurfsmusters sind die Möglichkeit, die interne Repräsentation des Objektes zu variieren, die Isolierung von Code zur Objekterstellung von dem zur Repräsentation des Objektes, sowie eine bessere Kontrolle über den Konstruktionsprozess.

3 Anforderungen

Im nachfolgenden Abschnitt werden die Ziele und Anforderungen beschrieben, welche die Markdown-Unterstützung für Sweble Hub erfüllen soll. Grundsätzlich gliedern sich diese in zwei logische Abschnitte, da letztendlich eine Transformation in zwei Richtungen erfolgen soll: Einerseits von Markdown nach WOM, andererseits von WOM zurück nach Markdown. Der erste Teil soll Aufgabe des Parsers sein, der zweite Teil soll vom Pretty Printer erfüllt werden. Beide sollten dabei in Java implementiert sein, um sich in die Codebasis des Sweble Hub einzufügen.

3.1 Anforderungen an den Parser

Die Anforderungen an den Parser lassen sich nach Recherche zum Aufbau existierender Parser in zwei Teile gliedern: Einerseits in den Parsing-Prozess, der aus einem Markdown-Dokument einen AST erstellt, andererseits in den Rendering-Prozess, der den AST in ein WOM-Dokument transformiert.

Das hauptsächliche Ziel von Parser und Renderer besteht in der Transformation von Markdown-Quelldateien zu einem inhaltlich gleichen WOM-Dokument. Dabei ist das WOM um Konstrukte, die zwar in Markdown existieren, im WOM jedoch nicht, zu erweitern. Zusätzlich soll nicht nur ein bedeutungsgleiches, sondern auch identisches Markdown-Dokument aus dem entstehenden WOM-Baum wiederhergestellt werden können. Dazu müssen im WOM-Dokument auch RTD-Knoten angelegt werden, mit deren Hilfe man das Originaldokument komplett nachvollziehen kann. Im Folgenden werden zusätzliche Ansprüche an Parser und Renderer getrennt gestellt.

Der Parser muss einige grundlegende Anforderungen auf jeden Fall erfüllen. Es ist notwendig, dass der Parser einen AST bereitstellt, auf dessen Basis das Ergebnis des Parsing-Prozesses weiterverarbeitet werden kann. Zusätzlich muss der AST für alle Elemente noch Informationen über die ursprüngliche Markdown-Repräsentation bereitstellen. Dies wird im weiteren Vorgang benötigt, um Round Trip Data generieren zu können. Der Parser sollte die CommonMark-Spezifikation

erfüllen, da diese Markdown besser standardisiert, wie in Abschnitt 2.3 beschrieben wurde. Das Vorhandensein von Erweiterungen oder zumindest die Möglichkeit der Erweiterbarkeit des Parsers wäre zwar vorteilhaft, aber nicht grundlegend notwendig.

Der Renderer soll die AST-Darstellung des geparsten Dokuments entgegennehmen und hat die Aufgabe, aus dieser ein WOM-Dokument zu erstellen. RTD-Knoten müssen mithilfe der Informationen aus dem AST angelegt werden.

Um für eine möglichst umfassende Menge an möglichen Markdown-Eingaben die korrekte Ausführung der Transformationen testen zu können, sollen die in der CommonMark-Spezifikation definierten Beispiele als Testfälle hergenommen werden.

3.2 Anforderungen an den Pretty Printer

Der Pretty Printer soll eine Rücktransformation eines WOM-Dokumentes nach Markdown ermöglichen. Im Gegensatz zur Wiederherstellung über die Inhalte von RTD-Knoten soll der Pretty Printer jedoch ein Markdown-Dokument auf Basis des Aufbaus eines WOM-Baumes erstellen. Eine Markdown-Quelldatei, welche zunächst mit dem Parser zu WOM konvertiert und danach mit dem Pretty Printer wieder in eine Markdown-Darstellung gebracht wird, ist danach möglicherweise nicht mehr identisch, bildet aber immer noch den gleichen WOM-Baum ab.

Der Pretty Printer sollte auch mit WOM-Elementen umgehen können, die keine entsprechende Markdown-Darstellung haben. In diesem Fall ist stattdessen HTML-Code zu generieren, der das entsprechende WOM-Konstrukt umsetzt.

Die korrekte Funktionalität des Pretty Printers ist wiederum durch eine möglichst umfassende Reihe an Testfällen sicherzustellen.

4 Architektur und Design

Das nachfolgende Kapitel beschreibt die Architektur- und Designentscheidungen aller benötigten Komponenten für die Markdown-Unterstützung der Sweble Hub Software, die im vorherigen Abschnitt mit ihren Anforderungen identifiziert wurden.

4.1 Parsersuche

Wie bereits in Abschnitt 2.3 angemerkt wurde, gibt es eine große Breite an schon existierenden Markdown-Parsern. Sollte sich darunter ein Parser befinden, der bereits die beschriebenen Anforderungen erfüllt, kann dieser anstelle einer kompletten Neuimplementierung als Basis verwendet werden. In Anbetracht der gestellten Kriterien bezüglich Features und Programmiersprache konnten zwei Parser als infrage kommend identifiziert werden, auf die im folgenden Abschnitt daher detaillierter eingegangen wird.

4.1.1 Atlassian CommonMark

commonmark-java[8] ist eine von Atlassian entwickelte Java-Bibliothek zum Parsen und Rendern von Markdown, veröffentlicht unter der 2-Klausel-BSD-Lizenz. Die Implementierung folgt der CommonMark-Spezifikation. Grundlegend ist die Benutzung in zwei Teile gegliedert: Der Parser erzeugt einen *AST*, während der Renderer diesen in eine HTML-Darstellung bringt. Beide Komponenten bieten Interfaces für Erweiterungen, ein komplettes Austauschen des HTML-Renderers durch eine alternative Implementierung zur Darstellung in einem anderen Format ist jedoch nicht vorgesehen. Ein Zugriff auf den AST ist zwar grundsätzlich möglich, allerdings bietet der AST nicht die Möglichkeit, eine komplette Rekonstruktion des Quelldokuments zu erlauben. Renderer und Parser sind durch diverse Optionen konfigurierbar.

[8] https://github.com/atlassian/commonmark-java

4.1.2 Flexmark

Flexmark[9] wurde von Vladimir Schneider als Fork von Atlassian CommonMark entwickelt. Die Bibliothek ist ebenfalls unter der 2-Klausel-BSD-Lizenz veröffentlicht. Hauptmotivation und Fokus für den Fork sind sowohl eine bessere Erweiterbarkeit, als auch die Möglichkeit, im AST noch komplett auf die zugrundeliegende Quelldatei zugreifen zu können und somit für jedes Element noch die genaue Markdown-Darstellung nachvollziehen zu können. Da Flexmark ursprünglich auf Atlassian CommonMark basiert, erfüllt es ebenfalls die CommonMark-Spezifikation, allerdings kann es auch das Verhalten anderer Markdown-Implementierungen emulieren.

4.1.3 Evaluation

Die folgende Tabelle bietet eine direkte Gegenüberstellung der für die Entscheidung relevanten Eigenschaften der beiden Parser-Bibliotheken.

	Flexmark	**Atlassian CommonMark**
Lizenz	BSD (2-clause)	BSD (2-clause)
Bietet AST an	ja	ja
Zugriff auf Quelldokument im AST	ja	nein
Unterstützter Markdown-Standard	CommonMark, Kramdown, Markdown.pl, MultiMarkdown, PegDown und mehr	CommonMark
Parser-Erweiterungen	29 mitgelieferte Erweiterungen (darunter alle, die commonmark-java auch anbietet); dazu Interfaces für eigene Erweiterungen	6 mitgelieferte Erweiterungen; dazu Interfaces für eigene Erweiterungen
Möglichkeit für eigene Renderer	Interfaces zur Neuimplementierung eigener Renderer; HTML-, DOCX- und PDF-Renderer dabei	nein, nur Erweiterung des vorhandenen HTML-Renderers vorgesehen
Abhängigkeiten	keine	keine
Relative Parse-Dauer (niedriger ist besser) (Schneider, 2018)	1x	0.6x bis 0.7x

Tabelle 4.1: Vergleich relevanter Eigenschaften von Flexmark und Atlassian CommonMark

[9] https://github.com/vsch/flexmark-java

Flexmark erfüllt im Gegensatz zu Atlassian CommonMark alle Anforderungen, die im vorherigen Kapitel aufgelistet wurden. Gerade die Nachvollziehbarkeit des Quelldokuments im AST ist ein signifikanter Vorteil, der als Grundlage für die weitere Implementierung wichtig ist. Round Trip Data könnte mit Atlassian CommonMark nicht zuverlässig konstruiert werden. Abgesehen von überschaubaren Performance-Einbußen ist Flexmark in allen Eigenschaften mindestens gleichwertig oder besser einzuordnen als Atlassian CommonMark. Daher wird im Folgenden Flexmark als Basis für die Sweble-Markdown-Unterstützung verwendet.

4.2 Aufbau von Flexmark

Wie im vorherigen Abschnitt bereits kurz beschrieben wurde, lässt sich der grundlegende Arbeitsablauf von Flexmark in zwei Abschnitte unterteilen: Zunächst bringt der Parser das Quelldokument in eine AST-Darstellung, danach nimmt ein Renderer diesen AST entgegen und erstellt daraus eine Zieldarstellung. In Flexmark direkt integriert sind unter anderem Renderer, die HTML-, PDF- sowie DOCX-Dokumente erstellen können.

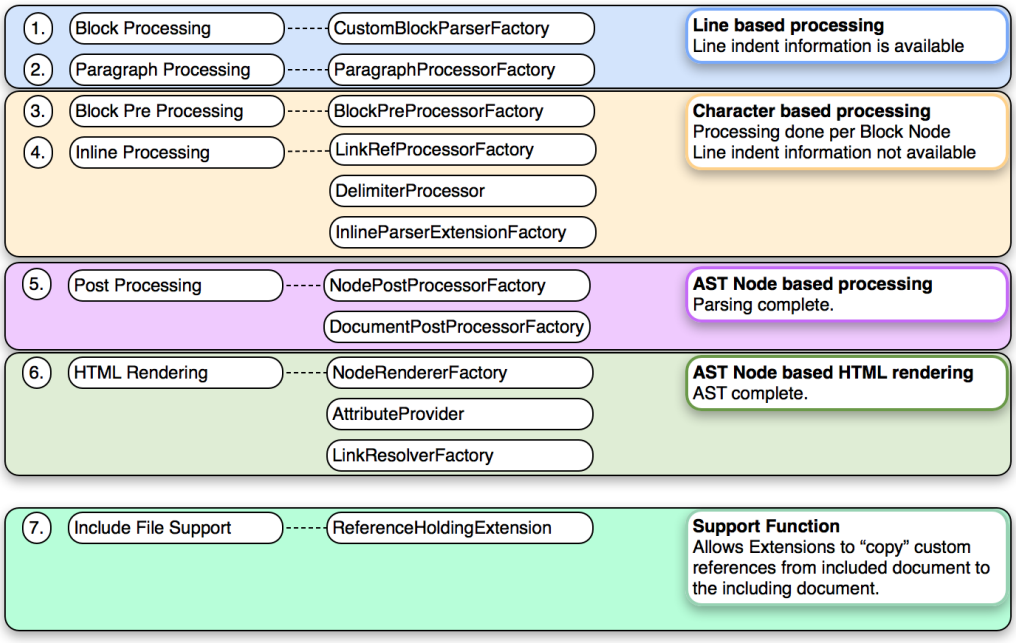

Abbildung 4.1: Parse- und Renderschritte von Flexmark (Schneider, 2017)

In Abbildung 4.1 ist eine Zusammenfassung der wichtigen Einzelschritte des Parse- und Rendervorgangs von Flexmark abgebildet. Die CommonMark-Spezifikation

schlägt eine Strategie für den Parsingprozess vor, die vom Flexmark-Parser angewandt wird. Im erstem Schritt des Parsing-Vorgangs wird das Quelldokument zeilenweise eingelesen und der Parser prüft, ob der Inhalt der aktuellen Zeile gegebene Bedingungen für den Start oder das Ende eines Blockknotens erfüllt. Alle Zeilen, die keine Bedingungen eines anderen Blocktypen erfüllen, werden als Paragraph behandelt. So wird schrittweise die Blockstruktur des Dokuments erstellt. Im zweiten Schritt können Paragraph-Blöcke zeilenweise gelesen und bearbeitet werden, um diese Zeilen noch anderen Blockknotentypen zuzuweisen. Der dritte Schritt erlaubt Erweiterungen, den AST noch vor dem Inline-Parsing weiter zu bearbeiten (MacFarlane, 2017).

Der vierte Schritt entspricht dem zweiten Teil des in der Spezifikation beschriebenen Prozesses. Die Inhalte jedes Blockes werden zeichenweise geparst, um Inline-Konstrukte zu identifizieren. Erweiterungen können hier entweder als *DelimiterProcessor* oder als *LinkRefProcessor* eigene Konstrukte anmelden. Nach diesem Schritt ist das grundlegende Parsen abgeschlossen, es können jedoch mit PostProcessoren - die entweder den ganzen AST, oder nur bestimmte Knotentypen durchlaufen - noch Veränderungen vorgenommen werden. Renderer nehmen diesen fertigen AST als Basis und erstellen daraus die entsprechende Zieldarstellung (MacFarlane, 2017).

Auf der Basis der von Flexmark gegebenen Interfaces wird ein neuer Renderer implementiert, der aus dem AST ein WOM-Dokument erstellt. Auf den genaueren Aufbau, den alle Renderer von Flexmark erfüllen, wird in Abschnitt 4.5, in dem die Architektur dieses WOM-Renderers beschrieben wird, genauer eingegangen.

4.3 Erweiterungen von Flexmark

Flexmark bietet eine große Zahl an Parser-Erweiterungen, die optional aktiviert werden können. Mithilfe dieser kann man Markdown um einige Konstrukte erweitern, die im WOM unterstützt sind, jedoch nicht in der CommonMark-Spezifikation. Sind diese Erweiterungen aktiviert, finden sich im AST von Flexmark neue Knotentypen wieder, deren Unterstützung vom Renderer implementiert sein muss, damit sie korrekt behandelt werden können. Die folgende Liste beschreibt eine Auswahl an Erweiterungen, deren Unterstützung durch den Renderer als sinnvoll erscheint, weil sie eine Syntax für Konzepte hinzufügen, die entweder durch Wikitext und WOM, oder durch HTML unterstützt werden:

- *AsideExtension*: Führt eine Syntax für HTML `<aside>`-Blöcke ein; Funktioniert analog zu *block quotes*, nur mit `|` anstatt `<`
- *AutolinkExtension*: Findet blanke URLs und E-Mails in Dokumenten und wandelt sie automatisch in Links um

- *DefinitionExtension*: Unterstützung für Definition Lists (`<dl>, <dt>, <dd>`)
- *InsExtension*: Unterstützung für `<ins>`; mit der Syntax `++Beispiel++`
- *StrikethroughSubscriptExtension*: Durchgestrichener Text mit der Syntax `~~Beispiel~~`, sowie tiefergestellter Text mit `~Beispiel~`
- *SuperscriptExtension*: Hochgestellter Text mit `^Beispiel^`
- *TablesExtension*: Unterstützung für Tabellen, der Syntax von GitHub Flavored Markdown folgend
- *WikiLinkExtension*: Unterstützung für Wiki Links anhand der Syntax `[[Beispiel]]`

4.4 Parser-Erweiterung

Bevor der AST an den Renderer weitergegeben wird, müssen noch leichte Transformationen vorgenommen werden, um die Arbeit des Renderers zu erleichtern. Dazu ist mit der *WomPostProcessingExtension* eine Parser-Erweiterung konzipiert worden. Diese meldet mit dem *WomPostProcessor* einen *DocumentPostProcessor* (siehe Schritt 5 in Abbildung 4.1) beim Parser an.

4.5 Aufbau des Renderers

Im folgenden Abschnitt wird auf die Architektur des *WomRenderer*s eingegangen. Flexmark bietet grundlegende Interfaces an, die von Renderern zu implementieren sind. Zusätzlich folgen die mit Flexmark mitgelieferten Renderer bestimmten Strukturen zur Ermöglichung von Erweiterbarkeit, indem sie wiederum neue Interfaces dafür definieren. Dies ist zum Beispiel dann hilfreich, wenn Parser-Erweiterungen neue AST-Knotentypen einführen. Diese können dann die Renderer leicht um Methoden für diese Knotentypen erweitern. Diesen Strukturen wird in der Architektur des *WomRenderer*s auch gefolgt, um die Erweiterbarkeit sicherzustellen.

Die Klasse *WomRenderer* ist der Einstiegspunkt. Sie implementiert das *IRender*-Interface von Flexmark und bietet diesem folgend die entsprechenden `render`-Methoden, mit denen der Rendering-Prozess aufgerufen werden kann, wie in Abbildung 4.2 zu sehen ist. Das Builder-Entwurfsmuster wird verwendet, um neue *WomRenderer*-Instanzen zu erstellen.

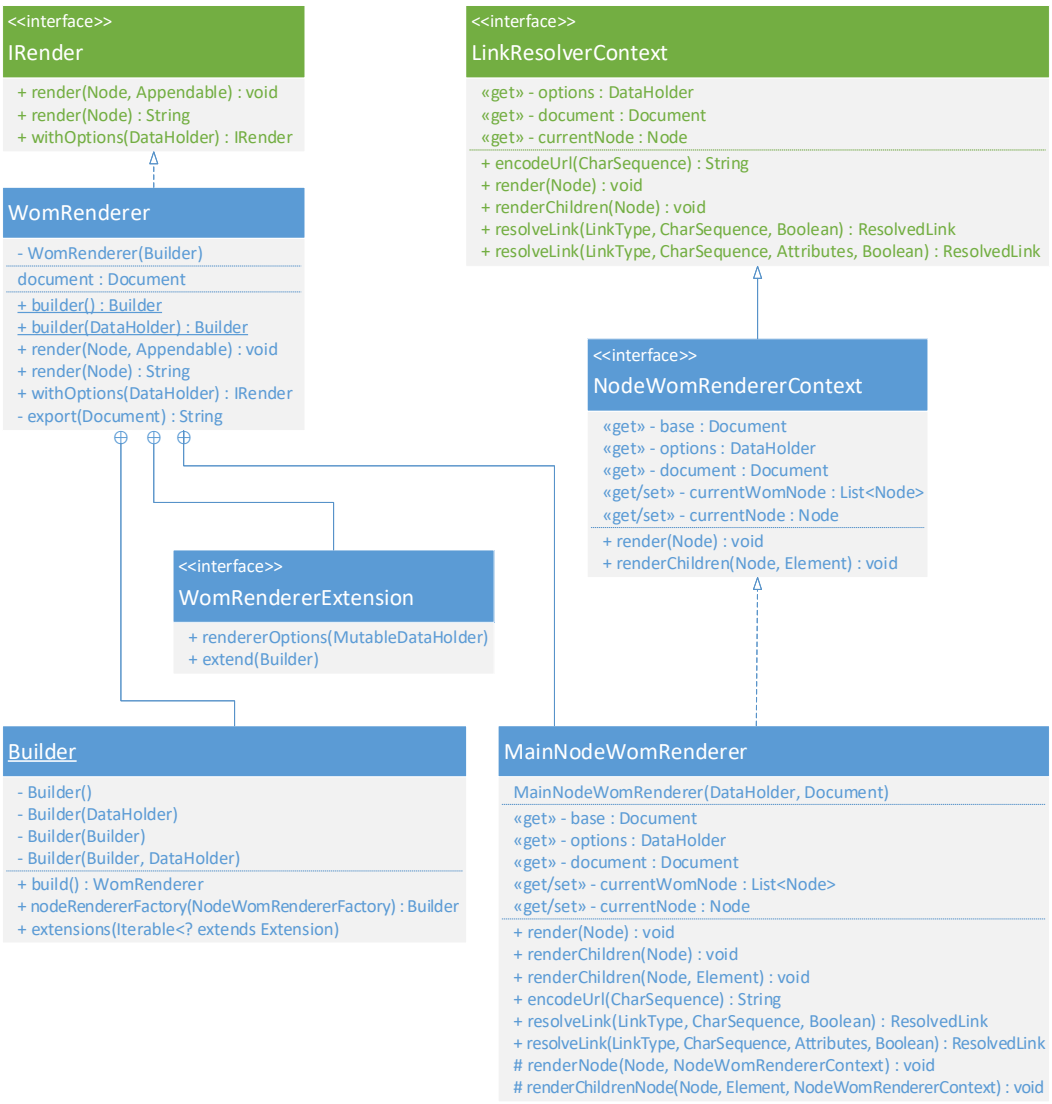

Abbildung 4.2: UML-Klassendiagramm des WomRenderer und zugehöriger Klassen

Grundlegend muss für jeden Knotentyp, mit dem der Renderer umgehen kann, ein *NodeWomRendererHandler* angemeldet werden. Dieser implementiert das *CustomWomRenderer*-Interface und ordnet einem spezifischen Knotentyp eine *render*-Methode zu. Der Aufbau wird in Abbildung 4.3 gezeigt.

Das Interface *NodeWomRenderer* dient als Basis für Klassen, die eine beliebige Anzahl an *NodeWomRendererHandler*n halten. Die Klasse *CoreWomRenderer* implementiert dieses Interface und enthält *NodeWomRendererHandler* für alle grundlegenden Knotentypen, sowie für einige Erweiterungen.

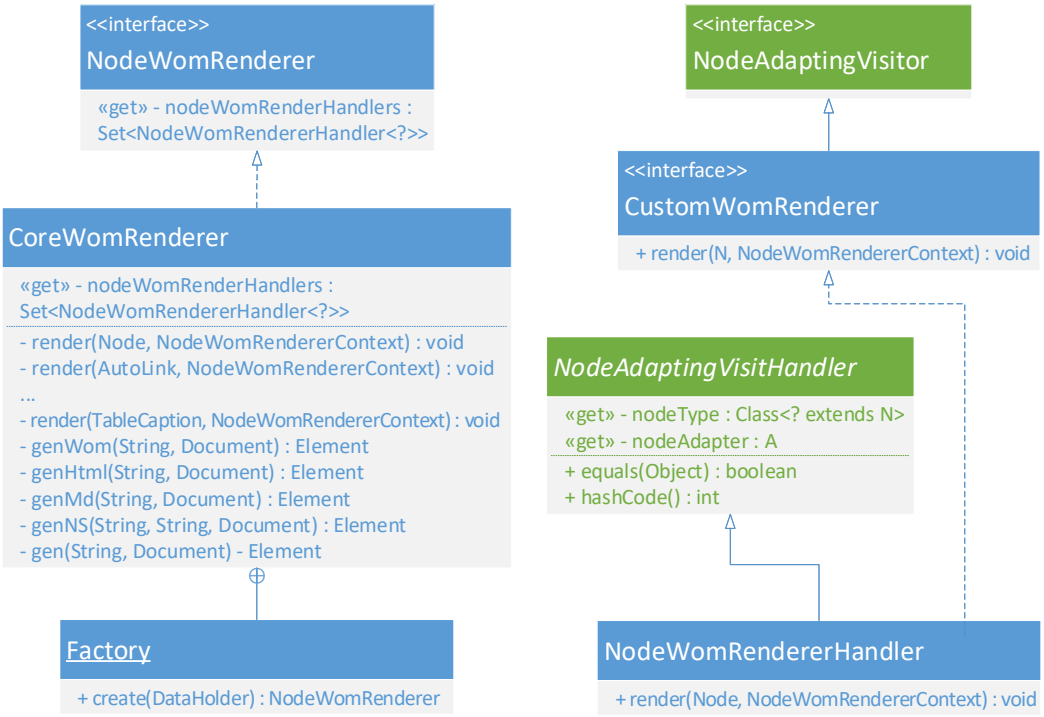

Abbildung 4.3: Restliche Klassen des Renderer-Packages

Der *MainNodeWomRenderer* bietet essentielle Methoden für das Durchlaufen des ASTs an. Mit `renderNode` wird für ein spezifisches Element je nach Art der richtige *NodeWomRendererHandler* aufgerufen, während mit `renderChildrenNode` die entsprechenden Aufrufe für alle Kinder eines Elementes gestartet werden. Mit Hilfe dieser Methoden kann der gesamte AST dem Visitor-Entwurfsmuster nach durchlaufen werden. Darüber hinaus sind mehrere Hilfsmethoden zu finden, wie z.B. `resolveLink`, welches einem *ReferenceLink* oder einer *ImageReference* die zugrundeliegende *ReferenceDefinition* zuordnet.

Analog zu den anderen Renderern von Flexmark bietet das Interface *WomRendererExtension* die Möglichkeit, dass Erweiterungen zum Renderer hinzugefügt werden können, indem diese entweder über `extend` neue *NodeWomRenderer* beim *Builder* anmelden, oder mit `renderOptions` die Optionen des Renderers auslesen oder bearbeiten.

4.6 Aufbau des Pretty Printers

Der Pretty Printer hat die Aufgabe, aus einem WOM-Dokument als Eingabe wieder ein Markdown-Dokument zu erstellen. Er besteht aus zwei Klassen: *PrettyPrinter* und *PrettyPrinterConfig*.

Über *PrettyPrinterConfig* können Konfigurationseinstellungen für den Pretty Printer vorgenommen werden, um Details des auszugebenden Markdowns zu beeinflussen. Eine solche Konfiguration kann dann dem Konstruktor des Pretty Printers übergeben werden.

Die Transformation eines WOM-Dokumentes kann über die `parse`-Methoden des *PrettyPrinter* aufgerufen werden. Dabei kann das Dokument entweder als `String`, oder als `org.w3c.dom.Document` übergeben werden. Auch hier wird die Eingabe nach dem Visitor-Entwurfsmuster durchlaufen. Zusätzlich existiert noch der *UnnecessaryTextNodeRemover*, welcher das Dokument vor der Transformation durchläuft, um Zeilenumbrüche und Einrückungen des Dokumentes zu entfernen, da diese für die weitere Verarbeitung im *PrettyPrinter* unnötig sind.

5 Implementierung

Das folgende Kapitel geht auf Implementierungsdetails der einzelnen Komponenten ein, die im vorherigen Kapitel beschrieben wurden.

5.1 Renderer

Eine Übersicht über die einzelnen Klassen des Renderers und deren Zusammenhänge wurde in Abschnitt 4.5 beschrieben. Hier wird nun auf Details zur Implementierung und Benutzung des Renderers eingegangen. Auflistung 5.1 zeigt zunächst beispielhaft, wie der Flexmark-Parser und der *WomRenderer* benutzt werden können, um einen Markdown-String in eine WOM-Darstellung zu bringen. Zunächst werden die Optionen konfiguriert, wobei im gezeigten Beispiel auch die vom Renderer unterstützten Erweiterungen aktiviert werden. Der Parser gibt den fertig geparsten AST zurück, welcher dann an den Renderer übergeben wird. Dieser gibt dann das fertige WOM-Dokument als String zurück.

```
String source = "[markdown-dokument]";
DataHolder options = new MutableDataSet()
          .set(Parser.EXTENSIONS, Arrays.asList(DefinitionExtension.create(),
          AsideExtension.create(),
          AutolinkExtension.create(),
          StrikethroughSubscriptExtension.create(),
          SuperscriptExtension.create(),
          InsExtension.create(),
          TablesExtension.create(),
          WikiLinkExtension.create(),
          WomPostProcessingExtension.create()));
Parser parser = Parser.builder(options).build();
WomRenderer renderer = WomRenderer.builder(options).build();
Document doc = parser.parse(source);
String wom = renderer.render(doc);
```

Auflistung 5.1: Grundlegende Benutzung von Parser und Renderer

5.1.1 WomRenderer

Wie in Auflistung 5.1 zu sehen ist, kann eine neue Instanz des *WomRenderer* über `builder(options).build()` erstellt werden. Diese Instanz bekommt vom *Builder* die an diesen übergebenen Optionen. Sollten Erweiterungen konfiguriert worden sein, übergibt der Builder außerdem jeweils eine Liste an *NodeWomRendererFactories*, sowie eine Liste an *LinkResolverFactories*, die von den Erweiterungen beim *Builder* angemeldet werden. Zusätzlich wird im Konstruktor des *WomRenderer* noch eine Factory des *CoreWomRenderer* angemeldet.

Der tatsächliche Render-Vorgang wird mit den `render`-Methoden eingeleitet. Es wird eine *MainNodeWomRenderer*-Instanz erstellt, die den AST durchläuft und danach über `renderer.getBase()` den fertigen WOM-Baum als `org.w3c.dom.Document` bereitstellt.

```
1 public void render(Node node, Appendable output) {
2    MainNodeWomRenderer renderer = new MainNodeWomRenderer(options,
      node.getDocument());
3    renderer.render(node);
4    document = renderer.getBase();
5    output.append(export(document));
6 }
```

Auflistung 5.2: `render`-Methode

Neben der in Auflistung 5.2 gezeigten Methode existiert noch eine weitere `render`-Variante, welche kein `Appendable` übergeben bekommt und das Ergebnis stattdessen als `String` zurückgibt.

Die `export`-Methode transformiert das Dokument mit Hilfe des XSLT-Prozessors Xalan in `String`-Form. Auflistung 5.3 zeigt die Konfigurationsoptionen, welche bei der Transformierung verwendet werden.

```
1 Transformer transformer = transformerFactory.newTransformer();
2 transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
3 transformer.setOutputProperty(OutputKeys.INDENT, "yes");
4 transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
5 transformer.setOutputProperty("{http://xml.apache.org/xalan}line-separator", "\n");
6 DOMSource source = new DOMSource(base);
7 StreamResult result = new StreamResult(new StringWriter());
8 transformer.transform(source, result);
9 return result.getWriter().toString();
```

Auflistung 5.3: `export`-Methode (Ausschnitt)

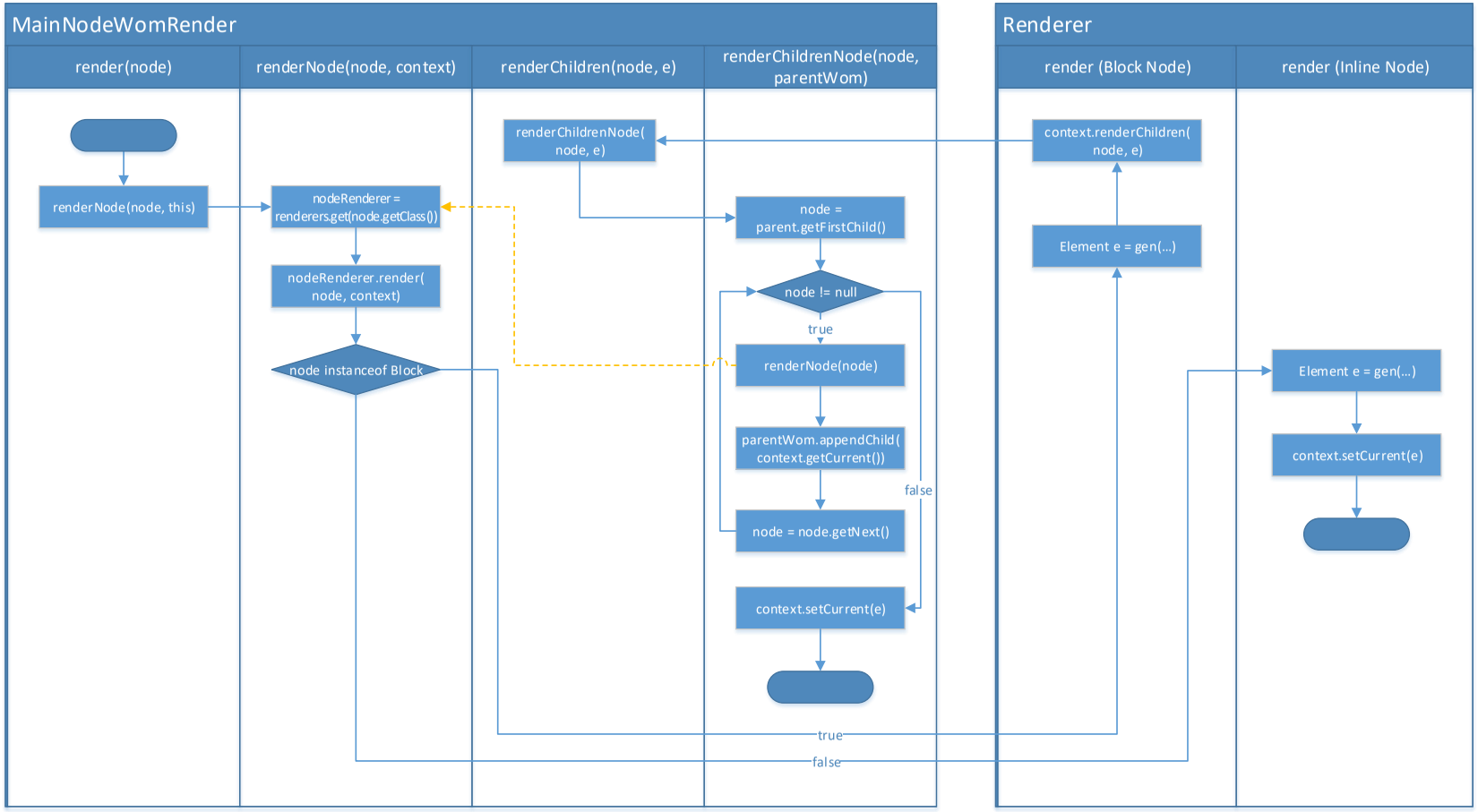

Abbildung 5.1: Ablaufdiagramm des Rendervorgangs

5.1.2 MainNodeWomRenderer

Der Konstruktor des *MainNodeWomRenderer* durchläuft die Liste an *NodeWomRendererFactories* des *WomRenderer*. Dabei wird bei jeder Factory ein *NodeWomRenderer* instanziiert und die Liste dessen *NodeWomRenderHandler* entgegengenommen. Diese werden dann der Liste an Renderern hinzugefügt. Analog wird bei der Liste an *LinkResolverFactories* verfahren.

```
1  for (int i = nodeWomRendererFactories.size() - 1; i >= 0; i--) {
2     NodeWomRendererFactory nodeWomRendererFactory =
       nodeWomRendererFactories.get(i);
3     NodeWomRenderer nodeWomRenderer =
       nodeWomRendererFactory.create(this.getOptions());
4     final Set<NodeWomRendererHandler<?>> rendererHandlers =
       nodeWomRenderer.getNodeWomRendererHandlers();
5     if (rendererHandlers == null) continue;
6
7     for (NodeWomRendererHandler nodeType : rendererHandlers) {
8        renderers.put(nodeType.getNodeType(), nodeType);
9     }
10 }
```

Auflistung 5.4: Ausschnitt aus dem Konstruktor des *MainNodeWomRenderer*

Der Ablauf des Rendervorgangs ausgehend vom *MainNodeWomRenderer* ist in Abbildung 5.1 dargestellt. Genauere Beschreibungen der einzelnen Methoden sind in den folgenden Abschnitten zu finden.

renderNode

Wie in Auflistung 5.2 gezeigt wurde, wird der Render-Vorgang mit einem Aufruf der `render`-Methode mit dem Basisknoten des AST - dem `Document` - als Parameter, gestartet. Diese Methode ruft wiederum `renderNode` auf, welche in der folgenden Auflistung gekürzt zu sehen ist.

Die Methode sucht in der Liste der angemeldeten Renderer nach einem passenden für den jeweiligen Knotentyp und weicht im Zweifelsfall auf den generischen Renderer aus und der Knoten wird gerendert.

```
protected void renderNode(Node node, NodeWomRendererContext context) {
   if (node instanceof Document) {
      ...
   } else {
      NodeWomRendererHandler nodeRenderer = renderers.get(node.getClass());

      if (nodeRenderer == null) {
         nodeRenderer = renderers.get(Node.class);
      }

      Node oldNode = this.renderingNode;
      context.setCurrentNode(node);
      nodeRenderer.render(node, context);
      context.setCurrentNode(oldNode);
   }
}
```

Auflistung 5.5: `renderNode` (gekürzt)

renderChildrenNode

Die Methode `renderChildrenNode` wird von einzelnen Block-Knoten-Renderern aufgerufen, um das Durchlaufen des AST fortzusetzen. Wie der Name dabei impliziert, ist eine Aufgabe der Methode, die Kinder des aufrufenden Elementes zu durchlaufen und für jedes dieser `renderNode` aufzurufen. Außerdem werden die von den Kindern konstruierten WOM-Elemente dem WOM-Element des Eltern-Knotens angehängt. Zusätzlich werden, falls notwendig, RTD-Knoten zwischen den einzelnen Kindern konstruiert.

```
protected void renderChildrenNode(Node parent, Element parentWom,
    NodeWomRendererContext context) {
   Node node = parent.getFirstChild();
   while (node != null) {
      renderNode(node, context);
      for (org.w3c.dom.Node child : current) {
         parentWom.appendChild(child);
      }
      node = node.getNext();
   }
   context.setCurrent(Collections.singletonList(parentWom));
}
```

Auflistung 5.6: `renderChildrenNode` (gekürzt)

5.1.3 CoreWomRenderer

Im *CoreWomRenderer* sind alle grundlegenden Implementierungen für einzelne Knotentypen des AST zu finden. Über `getNodeWomRenderHandlers` (Auflistung 5.7) wird ein *HashSet* zurückgegeben, welches die Methoden den einzelnen Klassen der Objekte im AST zuordnet. Alle diese Klassen erweitern die generische *Node*-Klasse.

```
1 public Set<NodeWomRendererHandler<?>> getNodeWomRendererHandlers() {
2    return new HashSet<>(Arrays.asList(
3       new NodeWomRendererHandler<>(Node.class, CoreWomRenderer.this::render),
4       new NodeWomRendererHandler<>(AutoLink.class,
     CoreWomRenderer.this::render),
5       ...
6       new NodeWomRendererHandler<>(TableCaption.class,
     CoreWomRenderer.this::render)
7    ));
8 }
```

Auflistung 5.7: `getNodeWomRenderHandlers` (gekürzt)

Insgesamt sind Methoden für 31 Knotentypen aus der Basisspezifikation zu finden. Außerdem sind Implementierungen für 16 Knotentypen von Parsererweiterungen vorhanden, sowie eine generische Methode, die bei unbekannten Knotentypen als Rückfallebene aufgerufen wird. Alle Methoden bekommen zwei Argumente übergeben: einerseits das Knoten-Objekt selbst, auf Basis dessen die `render`-Methode aufgerufen wird, andererseits die Instanz des aufrufenden *MainNodeWomRenderer*, die im Folgenden als Kontext bezeichnet wird.

```
1  private void render(Node node, NodeWomRendererContext context) {
2     if (node instanceof Block) {
3        Element e = genHtml("html:p");
4        context.renderChildren(node, e);
5     } else {
6        Element e = genWom("text");
7        e.setTextContent(node.getChars().toString());
8        context.setCurrent(Collections.singletonList(e));
9     }
10 }
```

Auflistung 5.8: Generische `render`-Methode (gekürzt)

Die grundlegenden Anforderungen an die Implementierungen einzelner Knoten-Renderer, je nach dem ob sie Block-Knoten oder Inline-Knoten sind, können aus der generischen `render`-Methode für unbekannte Knotentypen in Auflistung 5.8 abgeleitet werden:

- Block-Knoten: Erstellung des WOM-Elementes, welches das entsprechende AST-Element repräsentiert und danach der Aufruf von `renderChildren`, um das Ablaufen des AST fortzusetzen
- Inline-Knoten: Erstellung des WOM-Elementes; dieses wird dann mit `setCurrent` als aktuelles Element im Kontext festgelegt, damit der Elternknoten Zugriff auf das erstellte Element bekommt

Alles hier genannte zur Struktur der Klasse gilt analog für alle Renderer-Erweiterungen, die ebenfalls das *NodeWomRenderer*-Interface implementieren, um eigene Knoten-Renderer anzubieten.

HTML-Knoten

Die Transformation von HTML stellt eine besondere Schwierigkeit beim Render-Vorgang dar. CommonMark erlaubt die Benutzung von HTML in Quelldokumenten, welches bei der Darstellung auch als solches behandelt werden soll. Dabei sind Regeln für das Parsen von HTML-Blöcken sowie Inline-HTML definiert. Allerdings wird durch diese Regeln keine Validität des HTML sichergestellt. Auflistung 5.9 zeigt ein Dokument mit einen HTML-Block, der zwei Tags enthält, die geöffnet, aber nicht mehr geschlossen werden. Außerdem ist einer der Tags, zumindest nach XHTML und HTML5, kein gültiger HTML-Tag (Danilo, Leithead, Faulkner, Eicholz und Moon, 2017, §8.1.2). Neben diesen Problemen können außerdem auch ungültige Schachtelungen auftauchen.

```
1 <unknowntag>
2 <b>
```

Auflistung 5.9: HTML-Beispiel

Grundsätzlich soll das HTML aus HTML-Blöcken und Inlines in den WOM übernommen werden. Dazu muss es allerdings valide sein. Mit diesem Ziel werden die Inhalte solcher Elemente daher mit JSoup geparst. JSoup[10] ist eine Java-Bibliothek zum Parsen von HTML, die auch mit invalidem HTML umgehen kann und dieses auch gegebenenfalls repariert.

Der detaillierte Vorgang für HTML-Blöcke ist in Auflistung 5.10 zu sehen. Zunächst wird als Basis ein `rawhtml`-Element erstellt. Der Inhalt des Blocks wird einerseits unverändert in einen RTD-Knoten geschrieben und dann mit JSoup in ein leeres Dokument geparst. Das Parsing-Ergebnis wird dann in ein `dom`-Element importiert, welches - genauso wie der RTD-Knoten - als Kind an das `rawhtml`-Element angehängt wird.

[10] https://jsoup.org/

```
private Element renderHtml(String html, NodeWomRendererContext context) {
   Element e = genWom("rawhtml", context.getBase());

   Element rtd = genWom("rtd", context.getBase());
   rtd.setTextContent(html);
   e.appendChild(rtd);

   org.jsoup.nodes.Document doc = Jsoup.parseBodyFragment(html);
   org.w3c.dom.Document docw3c = new W3CDom().fromJsoup(doc);
   Element dom = gen("dom", context.getBase());

   NodeList children = docw3c.getFirstChild().getLastChild().getChildNodes();
   if (children != null) {
      for (int i = 0; i < children.getLength(); i++) {
         org.w3c.dom.Node imported =
    context.getBase().importNode(children.item(i), true);
         dom.appendChild(imported);
      }
   }
   e.appendChild(dom);

   return e;
}
```

Auflistung 5.10: `render`-Methode für HTML-Blöcke

5.2 Parser-Erweiterung

Die *WomPostProcessingExtension* meldet den *WomPostProcessor* beim Parser an. Dieser ist ein Visitor, der den vorläufigen AST durchläuft und bei zwei Knotentypen Veränderungen vornimmt:

- `TextBase`: Diese Knoten sind reine Container ohne eine vorgesehene Darstellung im gerenderten Dokument. Sie sind letztendlich nur hinderlich beim Render-Vorgang und werden daher entfernt und die enthaltenen Kinderknoten werden stattdessen an den Elternknoten der `TextBase` an dieser Stelle angehängt.
- `Paragraph`: Der Parser erstellt bei allen Inhalten von List Items in jedem Fall `Paragraph`-Knoten als Container. Die CommonMark-Spezifikation legt jedoch spezifische Regeln fest, in welchen Fällen Inhalte von List Items in einem `Paragraph` landen und in welchen diese Inhalte direkt im List Item darszustellen sind. Der Flexmark-Parser überlasst diese Entscheidung dem Renderer. Da es einfacher ist, dies nicht während dem Render-Vorgang durchzuführen, sondern bereits davor, geschieht es hier. Analog zum Entfernen der `TextBase`s werden `Paragraph`-Knoten, die Kinder eines List Items

sind und nicht gerendert werden sollen, gelöscht und deren Inhalte an den Elternknoten angehängt.

5.3 Pretty Printer

Im folgenden Abschnitt wird auf Details der Implementierung und Konfiguration des Pretty Printers eingegangen. In Abbildung 5.2 ist eine Übersicht über den Ablauf des Pretty Printer zu sehen.

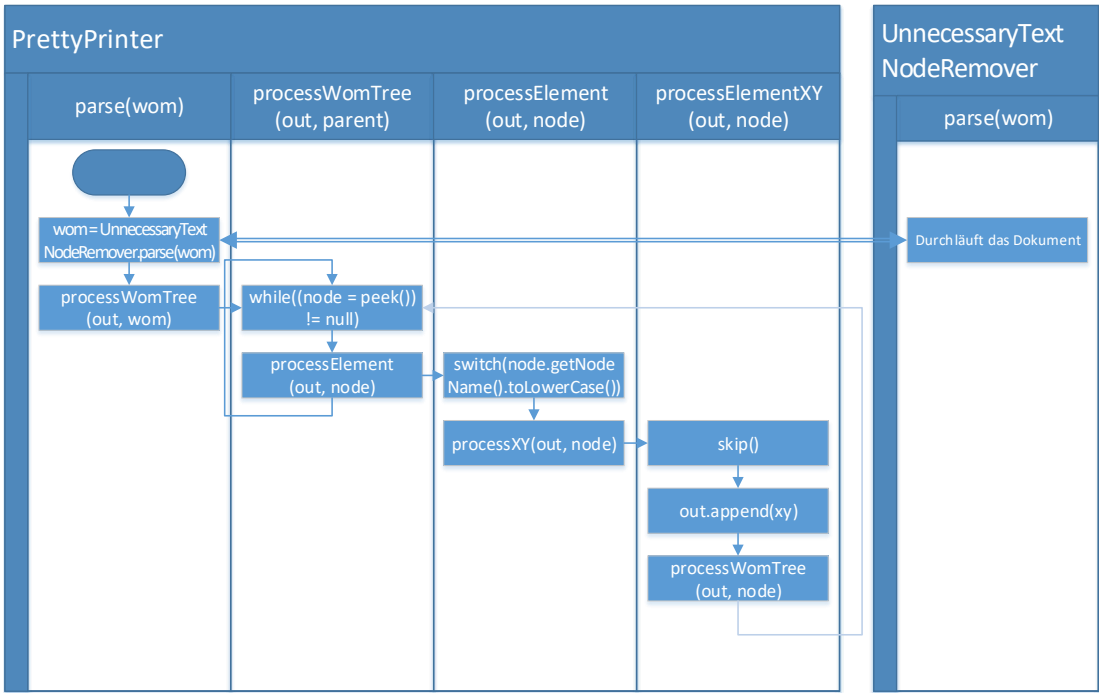

Abbildung 5.2: Ablaufdiagramm des Pretty Printers

Die grundlegende Benutzung ist simpel: der Pretty Printer kann entweder mit einem `PrettyPrinterConfig`-Objekt (mit dem sich Formatierungsoptionen festlegen lassen), oder ohne Parameter, instanziiert werden. Die Transformation kann daraufhin mit den `parse`-Methoden gestartet werden. Dabei gibt es wiederum zwei Varianten: Entweder wird das zu transformierende WOM-Dokument als `org.w3c.dom.Document` übergeben, oder als `String`.

Mit der Methode `processWomTree` erfolgt der Schritt in die nächste Ebene des Dokuments für einen übergebenen Knoten. RTD-Knoten werden dabei übersprungen. Diese Methode wird einerseits von `parse` aufgerufen, um das Ablaufen des Baumes zu starten, andererseits ruft jedes Element, das Kinderknoten haben kann,

wiederum diese Methode auf um das Ablaufen fortzusetzen. Der Ablauf ist in Auflistung 5.11 zu sehen.

```
1  private void processWomTree(FormattingAppendable out, Node parent) {
2     pushState(parent);
3     Node node;
4     while ((node = peek()) != null) {
5        if (!ignore.contains(node.getNodeName())) {
6           processElement(out, node);
7        } else {
8           skip();
9        }
10    }
11    popState();
12 }
```

Auflistung 5.11: `processWomTree`-Methode (gekürzt)

In `processElement` wird dem zu verarbeitenden Knoten die passende spezifische Methode zugewiesen. Wie in Auflistung 5.12 zu sehen ist, wird dabei mit `node.getNodeName()` der Typ des Knotens festgestellt und mit Hilfe einer switch-case-Struktur die richtige Verarbeitungsmethode aufgerufen.

```
1  private void processElement(FormattingAppendable out, Node node) {
2     switch(node.getNodeName().toLowerCase()) {
3        case "article" : processArticle(out, (Element) node); break;
4        case "body"    : processBody(out, (Element) node); break;
5        case "text"    : processText(out, node); break;
6        ...
7        default        : processUnknown(out, node); break;
8     }
9  }
```

Auflistung 5.12: `processElement`-Methode (gekürzt)

Die einzelnen Methoden zur Verarbeitung einzelner Knotentypen haben einerseits die Verantwortung, für das entsprechende Element eine Markdown-Darstellung zu erstellen und diese an `out` weiterzugeben, andererseits müssen sie die entsprechenden Hilfsmethoden aufrufen, um das Durchlaufen des Baumes fortzusetzen. Die folgenden Methoden und Klassen stehen dabei zur Verfügung:

- *State*-Klasse: ein Status, der einen Elternknoten, eine Liste dessen Kinderknoten, sowie einen Index enthält. Der Index gibt an, bei welchem Kinderknoten sich die Traversierung gerade befindet.
- `pushState`: muss aufgerufen werden, wenn in Kinderelemente eines Knotens gesprungen wird. Der aufrufende Elternknoten wird in den Status geschrieben, der Index wird auf 0 gesetzt.

- `popState`: muss wiederum aufgerufen werden, wenn die Traversierung der Kinderelemente abgeschlossen ist.
- `peek`: Gibt den Kinderknoten zurück, der nach dem aktuellen Index folgt (falls vorhanden); alternativ mit Parameter, um noch weiter voraus zu schauen.
- `skip`: Inkrementiert den Index; alternativ mit Parameter, um den Index um eine beliebige Zahl zu erhöhen.
- `next`: Wie `skip`, nur dass der Kinderknoten am neuen Index zurückgegeben wird.

Die spezifischen Verarbeitungsmethoden müssen dabei zumindest `next` oder `skip` aufrufen, um zu signalisieren, dass sie das Element verarbeitet haben. Zusätzlich muss, falls das Element Kinder haben kann, wiederum `processWomTree` aufgerufen werden, um die Kinder zu verarbeiten.

5.4 Beispiel

Im folgenden Abschnitt wird beispielhaft ein Dokument in den verschiedenen Start-, Zwischen- und Enddarstellungen gezeigt. Als Basis dient das Markdown-Dokument in Auflistung 5.13. Auflistung 5.14 zeigt die Struktur des AST, der vom Flexmark-Parser aus dem Dokument erstellt wird. In Auflistung 5.15 ist die vom Renderer generierte WOM-Darstellung zu sehen.

```
1 **Wichtig**
2
3 ---
4
5 > Zitat
```

Auflistung 5.13: Beispiel (Markdown)

Wie Auflistung 5.16 zeigt, kann aus der WOM-Repräsentation mit einer Hilfsmethode über die RTD-Knoten das Originaldokument wiederhergestellt werden. Alternativ kann man diese auch mit dem Pretty Printer durchführen, welcher statt der RTD-Knoten den Aufbau des Baumes betrachtet.

```
Document[0, 25]
  Paragraph[0, 12] isTrailingBlankLine
    StrongEmphasis[0, 11] textOpen:[0, 2, "**"] text:[2, 9, "Wichtig"]
    textClose:[9, 11, "**"]
      Text[2, 9] chars:[2, 9, "Wichtig"]
  ThematicBreak[13, 16]
  BlockQuote[18, 25] marker:[18, 19, ">"]
    Paragraph[20, 25]
      Text[20, 25] chars:[20, 25, "Zitat"]
```

Auflistung 5.14: Beispiel (AST-Aufbau, Debugausgabe)

```
<?xml version="1.0" encoding="UTF-8"?>
<article [...] version="3.5">
   <body>
      <html:p>
         <html:strong>
            <rtd>**</rtd>
               <text>Wichtig</text>
            <rtd>**</rtd>
         </html:strong>
      <rtd>
</rtd>
      </html:p>
      <rtd>
</rtd>
      <html:hr>
         <rtd>---</rtd>
      </html:hr>
      <rtd>

</rtd>
      <html:blockquote>
         <rtd>&gt; </rtd>
         <html:p>
            <text>Zitat</text>
         </html:p>
      </html:blockquote>
   </body>
</article>
```

Auflistung 5.15: Beispiel (WOM, um Namespace-Deklarationen gekürzt)

```
String roundtrip = Wom35Tools.restoreMarkup(document);
```

Auflistung 5.16: Rücktransformation von WOM nach Markdown

6 Evaluation

Basierend auf den Anforderungen aus Kapitel 3, die an Parser, Renderer und Pretty Printer gestellt wurden, wird in diesem Kapitel bewertet, inwiefern die entwickelten Lösungen, deren Architektur und Implementierung in Kapitel 4 und 5 beschrieben wurden, diese erfüllen.

Parser

Wie in Abschnitt 4.1 bereits beschrieben wurde, ist mit Flexmark ein existierender Parser gewählt worden, der die gestellten Anforderungen erfüllt: Der Parser ist Java-basiert, unterstützt die CommonMark-Spezifikation und stellt einen AST mit Originaldokument-Informationen bereit.

Renderer

Der Renderer erstellt bei allen Testfällen der Basisfunktionalität (basierend auf den Beispielen der CommonMark-Spezifikation), sowie in allen Testfällen der Erweiterungen die zu erwarteten WOM-Dokumente. Eine korrekte Wiederherstellung des Originaldokumentes, basierend auf den RTD-Knoten, ist in nahezu allen Fällen möglich.

Es gibt jedoch eine spezielle Situation, in der dies fehlschlägt. Die Konstellation wird in Auflistung 6.1 verdeutlicht. Im Beispiel ist ein *block quote* zu sehen, in dem sich wiederum ein *indented code block* befindet. Ein *block quote* benötigt nach dem öffnenden `>` ein Leerzeichen, der *intented code block* wird durch eine Einrückung von vier Leerzeichen gestartet.

```
1 >→→foo
```

Auflistung 6.1: Testfall Tabs (Zur Verdeutlichung werden Tabstopps als → dargestellt)

In diesem Beispiel ist die Einrückung durch zwei Tabstopps signalisiert. Tabstopps sind der CommonMark-Spezifikation nach in einem Einrückungskontext äquivalent mit 1 bis 4 Leerzeichen, sodass die gesamte Zeileneinrückung nach dem Tabstopp ein vielfaches von 4 ist. Dementsprechend signalisiert der erste Tabstopp sowohl das Leerzeichen, welches für das *block quote* benötigt wird, als auch zwei der vier Leerzeichen zur Einrückung des *indented code block*. Der zweite Tabstopp zeigt die letzten zwei benötigten Leerzeichen zur Einrückung an, sowie zwei Leerzeichen, die Inhalt des *indented code block* sind. Dieser Tabstopp ist somit teilweise RTD und teilweise Inhalt. Diese Situation kann im Augenblick nicht korrekt gelöst werden, stattdessen wird auch der zweite Tabstopp als Ganzes im RTD-Knoten eingetragen und zwei zusätzliche Leerzeichen werden in den *indented code block* als Inhalt geschrieben. Der Round Trip ist daher in diesem Fall nicht identisch mit dem Originaldokument.

Desweiteren sollte das Verhalten bei HTML-Blöcken hervorgehoben werden: Da, wie in Abschnitt 5.1.3 beschrieben wurde, HTML-Eingaben noch zusätzlich mit JSoup validiert werden, kann es dazu kommen, dass invalide Eingaben korrigiert und somit verändert werden. Dies ist gewolltes Verhalten, aber möglicherweise unerwartet und unintuitiv.

Pretty Printer

Der Pretty Printer nimmt beliebige WOM-Dokumente entgegen und erstellt darauf basierend ein Markdown-Dokument. Für einen Großteil der Testfälle wird das zu erwartende Dokument korrekt erstellt. Allerdings gibt es noch einige Sonderfälle, in denen der Pretty Printer nicht die erwartete Transformation durchführt. In insgesamt 44 der 897 definierten Testfälle erreicht der Pretty Printer nicht die erwartete Ausgabe.

Ein beispielhaftes Problem, welches bei mehreren Testfällen zu einem falschen Ergebnis führt, besteht in der Tatsache, dass WOM im Augenblick bei Links nicht zwischen Linktitel und Linktext unterscheidet. HTML und Markdown erlauben die Definition eines Linktitels, welcher nicht dem Text des Links entspricht, während WOM davon ausgeht, dass beide identisch sind.

7 Zusammenfassung & Ausblick

Das Ziel dieser Arbeit bestand in der Markdown-Unterstützung für Sweble Hub. In diesem Rahmen wurden zunächst Grundlagen zum Themengebiet behandelt. Die zu implementierenden Komponenten wurden identifiziert und Ansprüche an diese formuliert. Auf Architektur- und Designentscheidungen bei der Konzeption dieser Komponenten wurde eingegangen und tiefere Details zur Implementierung wurden dargelegt.

Grundsätzlich wurde im Rahmen dieser Arbeit eine funktionierende Lösung entwickelt, welche die gestellten Anforderungen erfüllt. Im aktuellen Zustand sind die Komponenten weit genug ausgereift, um sie zur Verwendung im Sweble Hub einzusetzen. Verbesserungen sind jedoch durchaus noch möglich: Wie in Kapitel 6 beschrieben wurde, sind noch wenige Sonderfälle vorhanden, in denen beide Komponenten nicht das gewünschte Ergebnis liefern.

Aktuell wandelt der Pretty Printer WOM-Dokumente direkt nach Markdown um. Eine mögliche zukünftige Neuimplementierung des Pretty Printers könnte erwägen, stattdessen aus dem WOM-Dokument einen Flexmark-AST zu konstruieren und die Transformation nach Markdown durch den Formatter von Flexmark vornehmen zu lassen. Dies wäre vorteilhaft, da der Formatter ausgereifter ist und eine starke Konfigurierbarkeit aufweist. Allerdings werden alle Zeichenketten in AST-Objekten von Flexmark als Teilmengen eines gemeinsamen Markdown-Quelldokumentes geführt. Bei der Transformation von einem WOM-Dokument hat man keine solche gemeinsame Dokumentquelle, die grundsätzlich vom AST benötigt wird. Es ist jedoch nicht auszuschließen, dass diese Variante dennoch umsetzbar ist.

Anhang Bill of Materials

Der folgende Abschnitt listet Fremdsoftware, die von den Implementierungen verwendet wird.

Software	**Lizenz**	**URL**
Flexmark	BSD (2-clause)	https://github.com/vsch/flexmark-java
JSoup	MIT	https://jsoup.org/
Sweble wom35-tools Sweble wikitext-to-wom35-pipeline	Copyright © by Professur für Open Source Software	https://osr.cs.fau.de/software/sweble-wikitext/
Xalan	Apache 2.0	https://xml.apache.org/xalan-j/
Xerces	Apache 1.1	https://xerces.apache.org/xerces-j/

Literaturverzeichnis

About writing and formatting on GitHub. (o.D.). Zugriff 25. Mai 2018 unter https://help.github.com/articles/about-writing-and-formatting-on-github/

commonmark-java. (2018). Zugriff 21. Juni 2018 unter https://github.com/atlassian/commonmark-java

Danilo, A., Leithead, T., Faulkner, S., Eicholz, A. & Moon, S. (2017). *HTML 5.2*. W3C. https://www.w3.org/TR/2017/REC-html52-20171214/.

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

Dohrn, H. & Riehle, D. (2011b). *WOM: An object model for Wikitext* (Techn. Ber. Nr. CS-2011-05). Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. Informatik.

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

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

Gruber, J. (2004). Markdown. Zugriff 4. Juli 2018 unter https://daringfireball.net/projects/markdown/

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

Hedley, J. (2018). jsoup: Java HTML Parser. Zugriff 17. Juli 2018 unter https://jsoup.org/

Leonard, S. (2016). *Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations* (RFC Nr. 7764). RFC Editor. RFC Editor. doi:10.17487/RFC7764

List of wikis by number of pages. (o.D.). Zugriff 28. Juli 2018 unter https://wikiindex.org/List_of_wikis_by_number_of_pages

MacFarlane, J. (o.D.). CommonMark. Zugriff 4. Juli 2018 unter http://commonmark.org/

MacFarlane, J. (2017, 1. August). CommonMark Spec. Zugriff 4. Juli 2018 unter https://spec.commonmark.org/0.28/

Mutel, A. (2016). babelmark3. Zugriff 1. Juli 2018 unter https://babelmark.github.io

Schneider, V. (2017). Writing Extensions. Zugriff 11. Juli 2018 unter https://github.com/vsch/flexmark-java/wiki/Writing-Extensions

Schneider, V. (2018). flexmark-java. Zugriff 21. Juni 2018 unter https://github.com/vsch/flexmark-java

snudown: reddit's markdown renderer. based on sundown. (o.D.). Zugriff 25. Mai 2018 unter https://github.com/reddit/snudown

Wiki Matrix: Markup Compare. (o.D.). Zugriff 28. Juli 2018 unter https://www.wikimatrix.org/syntax.php